

DB2

数据库管理 最佳实践

© 徐明伟 王涛 编著

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

IBM DB2 作为业界主流的数据库产品,广泛应用于金融、通信、烟草等行业。本书侧重于 DB2 数据库管理,以实战为主要目标,内容涵盖软件安装配置、数据库环境搭建、存储规划、数据迁移、备份恢复、锁、性能监控调优和常见的问题诊断等。通过循序渐进、深入浅出的讲解,力求让读者亲自动手实验,结合实际案例,快速掌握 DB2 知识,独立完成日常运行维护管理工作。本书作者均有 IBM 原厂的工作经历,实战经验非常丰富,本书将和大家分享他们的 DB2 数据库管理的最佳实践经验。

本书主要面向 DB2 DBA 和数据架构师。适用于具备一定数据库基础,有志于从事 DB2 DBA,或希望考取 DB2 认证,或从其他数据库转向 DB2 的读者。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

DB2 数据库管理最佳实践 / 徐明伟, 王涛编著. —北京: 电子工业出版社, 2011.9
ISBN 978-7-121-14485-1

I. ①D… II. ①徐… ②王… III. ①关系数据库—数据库管理系统, DB2 IV. ①TP311.138

中国版本图书馆 CIP 数据核字(2011)第 175740 号

策划编辑: 张春雨

责任编辑: 李利健

特约编辑: 赵树刚

印 刷: 涿州市京南印刷厂

装 订: 河北省涿州市桃园兴华装订厂

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 787×1092 1/16 印张: 35.25 字数: 909 千字

印 次: 2011 年 9 第 1 次印刷

印 数: 4000 册 定价: 89.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线: (010) 88258888。

序 一

炎炎夏日，泡上一杯香浓的咖啡，望着熙熙攘攘的人群来往于黄昏时分的大型超市、货柜车有条不紊地进出物流中心的停车场、遍布全市的银行网点仍在持续地运作……这就是我们数据库管理员的生活。这些大企业的背后是 IT 系统维系着业务的运行，而我们的数据库管理员则正维护着它们的核心系统。

在我看来，数据库管理员所从事的工作，挑战和高薪并存，令人自豪。我经常见到数据库管理员享受着不菲的待遇，特别是 DB2 管理员。尽管数据库管理员受人尊敬，大有前途，但是成为一名高水平的数据库管理员却并非易事。从国内最大的 DB2 社区 db2china 的反馈来看，缺乏高质量的 DB2 书籍是重要原因之一。

由于我的工作性质，我对数据库书籍非常关注。2011 年以前，书店里的 DB2 书籍相比 Oracle 书籍来讲，可谓寥寥无几。不过，这种情况在 2011 年有了很大改变，因为有一批 DB2 技术书籍将会陆续上市，这对有志成为高水平数据库管理员的读者来讲是一件幸事。

本书由徐明伟和王涛编写，注重实用，内容由浅及深，涵盖 DB2 的管理、运行和维护，将大量一线的实际服务和培训案例融入其中，从而将一系列相关的分散知识点真正形成了一个知识面。

今天，国内已有越来越多的技术人员在使用 DB2，相信这本书能对学习和使用 DB2 提供较大帮助，希望它能成为数据库管理员的良师益友，为您答疑解惑，点亮前进之路。

王飞鹏（IBM 中国开发实验室 DB2 资深顾问）

2011 年 7 月

序 二

关于本书作者

近些年，我主要负责人民银行部分全国大集中系统的数据库设计工作，其中多有接触 IBM 厂商的产品及相关技术专家。回想第一次与明伟接触是在 2007 年，当时明伟作为 IBM 专家，负责国库系统的数据库设计和支持，他对技术的炽热和求真态度，以及丰富的 DB2 实战经验给我留下了深刻印象。后来明伟离开 IBM 公司，从事独立的 DB2 咨询，为很多金融、烟草、通信运营商、钢铁等行业客户提供 DB2 培训和性能调优、咨询业务，获得了很好的口碑。近日，明伟嘱我为其与王涛合作的新作作序，在仔细阅读了部分章节之后，我欣然应允。

关于本书

很多朋友在谈及一本书时，时常会问“这是不是一本好书？”之类的问题，现在，借我几年前读过的一本关于数据库性能优化的书籍的阅读感受来跟大家分享一下：

2004-03-04 新书到手，彻夜通读，阅后收获颇丰。

2005-01-08 再读一年前所购之书，感觉的确是技术方面的好书，尤其针对实验指导、原理解析颇为透彻。

2006-01-14 至今日，再读性能优化相关书籍，备感自身“胸无点墨”，虽案牍颇丰，然皆为众家之言。

2007-06-07 纲举目张方能借左右而言“他物”（遍寻 DB2 优化器相关资料，无果，最后只能参考《Database Management Systems, Third Edition》）。

2008-05-01 同一位读者，不同时期读同一本书，感悟有所不同。

上述为我近些年在阅读相关书籍时的一个心路历程的缩影，单就本书而言，作为一本运维管理实践的技术书籍，涵盖了系统上线前规划、安装配置和上线后的运维管理、性能优化和问题诊断。本书不仅仅是从一个产品的视角来描述，更是以一个“从实践中来，到实践中去的”理念来与大家分享作者多年的工作积累与心得，虽部分章节只言片语一带而过，但细细品味，仍有值得回味之处。作者本着严谨、务实和求真的心态，一切案例从工作中提炼，由浅入深娓娓道来，但限于篇章及本书的受众，只好根据当前所描述的场景及专门论述的领域做一分支专题的论述。

最佳实践从来都不是一件独立而绝对的事情，要想在系统架构设计过程中设计出高性能、高并发并能满足用户需求的系统，必须在综合业务需求、专业技术特性、开发规范等众多系统建设参照物的前提下，开展系统规划、数据库架构、应用架构、存储规划等设计工作。而本书正是借助作者多年的故障诊断、性能优化、企业培训等诸多方面的丰富的经验，向大家展示了 DB2 在当前系统信息化建设过程中的最佳设计理念。理论与实践相结合是最好的学习方法，而本书正是希望通过这样的论述方式将精彩的实战内容展示给读者。但有道是“纸上得来终觉浅，绝知此事要躬行”，希望大家在“为伊消得人憔悴”的征途上，“衣带渐宽终不悔”地勇往前行。

明伟、王涛他们睿智、勤奋并乐于分享，经过深思熟虑奉献给大家一本技术与实践并重的实战指导书籍，让我们在阅读的过程中也一并感受到了作者的严谨与新颖的阐述问题的方式，同时，我们也透过作者对技术的分析，领略了他们对技术的热爱与乐于分享的激情。

希望本书能成为您数据库征途上的挚友，成为您起航前往罗马的灯塔。

李小平（人民银行金融电子化公司数据架构师）

序 三

2005 年夏天，我在 IBM 多伦多实验室认识王涛，作为“前辈”，在他加入 IBM 后我有幸成为他的指导者（mentor），与他共同工作过一段时间。

与王涛的最初接触中，他就给我留下了很深的印象。作为刚刚从大学毕业的新人，他对计算机技术与概念的领悟之快令人惊讶，很多基本的原理只需要简单解释一遍，他就可以从中推论出许多相对复杂的内部实现机制与设计初衷。同时，王涛是我见过最有天赋的 C 程序员之一，尽管技术支持的工作不需要亲自书写代码，但是我们需要阅读并理解大量可能隐藏着设计缺陷的代码，并从中找出发生问题的部分。王涛可以从 DB2 众多模块的复杂逻辑中找到隐藏极深的 bug，并在最短时间内提出有效的解决方案。

王涛于 2011 年初加入了 DB2 三线高级问题诊断小组（Advanced Problem Determination, APD），该小组在全球只有不到 15 人，由世界各地顶尖的 DB2 问题分析专家组成（亚太地区 4 人，分别在日本、韩国与澳大利亚），算是 IBM DB2 领域的“最后一道防线”。该小组负责分析世界各地最为紧急与复杂的问题。当 DB2 引擎的开发人员无法判定问题出现的模块，或者需要帮助理清复杂问题的思路时，通常会寻求高级问题诊断小组的帮助。而很多时候，作为提供问题解决方案的技术负责人，王涛需要协调各个不同模块与产品之间的开发人员，精确定位每一个模块各自所需研究的方向。

在短短的几年中，我欣喜地看到王涛从一个大学毕业的新人一步步成长为 DB2 世界级顶尖的专家，成功地为无数家全球顶尖企业解决过各种各样的疑难杂症，其中包括美国国防部与军方、波音、可口可乐、雀巢、SAP、HP、丰业银行、中国电信、美洲银行、汇丰银行、三星等知名机构与企业。

如今我作为数据库性能架构师，同样在不断寻找着市面上理论与实践并重的资料。王涛与徐明伟合写的这一本书在高层面用简单的语言阐述了许多 DB2 中苦涩难懂的模块，又能够结合王涛在多年的技术支持生涯中解决的经典案例，为读者带来实际的问题分析思路。

在阅读本书后，我个人的感觉是，本书并没有像很多其他的资料一样，如同“参考手册”般用严谨并枯燥的语言尝试覆盖每一个模块的所有细枝末节（比如各种参数与语法的使用），而是着重于最重要的“概念”的剖析，以及阐述“为什么”一个产品或者应用需要这样或者那样的设计。

客观地讲，本书并不是一种“随用随查”类型的参考资料。由于作者使用了大量的篇幅进行原理的剖析与案例的讲解，并没有在细节的知识点上做过多的纠缠。读者在需要某些特定的知识点时，可以参考 IBM 的信息中心得到语法信息。本书的目的是为了给读者一个高层面的理解，知道每一个模块“为什么这样设计”，以及对不同类型问题“应该如何思考”的启发。在读者阅读案例时，我建议并不要过于专注每一个案例的解决方法，而是要理清问题诊断的思路。同样现象的问题可能会有 100 种不同的原因，而作者在本书中并没有尝试列出所有可能发生问题的方面，而是给出一系列一步步的思考过程，让读者在日常工作中尽量能够做到真正的“问题诊断”，而不是“凭经验尝试”。

相信本书能够为真正希望了解 DB2 的读者打开一扇大门，在知道不同语法命令的同时，能够深入地理解产品本身的设计思路与问题诊断的思考过程，成为您在成长道路上的一个朋友。

唐 迅（数据库性能架构师）

2011 年 5 月于多伦多

序 四

从 2005 年至今，我主要负责中国移动通信集团海南有限公司 BOSS 系统和经营分析系统的数据库管理、维护和优化工作，也负责相关项目的架构设计。正如不能把所有的鸡蛋都放在一个篮子里一样，企业在选型的时候一般会平衡几家厂商，数据库产品也是如此。目前，在通信行业，BOSS 计费系统一般架构在 Oracle 数据库上，而一些仓库类系统选择 DB2 的比较普遍，当然，这几年一些交易型系统也开始越来越多地选用 DB2 数据库。

作为同时维护 Oracle 和 DB2 的一线 DBA，我与 IBM、Oracle 厂商或第三方的相关技术专家有过频繁接触，给我印象较深的是徐明伟，当时他负责我公司经营分析系统数据仓库的性能调优工作，帮助我们解决了困扰很长时间的日报和月报表运行时间慢的问题。他在 DB2 数据库优化工作中视野开阔、手段多样、实战经验丰富，这与他多年的实战经验积累是分不开的。

听闻他要和另外一位顶级 DB2 专家王涛写书的消息，我感觉这对一线 DBA 是个福音。回想自己当初自学 DB2 时，可供参考的资料和书籍少之又少，大大增加了学习曲线。

作为 DBA，需要的是实际操作和动手能力，而非一些枯燥的理论分析。徐明伟和王涛合著的这本书用朴素的语言无私地与我们分享了他们多年来积累的实践经验。这些实践涵盖从系统上线前的规划、上线的安装配置和上线后的运行维护管理、性能优化和问题诊断等多方面的丰富经验。在案例的分析上，着重思路、方法和手段，而非单纯的结论。

对于一线 DBA 和即将步入这个行业的工程师，本书可以作为一本绝佳的参考书，相信每个人都必将有所收获。同时也非常期待两位作者的高级数据库管理著作。

吉训尊（海南移动通信公司 首席 DBA）

2011 年 7 月

前 言

写作背景

随着信息技术的发展，企业对 IT 的依赖不断增长，数据规模不断扩大，对高可用、高性能也提出了更高的要求，这些都对 DBA 提出了更高的挑战。与此同时，随着投入的增大，IT 成本控制也成为每个公司的迫切需求。现在企业对 DBA 的要求已经不仅仅是单纯的维护，还要求从整体架构、存储规划、应用设计和性能优化等方面提供咨询建议，以减少数据库系统的软硬件支出，提高企业竞争力。

DB2 数据库是 IBM 信息管理家族的核心产品，与 Oracle、SQL Server 一起占据着国内关系数据库领域的大部分市场份额，广泛应用在金融、通信、电力、烟草等行业。但与其他数据库相比，DB2 相对封闭，学习曲线较陡，市场的书籍也不多，导致用户出现问题时很难独立诊断和解决。

这些我是深有感触的，在 IBM 原厂工作的 5 年到目前 DB2 独立咨询，我一直战斗在一线，曾经帮助很多银行、通信、电力、烟草、高速等客户进行 DB2 运维支持、性能调优和问题诊断，并为几十家企业提供 DB2 企业内训。在这个过程中，与很多 IT 部门经理、DBA 进行过沟通和交流，大家的普遍感觉是 DB2 比较稳定，在海量数据处理方面性能很好，但 DB2 的公开资料及懂 DB2 的工程师太少，导致运维成本相对较高。特别希望市场上能多出现一些 DB2 实战的书籍。

信息技术的发展日新月异，留给 DBA 学习的时间越来越短，独立承担运维管理任务并快速解决问题是对每个 DBA 的基本要求。为了让一些初学者少走弯路，让中级水平的人快速提升，高手也能有所收获，决定写一本 DB2 数据库运维管理的书，将自己多年的实战经验毫无保留地奉献给大家，也算为 DB2 的推广做一点微不足道的贡献。

有了想法，真正写起来却是比较孤寂和艰辛的，因为白天还要为客户做一些性能调优、问题诊断和培训工作，只能利用晚上的宝贵时间。当我邀请我的好朋友王涛一起来写时，立即得到了他的积极响应，当晚我们就完成了该书的大概框架。以后的进展就比较顺利了，我们将每天写的部分互相审阅，交流意见和想法，最大程度地保证本书的质量。

本书在创作之初就定位为“面向实践”，前前后后共串联了几十个命令和工具，每个重要的命令和工具均配有相关实例演示，便于实战操作。在写作过程中，我们也随时补充在客户现场遇到的一些调优和问题诊断案例，并积极听取客户和培训学员的一些宝贵意见和建议。

由于 DB2 的知识点太多，要想在一本书里兼顾所有几乎是不可能的，因此，决定将一些相对高级但又比较实用的特性，如数据库分区、表分区、压缩、MQT、高级优化器、HADR、DB2 联邦和 Q 复制等集成到《DB2 数据库高级管理》一书，这样大家可以根据自己的需要来选择。

本书组织结构

第一部分（第 1、2 章）DB2 概述

本部分共分两章，第 1 章概要介绍了 DB2 的产品发展和演变历史，第 2 章概述 DB2 体系结构，从静态和动态两个方面介绍了 DB2 的对象层次关系和执行流程。该部分起到提纲挈领、统领全局的作用，更有助于后续章节的把握。

第二部分（第 3～7 章）DB2 部署和规划

本部分共分 5 章，详细介绍了从产品安装到实例创建、数据库创建、表空间规划管理、表创建到访问数据的每一个步骤。存储规划是数据库系统上线前最重要的环节，如果规划不好会对以后的性能造成很严重的影响，并且系统一旦上线就很难改动。我们会与大家一起分享部署规划的最佳实践。

第三部分（第 8～11 章）DB2 运维管理

本部分共分 4 章，数据迁移章节介绍了数据迁移的方法和几个最常用的命令：`export`、`import`、`load`、`db2move`、`db2look`、`db2dart` 等，针对最常遇到的问题，我们用问答的方式演示了解决方案。日志原理较难理解，也是最容易出问题的地方，我们用最简洁、易懂的语言向大家阐述。作为一名 DBA，确保数据安全和可用是最重要的任务（注意：没有之一），在备份和恢复章节，通过十几个案例让你对备份恢复技术烂熟于心。

本部分是核心内容，信息量大，知识点多，需要读者多实践、多操作才能融会贯通。

第四部分（第 12～16 章）DB2 监控和调优

监控和调优本身是作为运维管理的一部分，但由于监控调优的内容相对高级和深入，因此另成体系。本部分共包含 5 章，DB2 通过锁来实现多用户并发情况下数据的一致性，了解和掌握 DB2 进程模型和内存模型对于调优和问题诊断都有重要的意义，在监控章节，重点介绍了几个命令：`snapshot` 快照、`db2pd`、`db2top` 等，并重点介绍一些常用的 KPI。在调优章节，通过几个真实案例介绍性能分析和调优的思路和方法。

第五部分（第 17、18 章）DB2 问题诊断及安全

本部分包含两章内容，在实际操作和运维中，几乎每天都不可避免地遇到各类问题，有些问题可能会对系统造成严重影响，第 17 章介绍了几种问题诊断的方法，学会这些方法和工具将大大加快问题诊断和处理的速度。第 18 章介绍了数据库安全，可以保证数据库安全稳定的运行，为企业和个人减小损失。

致初学者

学习 DB2 有前途吗？

DB2 好找工作吗？

没有环境，能学好 DB2 吗？

学 DB2 好还是 Oracle 好？

前 言

这些几乎是每个初学者都曾经遇到的问题。对于前两个问题，我们所能够回答的就是只要你学得好，工作就好找，就会有前途，现在很多公司在热招 DB2 职位，只要机会来的时候能够准备好、把握住就可以了。

对于第三个问题，有人可能会说，现在绝大多数的生产都是 UNIX 系统，没有环境。其实 Linux 上的 DB2 命令和 UNIX 上几乎完全一样，我们完全可以轻松地利用 Linux 虚拟机开始学习，包括一些问题的模拟、性能优化和了解仓库特性，都可以在 Linux 中实现。

对于第四个问题，这是个见仁见智的问题。如果是从来没有接触过数据库的读者，并且有志于在数据库领域发展，我们建议直接学 DB2，毕竟先入为主，比从 Oracle 转过来要轻松许多。这个市场上懂 Oracle 的人太多了，高手如云，想出人头地做出成绩太难了。

因此，对于初学者的建议就是抓紧行动，去除浮躁，踏踏实实地把基础打牢，待机会来临的时候抓住。

致谢

本书在写作过程中得到了很多朋友的支持、鼓励和帮助。特别感谢我的好兄弟管伟（北京快通高速路有限公司首席 DBA）、王敬东（北京电讯盈科首席 DBA）和孙云峰（深圳市共济科技有限公司产品经理），他们几乎逐字审阅了本书内容，从用户的视角提出了很多宝贵意见和建议。

感谢加拿大丰业银行 DB2 性能优化架构师唐迅，IBM 加拿大实验室的专家 Arthur Chen、Deliang Han、Angela Yang 和 Shen Li，他们从专家视角审核了本书的大部分章节。

最后，感谢我的家人和朋友在背后的默默支持，是他们激励着我不断前进。

徐明伟

2011 年 5 月

目 录

第一部分 DB2 概述

第 1 章 DB2 产品介绍	1
1.1 数据模型	1
1.2 DB2 历史	2
1.3 DB2 版本	3
1.4 DB2 9 主要功能增强	5
1.5 DB2 认证	8
1.6 DBA 的任务和职责	8
1.7 IBM 信息管理产品概述	9
1.8 小结	10
1.9 判断题	11
1.10 参考文献	11
第 2 章 DB2 体系结构	12
2.1 DB2 体系结构简介	12
2.2 对象层次关系	15
2.3 数据访问过程	16
2.4 数据库工具	18
2.5 小结	19
2.6 判断题	19
2.7 参考文档	20

第二部分 DB2 部署和规划

第 3 章 安装 DB2 软件	21
3.1 软件安装	21
3.1.1 软件获取	22
3.1.2 安装前检查	22
3.1.3 安装	23
3.1.4 补丁升级	25
3.1.5 版本升级	28
3.2 小结	30
3.3 判断题	31
3.4 参考文档	31

Contents

第 4 章 实例管理	33
4.1 什么是实例	33
4.2 创建实例	34
4.2.1 在 Windows 平台下创建实例	34
4.2.2 在 UNIX/Linux 平台下创建实例	35
4.3 启动/停止/列出实例	37
4.4 更新实例	38
4.5 删除实例	39
4.6 实例参数	39
4.7 管理服务器 (DAS)	40
4.8 小结	41
4.9 判断题	41
第 5 章 数据库创建和存储管理	43
5.1 数据库结构	43
5.2 建库、表空间	45
5.3 表空间维护管理	50
5.3.1 表空间监控	50
5.3.2 表空间更改	52
5.3.3 表空间状态	55
5.3.4 表空间高水位	59
5.3.5 深入 DMS 表空间	65
5.4 存储设计最佳实践	67
5.5 小结	71
5.6 判断题	71
第 6 章 数据库连接	73
6.1 远程连接概述	73
6.2 节点和数据库编目	74
6.3 常见的数据库连接问题	76
6.4 小结	78
6.5 判断题	79
第 7 章 数据库对象	80
7.1 模式	81
7.2 表	81
7.2.1 表约束	84
7.2.2 表状态	85
7.2.3 表压缩	86

7.2.4 表分区	87
7.3 索引	88
7.4 视图	94
7.5 昵称	94
7.6 序列 (Sequence)	94
7.7 自增字段	96
7.8 大对象 (LOB)	98
7.9 函数	101
7.10 触发器	102
7.11 存储过程	103
7.12 小结	109
7.13 判断题	109
 第三部分 DB2 运维管理	
第 8 章 数据迁移	110
8.1 数据迁移概述	111
8.2 文件格式	111
8.2.1 DEL 格式	111
8.2.2 ASC 格式	112
8.2.3 PC/IXF	112
8.2.4 Cursor	112
8.3 export	112
8.4 import	113
8.5 load	115
8.5.1 load 步骤及原理	115
8.5.2 load 表状态	118
8.5.3 load 的 copy 选项	119
8.5.4 set integrity 完整性检查	125
8.6 12 个怎么办	129
8.6.1 出现了 load pending 了怎么办	129
8.6.2 在客户端 load 问题	130
8.6.3 要加载的数据是 Excel 格式怎么办	131
8.6.4 要导出/加载的数据不是逗号/双引号分隔怎么办	131
8.6.5 文件中的列比要导入的表中的字段多怎么办	133
8.6.6 文件中的列比要导入的表中的字段少怎么办	133
8.6.7 要导入/导出大字段 (LOB) 怎么办	134
8.6.8 sequence 数据怎么办	135
8.6.9 导入 identity 数据怎么办	136
8.6.10 要加载的数据有换行符怎么办	139

Contents

8.6.11	迁移出现乱码怎么办	141
8.6.12	表数据从一个表空间迁移到另外一个表空间怎么办	143
8.7	db2look/db2move	146
8.7.1	db2move 工具介绍	146
8.7.2	db2look 工具介绍	146
8.7.3	db2look+db2move 迁移案例	147
8.8	db2dart	151
8.9	小结	153
8.10	判断题	153
第 9 章	备份恢复	155
9.1	备份恢复概述	155
9.2	DB2 日志	158
9.2.1	日志机制和原理	158
9.2.2	日志参数配置最佳实践	163
9.2.3	日志监控和维护管理	168
9.2.4	其他日志相关的考虑	171
9.2.5	经常遇到的日志问题	172
9.3	备份	176
9.3.1	离线备份	178
9.3.2	在线备份	178
9.3.3	表空间备份	179
9.3.4	增量备份	179
9.3.5	备份介质检查	180
9.3.6	备份监控	183
9.4	恢复	183
9.4.1	崩溃恢复	183
9.4.2	版本恢复	184
9.4.3	前滚恢复	192
9.4.4	删除表恢复 (dropped table recovery)	196
9.5	常见备份恢复场景及遇到的问题	199
9.5.1	宕机后数据库连接 hang 的处理	199
9.5.2	循环日志模式下的离线备份恢复	200
9.5.3	归档日志模式下的备份恢复	201
9.5.4	归档日志模式下前滚恢复的几个时间戳	203
9.5.5	同版本不同实例下的数据库备份恢复 (表空间是自动存储管理)	205
9.5.6	同版本不同实例下的数据库备份恢复 (表空间是非自动存储管理)	206
9.5.7	不同版本不同实例下的数据库恢复	206
9.5.8	从生产库到测试库恢复的案例分析	207

9.5.9	历史文件过大造成数据库停止响应案例分析	209
9.5.10	恢复时解压类包问题	210
9.5.11	备份失败问题	211
9.6	小结	212
9.7	判断题	212
第 10 章	DB2 日常运维	213
10.1	日常运维工具概述	213
10.2	Runstats	214
10.2.1	Runstats 原理	214
10.2.2	Runstats 用法	215
10.3	Reorg	217
10.3.1	为什么需要 Reorg	217
10.3.2	Reorg 用法	221
10.3.3	Reorg 最佳实践	225
10.4	Rebind	226
10.5	获取数据库占用空间的大小	227
10.6	获取某个表空间占用空间大小	228
10.7	获取某个表/索引占用空间的大小	229
10.8	小结	231
10.9	判断题	232
第 11 章	锁和并发	233
11.1	锁和隔离级别概述	233
11.2	锁的模式和兼容性	235
11.2.1	表锁模式	236
11.2.2	行锁模式	239
11.2.3	表锁和行锁兼容性	243
11.3	锁的各种问题	245
11.3.1	锁等	245
11.3.2	锁超时	246
11.3.3	死锁	246
11.3.4	锁升级	248
11.3.5	锁转换	249
11.4	锁监控和诊断	249
11.4.1	锁的分析思路和方法	249
11.4.2	锁升级(lock escalation)的诊断分析	250
11.4.3	锁等(lock wait)的捕获与诊断分析	250
11.4.4	锁超时(lock timeout)的捕获与诊断分析	254

Contents

11.4.5 死锁 (deadlock) 的捕获与诊断分析	259
11.4.6 9.7 锁事件监控器	263
11.5 锁和并发调优	269
11.6 Currently Committed 机制	270
11.7 小结	273
11.8 判断题	273

第四部分 DB2 监控和调优

第 12 章 DB2 进程/线程模型	274
12.1 提要	274
12.2 从操作系统看进程和线程	275
12.3 DB2 V8/V9.1 进程模型	278
12.3.1 代理进程	279
12.3.2 分区内并行	280
12.3.3 分区间并行 (DPF)	281
12.3.4 预取进程 (prefetcher)	282
12.3.5 页面清理进程 (Page Cleaner)	284
12.3.6 其他进程	285
12.3.7 实例 / 数据库启动步骤	287
12.4 DB2 9.5/9.7 线程模型	289
12.5 小结	291
12.6 判断题	291
第 13 章 DB2 内存模型	292
13.1 从操作系统看内存	292
13.2 DB2 8/9.1 内存模型	294
13.2.1 实例共享内存段	295
13.2.2 数据库共享内存	296
13.2.3 应用程序组共享内存	299
13.2.4 私有内存	300
13.3 DB2 9.5/9.7 内存模型	301
13.3.1 实例内存	302
13.3.2 应用程序内存	302
13.3.3 自动内存调节 (Self Tuning Memory Management, STMM)	303
13.4 内存监控	305
13.4.1 db2mtrk	305
13.4.2 db2pd -dbptnmem	306
13.4.3 db2pd -memset / db2pd -mempool	307
13.5 小结	310

13.6 判断题	310
第 14 章 DB2 监控工具	312
14.1 snapshot 命令行监控	313
14.2 snapshot 管理视图	314
14.3 db2pd	315
14.4 db2top	328
14.4.1 实时监测	329
14.4.2 历史信息收集	330
14.4.3 子窗口	331
14.5 DB2 事件监控器	340
14.6 小结	341
14.7 判断题	341
第 15 章 性能监控和分析方法	343
15.1 收集数据	343
15.1.1 操作系统级别性能监控	344
15.1.2 数据库级别性能监控	354
15.1.3 数据收集的频度	387
15.1.4 小结	389
15.2 分析数据	389
15.2.1 瓶颈分类与原理介绍	389
15.2.2 性能分析思路	397
15.2.3 性能分析案例	403
15.2.4 小结	437
15.3 判断题	437
第 16 章 优化器与性能调优	438
16.1 优化器简介	438
16.2 性能调优简介	450
16.2.1 索引	457
16.2.2 排序	463
16.3 KPI	477
16.3.1 缓冲池命中率 (bufferpool hit ratio)	477
16.3.2 有效索引读	479
16.3.3 包缓存命中率 (package cache hit ratio)	480
16.3.4 平均结果集大小	481
16.3.5 同步读取比例	482
16.3.6 数据、索引页清除	483

Contents

16.3.7	脏页偷取 (dirty page steal)	483
16.3.8	缓冲区读写 I/O 响应时间	484
16.3.9	Direct I/O 时间	485
16.3.10	直接 I/O 读取 (写入) 的次数	485
16.3.11	编目缓冲区插入比例	486
16.3.12	排序指标	486
16.3.13	基于事务的指标度量	487
16.3.14	检测索引页扫描	490
16.3.15	日志写入速度	491
16.3.16	查询执行速度	491
16.3.17	实例级性能指标	492
16.3.18	操作系统级指标	492
16.4	小结	494
16.5	判断题	494

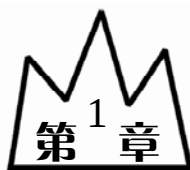
第五部分 DB2 问题诊断

第 17 章	问题诊断	495
17.1	概述	495
17.2	日志信息错误	496
17.3	宕机	498
17.4	挂起	503
17.5	错误信息	506
17.5.1	SQLCODE	507
17.5.2	db2trc	513
17.5.3	strace	519
17.6	分析数据收集工具	522
17.7	IBM 服务支持体系	528
17.8	小结	528
17.9	判断题	529
第 18 章	数据库安全	530
18.1	安全概述	530
18.2	认证机制	531
18.3	权限控制	532
18.3.1	管理权限	532
18.3.2	对象特权	535
18.3.3	权限设计案例	537
18.4	审计机制	540
18.5	DB2 安全最佳实践	545

目 录

18.6	其他安全技术增强	545
18.7	小结	545
18.8	判断题	545
18.9	参考文献	546

第一部分 DB2 概述



DB2 产品介绍

DB2 是 IBM 公司研发的关系数据库产品，目前广泛应用于金融、通信、烟草、交通等行业，在 IBM 随需应变的战略体系中扮演着重要角色。

本章组织内容如下：

- 数据模型。
- DB2 发展历史。
- DB2 产品版本。
- DB2 9 的主要功能增强。
- DB2 认证。
- DBA 的任务和职责。
- IBM 信息管理产品概述。

1.1 数据模型

在数据库技术发展和演变的过程中，出现了 3 种重要的数据模型：网状模型、层次模型和关系模型。

网状模型起源于 20 世纪 60 年代，它是用网络结构表示实体类型及其实体之间联系的模型，实体之间构成一张网状图。层次模型是在网状模型基础上发展起来的，它是用树形结构表示实体类型及实体间联系的模型，现实中如组织架构、家族族谱等都是这种树形（层次）结构的例子。最著名的层次数据库系统是 IBM 公司在 1968 年研制的 IMS（Information Management System），这是最早的大型机数据库系统，也称为 DB1，至今仍在一些银行发挥着重要作用。网状模型和层

次模型都是现实世界的真实模拟，但并无数学理论的支持，在使用上存在比较大的约束和限制。

1970 年，有“关系数据库之父”之称的 IBM 研究员 E.F.Codd 博士在刊物 *Communication of the ACM* 上发表了题为“A Relational Model of Data for Large Shared Data banks（大型共享数据库的关系模型）”的论文，文中首次提出了数据库关系模型的概念，奠定了关系模型的理论基础。后来 Codd 又陆续发表多篇文章，论述了范式理论和衡量关系系统的 12 条标准，用数学理论奠定了关系数据库的基础。IBM 的 Ray Boyce 和 Don Chamberlin 将 Codd 关系数据库的 12 条准则的数学定义以简单的关键字语法表现出来，里程碑式地提出了 SQL 语言。关系模型简单明了，具有坚实的数学理论基础，所以，一经推出就受到了学术界和产业界的高度重视和广泛响应，并很快成为数据库市场的主流。

关系模型就是二维表格模型，关系型数据库就是由二维表及其之间的联系组成的一个数据组织。20 世纪 80 年代以来，基于关系模型涌现出一批数据库产品，甲骨文公司（Oracle）于 1979 年开发了业界第一个关系数据库 Oracle，英孚美公司在 1981 年发布了 Informix 数据库，IBM 于 1982 年发布了 SQL/DS for VSE/VM，这是业界第一个以 SQL 作为接口的商用数据库管理系统，1983 年，DB2 for MVS 面世，DB2 正式诞生（IMS 是 IBM 第一代数据库，DB2 是第二代，因此取名 DB2）。1987 年，Sybase 公司推出了 Sybase SQL Server。1988 年，微软与 Sybase 合作开发了 SQL Server，1994 年两家公司分道扬镳，各自发展自己的产品，微软的就是 MS SQL Server，而 Sybase 的叫 Sybase ASE。1996 年，一个叫 MySQL 的小型开源关系数据库诞生，时至今日，它已经成为众多中小企业建站的首选。

时过境迁，30 多年后，数据库市场已经发生了巨变。IBM 于 2001 年收购了 Informix，并将很多 Informix 的优良特性整合到 DB2 中，Informix 的市场份额逐渐萎缩；Sybase 于 2010 年被 SAP 公司收购，目前在移动数据库领域占据 70% 的份额，但传统的数据库产品份额逐年递减；MySQL 命运坎坷，先被卖到 Sun 公司，后 Sun 公司又被 Oracle 公司收购，它的前途几乎没多少人看好。因此，当今的数据库市场几乎呈现出 Oracle、DB2 和 SQL Server 三足鼎立的局面。

1.2 DB2 历史

DB2 最早运行在 MVS 大型机系统，MVS（多虚拟存储）开创了操作系统的一个新纪元，我们现在听得较多的大型机 z/OS 系统就是来自 MVS，国内的五大行使用的都是 IBM 大型机，其上运行的是主机 DB2。

随后 IBM 研制出 AS400（Application System/400）中型机系统，AS400 具有先进的体系结构，并能不断吸收、融合新的技术，因而快速成为当时全球使用最广泛的中型机，AS400 使用的操作系统是 OS/400，OS/400 含有一个功能强大的数据库，即 DB2 数据库。AS400 以稳定性著称，时至今日，仍有一部分客户在使用，但它的性价比相对较低，随着小型机处理能力的日益增强，AS400 将逐步退出历史舞台。

尽管大型机和中型机功能强大，可靠性/稳定性高，但技术相对封闭，可移植性差。历史注定是强人创造的，20 世纪 80 年代初，贝尔实验室的肯·汤普逊（Kenneth Lane Thompson）和丹尼斯·里奇（Dennis MacAlistair Ritchie）用汇编语言开发了一个操作系统原型，即 UNIX 系统。为了便于移植，他们又开发了今天我们还在使用的大名鼎鼎的 C 语言，并成功地用 C 语言重写了 UNIX 的第三版内核。UNIX 操作系统的修改、移植相当便利，并且具有多用户、多任务的特点，这为 UNIX 日后的普及打下了坚实的基础。随后，UNIX 得到了学术界和产业界的极大支持，发展出很多版本，比较著名的有 IBM AIX、Sun Solaris 和 HP-UX 等。1993 年，支持 AIX 平台的 DB2 面世，1994 年，DB2 支持 HP-UX 和 Solaris。1996 年是标志性的一年，DB2 正式更名为 DB2 UDB 5（统一数据库）。1999 年，DB2 开始支持 Linux 系统。随后的 10 年中，IBM 顺应市场需求，通过收购和自主研发，将很多产品特性集成到 DB2 产品中。2004 年，DB2 8.2 版本发布，2006 年，DB2 9.1 版本发布，2007 年，DB2 9.5 版本发布，2008 年，DB2 9.7 版本面世，2009 年，DB2 9.8 版本发布，DB2 10 版本计划在 2011 年发布。DB2 版本演变历史如图 1.1 所示。

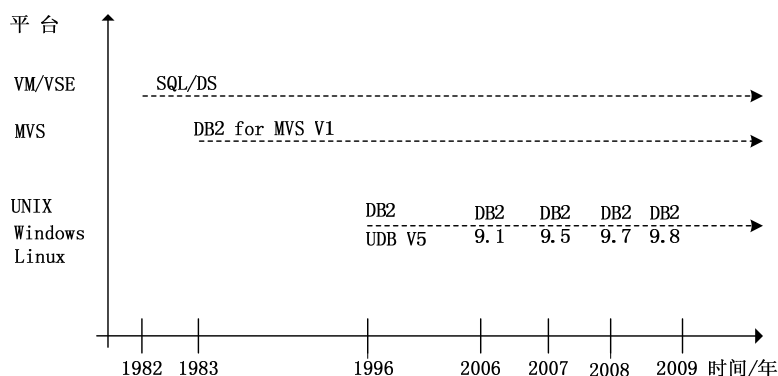


图 1.1 DB2 版本演变历史

IBM 机器型号包含 4 种系列：z 系列（zero，零宕机，IBM 大型机系统），i 系列（主要指 AS/400 中型机），p 系列（IBM 小型机，power 处理器，以前叫 RS/6000）和 x 系列（x86 处理器系列，主要是 PC Server）。针对 z 系列的 DB2，称为 DB2 for z/OS；针对 i 系列的 DB2，称为 DB2 for AS/400；针对 p 和 x 系列的，一般统称为 DB2 for LUW（DB2 for Linux 和 UNIX and Windows）。z 系列和 i 系列系统相对封闭，学习曲线较陡，而 p 系列和 x 系列则称为开放平台（或分布式平台）。在本书中，如果不特别强调，我们说的 DB2 专指 DB2 for LUW。

1.3 DB2 版本



DB2 for LUW 提供了很多版本满足不同客户的需求，为个人、部门和大型企业应用提供支持。除了 Everyplace Edition 和 Cloudscape 外，其余的各个版本底层核心代码是一样的，但由于支持的授权和价格不同，有些功能在某些版本受限，企业可根据自己的需求选择合适的版本。从节省成本考虑，小型企业可考虑 Express 或 Workgroup 版本，大中型企业考虑用 Enterprise

4 DB2 数据库管理最佳实践

版本，海量数据仓库系统用 Enterprise with DPF，而个人或小型企业可考虑用免费的 Express-C 版本。

DB2 版本如下：

- DB2 Everyplace Edition。
- DB2 Cloudscape。
- DB2 Personal Edition。
- DB2 Express-C Edition（免费版）。
- DB2 Express Edition。
- DB2 Workgroup Edition。
- DB2 Enterprise Edition。
- DB2 Enterprise Edition with DPF。

DB2 Everyplace Edition 主要用于移动设备，如手机、PDA、手提电脑、嵌入式系统等。Everyplace 版非常精简，只有 350KB 大小，但支持大部分 SQL 处理。在通常情况下，使用 Everyplace 存取实时移动数据，然后将其复制到后台数据系统中。

DB2 Cloudscape 最早是 Informix 公司的产品，被 IBM 收购后在很多产品中用它作为内嵌数据库。2004 年，IBM 将 Cloudscape 开源，赠送给了 Apache，更名为 Derby。Apache Derby 是一个完全用 Java 编写的数据库，非常小巧，核心部分只有 2MB，既可以作为单独的数据库服务器，也可以内嵌在应用程序中使用。

DB2 Personal Edition 个人版只支持单用户使用，不支持来自远程的数据库连接。对于一些终端应用，如 POS 机，可以安装个人版以节省成本。实际上，个人版的使用非常少。

DB2 Express Edition 提供了一个低成本、入门级的数据库服务器，主要用于小型企业或部门，支持 Windows 和 Linux 系统。Express 版与 DB2 工作组版有相同的功能和特性，但它只支持两颗 CPU 和 4GB 内存。

DB2 Express-C Edition 并不是一种“真正的”DB2 版本，它是为合作伙伴及开发社区所设计的，是当今世界上最高级的免费数据库管理系统之一，可通过 IBM 官方网站下载。Express-C 版的核心代码与其他版本并无差别，不受数据量大小和用户数限制，但只支持 2 颗 CPU 和 2GB 内存。与 Express 版本相比，Express-C 不支持 DPF（多分区），HADR 和复制功能。对于学生和个人爱好者，可使用 Express-C 版本学习 DB2 或考取认证，对于企业客户，可在该版本中进行应用程序开发和测试等。DB2 Express-C 只支持 Windows 和 Linux 系统。

DB2 Workgroup Edition 工作组版适用于中小企业或部门，可用于 Windows、Linux 和 UNIX 系统，最大可使用 16GB 内存和 4 颗 CPU。

DB2 Enterprise Edition（ESE）是使用最广泛、功能最完整的 DB2 版本，支持各类企业级需求，如表分区、物化视图、多维集群、HADR 等。

DB2 Enterprise Edition with DPF 基于 ESE 版本，如果购买了 DPF license，则支持数据库分区特性，一般用于数据仓库、商业智能等海量数据库系统。例如，国内很多省移动公司都是用 DB2 with DPF 构建经分系统，支持几 TB 到几百 TB 的数据分析处理（注：DPF 是 Database Partition Feature 的缩写）。

1.4 DB2 9 主要功能



由于 DB2 UDB V8 版本将逐渐退出历史舞台，本节我们重点介绍 DB2 9 版本的主要功能，包括 9.1、9.5、9.7、9.8 等。比起之前的版本，DB2 在 9 版本中对系统架构做了大量的调整。在 9 版中的每一个小版本中，DB2 都着重在某一特定的方面进行提高。当然，每一个版本中包含着大量的新特性与功能，总的来说，每一个版本都在包含了前一个版本功能的前提下，有着自己的针对性。

1. DB2 9.1 新特性

在 DB2 9.1 版本中，主要的新特性就是原生 XML、表分区与表压缩功能。从系统框架上看，DB2 9.1 与 V8 相比并没有特别多的改变，总体的内存和进程模型基本相同。

XML 则是 IBM 在 DB2 9.1 版本中主推的一个功能。如今在很多系统中，关系模型已经不能满足业务的需要了，而过去的 XML Extender 是将 XML 文档切分后用关系模型存储在关系数据库中，对性能和稳定性带来了很大的影响。在 9.1 版本中，XML 作为一种 DB2 内建的数据类型，支持 XML 文档的解析、存储和快速访问，在性能和稳定性上都有极大的提高。XML 特性目前在医疗行业有很多成功案例。

表分区则是一个非常有用的功能，能够让用户在最短的时间内将某一个表分区连接到一个表，或者从表中断开。这个功能在一些数据仓库和在线交易系统中被广泛使用。

DB2 9.1 引入的行数据压缩能够为用户节省大量的磁盘空间，同时由于每一个数据页可以容纳更多的数据，读取同样多的数据则需要更少的 I/O，也就是说，从某种程度上对性能也会带来提升。一般我们都认为性能与空间是成反比的关系，但是对于 DB2 9.1 的数据压缩，既可以提高性能也能够节省空间，因此是 DB2 9.1 中的一大亮点。DB2 提供的行压缩比可以高达 80%，已经获得很多客户的认可和好评。

2. DB2 9.5 新特性

在 DB2 9.5 版本中，抛开一些细节的改变不提，最主要的更改是架构的变更：在 UNIX 系统中原进程模型变为线程模型。实际上，这个改变并没有想象中的那样巨大。在 DB2 for Windows 中，DB2 一直使用着线程模型，因此该技术已经相当成熟。一般来说，在操作系统内部的进程间切换的开销要远远大于线程之间的切换。因此，由进程模型变为线程模型会对系统性能有所帮助。

在 DB2 9.5 之前，由于很多用户使用 32 位系统，而线程模型在有限的寻址空间（最大 4GB）中会造成过大的冗余开销（例如每个线程的堆栈空间），造成进程无法使用很大的共享内存。因此，在早期的 32 位系统中必须使用进程模型。但是到了 DB2 9.5，大部分用户都已经使用了 64 位系统（少量的用户使用 32 位 Linux 和 Windows），对于 64 位系统，进程的寻址空间已经几乎无限倍地增加，因此，从进程模型切换到线程模型已经具备了所需的条件（对于依然使用 32 位 Linux 的用户，一般从系统的业务量来说，也不会同时运行成百上千的并发连接，因此并不会造成过多的困扰）。

3. DB2 9.7 新特性

DB2 9.7 的系统架构与 DB2 9.5 相比变化不大，最主要的特点就是与 Oracle PL/SQL 兼容和锁机制的增强。过去，DB2 在锁机制上一直对很多开发人员造成困扰。与 Oracle 相对简单的锁机制相比，DB2 严格的锁机制让应用程序的开发过程相对复杂，开发人员需要经过大量的测试与调整才能够保证在逻辑上保证应用程序不会造成过多的锁等待。而对于一些已经为 Oracle 专门设计的应用程序，想要移植到 DB2 上几乎是天方夜谭。于是，DB2 9.7 的新特性则为应用程序的跨平台移植提供了方便。

首先，DB2 9.7 中的当前已落实（Currently Committed）功能避免了互斥锁与共享锁之间的等待，这样对于很多使用了该特性的 Oracle 应用程序，可以在不更改大量应用程序的逻辑下移植到 DB2 中。

另一方面，DB2 在 9.7 版本中增加了很多的存储过程与函数，支持 PL/SQL 语言，方便客户从 Oracle 迁移到 DB2 数据库中。

除了这两个方面，DB2 9.7 在存储优化、数据库可管理性、监控、高可用性等方面也都有很大增强，下面为大家简要介绍：

- 支持索引压缩、临时表数据压缩和 XML 压缩，更加降低了存储空间成本。
- 支持内联大对象（Inline Large Object）。大对象的使用越来越多，但大对象在读/写时无法通过内存池缓存数据。在很多情况中，LOB 字段存储的数据并不是很大，如果可以和普通字段存放在同一个数据页面就可以通过内存池缓存数据了，这就是内联 LOB 的好处。
- 在线表迁移（Online Table Move）功能对于 DB2 运维人员来说是一把瑞士军刀，现代 IT 系统对数据库系统的持续可用性提出了更高要求，运维时间窗口越来越短，在线表迁移功能可以保证表迁移时例如表从一个表空间迁移到另外一个表空间、更改表的定义（增加/删除字段、改变字段数据类型、改变字段顺序等）、改变分区表的分区键或范围、执行在线表压缩、执行在线 reorg 等，数据仍然是可用的。在线表迁移通过 ADMIN_TABLE_MOVE 存储过程实现。
- 支持实时表字段更改，比如字段定义变长或变短、字段类型更改、改变字段名等，在更改过程中，DB2 扫描每条记录，验证是否满足更改条件，并对一些依赖对象做校验。
- 在性能监控方面，DB2 9.7 有了极大增强。新的监控模型不仅可以快速找出问题瓶颈，而且对系统的影响（Overhead）非常小。特别是对锁的监控，通过新的 Locking Event

Monitor 可同时监控死锁、锁等待和锁超时。

- 移植性增强。在早期版本，要从 Oracle 或 Sybase 数据库迁移到 DB2 数据库是一个令人抓狂、痛不欲生的苦差事，不仅是数据库之间的许多数据类型不兼容，更痛苦的是程序对象的迁移，如存储过程、函数和触发器等。原因是各家厂商在过程语言的实现上都有自己独特的语法。为了支持 Oracle 数据库迁移，9.7 版本引入了 number、varchar2 等数据类型，引入了很多与 time、math 和 string 相关的函数，支持 package 等。另外，DB2 提供了一个 PL/SQL 编译器，可以直接编译 Oracle 的 PL/SQL，大大减少了迁移时间（注意：这个功能是与 Postgres 公司合作的结果），如图 1.2 所示。

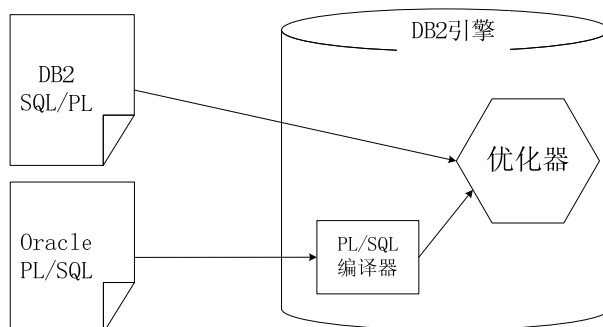


图 1.2 DB2 9.7 支持 PL/SQL 编译器

- HADR 备机可读。HADR 最早在 8.2 版本引入，目的是在主备机之间同步数据，当主机失败时，切换到备机，提供灾难恢复功能。但 HADR 存在的最大问题是主机数据库正常使用时，备机数据库不能使用，无法读/写，这对资源是极大的浪费。在 9.7 fp1 版本中，DB2 支持主机数据库使用时，备机数据库可进行读取操作，这样就可以充分利用备机数据进行一些数据分析、报表处理等工作。

4. DB2 9.8 新特性

DB2 的最新版本则是 PureScale，也就是 9.8。这个版本可以说是对底层架构的完全重构。

在 DB2 9.8 之前，DB2 的设计一直基于 Shared Nothing 架构，也就是多分区系统（DPF）中的每一个分区都拥有自己独有的资源。但是在 DB2 9.8 中，除了 DPF 架构外，DB2 又增加了共享磁盘（Shared Disk，SD）架构，也就是并不以分区键将数据分配到不同分区，而是所有的分区都可以访问同样的资源。这样对于高并发系统，可以有效地提高数据均匀负载平衡和高可用性的能力。

共享磁盘架构在多年前就被经典的 DB2 for z/OS 所使用，因此从技术的大方向上看相当成熟。但是对于一个使用了全新架构的产品来说，需要一定时间的调整与磨合。目前已经有很大行业客户对 PureScale 表现出浓厚兴趣，IBM 一直在致力于发展与改进 PureScale 系统，相信在不久的将来，越来越多的生产系统会使用 SD 架构。

1.5 DB2 认证

认证考试的利弊是个见仁见智的话题，笔者认为有总比没有强。原因有三，一是可以通过认证进行系统化、规划化的学习，考试起到督促的作用；二是增强自信心，毕竟有一个国际承认的专业技能；三是可充当敲门砖，企业在招人的时候会以此为参考。由于 IBM 并没有刻意去推广和宣传认证，导致很多人对 DB2 的认证体系不了解，本节将简要为大家介绍认证路线和相关情况，希望对准备考试的人有所帮助。

表 1.1 是 DB2 9 for LUW 认证路线，考试为全英文上机考试，如果是 DBA，考试路线图为 730→731→734。如果是开发人员，可以是 730→733 或 730→735，其中 730 是基础，必须先通过，否则即使过了 731 也无法拿到证书。

表 1.1 DB2 9 for LUW 认证路线

考试号	考试名称	认证解释	题目数量	通过考试需要答对的题目	考试时间(分钟)
730	DB2 9 Family Fundamentals	DB2 助理 DBA 认证	64	59%	90
731	DB2 9 Database Administration	DB2 DBA 认证	69	59%	90
733	DB2 9 Application Developer	DB2 应用开发认证	70	55%	90
734	DB2 9 Advanced Database Admin	DB2 高级 DBA 认证	51	64%	90
735	DB2 9 SQL Procedure Developer	DB2 存储过程开发认证	60	60%	90

关于考试报名和费用，IBM 将认证委托交给一个名为 Prometric 的公司，Prometric 公司又授权了国内一些机构做认证考试。可登录 Prometric 的中文网址 http://www.prometric.com.cn/aptcquery.asp?page_id=1011 查询各地的考点（选择考场，然后单击查询）信息，然后打电话预约和交费。目前，在国内，每门 DB2 考试的费用大概为 800~900 元不等。

关于认证考试内容的准备，网上有大量的资料可供参考，在此不做赘述。

1.6 DBA 的任务和职责

首先介绍一下 DBA 的分类，在国内有一个比较有意思的分类：甲方 DBA 和乙方 DBA。甲方 DBA 就是如通信、银行、保险、电力等单位的 DBA，这些人通常要维护一个或多个核心业务数据库，一般水平都较高，由于国内这些行业的优越性，甲方 DBA 也成为很多从业者向往和追求的目标岗位。乙方 DBA，一般指为甲方服务的第三方软件公司或集成公司的 DBA。

不管是甲方还是乙方，DBA 的任务主要分为两类：数据库管理和开发设计。管理类 DBA 主要从事数据库运维管理工作，负责数据的可用性、容量和存储规划、安装配置、版本(补丁)

升级、安全控制和权限管理、日常监控、数据备份恢复、性能调优和问题诊断等；开发类 DBA 主要负责数据库逻辑设计、数据库对象创建、SQL 开发等。在国内某些公司，特别是一些小公司，由于人力和其他资源限制，DBA 的职责分工并不明确，往往一个人要干几个人的工作，既要开发，又要兼职数据库维护，这样其实非常有助于一个 DBA 的成长。

至于做开发 DBA 好还是运维 DBA 好，这是由个人兴趣、技术背景等多方面因素决定的，有的人喜欢编码工作，不愿意在晚上睡的香的时候被电话吵醒去解决故障，也有的人同样不愿意在应用上线前疯狂加班改 Bug。尽管各自的分工不同，但还是建议开发 DBA 和运维 DBA 能够相互学习，参加与对方相关的培训。

本书面向的主要群体是运维 DBA。要成为一个优秀的运维 DBA 需要具备哪些素质呢？

- 积累丰富的经验。DB2 是实战性很强的学科，经验在某些问题的定位和诊断上往往发挥重要作用。经验是逐渐累积的，对于刚接触 DB2 或接触不久的初学者，建议找一个环境，多上机操作、演练，不要怕遇到问题，反而要庆幸及早遇到了问题，并且要重视遇到的每一个问题，而不是简单地用重启或恢复来规避问题，只有这样才能不断提高。
- 培养独立分析问题、解决问题的能力。在运维过程中，不可避免地会遇到各类问题，有些问题只凭经验是无法解决的，这时就需要学会分析问题。
- 快速的学习能力。信息技术的发展日新月异，新版本、新特性层出不穷，没有快速的学习能力将无法跟上时代的步伐。对于 DBA 来说，要掌握的不仅仅是数据库本身，还要了解系统、存储和网络等相关知识，这更要求 DBA 具备快速学习能力。
- 持续保持高度的认真态度，要细心，并且保持耐心。其实，DB2 技术本身的学习并不难，如果有很好的方法和指导，一个 DBA 专心学习一年就足以胜任基本运维任务，但比技术本身更重要的是工作态度。我们发现很多 DBA 遇到的问题是由于误操作引起的，比如误删文件、错误更新或删除数据等。DBA 掌握着企业的核心数据，在操作数据库时必须时刻保持清醒的头脑，粗心大意必将付出代价。当遇到棘手问题的时候，一定要沉着冷静，冲动和慌乱只会使事情越来越糟。
- 努力培养学习 DB2 的兴趣。兴趣是在学习的过程中慢慢培养的，很多 DBA 在初学时往往是工作需要不得不学，随着水平的提高和经验的积累，当解决了一些难题后，就会有种巨大的成就感，这时的学习往往就由被动变为主动。如果没有主动学习的兴趣，是不可能深入的。
- 健康的体魄。现在企业对数据的持续可用提出了越来越高的要求，一些常规的运维任务，如补丁升级、业务变更、空间扩容等都要求在晚上进行，这就要求 DBA 拥有健康的体魄，应付繁杂的工作。

1.7 IBM 信息管理产品概述

IBM 企业软件包含了五大产品线：WebSphere、DB2、Lotus、Tivoli 和 Rational。WebSphere 是中间件产品，DB2 属于信息管理家族，Lotus 是工作流和协作软件，Tivoli 是系统管理、运行、监控和安全家族产品，Rational 包含一系列软件生命周期产品。每个产品线包含几十个，甚至几

百个软件产品。比如信息管理家族，通过自主研发和并购，已经形成了非常完整的产品线和解决方案，具体来说，主要分为五大类：数据库服务器（Data Servers）、信息整合和集成（Information Integration）、数据仓库（Data Warehouse）、主数据管理（Master Management）和内容管理（Content Management），具体说明如表 1.2 所示。

表 1.2 信息管理家族的信息产品说明

信息产品分类	分 类 描 述	信 息 产 品	信息产品描述
数据库服务器	提供对数据安全、有效的管理	DB2	IBM 核心数据库产品
		Informix	2001 年收购
		IMS	IBM 主机数据库
信息整合和集成	提供数据的集成和处理	WebSphere Federation Server	数据库联邦
		WebSphere Replication Server	数据复制
		WebSphere DataStage	用做 ETL 处理
		WebSphere QualityStage	用做数据治理
数据仓库	提供数据仓库构建、报表展现和统计分析功能	InfoSphere Warehouse	数据仓库套件
		Cognos	报表展现和分析工具，2009 年收购
		SPSS	统计分析预测软件，2009 年收购
主数据管理	主数据管理软件	WebSphere Customer Center	客户主数据管理，收购于 2006 年
		WebSphere Product Center	产品主数据库管理
内容管理	提供企业内容管理	FileNet	收购于 XX 年
		IBM Content Management	内容管理产品
		Omnifind	企业搜索产品

1.8 小结

DB2 是 IBM 最核心的软件产品之一，在数据库领域，与 Oracle、SQL Server 等产品一起占据着接近 90% 的市场份额。DB2 支持几乎所有的硬件平台，从 IBM 大型机、中型机，到 UNIX、Linux 和 Windows 等开放平台。DB2 提供了很多版本满足各类客户的需求，尽管在国内 DB2 的市场占有率还不足以与 Oracle 相比，但不可否认的是，IBM 近年来在 DB2 产品线上投入了巨大的人力和物力，在功能上增加了很多高级特性，以增强自主管理功能、减少存储需求、降低运维成本。相信在不久的将来，DB2 一定会赢得更多客户的青睐。

1.9 判断题



(1) DB2 最早运行为 AIX 操作系统。

T: 正确

F: 错误

(2) DB2 的免费版叫做 DB2 Express Edition。

T: 正确

F: 错误

(3) XML 功能是 DB2 V9.1 中的新特性。

T: 正确

F: 错误

(4) DB2 9.7 能够兼容 Oracle PL/SQL。

T: 正确

F: 错误

(5) DB2 9.5 支持内联大对象。

T: 正确

F: 错误

1.10 参考文献



百度百科

认证考试信息:

<http://www-03.ibm.com/certify/tests/ovr730.shtml>

<http://www-03.ibm.com/certify/tests/ovr731.shtml>

<http://www-03.ibm.com/certify/tests/ovr733.shtml>

<http://www-03.ibm.com/certify/tests/ovr734.shtml>

<http://www-03.ibm.com/certify/tests/ovr735.shtml>



DB2 体系结构

本章主要讨论 DB2 的体系结构，只有深入理解了 DB2 的核心概念和结构，才能从宏观上把握 DB2 的全貌，避免一开始就陷入支离破碎的细节。与其他关系数据库产品类似，DB2 主要提供数据存储和数据访问能力。我们将从两条主线讲述 DB2 的体系结构，一条是 DB2 数据库对象层次，如实例、数据库、表空间、表、索引等，另一条是数据动态访问过程，包括数据库连接、应用、事务、SQL 语句等。

本章内容安排如下：

- DB2 体系结构。
- DB2 对象层次关系。
- 数据访问过程。
- 数据库工具。

2.1 DB2 体系结构简介

当前的应用系统主要分为两类：联机事务处理（OLTP）和联机分析处理（OLAP）。OLTP 主要执行日常的事务处理，比如银行存取款、商场买东西等，它的主要特点是对响应时间要求高，数据量一般较小，并发多，面向应用。OLAP 主要指数据仓库、决策分析类系统，主要特点是数据量大，对实时性要求不高，面向主题。

针对这两种典型的系统，DB2 提供了很好的支持。对于 OLTP 系统和数据量较小的 OLAP

系统，可以采用单分区架构，如图 2.1 所示。在这个单分区架构中，包含一台数据库服务器，该机器包含一些物理资源，如 CPU（左边）、一组内存（中间）和一组磁盘（下边）。DB2 在设计上的一个主要考虑就是充分利用这些资源并行处理一些任务。

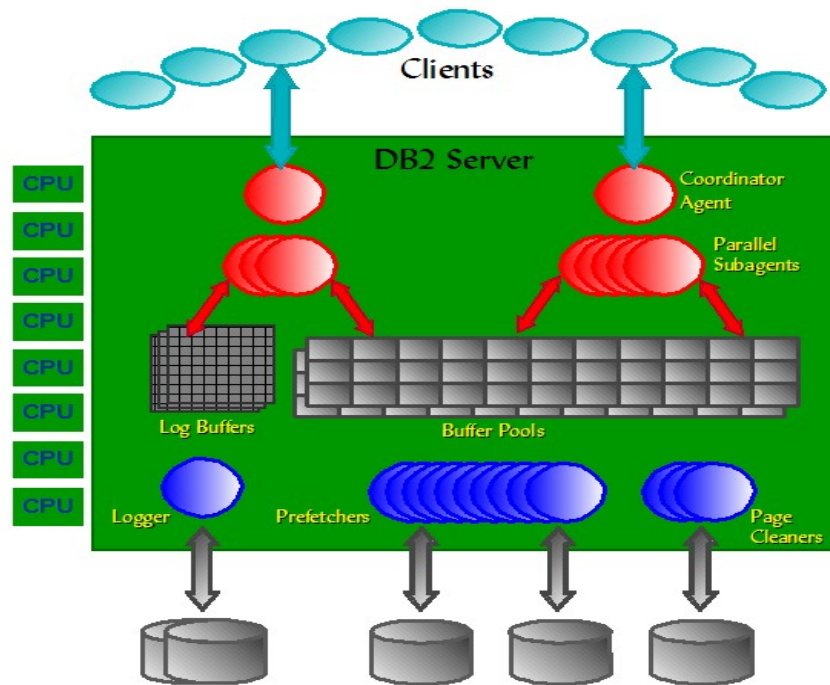


图 2.1 DB2 单分区架构

在图 2.1 中，一些 Client 客户端应用程序连接到 DB2 服务端。每个客户端连接都有一个或多个 DB2 代理（agent）进程或线程，代表应用执行一些任务。当读取数据时，这些进程或线程首先将数据从磁盘加载到缓冲池，然后返回给客户端。当写数据时，DB2 引擎内部还有一些独立的进程或线程，进行异步 I/O 操作和日志读/写操作。这些线程或进程直接或间接地与磁盘交互，将数据读出或写入磁盘。

一般来说，一个 DB2 代理进程或线程服务于一个应用程序。但是有的时候，当查询的数据量很大时，用户可以开启一个非常有用的功能，使得多个进程和线程在同分区内共同并行处理一个事务请求。这个功能叫做分区内并行（Intra-Parallel）。比如，一个复杂的 SQL 语句在 DB2 引擎中由两个子代理进程或线程同时处理，然后将得到的结果汇总给一个另外的代理（叫做协调代理），返回给用户。这个功能可以为大规模的数据处理提供很高的效率。

但是有一些 OLAP 系统，比如国内一些通信公司和电力公司的经营分析系统，包含的数据超过几十 TB（有的已经上千 TB），一台机器的处理性能根本无法满足要求。这时，可考虑 DB2 的多分区架构，即 Shared Nothing 架构。图 2.2 所示是 DB2 多分区架构图。

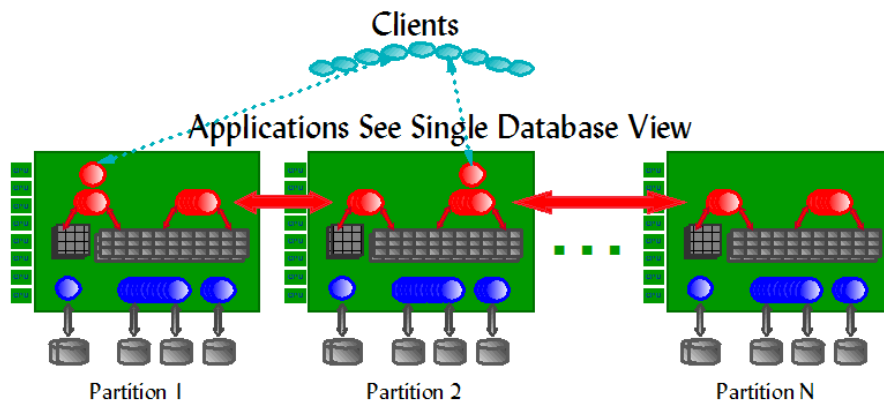


图 2.2 DB2 多分区 (DPF) 架构

从概念上讲，DB2 数据库多分区在很多时候可以看成是一系列单分区数据库的聚集，每个分区都包含独立的数据、日志、锁等，当一个任务到来后，由一个分区（这个分区叫做协调分区）将请求分发到各个分区上，每个分区都会启动一个或若干个子代理，同时执行查询处理，然后把结果集返回给协调分区进行汇总，最后返回给应用程序。分区数据库对应用是透明的，应用看到的还是一个数据库。

当新的数据被插入，或者老的数据被修改时，DB2 会通过一种叫做“分区键”的数据列（用户可以指定列作为分区键）对其进行散列（hash），然后根据散列结果插入相应的分区。也就是说，一条数据仅仅存在一个分区中，每个分区无法访问其他分区中的数据。在海量数据应用中，这种设计大大减少了每个分区需要处理的数据量，为查询性能的提升起到了重要的作用。

这种架构的优点就是能够充分利用系统资源，将一个大型的查询分解成若干个小查询并行运行在不同的系统中。由于每一个分区只能访问自己分区的数据，当查询数据需要关联时，需要在分区中交换必要的的数据，分区之间使用一种叫做 FCM（Fast Communication Manager）的通信机制。这种架构对系统设计人员的要求较高，一定要充分理解优化器与系统访问数据的规则，并且设计很好的“分区键”，才能够尽可能避免分区间大量的数据交换。

DB2 for LUW 是 Share-Nothing 架构的典型代表，当前比较流行的 Greenplum 也是这个架构。该架构的设计目标主要是解决数据扩展性和并发性能问题，而不是高可用性。目前主要用于包含海量数据的 OLAP 系统，如仓库和经营分析系统。

与 Share-Nothing 相对的另外一种常见的架构是 Share-Disk，典型代表是 Oracle RAC 和 DB2 for z/OS。Share-Disk 架构允许所有机器都可以访问全部的数据，好处是管理起来相对方便，而且一台机器宕机后，只要存储部分不出问题，其他系统可以照样访问数据。它的设计目标主要是提供高可用性，一般用于 OLTP 系统。

很多初学者都会问一个问题：这两个架构哪个好？答案也很明显，适合你自己业务的就是最好的。用 Share-Disk 架构来跑几百个 TB 的数据仓库明显是吃力不讨好，而使用 DB2 分区数据库跑只有几十 GB 的在线交易也有点大材小用。因此，在选择一个系统的时候，一定要对相

关的不同架构进行了解，然后通过自己的分析（别看广告，看疗效）得出相应的结论。

另外，为了满足 OLTP 系统对高可用性的需求，DB2 for LUW 对 DB2 9.8 版本（代号 pureScale）进行了体系结构的重大变革，即支持 Share-Disk 架构，它的体系结构更加先进，效率更高。IBM 一直在致力于发展与改进 pureScale 系统，相信在不久的将来，越来越多的生产系统会使用 SD 架构。

关于 DPF 和 pureScale 的更细节介绍，我们将在《DB2 数据库高级管理》一书中重点讲述。本书只专注于单分区数据库的学习。

2.2 对象层次关系

本节我们将简要概述 DB2 对象的层次关系，理解这些对象的概念，以及它们之间的关系，对于我们接下来深入学习 DB2 有很大的帮助。

图 2.3 所示是数据库对象逻辑层次关系图，最高层次的对象是系统。一个系统可以简单理解为一个 DB2 产品版本，如 DB2 9.5 版本的安装。从 DB2 9 开始，一台物理机器上可以安装多个产品版本，即多版本共存。

一个系统可以创建一个或多个实例（instance），每个实例可以管理一个或多个数据库（database）。

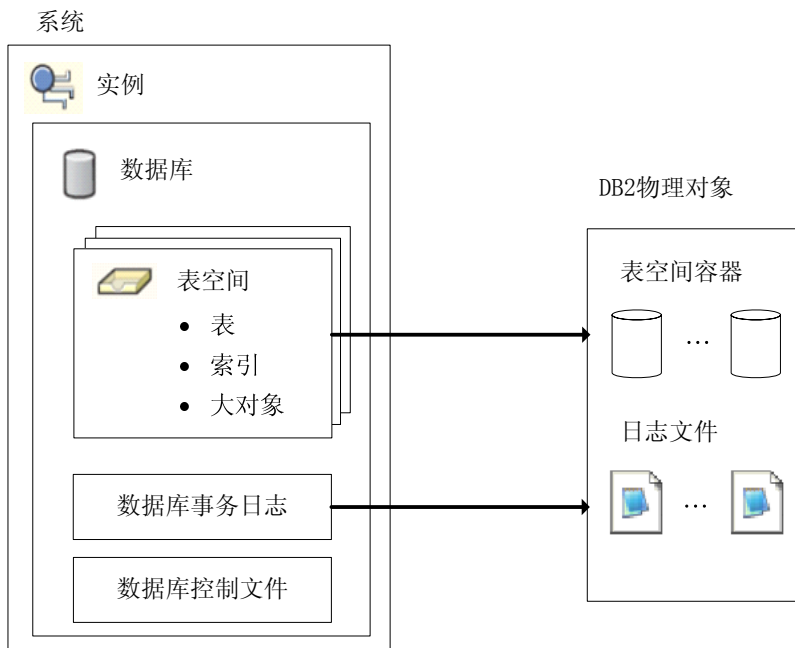


图 2.3 DB2 对象逻辑层次关系图

实例也叫数据库管理器 (Database Manager)，可以理解为一个运行时环境，包括一组进程/线程和内存。数据库是指信息的物理存储。

关于实例和数据库的关系，许多 DB2 初学者很容易混淆。我们通过一个比喻来理解。数据库好比一个岛屿，而实例就是通向岛屿的桥梁，出入岛屿必须经过桥梁，即使桥塌了，岛屿仍然还在，但却无法出入岛屿。对应到 DB2 的术语，只有实例在运行的时候，才可以访问数据库。如果实例挂起或停止了，虽然数据仍在，但却无法访问。

在 DB2 中，一个实例可以包含多个数据库，删除实例并不会删除数据库，这与 Oracle 的体系结构存在根本差别，但一个数据库只能属于一个实例。

表 (Table) 是数据库最重要的对象，表是二维结构，由行和列构成的。对于应用程序来说，对数据的访问和操作都是通过表。那么这些表数据存在什么地方呢？自然是磁盘上的某个地方，但是数据库如何能够管理哪个表存放在什么磁盘上，这就是表空间 (Tablespace) 的职责了。

可以把整个数据库想象成一个 N 层的房子，每一层就相当于一个表空间。每当用户创建一个表的时候，都需要指定创建在哪个表空间中，也就是去哪一层。

每一个表空间可以容纳一个或多个表与索引等对象。表空间实际上是逻辑上的概念，物理上对应着一个或多个容器。所谓容器是用来真正存放数据的地方，可以是文件，可以是裸设备（理解为没有格式化的磁盘），也可以是目录。一个表空间可以跨越多个容器，但每个容器只能归属于一个表空间。用房屋的比喻，则可以将每一个容器看成每一层中的各个单元。

相信大家对 DB2 的对象层次关系有了一个清楚的认识。总结一下，就是一个系统可以创建一个或多个实例，每一个实例中包含一个或多个数据库，每个数据库中包含一个或多个表空间，每个表空间包含一个或多个容器，而每个表和索引则可以存在于一个或多个表空间中。

DB2 数据库通过日志来保证一致性。当用户更改某些数据的时候，为了保证这些更改在系统崩溃或者回滚的时候能够再次被使用，则需要记入事务日志。事务日志中记录的是每一个数据页中一个给定记录的更改或者行数据的插入和删除。

除此之外，每个数据库还包括一些控制文件。

2.3 数据访问过程

如果说上一节介绍的对象是静态的话，那么数据访问过程则是从动态的角度了解数据访问的流程和步骤。譬如，客户端应用程序要执行一条 SQL 语句，需要经历哪些操作呢？本节我们要介绍连接、应用程序、事务、SQL 语句等概念。

一般来说，一个客户端在访问数据前，首先要知道访问的是哪台机器上的哪个数据库，这个操作叫做编目 (Catalog)。所谓编目，就是在客户端做个注册，把注册机器的动作叫做节点编

目，把注册数据库的动作叫做数据库编目。

一旦编目成功，客户端就可以携带用户名/密码连接数据库了，当编目信息与认证信息都正确的时候，数据库连接（Connection）就会被建立起来。对于每一个数据连接，从数据库角度看就是一个应用程序（Application）。

如果在客户端使用同一个可执行文件同时执行 10 次，尽管这些连接所发起的可执行文件相同，从数据库看依然接收到 10 个连接请求，也就是 10 个应用程序，而不是一个。每一个应用程序可以由一个或多个代理（Agent）负责完成。代理是真正运行在操作系统中的线程或者进程，负责真正运行应用程序的请求。

每一个应用程序在同一时刻只能运行一个事务（Transaction）。事务是指一组工作单元（Unit of Work, UOW），指的是一组要么全成功要么全失败的操作。比如工厂从仓库 A 向仓库 B 调运货物，第一步操作会是从仓库 A 的库存中减少货物的存量，而第二步操作是在仓库 B 中增加等量的存货。但是如果第二步操作失败了（比如仓库满了），那么我们要撤销第一步操作中在仓库 A 的更改，也就是让两步操作同时失败。同理，如果第二步操作成功了，那么我们会提交这个事务，让数据真实生效。

在每一个事务中，可以包含一个或多个语句（Statement）。其中有的语句可以用来插入数据，有的语句用来更新或者查询数据。当该事务中所有的操作都执行完毕以后，用户可以提交（Commit）或者回滚（Rollback）该事务，确保不会出现数据不完整的状况。

当一个语句从客户端应用程序发起的时候，首先应用程序会使用某种驱动来与 DB2 服务端交流。这种驱动可以是 CLI、JDBC 或者 ODBC。当一个语句被驱动拿到后，就会被打成一个网络包发送给数据库的接收端。

当服务端接收到这个包的时候，DB2 会根据这个包的数据结构解析出 SQL 语句。然后 DB2 编译器对这个语句进行解析，并通过优化器中得到执行计划。所谓执行计划就是一组告诉 DB2 如何运行一个查询（定义数据对象访问的顺序）的数据结构。然后数据库运行时类根据执行计划生成一个包。当执行这个包的时候，DB2 引擎就可以从磁盘上相应位置获取数据，或者与其他的表进行关联。当获得结果后，通过先前建立的 TCP/IP 连接把结果发送给客户端，图 2.4 所示演示了这个过程。

不论是什么类型的应用程序，也不管是本地还是远程（如果是本地就不是 TCP/IP，而是 IPC 共享内存），其连接方法都没有太大的差异。

总的来说，数据访问流程就是客户端首先建立数据库连接，每个连接相当于一个应用程序，这个应用程序由一个或几个代理负责完成。每个应用顺序执行一些事务处理，每个事务由一条或多条 SQL 语句构成。

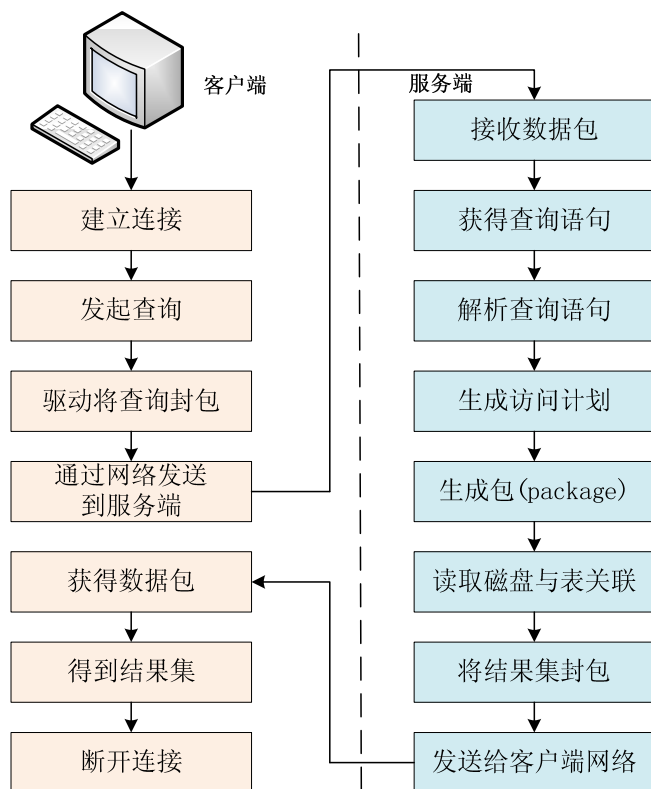


图 2.4 SQL 语句处理流程

2.4 数据库工具

DB2 提供了各类接口访问数据库，包括命令行、图形界面和应用程序接口，如图 2.5 所示。其中，命令行是有经验的 DBA 最愿意使用的方式，特别是在 UNIX/Linux 环境下，当一个成熟系统中的各种维护脚本被大量使用时，使用图形界面会使得简单的工作变得加倍复杂，同时会对一些命令细节的参数把握不够精确，当出现问题时调试的难度也会显著增加。

对于应用开发人员，大都喜欢用图形界面工具，因为查询和处理数据比较直观。DB2 自带了一组图形界面工具，但并不是很好用，从 DB 2 9.1 开始，IBM 推出了新的基于 Eclipse 的开发工具 Data Studio，替代以前的开发中心（Development Center）。DB2 9.5 版本更名为 Optim Development Studio 和 Optim Database Administrator。对于开发存储过程的用户，Data Studio 和 Optim 可以作为不错的选择。

除此之外，也有一些很好的第三方工具可供选择，比如 QC（Quest Central）和 Toad，用来查看数据库对象、处理 SQL 语句还是比较方便的。

还有一种访问数据库的方式是通过应用程序，针对主流的开发语言，DB2 都提供了访问接

口，如 JDBC 是 Java 访问数据库的标准接口；针对微软的平台，DB2 提供了一组调用接口；针对 C、Cobal、Fortran 等语言，提供了嵌入式 SQL 支持。由于篇幅所限，本节不会具体介绍每种编程方法，感兴趣的请参考信息中心或相关书籍。

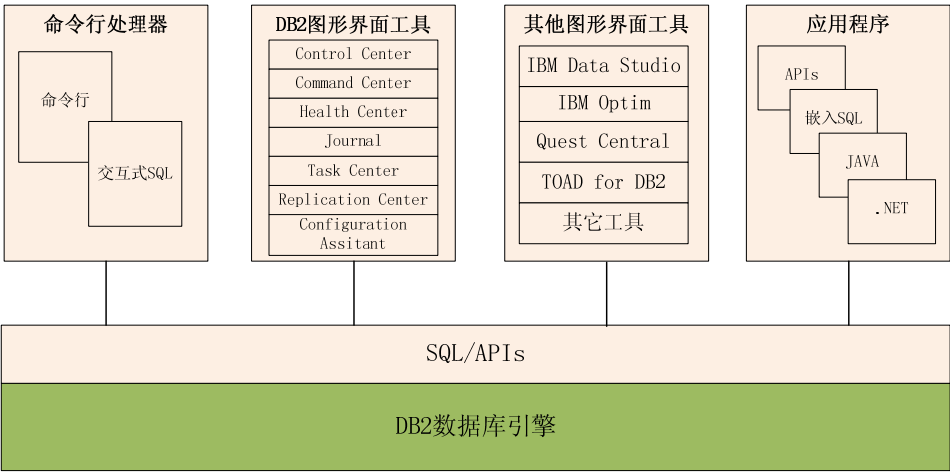


图 2.5 访问 DB2 的各种方法

2.5 小结

数据库系统的体系结构是一个非常广阔且深入的话题。在这一章我们只是从对象结构和数据访问流程这两个角度总体上讨论了 DB2 的工作原理。对于具体的进程内存模型，以及各个子模块的工作原理，我们将在本书的后面章节中进一步讨论。

2.6 判断题

DB2 支持 Shared Nothing 体系架构。

T: 正确

F: 错误

(1) DB2 既支持分区间并行也支持分区内并行。

T: 正确

F: 错误

(2) DB2 中每一个实例只能创建一个数据库。

T: 正确

F: 错误

(3) 数据库无须编目可以直接被使用。

T: 正确

F: 错误

(4) DBA 与应用程序开发人员只能够使用命令行访问数据库。

T: 正确

F: 错误

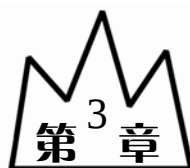
2.7 参考文档



Data Studio 的下载地址:

<http://www-01.ibm.com/software/data/optim/data-studio/>

第二部分 DB2 部署和规划



安装 DB2 软件

DB2 软件安装是操作数据库的第一步，不安装 DB2 产品，其他一切都是纸上谈兵。本章我们将介绍 DB2 安装介质的获取、安装前的准备工作、安装步骤和安装后的检查。任何软件都不避免地存在一些故障，DB2 也不例外，IBM 每隔一段时间会发布一个补丁包修订错误，本章会介绍补丁升级的步骤和注意事项。另外，DB2 版本也存在一定的生命周期，从低版本到高版本的升级方法也将在本章介绍。

对于初学者，从连贯性角度，建议先学习“3.1 软件安装”，当熟悉了第 4 章实例的概念后，再根据自己的需求，学习补丁升级和版本升级。

本章内容组织如下：

- DB2 软件安装。
- 补丁升级。
- 版本升级。

3.1 软件安装



不同版本的 DB2 软件安装略有差异，不过大都会遵循以下几个步骤：软件获取、安装前检查、安装和安装后检查。

3.1.1 软件获取

IBM 官方网站上提供了 DB2 Express-C 版本的软件免费下载, 该版本支持 2 CPUs/4GB 内存硬件配置, 基本能够满足个人学习和测试。可通过如下链接下载当前 DB2 express-c 版本:

<http://www.ibm.com/developerworks/cn/downloads/im/udbexp/>

下载界面如图 3.1 所示。

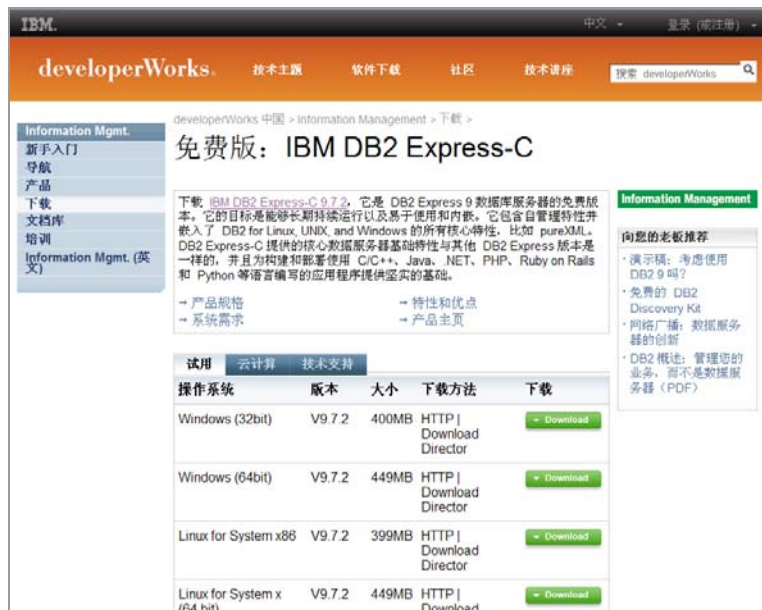


图 3.1 DB2 Express-C 下载

注意: Express-C 版本只支持 Windows 和 Linux 平台, 在下载介质时需要根据硬件平台选择 32bit 还是 64bit。

如果需要使用 HADR、DPF 等高级特性, 则需企业版安装, 企业版需要 License。在 UNIX 平台中, 企业版要求系统支持 64bit。

3.1.2 安装前检查

DB2 需要软/硬件环境的支持, 在安装之前, 需确保系统满足操作系统、硬件、软件、网络、磁盘和内存需求。不同版本、不同平台对环境的要求不同, 可通过 IBM 信息中心查看:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/index.jsp>

以下是 IBM 官方发布的 DB2 for AIX/Solaris/HP-UX 平台上已知的问题, 强烈建议在安装前仔细检查确认:

<https://www-304.ibm.com/support/docview.wss?uid=swg21165448> (AIX)
<https://www-304.ibm.com/support/docview.wss?rs=0&uid=swg21257606> (SOLARIS)
<https://www-304.ibm.com/support/docview.wss?rs=0&uid=swg21257602> (HP-UX)
<https://www-304.ibm.com/support/docview.wss?rs=0&uid=swg21224762> (old AIX)

图 3.2 所示是在 AIX 上安装 DB2 时对操作系统和硬件的要求。比如在 AIX 5.3 系统上安装 DB2，需要满足：

- 必须是 64bit 平台。
- 要求系统补丁是 5.3.06.2，并且要求打上了 IZ03063 APAR。
- 要求 C++运行时环境，如果系统没有安装，可到 IBM 网站下载，并按指定的步骤安装。

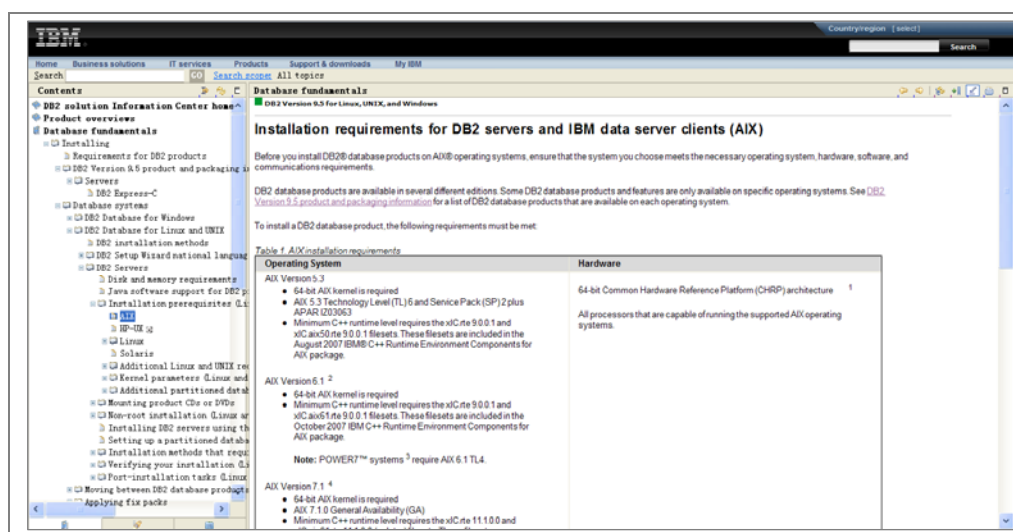


图 3.2 DB2 安装需求

3.1.3 安装

当获得了软件安装包，并且满足了软/硬件环境后，可以安装了。在 Windows 平台的安装比较简单，根据图形界面指示一步步安装即可。Linux/UNIX 平台有两种安装方法：图形界面和命令行，一般选择命令行安装方法，以下是 Linux 平台安装步骤，UNIX 平台安装方法完全相同。

首先解压安装包，然后安装，安装大概需要 5~10 分钟，安装完毕可检查安装日志确认是否成功安装。

```
db2server:/soft # cp db2ese.tar /soft
db2server:/soft # tar -xf db2ese.tar
db2server:/soft/ese # ./db2_install -b /opt/ibm/db2/V9.5
```

Specify one of the following keywords to install DB2 products.

```

ESE                                --选择 ESE (Enterprise Server Edition)
CLIENT
RTCL

Enter "help" to redisplay product names

Enter "quit" to exit
*****
ESE
.....
The execution completed successfully

For more information see the DB2 installation log at
"/tmp/db2_install.log.7128"

```

从 DB2 9 开始，一台机器上可以装多个 DB2 版本。通过 `db2ls` 命令可查看安装的版本及补丁信息，在以下环境中，安装了 DB2 9.1、9.5 和 9.7.3 个版本，通过 `db2ls -q` 选项可查看安装的组件：

```

db2server:/soft/ease # db2ls

```

Install Path	Level	Fix Pack	Special	Install Number	Install Date
/opt/ibm/db2/V9.7		9.7.0.1	1	Mon Dec 21 13:16:57 2009 PST	
/opt/ibm/db2/V9.1		9.1.0.3	3	Wed Mar 24 22:09:17 2010 PDT	
/opt/ibm/db2/V9.5		9.5.0.0	0	Wed Aug 25 23:09:27 2010 PDT	

```

db2server:/soft/ease # db2ls -q -b /opt/ibm/db2/V9.5

Install Path: /opt/ibm/db2/V9.5

```

Feature Response File ID	Level	Fix Pack	Feature Description
BASE_CLIENT	9.5.0.0	0	Base client support
JAVA_SUPPORT	9.5.0.0	0	Java support
SQL_PROCEDURES	9.5.0.0	0	SQL procedures
BASE_DB2_ENGINE	9.5.0.0	0	Base server support
JDK	9.5.0.0	0	IBM Software Development Kit (SDK)
for Java			
CONNECT_SUPPORT	9.5.0.0	0	Connect support
COMMUNICATION_SUPPORT_TCPIP	9.5.0.0	0	Communication support - TCP/IP
REPL_CLIENT	9.5.0.0	0	Replication tools
CONTROL_CENTER	9.5.0.0	0	Control Center
DB2_DATA_SOURCE_SUPPORT	9.5.0.0	0	DB2 data source support
LDAP_EXPLOITATION	9.5.0.0	0	DB2 LDAP support
INSTANCE_SETUP_SUPPORT	9.5.0.0	0	DB2 Instance Setup wizard
SPATIAL_EXTENDER_CLIENT_SUPPORT	9.5.0.0	0	Spatial Extender client
XML_EXTENDER	9.5.0.0	0	XML Extender
APPLICATION_DEVELOPMENT_TOOLS	9.5.0.0	0	Base application development tools
FIRST_STEPS	9.5.0.0	0	First Steps

DB2_SAMPLE_DATABASE	9.5.0.0	0	Sample database source
INFORMIX_DATA_SOURCE_SUPPORT	9.5.0.0	0	Informix data source support

3.1.4 补丁升级 ■ ■ ■

同所有软件一样，DB2 也不可避免地会出现一些错误，包括用户使用中发现的错误或者实验室测试出的错误。为修正这些错误，IBM 一般每隔 3 个月出一个补丁包(fixpack)，后一个补丁包包含前一个补丁包的内容，比如 fp2 会包含 fp1 的内容。一般情况下，对于一个大版本来说，fixpack 3 以后的才是比较稳定的，否则可能会遇到各类问题，因此，建议在初始安装时即升级到较新的补丁包。

升级补丁的过程比较简单，但在生产环境升级前，建议先在测试机上进行测试，因为不是每次打补丁都能成功，而且升级过程中需要停止实例，数据不可用。同时，也要做好升级失败的回退机制，确保生产系统的稳定运行。

不同版本的升级方法略有不同，强烈建议在升级时参考 IBM 官方文档：

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp> (9.7 版本)
<http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/index.jsp> (9.5 版本)
<http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp> (9.1 版本)

展开左边的内容，会看到“Applying fix packs”节点，展开该节点后有详细的说明，如图 3.3 所示。

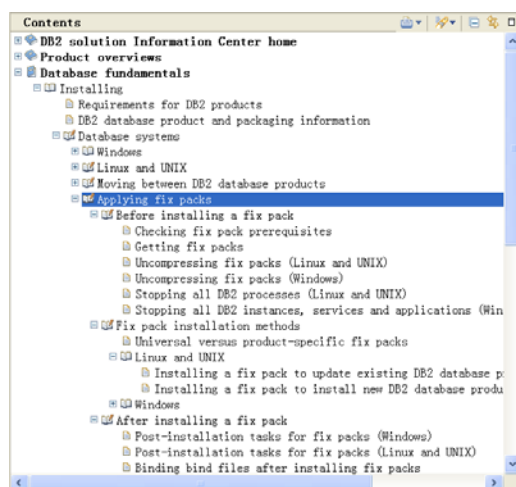


图 3.3 DB2 补丁升级文档

我们以 DB2 9.5 版本的补丁升级为例，为大家介绍升级步骤（注：如果还没有创建实例，则只操作步骤（1）和（5）即可）。

（1）下载并解压补丁包。补丁可从 IBM 官方网站免费下载，链接如下：

<https://www-304.ibm.com/support/docview.wss?rs=0&uid=swg27007053>

下载界面如图 3.4 所示。

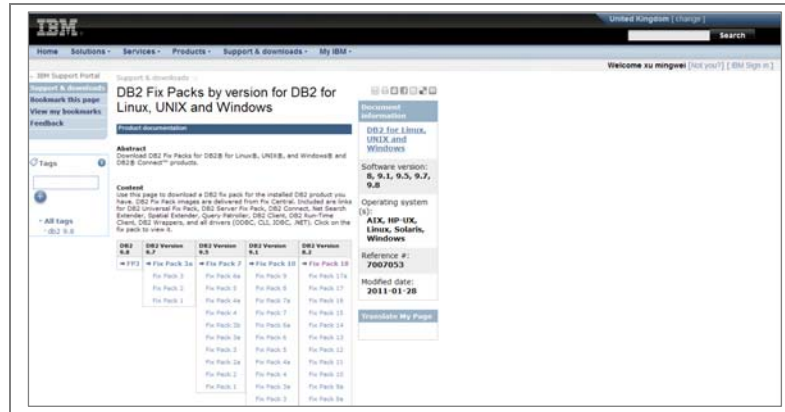


图 3.4 DB2 补丁下载

(2) 备份当前实例和数据库配置信息，此步骤虽是可选的，但强烈建议备份。

```
db2support.-d sample -cl 0
```

参数-cl 0 会收集数据库系统目录、数据库和实例配置参数、DB2 注册变量设置等。这些信息会打包到 db2support.zip 文件中。

备份每个数据库的 package 信息：

```
db2 list packages for all show detail > packages.txt
```

备份数据库 DDL 语句：

```
db2look -d sample -e -l -x -o sample.ddl
```

(3) 备份数据库，对于关键业务系统，强烈建议做好数据库备份。

```
[db2inst1@db2server]$ db2 backup database sample
```

(4) 停止实例、服务和应用，准备升级。如果要升级的 DB2 版本包含了多个实例，则需要将所有实例都停止。注意：如果采用了双机软件配置，需要检查双机应用脚本，确保停止 DB2 实例时不会发生主备机切换。

```
su - iname          # 切换到实例用户
db2 force applications all  # 将所有连接断开
db2 terminate
db2stop             # 停止实例
db2licd -end        # 停止 License 管理进程
```

如果机器上启动了 DAS(Database Administration Server)，则需要停止。我们将在第 4 章介绍 DAS。

```
su - dasusr1        # 切换到 DAS 实例用户
```

```
db2admin stop # 停止 DAS 管理服务器
```

然后通过 `ps -ef |grep -i db2` 检查是否仍然存在 DB2 进程，通过 `ipclean` 进行清理，如果清理后仍然存在 DB2 进程，则可通过 `kill -9 <db2_pid>` 清除。

(5) 安装补丁包。

进入补丁包安装目录，通过 root 用户执行 `installFixPack` 命令。

```
./installFixPack -b DB2DIR #DB2DIR 是要升级的 DB2 软件目录
```

(6) 安装后任务。

在补丁升级过程中，一些工具（如 Import、Export、Load 等）会自动绑定，DB2 实例也会自动升级。但如果发生错误，则需要手工绑定和升级实例，命令如下：

```
DB2DIR/instance/db2iupdt iname # 升级实例
DB2DIR/instance/dasupdt # 升级 DAS 实例
```

如果希望使用补丁包中的新特性，强烈建议升级数据库，此命令需要实例用户执行：

```
db2updv95 -d dbname
```

重启实例和 DAS 实例：

```
su - iname # 切换到实例用户
db2start # 启动实例
su - dasusr1 # 切换到 DAS 用户
dasstart # 启动 DAS
```

接着对一些工具进行绑定（bind），path 是绑定文件目录，比如 `/home/db2inst1/sqllib/bnd`。

```
db2 terminate
db2 CONNECT TO dbname
db2 BIND path\db2schema.bnd BLOCKING ALL GRANT PUBLIC SQLERROR CONTINUE
db2 BIND path@db2ubind.lst BLOCKING ALL GRANT PUBLIC ACTION ADD
db2 BIND path@db2cli.lst BLOCKING ALL GRANT PUBLIC ACTION ADD
db2 terminate
```

并重新绑定 packages：

```
db2rbind dbname -l logfile all
```

(7) 回退机制。

当 DB2 补丁升级出现异常时，需要回退到初始的版本。在 Windows 平台中，只能通过控制面板将高版本补丁删除，再重新安装低版本补丁。在 UNIX/Linux 平台，可通过重新安装初始的补丁实现回退，`-f` 选项表示忽略版本的检查，`DB2DIR` 表示初始补丁安装目录。

```
[root@db2server]#./installFixPack -f level -b DB2DIR
```

(8) 检查 DB2 版本，确认升级成功。

```
[db2inst1@db2server] $ db2level
DB21085I Instance "inst20" uses "32" bits and DB2 code release "SQL09055" with
level identifier "06060107".
Informational tokens are "DB2 v9.5.0.5", "s091123", "MI00315", and Fix Pack
"5".
Product is installed at "/opt/ibm/db2/V9.5".
```

3.1.5 版本升级 ■ ■ ■

软件都有一定的生命周期，表 3 是 DB2 各主要版本的发布和终止支持的时间。对于购买了 IBM 电话支持的客户，建议在支持结束前升级到更高版本，否则将为此支付更高的支持费用。对于新项目，建议直接选择 9.5 或 9.7 版本。

表 3 DB2 各主要版本发布和终止支持的时间

DB2 版本	发布时间	IBM 终止支持时间	最新补丁(截止到 2011.2)
8.2	2004-12-10	2009-04-30	Fp18
9.1	2006-09-22	2011-09-30	Fp10
9.5	2007-12-14	2013-04-30	Fp7
9.7	2009-08-28	2014-09-30	Fp3

对于大版本的升级，如 8.2→9.1、9.1→9.5、9.1→9.7、8.2→9.7 等，它们之间会有很多变化，可能会出现在低版本执行正常，到高版本反而有问题的情况，这时详细的功能、性能测试就非常有必要了。

版本升级需要仔细规划和测试，强烈建议在生产环境升级前先在测试机上测试，这样可以提前发现升级过程中潜在的问题，并可以估计升级的时间和对系统的影响。DB2 不同版本的升级命令有差异，建议读者参看 DB2 信息中心。

以下通过实例为大家演示从 9.1 版本升级到 9.7 版本的过程，假定 9.1 版本安装目录为 /opt/ibm/db2/V9.1，该版本有一个实例名为 inst20，inst20 实例有一个数据库 SAMPLE。

升级步骤如下：

(1) 升级前的任务。

建议升级前对数据库做离线完全备份，以便升级出现问题时可以进行数据恢复。备份恢复将在第 9 章详细讲解，然后通过 db2support 备份一些重要的配置信息：

```
db2support.-d sample -cl 0
```

升级时确保系统表空间和临时表空间有足够的剩余空间，并且确保有足够的日志空间，因为升级时会把系统对象的变化作为一个完整的事务，当日志空间不够时，该事务会回滚，导致升级失败。

(2) 安装新版本。

安装 DB2 9.7 版本，假定安装目录为/opt/ibm/db2/V9.7。本步骤不需创建实例，因为接下来我们要升级 9.1 的实例。

(3) 升级前检查。

在升级前，最好通过 db2ckupgrade 命令检查是否可以成功升级。采用 inst20 实例用户进入 /opt/ibm/db2/V9.7/instance 目录，执行 db2ckupgrade。注意：该命令要求 inst20 实例必须是启动的。

```
db2server:/opt/ibm/db2/V9.7/instance # su - inst20
inst20@dpf1:~> cd /opt/ibm/db2/V9.7/instance
inst20@dpf1:/opt/ibm/db2/V9.7/instance> /db2ckupgrade sample -l /tmp/db2ckupgrade.
log -u inst20 -p password
db2ckupgrade was successful. Database(s) can be upgraded.
```

注：9.7 之前版本的升级检查的命令是 db2ckmig，到 9.7 版本替换为 db2ckupgrade。

(4) 升级实例。

当所有检查完毕后，开始迁移 inst20 实例到 9.7 版本。升级前必须先停止 inst20 实例，升级命令是 db2iupgrade，使用 root 用户执行。

```
dpf1:/opt/ibm/db2/V9.7/instance # ./db2iupgrade -u db2fenc1 inst20
db2ckupgrade was successful. Database(s) can be upgraded
DBI1070I Program db2iupgrade completed successfully
```

升级产生的信息会记录在实例用户目录下的 upgrade.log 日志文件中，如果升级出现故障，可检查该文件诊断错误原因。

```
inst20@dpf1:~> pwd
/home/inst20
inst20@dpf1:~> cat upgrade.log
Version of DB2CKUPGRADE being run: VERSION 9.7
```

实例升级成功后，切换到实例用户并启动，检查版本，发现已经升级到 9.7。

```
dpf1:/opt/ibm/db2/V9.7/instance # su - inst20
inst20@dpf1:~> db2start
SQL1063N DB2START processing was successful
inst20@dpf1:~> db2level
DB21085I Instance "inst20" uses "32" bits and DB2 code release "SQL09071" with
level identifier "08020107".
Informational tokens are "DB2 v9.7.0.1", "s091114", "IP23033", and Fix Pack
"1".
Product is installed at "/opt/ibm/db2/V9.7"
```

注：9.7 之前版本的升级实例的命令是 db2iupdt，到 9.7 版本替换为 db2iupgrade。

(5) 升级数据库。

实例升级成功后，需要将实例下的所有数据库都升级到最新版本。数据库升级命令是 `upgrade`。

```
inst20@dpf1:/home> db2 upgrade database sample
DB20000I The UPGRADE DATABASE command completed successfully
```

数据库升级可能出现以下两类问题：日志空间不足（错误号是：SQL1704N, RC=3）和表空间不足（SQL1704N, RC=17）。

```
inst20@dpf1:~> db2 upgrade database sample
SQL1704N Database upgrade failed. Reason code "3".
```

```
inst20@dpf1:~> db2 upgrade database sample
SQL1704N Database upgrade failed. Reason code "17". LINE NUMBER=1
```

注：9.7 之前版本的升级数据库的命令是 `migrate`，到 9.7 版本替换为 `upgrade`。

（6）升级后检查。

至此，升级过程已经完成，但仍有一些工作要做。

①通过 `db2rbind` 重新绑定 `package`。

```
db2rbind dbname -l logfile all
```

②如果之前版本创建了事件监控器，并且将结果写到表中，升级后需要删除并重新创建。

③9.7 版本与 9.1 版本相比在锁机制上发生了比较大的变化，在默认情况下，升级后仍然采用 9.1 版本锁机制。如果需要使用 9.7 版本的锁机制，则需要手工配置，并进行充分测试。

注意：DB2 不支持高版本向低版本的回退，比如从 9.5 版本升级到 9.7 版本后，再回退到 9.5 版本就只能通过数据库恢复了。

3.2 小结



安装 DB2 是使用 DB2 数据库的第一个步骤。一般来说，安装 DB2 的过程并不复杂。用户唯一要特别注意的就是在安装前，确保操作系统完全达到 DB2 当前版本的需求。另外，为减少使用中的问题，尽量将数据库升级到较新的补丁和版本，升级和版本升级需要做好备份，以便在升级出现问题时能够正确回退。

3.3 判断题



(1) DB2 Express-C 版本支持 64bitAIX。

T: 正确

F: 错误

(2) DB2 在安装时会自动安装操作系统所需的补丁文件。

T: 正确

F: 错误

(3) 用户应当尽可能使用最新的补丁包。

T: 正确

F: 错误

(4) 在大版本之间的升级时用户需要进行大量测试，确保新版本不会造成其他问题。

T: 正确

F: 错误

(5) DB2 支持高版本向低版本的自动回退。

T: 正确

F: 错误

3.4 参考文档



软件下载:

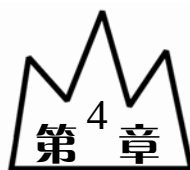
<http://www.ibm.com/developerworks/cn/downloads/im/udbexp/>

信息中心:

<http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp>
<http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp>
<http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/index.jsp>
<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp>
<http://publib.boulder.ibm.com/infocenter/db2luw/v9r8/index.jsp>

安装前检查:

```
https://www-304.ibm.com/support/docview.wss?uid=swg21165448 (AIX)
https://www-304.ibm.com/support/docview.wss?rs=0&uid=swg21257606 (SOLARIS)
https://www-304.ibm.com/support/docview.wss?rs=0&uid=swg21257602 (HP-UX)
https://www-304.ibm.com/support/docview.wss?rs=0&uid=swg21224762 (old AIX )
```



实例管理

什么是实例？为什么需要实例？实例和数据库有什么关系？相信很多初学者都会有这样的疑问。特别是熟悉 Oracle 的朋友，会发现两者在实例概念，以及实例与数据库之间的关系上，存在很大差别。本章我们将介绍实例的概念、创建和一些常用的实例命令。

本章内容组织如下：

- 什么是实例。
- 创建实例。
- 启停实例。
- 更新、删除实例。
- 实例参数。
- 管理服务器。

4.1 什么是实例



从操作系统的角度看，DB2 的实例是一组进程和一组共享内存。进程和内存的细节将在第 12 章和第 13 章详细探讨。在这里，可把实例想象为一个数据库的集合，共同运行在一个逻辑服务单元中（同一个端口）。在一个系统中，用户可以创建若干个实例，每一个实例使用各自不同的端口服务于远程应用程序。每一个实例可以包含若干个数据库。图 4.1 描述了系统、实例和数据的关系。

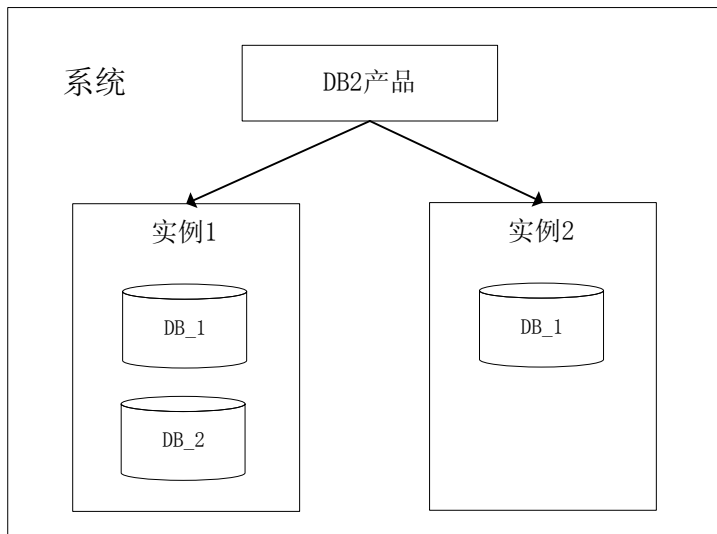


图 4.1 系统、实例和数据的关系

如果一个实例停止了，那么实例下所有的数据库将不可用。一个实例的停止将不会影响到其他实例下的数据库。

4.2 创建实例

在 Windows 和 UNIX/Linux 平台下，DB2 实例有很大差别。UNIX/Linux 系统下实例名需要与用户绑定，不同实例需要对应不同的用户名，实例创建在用户目录下。这样，当需要切换不同实例时，只需切换到相应用户即可。在 Windows 系统下，实例不需要与用户绑定，当需要在不同实例间切换时，通过 `db2instance` 环境变量进行设置。以下分别讲述两种平台的实例创建过程。

4.2.1 在 Windows 平台下创建实例 ■ ■ ■

在 Windows 平台下，DB2 软件安装过程会自动创建一个名为 `DB2` 的实例，用户可以使用这个实例，也可以选择自己创建。实例创建的命令是 `db2icrt`，语法如下：

```
db2icrt InstName [-s {ese|wse|standalone|client}]
                [-p instance profile path]
                [-u username,password]
                [-h hostname]
                [-r baseport,endport]
                [-j textSearchConfig]
                [-?]
```

其中，`-s` 选项用于指定实例的类型；`-p` 选项用于指定实例目录的位置，如果不指定，默认

的实例会创建在 DB2INSTPROF 注册变量指定的位置（通过 db2set 可查看）。

以下在 E 盘创建一个名为 TESTINST 的实例，创建后，观察 E 盘实例目录情况。

```
D:\temp>db2icrt TESTINST -p e:\
DB200000I DB2ICRT 命令成功完成
D:\temp>cd E:\TESTINST
E:\TESTINST>dir
E:\TESTINST 的目录

2011-02-13 14:51 <DIR>      .
2011-02-13 14:51 <DIR>      ..
2011-02-13 14:51 <DIR>      ctrl
2011-02-13 14:51          16,384 db2system
2011-02-13 14:51 <DIR>      log
2011-02-13 14:51 <DIR>      security
2011-02-13 14:51 <DIR>      TMP
2011-02-13 14:51 <DIR>      UIF
                1 个文件      16,384 字节
                7 个目录      1,848,221,696 可用字节

D:\temp>db2ilist
TESTINST
DB2
```

4.2.2 在 UNIX/Linux 平台下创建实例 ■ ■ ■

在 UNIX/Linux 平台下，DB2 实例需要与一个系统用户关联在一起，实例名与用户名相同。UNIX/Linux 下 db2icrt 的语法如下：

```
db2icrt [-h|-?]
        [-d]
        [-a AuthType]
        [-p PortName]
        [-s InstType]
        [-u FencedID InstName]
```

其中，-p 选项用于指定实例端口号；-s 选项用于指定实例类型；-u 用来指定 fenced user，该用户用来执行存储过程等。

与 Windows 不同，InstName 必须是一个事先建好的操作系统用户，该用户也叫实例用户。当实例创建后，会在该用户目录创建一些文件和子目录。

接下来，为大家演示 Linux 下用户/组、实例的创建过程。不同的 UNIX 平台创建组和用户的命令略有不同。在本例中，我们将用户目录创建在本地/home/db2inst1 和/home/db2fenc1 中，在实际生产中，如果要配置双机环境，建议创建在共享存储中。

```
[root@db2server]# groupadd -g 1100 db2iadm1 # 创建实例管理组
```

```
[root@db2server]# groupadd -g 1101 db2fadm1          # 创建 fenced 组
# 创建实例用户
[root@db2server]# useradd -g db2iadm1 -u 1100 -m -d /home/db2inst1 db2inst1
# 创建 fence 用户
[root@db2server]# useradd -g db2fadm1 -u 1101 -m -d /home/db2fenc1 db2fenc1
```

然后切换到 DB2 软件安装目录，使用 root 用户创建实例。

```
[root@db2server]# cd /opt/ibm/db2/V9.5/instance
db2server:/opt/ibm/db2/V9.5/instance # ./db2icrt -p 50000 -u db2fenc1 db2inst1
DBI1070I Program db2icrt completed successfully
```

可能有的读者会问：为什么需要 fence user 呢？我们知道，在有些应用中，开发人员会使用非 SQL 语言设计一些存储过程（Stored Procedure）实现某些业务逻辑。这些存储过程包含用户自己的代码，但是如果这些代码没有被完美地调试过，或者被别有用心的人包含了恶意代码，在 DB2 引擎进程中执行这些代码可能会破坏一些数据结构，造成 DB2 的崩溃。为了保护引擎，让这些存储过程在单独的进程下执行，就避免了这个问题，起到了隔离和保护作用，这就是 fence user 的目的（譬如说 DB2 引擎是以 setuid 权限执行的，但是如果在用户的 C 代码写成的存储过程中恶意地调用了类似 Shell，或者后台启动一些 root 才能执行的操作，会对操作系统造成很大的安全隐患）。

以下是 DB2 进程执行情况，两个 Java 存储过程进程（db2fmp）是在 db2fenc1 用户下执行的。

```
db2inst1@db2server:~> ps -ef |grep -i db2
root          7447      1          0 Dec06 ?        00:00:00 db2wdog 0
db2inst1      7449      7447      0 Dec06 ?        00:08:25 db2sysc 0
db2inst1      7849      1          0 06:52 pts/1    00:00:02 /home/inst20/sqlllib/bin/db2bp
7443A901 5 A
db2fenc1      12126     7447      0 Dec06 ?        00:00:01 db2fmp ( ,1,0,0,0,0,0,0,1,0,
8a1984,14,1e014,2,0,1,31fd0,0x21e90000,0x21e90000,1600000,18002,2,540021
db2fenc1      20774     7447      0 Dec06 ?        00:00:01 db2fmp ( ,0,0,0,0,0,0,0,0,1,0,
8a1984,14,1e014,2,0,1,91fd0,0x21e90000,0x21e90000,1600000,18002,2,db801c
```

实例一旦创建成功，会在实例用户目录下生成 sqllib 目录，进入后出现图 4.2 所示的目录结构。



图 4.2 目录结构

在图 4.2 所示的目录下包含了很多文件，这些文件基本都是二进制格式，不允许用户手工修改，但大部分可以通过 DB2 提供的命令来进行浏览和修改。

以下是一些常见的配置文件。

- db2system: 实例配置文件，使用 `get dbm cfg/update dbm cfg` 控制。
- profile.env: 实例 DB2 环境变量，使用 `db2set` 控制。
- sqldbdir/sqldbdir: 实例数据库编目，使用 `list db directory / catalog database` 控制。
- sqlnodir/sqlnodir: 实例节点编目，使用 `list node directory / catalog tcpip node` 控制。
- /var/db2/global.env: 全局安装、实例、环境变量列表，使用 `db2ilist` 浏览，使用 `db2greg` 控制。

4.3 启动/停止/列出实例

实例创建后，需要通过 `db2start` 命令启动才能工作。在 Windows 平台下，通过 `db2cmd` 进入命令行窗口执行 `db2start`。如果存在多个实例，则需要先切换到相应实例后才能启动，切换的命令是通过 `db2instance` 环境变量实现的。

```
D:\temp>set db2instance=testinst           # 设置实例
D:\temp>db2 get instance                   # 查看当前实例
当前数据库管理器实例是: TESTINST
D:\temp>db2start                           # 启动实例
SQL1063N DB2START 处理成功。
```

在 UNIX 平台下，切换到实例用户的命令是 `db2start`。

```
[root@db2server]# su - db2inst1
db2inst1@db2server:~> db2start
08/26/2010 00:26:40    0    0    SQL1063N DB2START processing was successful.
SQL1063N DB2START processing was successful.
```

停止实例的命令是 `db2stop`。

```
inst20@dpf1:/home> db2stop
02/13/2011 00:27:16    0    0    SQL1064N DB2STOP processing was successful.
SQL1064N DB2STOP processing was successful.
```

如果当前实例下某数据库有应用连接，则 `db2stop` 会报错。

```
inst20@dpf1:/home> db2stop
02/13/2011 00:26:51    0    0    SQL1025N The database manager was not stopped because
databases are still active.                                     # UNIX/Linux 下
```

这时可通过 `db2 force applications all` 把所有应用连接断开，或通过 `db2stop force` 强制停止实例。

查看某个 DB2 版本下有哪些实例，可通过 `db2ilist` 命令实现。

```
db2inst1@dpf1:~> db2ilist
inst20
db2inst1
```

要查看某台机器上有哪些实例，可通过 `db2greg` 命令实现，在以下的输出中，观察第一列为“`I`”的行，第三列为版本，第四列为实例名，第五列为实例目录，`db2inst1` 和 `inst20` 是 9.7 版本的实例，而 `db2inst2` 是 9.5 版本的实例。

```
db2inst1@dpf1:~> db2greg -dump
V,DB2GPRF,DB2SYSTEM,db2server,/opt/ibm/db2/V9.7,
S,DB2,9.7.0.1,/opt/ibm/db2/V9.7,,,1,0,,1261430217,0
S,DB2,9.1.0.3,/opt/ibm/db2/V9.1,,,3,0,,1269493757,0
S,DB2,9.5.0.5,/opt/ibm/db2/V9.5,,,5,0,,1283676115,0
I,DB2,9.5.0.5,db2inst2,/db2home/db2inst2/sqllib,,1,0,/opt/ibm/db2/V9.5,,
I,DB2,9.7.0.1,db2inst1,/home/db2inst1/sqllib,,1,0,/opt/ibm/db2/V9.7,,
I,DB2,9.7.0.1,inst20,/home/inst20/sqllib,,1,0,/opt/ibm/db2/V9.7,,
```

实例停止不了的问题

大家一定遇到过 `db2stop force` 无法停止，而 `db2start` 也无法启动的问题，在 UNIX/Linux 下，可通过 `db2_kill` 强制终止所有分区上执行的进程，然后执行 `ipclean`，当重新启动数据库时，DB2 会做崩溃恢复。Windows 下没有 `db2_kill` 命令，可通过 `db2stop -kill` 实现。不建议通过操作系统命令杀进程。

4.4 更新实例

实例更新的命令是 `db2iupdt`，一般在打补丁或版本升级时使用。`db2iupdt` 命令需要 root 用户执行，执行前需要首先停止实例。

```
[root@db2server]# cd /opt/ibm/db2/V9.5/instance
[root@db2server]# ./db2iupdt db2inst1
```

还有一种情况可能有些读者遇到过，就是更改了实例用户目录权限后，再连接数据库时报 SQL30082N rc=24 或 rc=15 错误。SQL30082N 错误一般是用户名/密码错误，如果确认用户名/密码没有问题，则很大的可能是更改了实例目录权限。这时，检查实例 `security` 子目录的 `db2ckpw` 和 `db2chpw` 文件权限，当某个用户或应用程序连接数据库时，使用 `db2ckpw` (check password 的缩写) 进程检查用户名和密码。当文件权限更改后，就可能出现数据库连接问题。正确的权限如下，如果发现不符，则可通过 `db2iupdt` 更新实例。

```
db2inst1@dpf1:~ > ls -l /home/db2inst1/sqllib/security
-r-s--x--x 1 root db2iadml 51675 2010-10-08 08:19 db2chpw
-r-s--x--x 1 root db2iadml 3645066 2010-10-08 08:19 db2ckpw
```

4.5 删除实例



删除实例的命令是 `db2idrop`，使用 `root` 用户执行，删除前必须停止实例。

```
db2inst1@db2server:~> db2stop force
08/26/2010 00:27:21    0    0    SQL1064N  DB2STOP processing was successful.
SQL1064N  DB2STOP processing was successful.

[root@db2server]# cd /opt/ibm/db2/V9.5/instance
[root@db2server]# ./db2idrop db2inst1
```

注意：删除实例并不会删除该实例下的数据库。

4.6 实例参数



每个实例都有一个配置参数文件，用于控制实例相关的参数，如诊断路径、监控开关、安全相关的控制及服务端口号等。通过 `db2 get dbm cfg` 命令查看实例参数，以下对一些重要的参数进行了标注：

```
db2inst1@db2server:~> db2 get dbm cfg

      Database Manager Configuration

      Node type = Enterprise Server Edition with local and remote clients

      Database manager configuration release level = 0x0c00

      CPU speed (millisec/instruction)      (CPUSPEED) = 2.519169e-07
      ...
      -- 诊断路径位置
      Diagnostic data directory path          (DIAGPATH) = /home/db2inst2/sqlllib/db2dump
      ...
      -- 实例级监控参数，建议将这些参数都打开
      Default database monitor switches
      Buffer pool                             (DFT_MON_BUFPOOL) = OFF
      Lock                                   (DFT_MON_LOCK) = OFF
      Sort                                   (DFT_MON_SORT) = OFF
      Statement                             (DFT_MON_STMT) = OFF
      Table                                 (DFT_MON_TABLE) = OFF
      Timestamp                             (DFT_MON_TIMESTAMP) = ON
      Unit of work                           (DFT_MON_UOW) = OFF
      ...
      -- 用于为 health center 提供数据，这个参数可能造成实例出现问题，建议设成 OFF
      Monitor health of instance and databases (HEALTH_MON) = ON
      ...
```

```
--这组参数用于设置实例级管理权限控制
SYSADM group name           (SYSADM_GROUP) = DB2IADM1
SYSCTRL group name          (SYSCTRL_GROUP) =
SYSMAINT group name         (SYSMAINT_GROUP) =
SYSMON group name           (SYSMON_GROUP) =
...
--这组参数用于设置用户认证方法
Database manager authentication (AUTHENTICATION) = SERVER
Cataloging allowed without authority (CATALOG_NOAUTH) = NO
Trust all clients            (TRUST_ALLCLNTS) = YES
Trusted client authentication (TRUST_CLNTAUTH) = CLIENT
Bypass federated authentication (FED_NOAUTH) = NO
...
-- 默认数据库目录
Default database path        (DFTDBPATH) = /home/db2inst2
...
--实例服务端口, 确保唯一
TCP/IP Service name          (SVCENAME) = 50010
```

实例参数只能通过 `update dbm cfg` 设置，绝大多数参数的更改都需要重启实例。为支持远程数据库连接，需要配置通信协议和端口。从 DB2 9 开始，DB2 唯一支持的通信协议是 TCP/IP，IP 端口号必须唯一，一般从 50 000 开始，但不能超过 65 535。以下参数更改后需要重启实例才生效。

```
db2inst1@db2server:~> db2set DB2COMM=TCPIP
db2inst1@db2server:~> db2 update dbm cfg using SVCENAME 50000
DB20000I The UPDATE DATABASE MANAGER CONFIGURATION command completed successfully.
```

如需将某个参数设为空值，可采用如下命令，注意 NULL 要大写。

```
db2inst1@db2server:~>db2 update db cfg for sample using svcename NULL
```

4.7 管理服务器 (DAS)

DB2 DAS (Database Administration Server) 相当于一个代理，主要用来配合 DB2 图形界面完成一些操作，比如如果远程 DB2 客户端想通过图形界面对数据库进行定时备份，那么 DAS 需要启动。一台主机上可以有多个 DB2 实例，但是 DAS 只能有一个，可以同时管理并服务于多个 DB2 实例。

我们举例说明 DAS 的使用。与实例创建类似，DAS 也需要一个用户与之关联：

```
[root@db2server]# groupadd dasadm
[root@db2server]# mkdir -p /home/dasusr1
[root@db2server]# useradd -d /home/dasusr1 -g dasadm dasusr1
[root@db2server]# passwd dasusr1
```

然后创建 DAS：

```
[root@db2server] # pwd
/opt/ibm/db2/V9.5/instance
[root@db2server]# ./dascrt -u dasusr1
SQL4406W The DB2 Administration Server was started successfully.
DBI1070I Program dascrt completed successfully.
```

通过 db2admin start/stop 命令开始/停止 DAS:

```
[root@db2server]# su - dasusr1
[dasusr1@db2server]$ db2admin
dasusr1
[dasusr1@db2server]$ db2admin stop
SQL4407W The DB2 Administration Server was stopped successfully.
[dasusr1@db2server]$ db2admin start
SQL4406W The DB2 Administration Server was started successfully.
```

通过 dasdrop 命令删除 DAS:

```
[root@db2server]# pwd
/opt/ibm/db2/V9.5/instance
[root@db2server]# ./dasdrop
SQL4407W The DB2 Administration Server was stopped successfully.
DBI1070I Program dasdrop completed successfully.
```

注意, 如果不使用 DB2 提供的图形界面, 可以不创建 DAS。从 9.7 版本开始, DAS 将逐渐退出历史舞台。

4.8 小结

本章我们主要讨论了实例的概念, 并分别介绍了在 UNIX/Linux/Windows 上如何创建、删除、更新实例, 以及实例的启停。在日常的工作中, 除非有新的项目需要使用全新的实例, 或者系统迁移到其他硬件中, 或者安装新的 DB2 补丁, 一般来说不需要经常对实例进行操作。

4.9 判断题

(1) 数据库需要创建实例后才能使用。

T: 正确

F: 错误

(2) 用户可以使用 db2start/db2stop 命令, 来启动/停止实例。

T: 正确

F: 错误

(3) db2idrop 命令可以用来更新实例。

T: 正确

F: 错误

(4) db2 get dbm cfg 命令可以得到实例配置信息。

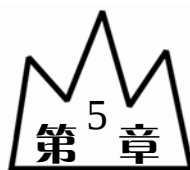
T: 正确

F: 错误

(5) 多个实例可以使用同一个端口号与应用程序通信。

T: 正确

F: 错误



数据库创建和存储管理

影响数据库性能的因素有很多，比如逻辑设计、物理设计、应用程序设计、SQL 语句、硬件资源配置等，其中存储规划物理设计是非常重要的一个方面，但是在实战中，发现很多企业对存储规划的重视不够，导致系统上线后存在严重的 I/O 资源瓶颈。对于 DBA 来说，对系统、存储方面的知识了解得越多，对数据库的管理就越有把握。

在 DB2 中，数据库的数据是存在表空间中的。本章从一个建库和表空间的例子入手，给大家一个直观的感觉。然后从表空间监控、扩容、状态和高水位标记等几个角度阐述表空间的运维管理。最后分享数据库存储规划和设计的最佳实践。

本章内容组织如下：

- 数据库结构。
- 创建数据库。
- 创建表空间。
- 表空间运维管理。
- 存储规划设计最佳实践。

5.1 数据库结构



当创建完实例后，就可以创建数据库，一个实例可以包含多个数据库，但一个数据库只能归属于一个实例。每个数据库是由一组对象构成的，如表、视图、索引等。表是二维结构，由一些行和列构成，表数据存放在表空间里，表空间是数据库的逻辑存储层，每个数据库包含多个表空间，每个表空间只能归属于一个数据库。从实例→数据库→表空间→表构成了 DB2 的逻

辑层次关系，从物理存储上，每个表空间由一个或多个容器构成，容器映射到物理存储，容器可以是目录，也可以是文件或裸设备，每个容器只能属于一个表空间。根据数据的管理方式，表空间分为系统管理（SMS）和数据库管理（DMS）。图 5.1 是 DB2 数据库对象逻辑结构图。

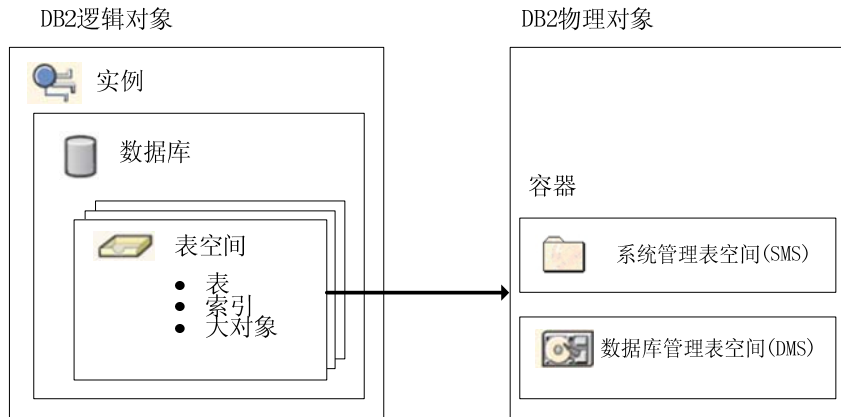


图 5.1 数据库对象逻辑层次关系

从应用的角度，并不需要关心表数据在内部是如何存储的，而只关心表的逻辑结构和表之间的关系。但是对于 DBA 来说，为了更好地规划存储设计和优化 I/O 性能，有必要深入理解 DB2 的内部存储结构，即存储模型。

DB2 将表和索引数据存在页（page）里，页是 DB2 的最小物理分配单元，表中每行数据只能包含在一页，不能跨页。DB2 支持的页大小共 4 种：4K、8K、16K 和 32K 页，假定一行数据大小为 20K，那么需要的页大小应该是 32K。

DB2 在读取数据时，从性能考虑，并不会每次读取一页，而是按块（extent）读取，extent 是一组连续的页。当一个表空间有多个容器，为了数据的均衡分布，DB2 在写数据时，按照循环的方式在各个容器里写数据，即在一个容器写满一个 extent 后，开始在第二个容器继续写 extent，周而复始，这样能够确保数据均匀分布在多个容器上，提高读/写效率。

每个表空间是由 1 个或多个容器构成的，表空间仅仅是逻辑存储层，具体的数据是存在容器里。容器是由多个 extent 构成的。

这样，从表空间→容器→extent→page 就构成了 DB2 的存储模型，图 5.2 解释了这个层次关系。

那么每个表空间的大小是否存在容量限制？在 v8 版本中，这是一个比较普遍的问题，因为 V8 采用的常规表空间最多只能支持 256GB 大小（页大小为 32K）。随着用户数据量的快速增长，这个限制已经无法满足用户需求。因此，从 DB2 9 开始，引入了大表空间(large)，它支持的最大容量可达到 16TB（针对 32K 页大小），从 9 开始，默认创建的数据表空间均为大表空间。

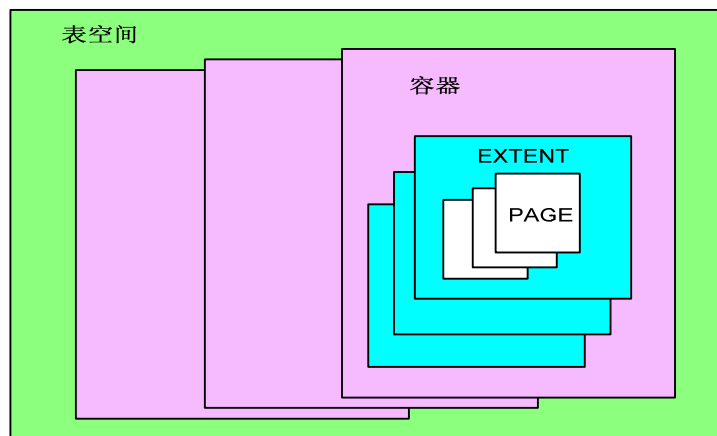


图 5.2 DB2 存储模型

5.2 建库、表空间



以下是一个客户建库、表空间的例子，后面会对每个语句的含义详细解释。

```
(1) db2 "CREATE DATABASE testdb AUTOMATIC STORAGE YES ON /dbauto DBPATH ON /database
USING CODESET UTF-8 TERRITORY CN COLLATE USING SYSTEM"

(2) db2 connect to testdb

(3) db2 "CREATE BUFFERPOOL BP32K SIZE 10000 PAGESIZE 32K"

(4) db2 "CREATE LARGE TABLESPACE tbs_data PAGESIZE 32K MANAGED BY DATABASE USING
(FILE '/data1/tbs_data/cont0' 100M, FILE '/data1/tbs_data/cont1' 100M ) EXTENTSIZ
32 PREFETCHSIZE automatic BUFFERPOOL BP32K NO FILE SYSTEM CACHING"

(5) db2 "CREATE TEMPORARY TABLESPACE tbs_temp PAGESIZE 32K MANAGED BY SYSTEM USING
('/data1/tbs_temp') BUFFERPOOL BP32K "

(6) db2 "CREATE USER TEMPORARY TABLESPACE tbs_user_temp PAGESIZE 32K MANAGED BY SYSTEM
USING ('/data1/tbs_usertemp') BUFFERPOOL BP32K "

(7) db2 "CREATE TABLESPACE tbs_index PAGESIZE 32K BUFFERPOOL BP32K"

(8)db2 "CREATE TABLESPACE tbs_data2 INITIALSIZE 100M INCREASESIZE 100M MAXSIZE 1000G"
```

本例中，共包含了 8 条语句，第（1）个语句创建数据库，第（2）语句连接数据库，第（3）语句创建缓冲池，第（4）条语句创建数据库管理（DMS）的数据表空间，第（5）语句用来创建系统管理（SMS）的临时表空间，第（6）语句创建 SMS 管理的用户临时表空间，（7）、（8）条语句创建自动存储管理（Automatic Storage）的表空间。

创建数据库 ■ ■ ■

对于建库语句，数据库名字是唯一必须提供的选项，并且长度不能超过 8bit。除此之外，以下几个选项也非常关键。

1. DBPATH ON

DBPATH ON 表示数据库目录，各位读者可以在建完库后浏览目录结构层次关系，如图 5.3 所示。

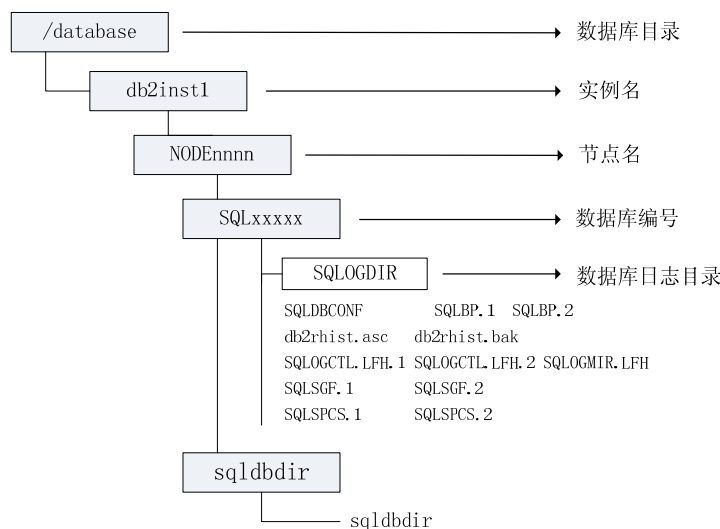


图 5.3 数据库目录结构图

在 /database 目录下，创建了名字为实例名的子目录，然后又创建了 NODExxxx 子目录，NODExxxx 是节点编号，在单分区环境，固定为 NODE0000，而多分区环境下每个分区对应一个节点目录。SQLxxxxx 则是数据库编号，每个数据库对应这样一个目录，sqldbdir 是本地数据库编目。

在 SQLxxxxx 数据库目录下包含一个 SQLOGDIR 子目录，默认情况下该目录用来存放数据库日志文件，在建库后可以更改日志目录的位置，我们将在第 8 章讲述。除此之外，SQLxxxxx 数据库目录还包含一些控制文件，这些文件都是二进制格式，不要尝试通过文件编辑工具打开，更不能修改、移动和删除。以下是一些重要配置文件的说明。

- SQLDBCONF：数据库参数配置文件，使用 `get db cfg/update db cfg` 读取和修改数据；
- SQLBP.*：缓冲池控制文件，使用 `alter bufferpool` 控制；
- SQLSPCS.*：表空间控制文件，使用 `list/alter tablespace` 控制；
- db2rhist.asc：数据库历史文件，使用 `LIST HISTORY` 浏览；
- SQLOGCTL.LFH.*：数据库日志控制文件，无法浏览或人工修改；

2. USING CODESET codeset TERRITORY territory

指定数据库编码集（Codeset）和区域（Territory）。数据库一旦创建，编码就无法改变，因此建库前需要根据应用和数据特点决定合适的编码集。目前国内客户常用的数据库编码集是 GBK 和 UTF-8，GBK 支持中文字符，UTF-8 支持几乎所有语言，ISO8859-1 支持英文字符。在 9.5 版本之前，如果不指定编码，DB2 默认根据系统的 Locale 来设置；从 9.5 版本开始，默认的数据库编码为 UTF-8。可以通过数据库参数查看数据库的编码集、代码页等信息。

```
db2inst1@db2server:~> db2 get db cfg for testdb | more
```

Database Configuration for Database testdb

```
Database configuration release level      = 0x0c00
Database release level                  = 0x0c00

Database territory                      = CN
Database code page                      = 1208
Database code set                       = UTF-8
Database country/region code           = 86
Database collating sequence             = IDENTITY
```

那么 Codepage 代表什么？与 Codeset 有什么关系？Codeset 编码集是操作系统级别的编码设置，是通用的；而 Codepage 代码页，是 DB2 对编码集的一种内部数字表示，只在 DB2 内有意义。本例中，编码集是 UTF-8，对应的 db2 数据库代码页是 1208。

感兴趣的读者，可通过信息中心查看 DB2 支持的代码页列表：

```
http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=/com.ibm.db2.luw.admin.nls.doc/doc/r0004565.html
```

在图 5.4 中，每个代码页都有对应的组、编码集和地域代码等。如果两个代码页属于同一个组，则它们可以互相转换，否则无法转换。需要注意的是：单字节（S）组和双字节（D）组都可以转换成中性（N）组，但 N 组不一定能转成 S 组或 D 组。如代码页 1386 属于 D 组，1208 属于 N 组，这两个代码页是可以相互转换的。

Code page	Group	Code set	Territory code	Collation	Locale	Operating system
437	S-1	IBM-437	86	Code page 437, Generic (SYSTEM_437)	-	-
819	S-1	ISO8859-1	86	Code page 819, Generic (SYSTEM_819)	-	-
850	S-1	IBM-850	86	Code page 850, Generic (SYSTEM_850)	-	-
923	S-1	ISO8859-15	86	Code page 923, Generic (SYSTEM_923)	-	-
1051	S-1	roman8	86	Code page 1051, Generic (SYSTEM_1051)	-	-
1383	D-4	IBM-eucCN	86	Code page 1383, Generic (SYSTEM_1383)	zh_CN	AIX
1386	D-4	GBK	86	Code page 1386, Generic (SYSTEM_1386)	Zh_CN.GBK	AIX
1208	N-1	UTF-8	86		ZH_CN	AIX
935	D-4	IBM-935	86		-	Host
1388	D-4	IBM-1388	86		-	Host
1383	D-4	hp15CN	86	Code page 1383, Generic (SYSTEM_1383)	zh_CN.hp15CN	HP-UX
1386	D-4	GBK	86	Code page 1386, Generic (SYSTEM_1386)	zh_CN.GBK	Linux

图 5.4 数据库代码页映射表

3. 创建表空间

当建库时，DB2 会创建三个默认表空间：系统表空间（system tablespace），用来存储系统表，也就是数据字典的信息，一个数据库只能有一个系统表空间；临时表空间（temporary tablespace），用来保存语句执行时产生的中间临时数据，如 join、排序等操作都可能产生一些临时数据；用户表空间（user tablespace），用来存储表、索引、大对象等数据。

在第（4）-（8）语句，我们创建了几个表空间。从管理方式上，表空间分为 SMS、DMS 和自动存储管理；从存储的数据类型上，表空间分为系统表空间、数据表空间、系统临时表空间 and 用户临时表空间。

第（4）条语句创建了一个数据表空间，managed by database 表示空间的分配和管理由 DB2 负责，即 DMS（database-managed space）。Using 指定表空间的容器，DMS 支持的容器类型是文件和裸设备（raw device，可理解成没有格式化的盘）。DMS 类型的表空间在创建时即分配空间，创建后可通过命令对表空间容器进行增删改。对于数据来说，建议用 DMS 管理。

第（5）条语句创建了一个系统临时表空间，系统临时表空间用来存储 DB2 产生的一些临时数据。managed by system 表示空间的分配和管理由操作系统负责，即 SMS（system-managed space）。Using 指定表空间的容器，SMS 支持的容器类型只能是目录，并且无需指定大小，只要路径所属的文件系统有空间，就可以被表空间使用。SMS 的优点是比较好管理，缺点是性能比 DMS 差一些，大概差 5%~10% 左右。对于临时表空间来说，建议用 SMS 管理。

第（6）条语句创建了一个用户临时表空间，与系统表空间类似，用户临时表空间也用来保存临时数据，但它存储的是用户自定义的临时表。对于用户临时表空间，建议用 SMS 管理。

第（7）条语句创建了一个自动存储管理表空间。自动存储管理（automatic storage）是 DB2 v8.2 就有的特性，它的目的是简化表空间的监控和管理。当表空间创建时，只需提供表空间名，而无需指定容器类型和大小。实际上，自动存储管理底层使用的仍然是 DMS 或 SMS 类型，只不过是容器不需指定而已。对于数据表空间，DB2 会选用 DMS 方式管理；对于临时表空间，会选用 SMS 方式管理。那么自动存储管理表空间的数据存在哪里呢？答案就是建库时指定的 ON 目录，也叫自动存储路径。只有建库时启用了 automatic storage yes，表空间才支持自动存储管理。

本例中，/dbauto 是自动存储管理目录，T0000000 是系统表空间目录，T0000001 是临时表空间目录，T0000002 是用户表空间目录。目录结构参看图 5.5。

那么对于一个已经创建的表空间，如何确定是否采用了自动存储管理呢？答案是表空间快照的“Using automatic storage”选项，“Yes”表示使用了自动存储管理，“No”则表示非自动存储管理。

```
db2 get snapshot for tablespaces on testdb |more
... ..
Tablespace name           = tbs_data
Tablespace ID             = 3
Tablespace Type           = Database managed space
```

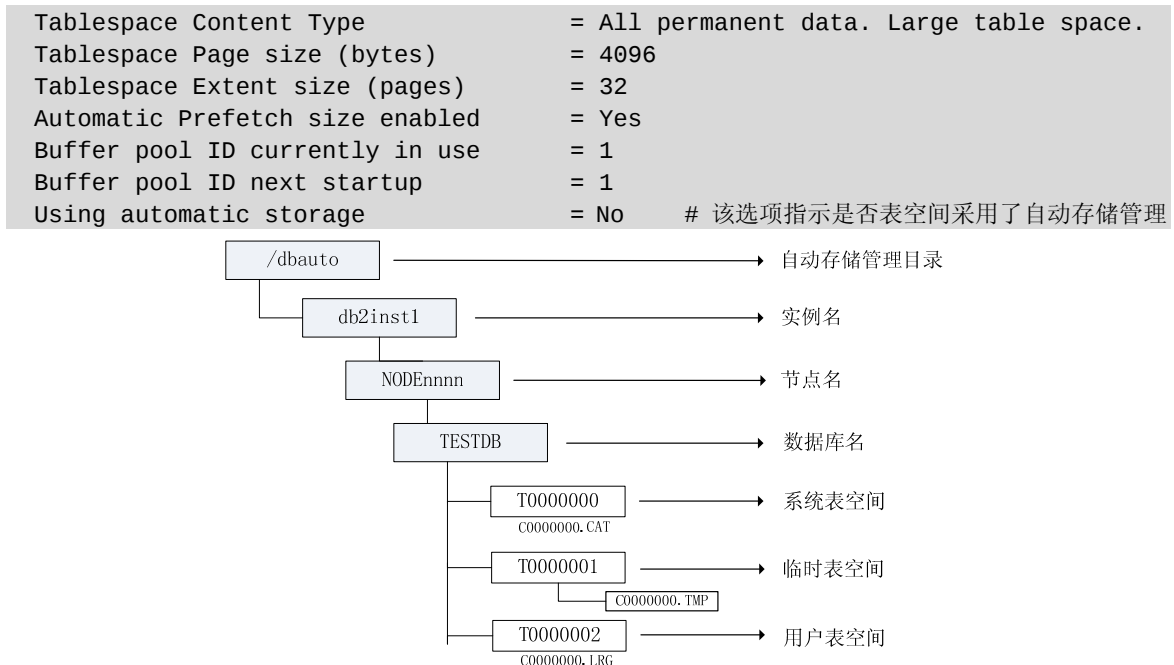


图 5.5 自动存储管理表空间目录结构

在（3）步，我们创建了一个缓冲池（bufferpool），Bufferpool 是数据库最重要的内存区，用来缓存数据，提高性能。创建缓冲池时，size 表示页数，pagesize 表示页大小，size*pagesize 就是缓冲池的内存大小。在建库时，DB2 会自动创建一个名为 IBMDEFAULTBP 的缓冲池，如果在建库时没有指定 pagesize，那么 IBMDEFAULTBP 的默认 pagesize 是 4K。

在（4）～（7）语句中可以看到，每个表空间都有一个 Bufferpool 与之对应，多个表空间可以共享同一个 Bufferpool，但要求 Bufferpool 的 pagesize 大小必须与表空间的 pagesize 匹配，否则无法创建。本例中，由于（4）～（7）的表空间页大小均为 32K，而 IBMDEFAULTBP 页大小为 4K，无法匹配，所以我们事先创建了一个页大小为 32K 名为 BP32K 的缓冲池。缓冲池的大小可以更改，命令如下，其中 20000 是页大小，即 20000*32K=640M。

```
inst20@db2server: > db2 alter bufferpool bp32k size 20000
```

在第（4）个语句，有一个 no file system caching 选项。该选项的目的是关闭文件系统缓存，原因是 DB2 使用 Bufferpool 缓存数据，为了减小额外的开销，不必使用文件缓存，no file system caching 也是 DB2 9.5 版本的默认选项。注意：大对象(CLOB,BLOB)数据的获取直接通过磁盘，无法通过 Bufferpool 缓存，因此可考虑将大对象数据创建在独立的表空间上，并使用 file system caching 选项。

5.3 表空间维护管理

5.3.1 表空间监控 ■ ■ ■

表空间的管理方式不同，监控的结果也有很大差异。对于 SMS 表空间，只要容器目录有空间，就不会出现表空间满；对于自动存储管理表空间，只要自动存储路径有空间，就不会出现表空间满；而对于 DMS 表空间，要随时监控表空间的使用，防止出现空间满无法插入数据的情况。

常用的表空间监控方法：

1. db2 list tablespaces 或 db2 list tablespace containers for <tablespace_id>

list tablespaces 会显示每个表空间的最核心的信息，比较简单和直观，以下是一个表空间片段：

```
db2inst1@dpf1 > db2 list tablespaces show detail
...
Tablespace ID          = 7                # 表空间 ID，每个表空间 ID 唯一
Name                   = DATA_TS3          # 表空间名字
Type                   = Database managed space # 表空间管理类型
# 表空间使用类型
Contents               = All permanent data. Large table space.
State                  = 0x0000          # 表空间状态，比较关键，需关注
Detailed explanation:
  Normal
Total pages            = 8192            # 表空间总的页数
Useable pages          = 8160            # 表空间可用页数
Used pages             = 96              # 表空间已使用页数
Free pages             = 8064            # 表空间剩余可用页数，需关注该值
High water mark (pages) = 96            # 表空间高水位标记 (HWM)，需关注
Page size (bytes)      = 4096            # 页大小
Extent size (pages)    = 32              # 块大小
Prefetch size (pages)  = 32              # 预取大小
Number of containers   = 1              # 表空间容器个数
```

表空间容器的相关信息可通过 list tablespace containers 命令查看。

```
db2inst1@dpf1 > db2 list tablespace containers for 7 show detail # 7 是表空间 ID

Tablespace Containers for Tablespace 7

Container ID          = 0                # 容器 ID
Name                  = /data1/db2inst1/NODE0000/TESTDB/T00000007/C00000000.LRG # 容器路径
Type                  = File             # 容器类型，支持的类型包括 file, path 和 device
Total pages           = 8192            # 容器大小
Useable pages         = 8160            # 容器可用页数
Accessible            = Yes             # 容器是否可访问，当表空间状态出现异常时，可
```

查看容器是否可访问，因为容器对应的是存储。

2. db2pd -d <db_name> tablespaces

该命令可以直观地显示表空间的配置信息、使用情况和容器信息。

```
db2inst1@dpf1:/data1/db2inst1/NODE0000/TESTDB> db2pd -d testdb -tablespaces
```

Database Partition 0 -- Database TESTDB -- Active -- Up 1 days 00:05:09

Tablespace Configuration:

Id	Type	Content Name	PageSz	ExtentSz	Auto	Prefetch	BufID	NumCntrs	MaxStripe
0	DMS	Regular	4096	4	Yes	4	1	1	0
SYSCATSPACE									
1	SMS	SysTmp	4096	32	Yes	32	1	1	0
TEMPSPACE1									
2	DMS	Large	4096	32	Yes	32	1	1	0
USERSPACE1									
3	DMS	Large	4096	32	Yes	32	1	1	0
DATA_TS									
4	DMS	Large	4096	32	Yes	96	1	4	1
DATA_TS2									
5	DMS	Large	4096	4	Yes	4	1	1	0
SYSTOOLSPACE									
6	SMS	SysTmp	4096	32	Yes	32	1	1	0
TEMP_TS1									
7	DMS	Large	4096	32	Yes	32	1	1	0
DATA_TS3									
8	DMS	Large	4096	32	Yes	32	1	1	0
DATA_TS4									
9	SMS	UsrTmp	4096	4	Yes	4	1	1	0
SYSTOOLSTMPSPACE									

Tablespace Statistics:

Id	TotalPgs	UsablePgs	UsedPgs	PndFreePgs	FreePgs	HWM	Max HWM	State
0	24576	24572	19932	0	4640		19932	19932
0x00000000								
1	1	1	0	0	0	0x00000000		
2	8192	8160	96	0	8064	96	96	0x00000000
3	2560	2528	288	0	2240	288	288	0x00000000
4	15360	15232	96	0	15136	96	96	0x00000000
5	8192	8188	152	0	8036	152	152	0x00000000
6	1	1	0	0	0	0	0	0x00000000
7	8192	8160	96	0	8064	96	96	0x00000000
8	2560	2528	96	0	2432	96	96	0x00000000
9	1	1	0	0	0	0	0	0x00000000

Tablespace Autoresize Statistics:

Id	AS	AR	InitSize	IncSize	IIP	MaxSize	LastResize	LRF
----	----	----	----------	---------	-----	---------	------------	-----

0	Yes	Yes	33554432	-1	No	None	None	No
1	Yes	No	0	0	No	0	None	No
2	Yes	Yes	33554432	-1	No	None	None	No
3	No	No	0	0	No	0	None	No
4	No	No	0	0	No	0	None	No
5	Yes	Yes	33554432	-1	No	None	None	No
6	No	No	0	0	No	0	None	No
7	Yes	Yes	33554432	-1	No	None	None	No
8	Yes	Yes	10485760	10485760	No	107374182400	None	No
9	Yes	No	0	0	No	0	None	No

Containers:

TspId	ContainNum	Type	TotalPgs	UseablePgs	StripeSet	Container
0	0	File	24576	24572	0	/data1/db2inst1/NODE0000/TESTDB/T00000000/C00000000.CAT
1	0	Path	1	1	0	/data1/db2inst1/NODE0000/TESTDB/T00000001/C00000000.TMP
2	0	File	8192	8160	0	/data1/db2inst1/NODE0000/TESTDB/T00000002/C00000000.LRG
3	0	File	2560	2528	0	/data1/ts1/cont0
4	0	File	5120	5088	0	/data1/ts2/cont0
4	1	File	5120	5088	0	/data1/ts2/cont1
4	2	File	2560	2528	0	/data1/ts2/cont2
4	3	File	2560	2528	1	/data1/ts2/cont3
5	0	File	8192	8188	0	/data1/db2inst1/NODE0000/TESTDB/T00000005/C00000000.LRG
6	0	Path	1	1	0	/data1/temptps
7	0	File	8192	8160	0	/data1/db2inst1/NODE0000/TESTDB/T00000007/C00000000.LRG
8	0	File	2560	2528	0	/data1/db2inst1/NODE0000/TESTDB/T00000008/C00000000.LRG
9	0	Path	1	1	0	/data1/db2inst1/NODE0000/TESTDB/T00000009/C00000000.UTM

3. db2 get snapshot for tablespaces on <db_name>

与 list tablespaces 结果相比, get snapshot for tablespaces 包含的内容更全面。比如, 是否启用了自动存储功能, 以及表空间 map 信息。

4. sysibmadm.snaptbody 和 sysibmadm.snapcontainer 管理视图

管理视图显示内容与 snapshot 快照类似, 但以视图的形式更容易做成脚本执行。

注意: SMS 的表空间使用情况无法通过命令监视, 只受文件系统限制。

5.3.2 表空间更改 ■ ■ ■

在运维过程中, 随着数据量的增长, 不可避免地会遇到当前表空间使用过高、需要扩容的情况。对于 SMS 类型表空间, 不支持表空间容器的更改, 只能更改容器路径所属的文件系统大小。

对于 DMS 表空间, 提供了几个方法更改表空间容器。其中 Add 用来增加新的容器, Drop 删除容器, Extend 用来扩展已有容器大小, Reduce 用来缩减已有容器大小, Resize 重新设定容

器大小。对于 Add 和 Drop 操作，表空间容器间会发生数据重新平衡（Rebalance）。对于 Reduce 和 Resize 操作，需要确保更改后的表空间容器有足够的空间，否则 DB2 会拒绝该操作。

在 DBA 日常运维中，经常会遇到 DMS 表空间满的情况，这时根据存储空间和对运维的影响，有以下 3 种方案：

（1）如果表空间容器对应的存储中还有未分配空间，可通过 alter tablespace 的 extend 或 resize 选项扩展已有表空间容器的大小。下例是在每个容器上扩展了 50GB：

```
db2inst1@dpf1> db2 "alter tablespace data_ts2 extend (file '/data1/ts2/cont0' 10M,
file '/data1/ts2/cont1' 50G )"
DB20000I The SQL command completed successfully
```

（2）如果表空间容器对应的存储中没有剩余空间时，可通过 alter tablespace 的 add 选项增加新的容器。需要注意的是：通过 add 增加容器会在容器间进行数据 Rebalance，即数据重新平衡。如果数据量很大，rebalance 的时间会很长，对系统性能会造成很大影响，我们曾经遇见过一个 10T 的系统，增加了一个新的容器后，Rebalance 操作花费了接近 30 小时才完成。

下例增加了一个新的容器，表空间发生 rebalance：

```
db2inst1@dpf1> db2 "alter tablespace data_ts2 add (file '/data1/ts2/cont2' 50G )"
DB20000I The SQL command completed successfully
```

（3）通过 alter tablespace begin new stripe set 选项。Begin new stripe set 选项是当已有容器使用完后，再使用新增加的容器。与方法 2 不同，该选项不会在容器间做 Rebalance，不会对系统造成性能影响，但它会造成数据偏移。举例如下：

```
db2inst1@dpf1> db2 "alter tablespace data_ts2 begin new stripe set (file
'/data1/ts2/cont3' 10M )"
DB20000I The SQL command completed successfully
```

对于自动存储管理的表空间，无法在表空间级进行容器更改，只能在数据库级别，因为自动存储路径是在建库时指定的。可以使用 add storage on 选项为数据库添加新的存储路径，命令是：db2 alter database db_name add storage on db_path3。

9.7 版本之前，对自动存储路径的管理有一定局限性。比如，只能增加路径，而不能删除；新增加的存储路径不会被表空间立即使用，只有已有的存储路径文件系统空间满了，才会使用新增加的存储路径，增加存储路径只是为了解决容量问题。

9.7 版本在自动存储路径的管理方面进行了增强，当新增了存储路径，只要对自动存储表空间执行了 rebalance 操作，就可以立即使用这个存储路径。以下是增加存储路径的例子：

```
db2inst1@dpf1:~> db2pd -d sample -storagepaths

Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:16:49

Database Storage Paths:
Number of Storage Paths      1
```

```

Address    PathID      PathState  PathName
0xA40F4000 0          InUse      /data1

db2inst1@dpf1:/data1> db2 "alter database sample add storage on '/data2/' "
DB20000I The SQL command completed successfully

db2inst1@dpf1:/data1> db2pd -d sample -storagepaths

Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:19:21

Database Storage Paths:
Number of Storage Paths      2

Address    PathID      PathState  PathName
0xA40F4000 0          InUse      /data1
0xA6649420 1          NotInUse   /data2

db2inst1@dpf1:/data1> db2 alter tablespace ts1 rebalance
DB20000I The SQL command completed successfully

db2inst1@dpf1:/data1> db2 list tablespace containers for 7 show detail

          Tablespace Containers for Tablespace 7

Container ID      = 0
Name              = /data1/db2inst1/NODE0000/SAMPLE/T0000007/C0000000.LRG
Type              = File
Total pages       = 192
Useable pages     = 160
Accessible        = Yes
Container ID      = 1
Name              = /data2/db2inst1/NODE0000/SAMPLE/T0000007/C0000001.LRG
Type              = File
Total pages       = 192
Useable pages     = 160
Accessible        = Yes

```

既然自动存储管理表空间有这么多优点，那么是否有办法把已有的 DMS 表空间转换为自动存储管理呢？答案是肯定的。在转换之前需要确保数据库启动了自动存储管理。转换方法如下：

(1) 更改表空间。这种方法保持表空间持续可用，但从 DMS 容器向自动存储路径迁移数据时需要花费一些时间做数据重新平衡操作，所需时间依赖于数据量的大小和存储 I/O 的性能，更改方法如下：

```

db2inst1@dpf1:/data1> db2 "create tablespace ts4 managed by database using (file
'/data1/ts2/cont0' 10M) "
DB20000I The SQL command completed successfully

db2inst1@dpf1:/data1> db2 alter tablespace ts4 managed by automatic storage

```

```
DB20000I The SQL command completed successfully

db2inst1@dpf1:/data1> db2 alter tablespace ts4 rebalance
DB20000I The SQL command completed successfully
```

(2) 采用重定向恢复，正在恢复的表空间不允许访问。

```
$ db2 RESTORE DATABASE database_name TABLESPACE table_space_name REDIRECT
$ db2 SET TABLESPACE CONTAINERS FOR tablespace_id USING AUTOMATIC STORAGE
$ db2 RESTORE DATABASE database_name CONTINUE
$ db2 ROLLFORWARD DATABASE database_name TO END OF LOGS AND STOP
```

5.3.3 表空间状态 ■ ■ ■

在 DBA 运维过程中，需要随时监控表空间的状态。当状态出现异常时，表数据一般都是不可访问的，因此需要深入理解各种状态出现的原因，这样当出现某种状态时能够迅速的定位问题并找到解决方案。可通过 `list tablespaces show detail` 查看表空间的状态。表 5-1 汇总了各种表空间状态、十六进制值和状态描述信息。可以看到，表空间状态主要分两类：一类是某些命令操作期间使用的临时状态，一类是挂起（Pending）状态。一般当出现什么样的挂起，就执行什么操作，如 Backup pending 就做 backup，Rollforward pending 就执行 rollforward。

表 5-1 各种

表空间状态	十 六 进 制	描 述
Normal	0x0	正常状态。一般情况下，只有表空间处于正常状态时，数据才是可以访问的
Quiesced Share	0x1	静默共享状态。当对表空间发出 Quiesce ... share 时，表空间就处于此状态。各应用只能读取该表空间的数据，但无法写
Quiesced Share	0x4	静默排它状态。当对表空间发出 Quiesce ... exclusive 时，表空间就处于此状态。调用该命令的应用可以读/写该表空间数据，但其他应用无法读/写
Backup Pending	0x20	在归档日志模式下，当执行了表空间前滚后，或使用了 Load ... copy no 选项，表空间将处于该状态
Rollforward In Progress	0x40	在执行前滚时出现的临时状态
Rollforward Pending	0x80	在归档日志模式下，当执行了数据库恢复后，表空间就处于此状态
Restore Pending	0x100	当执行重定向恢复时，在 set tablespace containers 之前，表空间将处于此状态
Disable Pending	0x200	在执行数据库前滚操作期间，表空间可能处于这种状态，但在前滚操作结束之时，就不应再处于该状态
Reorg In Progress	0x400	在进行 reorg 操作期间才出现的临时状态
Backup In Progress	0x800	在进行 backup 操作期间才出现的临时状态

续 表

表空间状态	十 六 进 制	描 述
Storage Must Be Defined	0x1000	使用重定向恢复到一个新的数据库期间, 如果省略了设置表空间容器的阶段, 或者在设置表空间容器的阶段无法获得指定的容器, 那么数据库的表空间就会处于这种状态
Restore In Progress	0x2000	在进行 restore 操作期间才出现的临时状态
Offline and not accessible	0x4000	如果表空间的一个或多个容器存在问题, 比如被重命名、移动、修改或损坏时时, 表空间就处于该状态。当修复容器后, 可以重启数据库来消除该异常状态。或者执行 ALTER TABLESPACE ... SWITCH ONLINE 消除该状态
Drop Pending	0x8000	在重启数据库时, 如果数据库的一个或多个容器有问题, 那么表空间就会处于该状态, 这时该表空间将不再可用。很多情况下, 出现该状态时并不一定要删除表空间, 看我们稍后的例子
Storage May Be Defined	0x2000000	在执行了重定向恢复操作的第一部分之后, SET TABLESPACE CONTAINERS 命令之前, 表空间就处于这种状态
DMS Rebalance in Progress	0x10000000	在执行数据重新平衡操作 (Rebalance) 期间才出现的临时状态
Table Space Deletion in Progress	0x20000000	在执行删除表空间的操作期间才出现的临时状态
Table Space Creation in Progress	0x40000000	在执行创建表空间的操作期间才出现的临时状态

下面介绍几种常用表空间状态的诊断和模拟。

1. Quiesced 状态

当进行运维时, 可通过 quiesce 命令锁定表空间防止其他用户对该表空间的表数据进行更改。

```
db2inst1@dpf1:~> db2 quiesce tablespaces for table db2inst1.t1 share
DB20000I The QUIESCE TABLESPACES command completed successfully

db2inst1@dpf1:~> db2 "insert into t1 values('adadf','dadaf') "
DB21034E The command was processed as an SQL statement because it was not a
valid Command Line Processor command. During SQL processing it returned:
SQL0290N Table space access is not allowed. SQLSTATE=55039

db2inst1@dpf1:~> db2 list tablespaces show detail
Tablespace ID          = 7
Name                   = TS1
Type                   = Database managed space
Contents               = All permanent data. Large table space
State                  = 0x0001
Detailed explanation:
```

```

Quiesced: SHARE
Total pages                      = 4096

db2inst1@dpf1:~> db2 quiesce tablespaces for table db2inst1.t1 reset
DB20000I  The QUIESCE TABLESPACES command completed successfully.

```

2. Drop pending 状态

这是一个在 Windows 平台发生的案例，客户将数据库备份介质复制到另外一台机器上恢复，结果恢复后很多表空间出现 Offline+Drop pending 状态。

```

Tablespace ID = 3
Name = SDCSM_TABLE_SPACE
Type = Database managed space
Contents = All permanent data. Large table space
State = 0xc000
Detailed explanation:
Offline
Drop Pending

```

检查 db2diag.log 日志，提示磁盘空间满，但检查磁盘空间时，发现还有 60GB。继续检查，发现文件系统是 FAT32 格式，我们知道 FAT32 能支持的最大文件大小为 4GB，因此会出现表空间满的问题。通过 convert 命令将 FAT32 格式转换为 NTFS 后，重新进行数据库恢复，问题解决。

```

...
2010-10-25-19.59.40.843000+480 I303661H1328          LEVEL: Error
PID      : 1608                TID : 3376          PROC : db2syscs.exe
INSTANCE: DB2                 NODE : 000
EDUID    : 3376                EDUNAME: db2bm.4060.0 (QDWSJ) 0
FUNCTION: DB2 UDB, buffer pool services, sqlbASDefineContainersForDMS, probe:10
MESSAGE  : ZRC=0x850F000C=-2062614516=SQL0_DISK "Disk full."
          DIA8312C Disk was full.
DATA #1 : String, 878 bytes
...
2010-10-25-20.43.13.656000+480 E311606H558          LEVEL: Info
PID      : 1608                TID : 3064          PROC : db2syscs.exe
INSTANCE: DB2                 NODE : 000          DB  : QDWSJ
APPHDL   : 0-122              APPID: *LOCAL.DB2.101025123630
AUTHID   : ADMINISTRATOR
EDUID    : 3064                EDUNAME: db2agent (QDWSJ) 0
FUNCTION: DB2 UDB, buffer pool services, sqlbStartPools, probe:15
DATA #1 : <preformatted>
Non-recoverable database detected and table space SDCSM_TABLE_SPACE
is in RESTORE_PENDING. Switching to DROP_PENDING.

2010-10-25-20.43.13.656000+480 E312166H676          LEVEL: Warning
PID      : 1608                TID : 3064          PROC : db2syscs.exe
INSTANCE: DB2                 NODE : 000          DB  : QDWSJ
APPHDL   : 0-122              APPID: *LOCAL.DB2.101025123630
AUTHID   : ADMINISTRATOR

```

```

EDUID   : 3064                      EDUNAME: db2agent (QDWSJ) 0
FUNCTION: DB2 UDB, buffer pool services, sqlbStartPools, probe:19
MESSAGE : ADM6047W The table space "SDCSM_TABLE_SPACE" ( ID "3") is in the
DROP_PENDING state. The table space will be kept OFFLINE. The table space state is
0x"8000". This table space is unusable and should be dropped

```

3. Offline and not accessible

当表空间容器出现异常时，状态可能会变为 Offline and not accessible 状态。以下进行模拟实验：

```

inst20@db2server:~> db2 "create tablespace ts1 managed by database using (file
'/data1/dms/cont2' 10M) "
DB20000I The SQL command completed successfully
inst20@db2server:~>
inst20@db2server:~> db2 "create table t1 ( id char(10), name char(10) ) in ts1"
DB20000I The SQL command completed successfully
inst20@db2server:~>
inst20@db2server:/data1/dms> mv cont2 cont3                # 将 cont2 容器文件改为 cont3

inst20@db2server:/data1/dms> db2 connect reset
DB20000I The SQL command completed successfully
inst20@db2server:/data1/dms> db2 connect to testdb          # 重新进行数据库连接
Database Connection Information

Database server          = DB2/LINUX 9.5.5
SQL authorization ID     = INST20
Local database alias     = TESTDB

tinst20@db2server:/data1/dms> db2 list tablespaces

Tablespace ID           = 6
Name                     = TS1
Type                     = Database managed space
Contents                 = All permanent data. Large table space
State                    = 0x4000
Detailed explanation:
  Offline                # 表空间处于 Offline 状态

inst20@db2server:/data1/dms> db2 list tablespace containers for 6 show detail

Tablespace Containers for Tablespace 6

Container ID             = 0
Name                     = /data1/dms/cont2
Type                     = File
Total pages              = 2560
Useable pages            = 2528
Accessible                = No                # 表空间容器处于不可访问状态

```

```

inst20@db2server:/data1/dms> db2tbst 0x4000          # db2tbst 对表空间状态进行解释
State = Offline and not accessible

inst20@db2server:/data1/dms> mv cont3 cont2          # 将容器文件恢复成原来样式
# 更改表空间，改回在线状态
inst20@db2server:/data1/dms> db2 alter tablespace ts1 switch online
DB20000I The SQL command completed successfully

```

5.3.4 表空间高水位

在表空间使用过程中，经常遇到高水位标记（High Water Mark, HWM）问题。HWM 是 DMS 表空间的一个属性，代表表空间当前分配的最高页数，这个值可能会大大高于已经使用的页数（used pages）。以图 5.6 为例，一个表空间中创建了 Table1 和 Table2 两张表，HWM 指向 Table2 的最后一页。如果 Table1 被删除，则可用空间会增加，但并不会被 DB2 自动回收，HWM 仍然指向 Table2 的最后一页。

以下几种情况，HWM 会对表空间管理产生比较大的影响。

- 当通过 alter tablespace 的 reduce、resize 或 drop 选项对表空间进行更改时，如果更改后的页数小于 HWM 的值，则操作将会失败。
- 备份时，DB2 会复制每个表空间 HWM 以下的所有数据块。在进行表空间重定向恢复时，如果重新设定的容器大小小于 HWM 值，即使有很多空闲块，恢复也无法进行。
- 表空间满了，删除了很多表数据后，却发现 HWM 仍然很高。

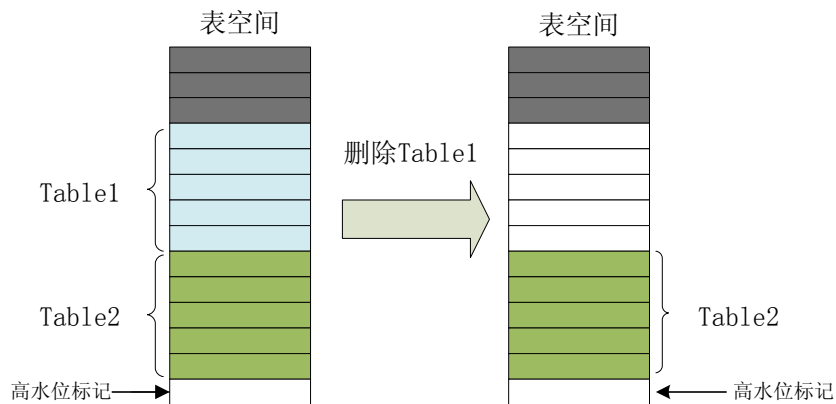


图 5.6 表空间高水位标记

1. 9.7 版本之前降低 HWM

9.7 版本之前，找到占据 HWM 的对象并降低 HWM 不是一件容易的事。好在 db2dart 中提供了 3 个选项可以协助我们处理 HWM 问题，分别是、/DHWM/LHWM 和/RHWM 选项。

1) DHWM (Dump HWM) 选项

该选项显示表空间和高水位标记信息，如定位哪个对象占据 HWM、HWM 以下剩余 extents 和已用 extents 的数量等。以下举例说明：

```
db2inst1@dpf1: > db2 "create table t1 (col1 char(100) ) in data_ts3"  # 在 data_ts3
中创建 T1 表
db2inst1@dpf1: > db2 "call sp_insert(10000) " # 调用存储过程，在 T1 表中插入 1 万行数据
db2inst1@dpf1: > db2 "create table t2(col1 char(100) ) in data_ts3 "  # 在 data_ts3
中创建 T2 表

# 查看表空间 DATA_TS3 的使用页数和 HWM 页数
db2inst1@dpf1: > db2 list tablespaces show detail
Tablespace ID          = 7
Name                   = DATA_TS3
Total pages            = 8192
Useable pages          = 8160
Used pages             = 480
Free pages             = 7680
High water mark (pages) = 480

# 然后将表 T1 删除，观察表空间使用页数和 HWM
db2inst1@dpf1: > db2 drop table t1

# 注意到 Used pages 已经降低到 160 页，但 HWM 仍然是 480 页
db2inst1@dpf1: > db2 list tablespaces show detail
Tablespace ID          = 7
Name                   = DATA_TS3
Total pages            = 8192
Useable pages          = 8160
Used pages             = 160
Free pages             = 8000
High water mark (pages) = 480
```

接下来用 db2dart DHWM 选项显示 HWM 相关信息，可以看到 HWM 以下有 10 个空余 extents，有 5 个已用块，占有 HWM 的对象是 tableid=5 的表。根据系统表可知，是 T2 占据了 HWM。

```
db2inst1@dpf1: > db2dart testdb /DHWM
Please enter tablespace ID:7
    The requested DB2DART processing has completed successfully!
    Complete DB2DART report found in:
/home/db2inst1/sqllib/db2dump/DART0000/TESTDB.RPT

db2inst1@dpf1: > more /home/db2inst1/sqllib/db2dump/DART0000/TESTDB.RPT

Highwater Mark: 480 pages, 15 extents (extents #0 - 14)

[0000] 65534 0x0e [0001] 65534 0x0e [0002] 65535 0x00 [0003] == EMPTY ==
[0004] == EMPTY == [0005] == EMPTY == [0006] == EMPTY == [0007] == EMPTY ==
[0008] == EMPTY == [0009] == EMPTY == [0010] == EMPTY == [0011] == EMPTY ==
[0012] == EMPTY == [0013] 5 0x40* [0014] 5 0x00*
```

```

Dump highwater mark processing - phase start.

Number of free extents below highwater mark: 10
Number of used extents below highwater mark: 5

Object holding highwater mark:

Object ID: 5
Type:      Table Data Extent
Note: Extent contains page #0 for object.

# 通过系统表找到 tbspaceid=7、tableid=5 的表
db2inst1@dpf1: > db2 "select substr(tabname,1,18) as tabname, tbspaceid, tableid from
syscat.tables where tbspaceid=7 and tableid=5"

TABNAME          TBSPACEID TABLEID
-----
T2                7         5

1 record(s) selected.

```

2) LHWMM (Lower HWM) 选项

该选项提供了降低 HWM 的建议和方法，如重组 (reorg)、数据导出加载 (export/load)、删除重建 (export/drop/create/load) 等，需要注意的是，LHWM 给出的建议并不能总是获得预期的结果。

继续以上的例子，LHWM 选项建议对 db2inst1.T2 做离线 reorg 来降低 HWM。

```

db2inst1@dpf1:~> db2dart testdb /LHWM
Please enter tablespace ID, and number of pages (desired highwater mark): 7,0
The requested DB2DART processing has completed successfully!
Complete DB2DART report found in:
/home/db2inst1/sqllib/db2dump/DART0000/TESTDB.RPT

db2inst1@dpf1:~> more /home/db2inst1/sqllib/db2dump/DART0000/TESTDB.RPT
Highwater Mark: 480 pages, 15 extents (extents #0 - 14)

Lower highwater mark processing - phase start.
Current highwater mark:                14
Desired highwater mark:                0
Number of used extents in tablespace:  5
Number of free extents below original HWM: 10
Number of free extents below desired HWM: 0
Number of free extents below current HWM: 10

Step #1: Object ID = 5

=> Offline REORG of this table (do not specify a temporary tablespace

```

```
and do not use the LONGLOBDATA option).
```

```
Table: DB2INST1.T2
```

```
DAT object size: 2
INX object size: 0
XDA object size: 0
LF object size: 0
LOB object size: 0
LOBA object size: 0
BMP object size: 0
```

```
Total size of object parts: 2
```

```
Minimum number of extents that will move by this operation: 2
```

```
Current highwater mark:          4
Desired highwater mark:          0
Number of used extents in tablespace: 5
Number of free extents below original HWM: 10
Number of free extents below desired HWM: 0
Number of free extents below current HWM: 0
```

```
** Run the suggested offline REORG for the table first, and then run LHWM
for the suggestion on other objects.
```

```
Lower highwater mark processing - phase end.
```

```
Highwater mark processing - phase end.
```

由于 T2 占用两个 extents，而 HWM 下有 10 个空余 extents，这种情况下 LHWM 会建议通过离线 reorg 降低 HWM。如果 HWM 下没有足够的空间完成离线 reorg，reorg 操作反而会增大 HWM 的值。这时，db2dart LHWM 选项就会建议导出数据并重建来降低 HWM。

对 T2 执行离线 reorg，并观察 HWM 信息，发现 HWM 已经降到了 160 页。

```
db2inst1@dpf1:~> db2 reorg table db2inst1.T2
DB20000I The REORG command completed successfully.
```

```
db2inst1@dpf1:~> db2 list tablespaces show detail
```

```
Tablespace ID          = 7
Name                   = DATA_TS3
Total pages            = 8192
Useable pages          = 8160
Used pages             = 160
Free pages             = 8000
High water mark (pages) = 160
```

关于 Reorg 的详细信息，请参看第 10 章。

3) RHWL (Remove HWM) 选项

该选项用来删除占据 HWM 的空 SMP 块。SMP 块用来标识该块映射的一组 extents 是否可

用，如果一个空 SMP 占据了 HWM，可通过 RHWMM 选项来降低。

```
db2inst1@dpf1:~> db2dart testdb /RHWMM
Please enter tablespace ID:
7
    The requested DB2DART processing has completed successfully!
    Complete DB2DART report found in:
/home/db2inst1/sqllib/db2dump/DART0000/TESTDB.RPT

db2inst1@dpf1:~> more /home/db2inst1/sqllib/db2dump/DART0000/TESTDB.RPT
Connecting to Buffer Pool Services...
Highwater mark processing - phase start.
Reduce highwater mark processing - phase start.
The Highwater mark cannot be reduced.
The index of the last SMP extent used is: (0)
Reduce highwater mark processing - phase end.
```

注意：表在离线重组（Reorg）时会保留原数据，同时在表空间内进行一份数据复制，当复制结束后删除原表数据块。如果 HWM 下没有足够的空间保存数据复制，则重组不但不能降低 HWM，反而会导致 HWM 增加。

2. 9.7 版本降低 HWM

9.7 简化了降低 HWM 的方法。以图 5.7 为例，Table1 和 Table2 分别占据一些 extent，当 Table1 对象被删除后，通过 alter tablespace 可回收空余空间，占据 HWM 的 Table2 会移动到较低位置。

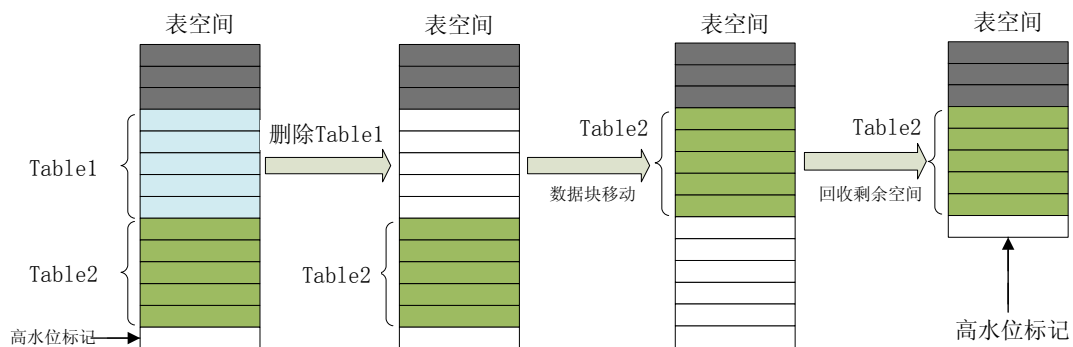


图 5.7 9.7 版本降低高水位标记

只有 9.7 版本创建的自动存储表空间或 DMS 表空间才允许使用这个特性，从 9.7 版本之前迁移过来的表空间目前不支持。

1) 自动存储管理表空间

降低表空间 HWM 的语法如下：

```
>>-ALTER TABLESPACE tablespace-name REDUCE-----+
|-----+--MAX-----+
|-----+--STOP-----+
|-----+--integer-----+
|
|               +-K-----+
|               +-M-----+
|               +-G-----+
|               +-PERCENT-+
|
```

一旦执行该操作，表空间扩展块的迁移操作就会在后台异步进行，如果指定了 **MAX** 选项，DB2 将会尽最大可能降低高水位。

仍以同样的例子模拟一个 HWM：

```
db2inst1@dpf1:~> db2 "create tablespace data_ts1"
db2inst1@dpf1:~> db2 "create table t1 (col1 char(100) ) in data_ts1"
db2inst1@dpf1:~> db2 "call sp_insert(10000) "
db2inst1@dpf1:~> db2 "create table t2 (col1 char(100) ) in data_ts1"
db2inst1@dpf1:~> db2 drop table t1

db2inst1@dpf1:~> db2 list tablespaces show detail
Tablespace ID          = 4
Name                   = DATA_TS1
Total pages            = 8192
Useable pages          = 8160
Used pages             = 160          # 使用的页数是 160 页
Free pages             = 8000
High water mark (pages) = 480          # HWM 是 480 页
```

通过 alter tablespace 的 reduce 选项降低 HWM：

```
db2inst1@dpf1:~> db2 alter tablespace data_ts1 reduce MAX
DB20000I The SQL command completed successfully.

db2inst1@dpf1:~> db2 list tablespaces show detail

Tablespace ID          = 4
Name                   = DATA_TS1
Total pages            = 192
Useable pages          = 160
Used pages             = 160
Free pages             = 0
High water mark (pages) = 160          # HWM 降为 160 页
```

2) DMS 表空间

降低 HWM 的语法是如下：

```
ALTER TABLESPACE tablespace-name LOWER HIGH WATER MARK
```

一旦执行以上操作，表空间扩展块的迁移操作就会在后台异步进行。感兴趣的读者可以自行模拟。

5.3.5 深入 DMS 表空间 ■ ■ ■

每个表空间至少要求需要 5 个数据块，前三个存储表空间元数据，第一个数据块是表空间头，第二个数据块是第一个 SMP 块（SMP=Space Map Page，用来跟踪第一组数据块的使用情况），第三个数据块是对象表，用来定位表空间中定义的对象。对于每个表对象，至少分配两个数据块，一个是这个对象的 extent map（EMP 用来描述这个表对象的 extents），另外一个用来存储数据。

在 DMS 表空间中，一个数据块只能被一个表对象占据，而不能被不同的表共享。表对象由下列独立的对象构成，每个表对象都是独立存储的。

- 表数据对象，存储常规列数据。
- 表索引对象，存储所有的索引数据
- 表长字段对象（Long Field，LF），用来存储表中字段类型为 Long 的数据。
- 两个大对象（LOB）数据对象，如果表中有 1 个或多个大对象，它们存储在这两个独立的表对象中。
 - 一个表对象用来存储 LOB 数据。
 - 另一个表对象用来存储 LOB 数据的描述数据，即元数据。
- 一个 block map 对象用来存储 MDC 数据。
- 一个 XDA 对象，用来存储 XML 数据。

有一道认证考题，用来考查表对象 extent 的分配情况。

```
How many extents are allocated when the following Employee table is
created in the DMS tablespace TBS03 ?
create table employee
( ID smallint not null,
  name varchar(9),
  DEPT smallint check (dept between 10 and 100),
  job char(5) check (job in ('Sales', 'Mgr', 'Clerk')),
  hiredate date,
  salary decimal(7,2),
  Comm decimal (7,2),
  Notes long varchar,
  resume blob,
  constraint yearsal check (year(hiredate) > 1986 or salary < 40500)
) in TBS03;
create unique index inx_u_emp on employee (id);
create index inx_emp_01 on employee (name);
```

这个表中包含 5 个对象，分别是表数据对象、索引对象、Long varchar 长字段对象、两个 LOB 对象（LOB 数据对象和 LOB 数据描述对象）。每个对象在创建时需要两个 extent，第一个是 EMP，另外一个数据 extent。因此这张表创建后需要 $5 \times 2 = 10$ 个 extent。

通过 db2dart demp(dump emp)选项可验证每个对象的 EMP。

```
db2inst1@dpf1:~> db2dart db1 /demp
```

```
Object specific mapping info:
```

```
-----
```

```
DAT extent anchor: 96
```

```
Traversing extent map for object type: 0
```

```
Tablespace ID: 9, Tablespace Seed: 9, Object: 4 EMP page class: 64,  
EMP pool page: 96, # entries: 1
```

```
Page LSN = 00000000109D7897
```

```
Pool relative page #'s :  
128
```

```
INX extent anchor: 352
```

```
Traversing extent map for object type: 1
```

```
Tablespace ID: 9, Tablespace Seed: 9, Object: 4 EMP page class: 65,  
EMP pool page: 352, # entries: 1
```

```
Page LSN = 00000000109DCB47
```

```
Pool relative page #'s :  
384
```

```
XDA extent anchor: 0
```

```
LF extent anchor: 160
```

```
Traversing extent map for object type: 2
```

```
Tablespace ID: 9, Tablespace Seed: 9, Object: 4 EMP page class: 66,  
EMP pool page: 160, # entries: 1
```

```
Page LSN = 00000000109D7E0D
```

```
Pool relative page #'s :  
192
```

```
LOB extent anchor: 224
```

```
Traversing extent map for object type: 3
```

```
Tablespace ID: 9, Tablespace Seed: 9, Object: 4 EMP page class: 67,  
EMP pool page: 224, # entries: 1
```

```
Page LSN = 00000000109D8365
```

```
Pool relative page #'s :  
256
```

```
LOBA extent anchor: 288
```

```
Traversing extent map for object type: 4
```

```
Tablespace ID: 9, Tablespace Seed: 9, Object: 4 EMP page class: 68,  
EMP pool page: 288, # entries: 1
```

```
Page LSN = 00000000109D8425
```

```
Pool relative page #'s :  
320
```

5.4 存储设计最佳实践

数据库的主要目标是快速有效的存储和获取数据，数据一旦入库，以后再想修改存储将会变得异常困难，存储设计的优劣可能直接决定了数据库的性能。DBA 并不是一个人在战斗，特别是在前期规划和上线时，可能涉及很多相关部门协同作业，比如存储、系统等相关部门。本节我们从存储、系统和数据库角度分享一些最佳实践。

对于每个数据库系统来说，以下几个分类需要规划存储空间。

- 实例目录：即实例用户目录，所占空间较小，在创建实例用户时就要规划，一般建议放在共享存储上。
- 数据库目录：存放一些控制文件，占用空间较小，建议放在共享存储上。
- 数据：包括表数据、索引数据，以及临时数据。这部分数据占用的存储空间最大，一般放在磁盘阵列上。
- 活动日志：从性能角度考虑，建议和数据从物理上分离，并放在速度较快的盘阵上。活动日志的初始空间可考虑设置为数据大小的 10-20%，然后通过监控调整它的空间大小。
- 归档日志和数据备份：比较重要，需要规划一定的磁盘空间。
- 每天产生的一些临时数据或文件。

图 5.8 是针对 OLTP 系统比较典型的存储布局，以 AIX 系统为例，展示了从存储层、系统层到数据库层的映射关系。在相对完善的 IT 数据中心，系统、存储和数据库系统是由不同的小组负责的，因此，一个新项目的上线，一般由几个部门协同规划：DBA 配合业务部门预测未来 3~5 年的数据量，并提供数据存储建议，存储部门根据需求划盘、配置阵列，并将逻辑盘映射到主机，系统工程师通过 AIX LVM 卷管理器配置卷组（VG）、逻辑卷（LV）和文件系统，DBA 将其映射到表空间容器中。

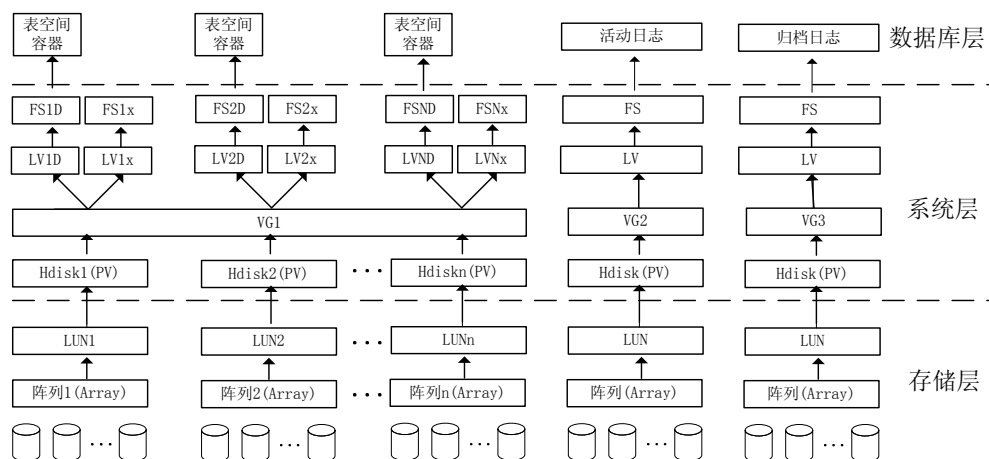


图 5.8 数据库存储布局规划

1. 存储层设计

目前几乎所有的存储服务器都采用 RAID（磁盘冗余阵列）技术保护磁盘数据。每个阵列由一组磁盘构成，阵列的选择需要综合考虑业务特点、容量、性能、冗余性和成本等。RAID 包含 RAID 0、RAID 1、RAID 5、RAID 6 和 RAID 10 几种级别，表 5.1 对这种常见的级别进行了对比。

表 5.1

RAID 级别	描 述	优 点	缺 点	适合的负载
0	数据条带，适用于对性能要求较高，而对数据安全不太在乎的领域	提供最好的性能	不提供任何冗余，任何一个盘损坏都将导致数据丢失	实际生产中用的很少
1	磁盘镜像，适用于存放非常重要数据的场景	性能好	每一块盘都需要镜像，存储成本高	交易系统
5	奇偶校验，RAID 5 是一种存储性能、数据安全和存储成本兼顾的存储解决方案	存储成本相对较低	写入数据的速度比对单个磁盘进行写入操作稍慢，不支持同时两块盘损坏	数据仓库或交易系统
10	Raid 0+1，特别适用于既有大量数据需要存取，同时又对数据安全性要求严格的领域	在提供与 RAID 1 一样的数据安全保障的同时，也提供了与 RAID 0 近似的存储性能	存储成本高	交易系统

RAID 级别的选择需要考虑应用的类型。交易系统(OLTP)对数据的安全性和读写性能要求更高，建议选择 RAID 10，因为 RAID 10 比 RAID 5 的写性能要好很多。对于仓库系统(OLAP)，一般写操作比较小，数据量比较大，占用存储空间大，从节省存储的角度，可考虑用 RAID 5。

对于活动日志，由于写操作比较频繁，而且对日志的可靠性要求比较高，可考虑用 RAID 10。

当确定了 RAID 级别，接下来就要规划存储容量和每个 RAID 组的磁盘个数。在这里，需要重点强调的是每块磁盘的容量并不重要，重要的是磁盘个数。磁盘数量越多，提供的并发 I/O 性能越好。

当在存储层选择了一组磁盘配成了 RAID 后，操作系统如何识别这个存储呢？答案是在存储层将 RAID 划分为一个或几个逻辑盘，逻辑盘通过 LUN（Logical Unit Number，逻辑单元号）来识别，然后将逻辑盘映射到主机，这样主机就可以识别存储盘了。每个逻辑盘跨越 RAID 组的所有盘，建议每个 RAID 只划一个逻辑盘。

2. 系统层设计

在 AIX 系统中, 存储管理是通过 LVM (Logical Volume Management) 管理器实现的, LVM 独立于底层存储, 为用户提供一个灵活的逻辑存储视图。LVM 采用层次化存储结构, 底层是 PV (Physical Volume, 物理卷), 对应存储层的每个 LUN, 在系统中是/dev/hdiskx 设备。一个或多个 PV 形成 VG (Volume Group, 卷组)。每个 VG 可划分一个或多个 LV (Logical Volume, 逻辑卷)。然后又可在 LV 上创建文件系统。这就是 LVM 的层次结构, 关于使用命令, 请参考相关手册。

在 OLTP 系统中, 建议在每个 hdisk(PV)上创建两个 LV, 其中一个用做表空间容器, 另外一个用做数据备份或其他用途。这样既不会造成 I/O 竞争, 又可充分利用存储空间。

3. 数据库层设计

数据库物理设计包括表空间的存储规划和表的设计。每个表空间包含一个或多个容器, 数据在容器之间做均衡, 每个容器可以映射到一个 LV, 即裸设备 (raw device), 也可以映射到 LV 上的文件系统。随着 CIO/DIO 的广泛应用, 文件系统的性能已经可以和裸设备相媲美, 但管理起来更容易些, 建议采用文件作为表空间容器。

对于事务日志, 由于 I/O 操作比较频繁, 为减少 I/O 竞争, 建议将其放在独立的物理存储上。这样既可以提升性能, 同时又减少了由于物理盘损坏导致日志和数据同时丢失的可能性。

对于表空间的设计, 有几个参数值得特别关注: extentsize 和 prefetchsize。

1) extent size 大小设置

在本章的存储模型中, 我们已经知道 extent 是由一组连续的页构成, DB2 按照 extent 在各容器循环写数据。那么 extent 大小设成多大合适? 与存储有什么关系?

在配置 RAID 时, 可以通过存储管理软件指定每块盘的条带 (叫做 Segment 或 Strip)。设置了每块盘的条带大小后, 就可以计算出一个 RAID 的条带大小: $\text{RAID Stripe} = \text{Strip} * n(\text{Disks})$ 。DB2 在表空间的各个容器循环写数据时, 是按照 Extent (块) 大小, 当第一个容器写完一个 Extent 后, 到第二个容器继续写, 以此类推。如果 Extent 块大小和 RAID 条带大小匹配的话, 在写数据时只需一次操作就可写到存储, 这样性能是比较优化的。

举个例子: 一个包含 4d + 1p 的 RAID 5 阵列, 每块磁盘的条带 (strip) 大小为 128KB, 则 $\text{RAID Stripe} = 128\text{KB} * 4 = 512\text{KB}$ 。如果表空间页大小 (Pagesize) 是 16KB, 那么可设置 $\text{Extent Size} = \text{RAID Stripe} = 512\text{KB} / 16\text{KB}(\text{页}) = 32(\text{页})$ 。这样可以在创建表空间时指定 Extent 的大小。

2) prefetch size 大小设置

关于数据库的预取功能, 读者可以将其理解为后台的一组任务在不停地异步读取数据, 尽量使得应用程序在读取数据时发现数据已经存在于缓冲区内存, 以提高系统性能。

而预取的单位则是数据页。由于 DB2 中每个数据对象分配数据的单位为数据块，因此预取的大小最好设置为数据块大小的整数倍。一般来说，建议的设置数据块大小乘以表空间容器再乘以每个容器所对应磁盘的数量。这样每次预取的时候可以在每块物理磁盘上并行读取一个数据块的单位，即：

```
Prefetch size=extent size * (# of containers) * (# of disk per container)
```

其中 extent size 代表块大小，# of containers 表示容器个数，# of disk per container 代表每个容器的磁盘数，DB2 如何知道每个容器有多少个磁盘呢？答案是通过 DB2_PARALLEL_IO 注册变量。

简单地讲，对于 extent 和 prefetch size 的大小，如果 extent 的大小等于一个 Raid 条带的话，这时可把整个 Raid 想象成一块盘，那么 DB2_PARALLEL_IO 参数不必设置；如果 extent 的大小设置为 Raid 中一个磁盘的条带大小，那么可设置 DB2_PARALLEL_IO 等于该阵列中磁盘的个数。推荐的做法是将块大小设置为一个 Raid 条带大小，而不设置 DB2_PARALLEL_IO。

总结一下，存储规划的最佳实践包括：

- 存储规划时，优先考虑磁盘个数而不是容量，让数据尽可能跨越多块盘。随着存储技术的发展，单块磁盘的存储容量越来越大，但每块盘的 I/O 读写速度（IOPS）和吞吐量（MB/s）仍然受到限制。采用 RAID 技术，可以充分发挥多块盘的 I/O 并行处理能力。曾经遇到过一个客户，开始使用的是一组包含 14 块盘、每块 147GB 容量 RAID 组，后来由于某些原因，更换了存储设备，将数据迁移到一组包含 3 块盘、每块容量接近 1TB 的 RAID 组。尽管整体容量比以前还要大，但盘的数量减少到 3 块，整体的 I/O 处理能力和吞吐量降低了很多，性能较以前降低了好的。
- 最多做两层条带。从存储系统到数据库层都提供了条带化机制，存储层通过 RAID 实现条带化，系统层通过卷管理器，而 DB2 通过表空间容器实现条带。从最佳实践的角度，建议在存储层和数据库层做条带。
- 在磁盘个数有限的情况下，建议将数据和索引跨越所有的物理存储，而不是单独存储，这样可以充分利用多块磁盘的并发性能。但从维护的角度，在建表时仍建议为数据、索引、大对象建立独立的表空间。
- 在使用 Raid 阵列时，将 Extent size 的大小设置为 Raid 条带大小，这样可以提高写的性能。而 Prefetch size 参数使用默认的 Automatic 值。
- 表空间的选择建议。对于系统表空间，从 V9 开始默认使用 DMS；对于临时表空间，由于产生的数据是临时的，对空间的需求是动态的，有的时候可能会增长的很大，我们希望它在不用自动的时候能自动缩减，所以推荐 SMS；对于大对象（LOB），由于不能通过 bufferpool 缓存数据，可采用 SMS 或 DMS（File 类型的容器），利用文件系统的缓存机制提升性能；存放数据的表空间，建议使用 DMS 管理。对于小型数据库，建议采用自动存储管理。
- 对于 DMS 容器类型，建议选择文件类型，而不是裸设备（Raw Device）。尽管很多资料都声称裸设备的性能好，但随着 CIO/DIO 的发展，文件类型的性能与裸设备几乎相差无

几，而且文件类型比裸设备管理起来要方便很多，因此建议使用。

- 当创建 DMS 表空间时，如果存储的是普通数据，可指定 `no file system caching` 避免文件缓存；如果存储的是大对象（LOB）数据，可考虑设置 `using file system caching`，充分利用文件缓存（LOB 无法通过 `bufferpool` 缓存）。

在 DB2 数据库中，`bufferpool` 是最重要的内存区，那么应该分配多大的内存给 `bufferpool` 呢？对于交易型系统来说，可以设置 `bufferpool` 的初始大小为机器可用内存的 30%，然后根据负载进行监控，并进行相应调整。

5.5 小结



本章着重介绍了创建数据库的命令，以及一些表空间的管理。深入理解表空间结构对 DBA 有着至关重要的作用。单纯地知道“数据存放在某一个表空间中”，与明白“什么表存放在表空间的哪一个数据页上，索引存放在哪些数据页上”，当严重问题发生时对用户所提供的帮助截然不同。在日常工作中，一个维护得很优秀的表空间，其中数据的连续性往往比没人看管的表空间高很多，这也就意味着 I/O 读写性能的提高。作为一名优秀的 DBA，在系统上线前，一定要对数据库的存储进行很好的规划和设计，否则由于 I/O 引起的性能问题将给日后的维护工作带来很大的隐患。

5.6 判断题



(1) DB2 的表空间分为数据库管理（DMS）与系统管理（SMS）。

T: 正确

F: 错误

(2) 创建数据库时会产生 4 个默认的表空间。

T: 正确

F: 错误

(3) 表空间状态只有两种，分别为在线与离线状态。

T: 正确

F: 错误

(4) 降低表空间高水位只能使用 REORG 命令完成。

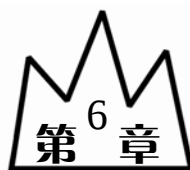
T: 正确

F: 错误

(5) 将所有数据存储在一块单独的硬盘中会得到最佳性能。

T: 正确

F: 错误



数据库连接

数据库已经建好了，但是如何让远程的客户端来访问呢？这就需要在远程客户端配置远程连接，也就是让客户端知道数据库服务器的信息。在 DB2 中，通过两个命令来实现，节点编目和数据库编目。

本章内容安排如下：

- 远程连接概述。
- 节点和数据库编目。
- 经常遇到的连接问题。

6.1 远程连接概述

在操作数据库对象前，必须建立对数据库的连接。连接分为本地连接和远程连接，本地连接是登录到服务器操作，不需要通过网络，采用 IPC 协议（IPC 是进程间通信）。远程连接一般采用 TCP/IP 协议，在客户端需要对节点和数据库进行编目。图 6.1 所示是数据库连接结构图。

在配置连接前，需要确保两边的网络是通的，排除网络故障和防火墙等问题。然后需要分别在 DB2 服务端和客户端进行相应配置。

在 DB2 服务端，主要配置通信协议和 IP 端口号。通信协议是通过 `db2comm` 注册变量设置，IP 端口号是针对实例，每个实例有一个 `SVCENAME` 参数指定端口号。TCP/IP 是从 DB2 9.5 之后唯一支持的通信协议。

在 DB2 客户端，要通过 `Catalog` 命令进行编目。什么是编目（Catalog）呢？通俗地说，就

是将服务端的一些信息在客户端登记一下，比如服务端机器 IP、实例的端口号，以及要连接的数据库名。对 IP 和端口号登记的操作叫做节点编目（Node Catalog），而对数据库信息登记的操作就是数据库编目（Database Catalog）。

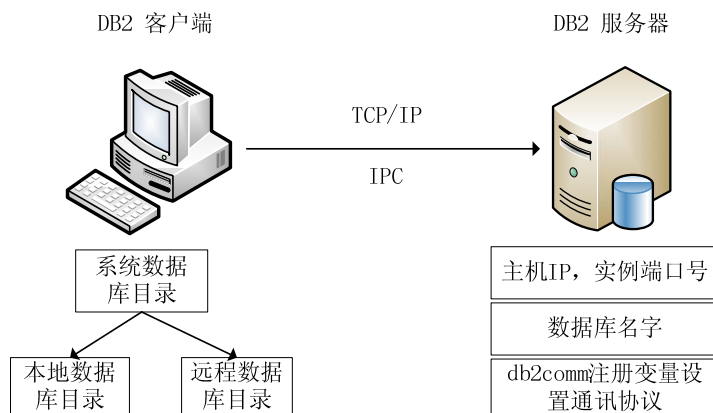


图 6.1 数据库连接结构图

6.2 节点和数据库编目



节点编目（Node Catalog）的语法如下：

```
CATALOG [ADMIN] {TCPIP | TCPIP4 | TCPIP6} NODE node-name REMOTE hostname [SERVER service-name]
```

node-name 是节点名，不能超过 8 位，建议使用服务端主机名作为节点名；hostname 是服务端主机名或 IP，service-name 是服务端实例端口号。

当节点编码完成后，节点编目的信息将存在客户端的实例目录下，<insthome>/sqllib/sqlnodir，通过 list node directory 可查看节点编目信息。

数据库编目（Database Catalog）的语法如下：

```
CATALOG DATABASE database-name [AS alias] [ON path | AT NODE node-name]
```

database_name 是数据库名，不能超过 8 位；AS alias 是数据库别名，ON path 用来编目本地数据库；node-name 是节点编目名，即哪个实例。通过 list db directory 查看数据库编目信息。

数据库编目信息存放在<insthome>/sqllib/sqlldir 系统数据库目录中，该目录既包含远程数据库的编目信息，也包括本地数据库的编目信息。本地数据库在创建时会自动编目，除了在系统数据库目录保存编目信息外，在本地数据库目录也有记录，保存在<dbdir>/<insthome>/NODE0000/sqlldir 目录下。

有些时候，在编目的时候可能会输入错误信息，比如敲错了 IP 或端口号等。这时就需要重新编目，在重新编目前，需要删除之前的编目配置，这就是反编目（uncatalog）。同 catalog 命令类似，uncatalog 可以针对节点和数据库，命令如下：

```
DB2 UNCATALOG DATABASE database-name # 反编目数据库
DB2 UNCATALOG NODE node_name        # 反编目节点
DB2 terminate
```

以下是一个编目的例子：

服务端是 AIX 机器，IP 为 192.168.220.129，实例端口号为 50000，该实例包含 testdb 数据库。客户端是远程的一台 PC，要访问 testdb 远程数据库，是服务端和客户端的配置步骤如下。

（1）首先在 DB2 服务端配置（如果没有配置的话）。

```
# 配置监听协议
inst20@db2server: > db2set db2comm=tcPIP
# 配置实例端口号
inst20@db2server: > db2 update dbm cfg using svcename 50000
# 重启实例才会使得参数修改生效
inst20@db2server: > db2stop
inst20@db2server: > db2start
```

（2）然后在 DB2 客户端编目。

```
# 编目节点
G:\Documents and Settings\db2admin> db2 catalog tcPIP node node_129 remote
192.168.220.129 server 50000
DB20000I  CATALOG TCPIP NODE 命令成功完成。
DB21056W  直到刷新目录高速缓存之后，目录更改才生效。

# 编目数据库
G:\Documents and Settings\db2admin>db2 catalog db testdb as testdb at node node_129
DB20000I  CATALOG DATABASE 命令成功完成。
DB21056W  直到刷新目录高速缓存之后，目录更改才生效。

# terminate 使操作生效
G:\Documents and Settings\db2admin>db2 terminate
DB20000I  TERMINATE 命令成功完成。
```

在客户端，通过 List nod directory 查看节点信息，node_129 就是刚才配置的节点名。

```
G:\Documents and Settings\db2admin>db2 list node directory
节点目录
目录中的条目数 = 1

节点 1 条目:
节点名           = NODE_129
注释             =
目录条目类型     = LOCAL
```

协议	= TCPIP
主机名	= 192.168.220.129
服务名称	= 50000

在客户端，通过 `list db directory` 查看数据库信息，其中一个就是刚才配置的 `testdb` 远程数据库，另外一个是在本地创建的 `sample` 数据库。

```
G:\Documents and Settings\db2admin>db2 list db directory

系统数据库目录
目录中的条目数 = 2

数据库 1 条目:
数据库别名           = TESTDB
数据库名称           = TESTDB
节点名               = NODE_129      # 远程节点名
数据库发行版级别     = d.00
注释                 =
目录条目类型         = 远程           # 远程 (remote) 表示该库位于远程服务器
目录数据库分区号     = -1
备用服务器主机名     =
备用服务器端口号     =

数据库 2 条目:
数据库别名           = SAMPLE
数据库名称           = SAMPLE
本地数据库目录       = G:           # 本地数据库目录位置
数据库发行版级别     = d.00
注释                 =
目录条目类型         = 间接           # 间接 (indirect) 表示本地数据库
目录数据库分区号     = 0
备用服务器主机名     =
备用服务器端口号     =
```

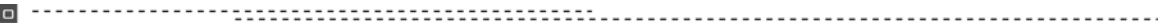
(3) 在客户端进行数据库连接，远程连接需要提供用户名和密码。

```
G:\Documents and Settings\db2admin>db2 connect to testdb user db2inst1 using password

数据库连接信息

数据库服务器         = DB2/LINUX 9.7.1
SQL 授权标识         = DB2INST1
本地数据库别名       = TESTDB
```

6.3 常见的数据库连接问题



1. 通信协议错误问题

```
G:\Documents and Settings\db2admin>db2 connect to testdb user db2inst1 using password
```

SQL30081N 检测到通信错误。正在使用的通信协议: "TCP/IP"。正在使用的通信 API: "SOCKETS"。检测到错误的位置: "192.168.220.129"。检测到错误的通信功能: "connect"。特定于协议的错误代码: "10061"、"*" 和 "*"。 SQLSTATE=08001

当出现该错误后,可依据如下几个步骤分析原因:

①检查网络是否正常。

②是否存在防火墙问题,这也是比较常见的问题。

③检查服务端 db2diag.log 日志,并确认实例端口(svcename 参数)和 db2comm 注册变量是否正确设置。

④检查客户端节点编目和数据库编目是否正确。

2. 数据库连接编码问题

在默认情况下,客户端在访问服务端数据库时,根据客户端操作系统 Locale 获得编码页,如果客户端代码页与服务端数据库代码页不兼容,连接时会出现字符转换问题。这时,可在客户端设置 db2codepage 注册变量。

国内用户最常用的代码页是 1208、1386 和 819。1208 是 UTF-8 编码集,UTF-8 与其他编码一般是兼容的,比如 819 和 1208、1386 和 1208 是兼容的,可以互相转换,而 819 和 1386 则不兼容。

例如,服务端数据库 test1 代码页是 1386。客户端是 Windows,db2codepage 设为 819(英文操作系统),访问时会出现编码转换问题。

```
G:\Documents and Settings\db2admin>db2set db2codepage=819
G:\Documents and Settings\db2admin>db2 terminate
DB20000I  TERMINATE 命令成功完成。

G:\Documents and Settings\db2admin>db2 connect to test1 user db2inst1 using password
SQL0332N  不支持从源代码页 "1386" 到目标代码页 "819" 的字符转换。
SQLSTATE=57017

# 在客户端设置 db2codepage 变量为 1386,再次连接,成功
G:\Documents and Settings\db2admin>db2set db2codepage=1386
G:\Documents and Settings\db2admin>db2 terminate
DB20000I  TERMINATE 命令成功完成。
G:\Documents and Settings\db2admin>db2 connect to test1 user db2inst1 using password
数据库连接信息

数据库服务器          = DB2/LINUX 9.7.1
SQL 授权标识           = DB2INST1
本地数据库别名        = TESTDB
```

3. 实例目录被删或破坏,重新编目

曾经在某烟草客户遇到一个案例,由于机器突然断电导致实例目录下的某些文件被误删,

实例无法启动。由于没有实例文件备份，因此考虑重建实例。重建实例并不会删除数据库，但实例目录中保存的数据库编目信息都会丢失，因此无法进行数据库连接。这时，可考虑对本地数据库进行重新编目，方法如下：

```
db2 catalog db <db_name> on <db_path>
```

其中<db_path>是数据库目录。生产环境中，为保证实例目录文件被删或出现异常，建议对实例目录做备份。

4. 删除数据库目录，重建库失败

这是一个实际案例，某客户要删除并重建数据库，但删除时没有采用 drop database 命令，而是直接删除了数据库目录，当重建库时总是报 SQL1005N 错误：数据库名已经在系统数据库目录或本地数据库目录存在了。

```
db2inst1@dpf1:/data1/archlog> db2 ? sql1005n
SQL1005N The database alias"<name>"already exists in either the local
database directory or system database directory.
```

对于 SQL1005N 错误，有 3 种可能：

- 需要创建的数据库别名已经存在于系统数据库目录和本地数据库目录。需要 drop 数据库。

```
db2 drop database database_alias
```

- 需要创建的数据库别名已经存在系统数据库目录，但不在本地数据库目录。需要 uncatalog 别名。

```
db2 uncatalog database your_database_alias
```

- 需要创建的数据库别名存在本地数据库目录了，但不在系统数据库目录中。需要先编目到系统数据库目录，然后删除数据库。

```
db2 catalog database your_database_alias
db2 drop database your_database_alias
```

对于本例，客户执行了这 3 种方式都未能成功，是否就没有办法了呢？一个小技巧是手工创建一个空的数据库目录 SQL0000X (X 为数据库编号) 欺骗 DB2，然后进行第 3 个方法 catalog db <db_name> on <db_path>, 然后执行 drop database 命令。

这也告诉我们，当删除数据库时，一定要使用 drop database 命令，而不要直接删除数据库目录，因为删目录无法干净地清除编目信息。

6.4 小结



要在远程客户端对服务端数据库进行操作，就需要配置远程连接。远程连接需要在客户端

和服务端分别做一些配置。本章重点介绍了配置命令、原理和出现连接问题的诊断方法。

6.5 判断题



(1) 客户端需要在编目远程节点后才能编目远程数据库。

T: 正确

F: 错误

(2) 在客户端可以使用 `list db directory` 命令查看数据库编目信息。

T: 正确

F: 错误

(3) 远程节点编目保存在 `sqlbdbir` 目录下。

T: 正确

F: 错误

(4) 远程数据库编目与本地数据库编目均保存在 `sqlbdbir` 目录中。

T: 正确

F: 错误

(5) 手工删除数据库目录会隐式自动清空编目信息。

T: 正确

F: 错误



数据库对象

到目前为止，我们已经介绍了 DB2 的安装配置和存储规划，可以说“壳子”已经搭好了，但是对于设计/开发人员来说，更关心的是壳子里面的“肉”，即数据库对象。因为无论是数据建模、应用开发还是数据访问，都是围绕着数据对象展开的。尽管各个数据库产品都存有类似的概念，但是一些对象的概念和用法仍然存在很大差别，比如模式、存储过程等。深入理解各个对象的概念、用法，是学习 DB2 的基础。

本章要介绍的数据库对象如下：

- 模式。
- 表、字段类型、表约束、表的状态。
- 表压缩、表分区。
- 索引。
- 视图。
- 别名。
- 序列。
- 自增字段。
- 大对象。
- 函数。
- 触发器。
- 存储过程。

7.1 模式

数据库中每一个表和索引等对象都有自己唯一的名字。但是，有的时候一个数据库可能被多个用户使用，如果用户 A 把自己的一个表叫做 TEST，而用户 B 也想用同样的名字怎么办？

这种情况下就引入了模式的概念。模式英文名为 **schema**，简单地说，模式是对数据库对象的逻辑分组。可以把模式想象成一个对象的“姓”，而对象的名称则是对象的“名”。如果一个人叫做张三，另一个人叫做李三，那么两个人的名字都是“三”，但是不同的姓则可以将两个人区分开。数据库对象的命名由模式名加上对象名构成（模式名.对象名）。

因此，如果用户 A 想要创建一个 TEST 表，则可以指定这个表的模式为 A，也就是 A.TEST。而 B 则可以用 B.TEST 来命名他的表。而在应用程序中，如果模式没有被显式地指定，那么就会用运行该应用程序的用户名作为默认模式，比如查询：SELECT * FROM TEST，当用户 A 执行的时候就会从 A.TEST 中选择数据；而用户 B 执行的时候就会从 B.TEST 中选择。

DB2 中的模式与 Oracle 模式有根本差别，Oracle 模式名是用户名，DB2 中的模式不一定是用户名，因为 DB2 内部没有用户的概念，连接用户必须是操作系统用户。

DB2 模式支持两种创建方式，一种是通过 create schema 命令创建，这种叫显式创建；另外一种是在创建对象时指定 schema 名，如果该 schema 以前没有创建，那么在创建对象时会自动创建（前提是有限权），这种叫隐式创建。

```
# 显式创建 prod 模式
db2inst1@dpf1:~/db2inst1/NODE0000> db2 "create schema prod "
DB20000I The SQL command completed successfully.

# 隐式创建 test 模式
db2inst1@dpf1:~/db2inst1/NODE0000> db2 "create table test.t1 (col1 char(10) )"
DB20000I The SQL command completed successfully.
```

可通过 syscat.schemata 视图查看数据库创建了哪些模式。

DB2 内部提供了以下几个系统模式用来对相关对象进行分组：

- SYSIBM 模式下的对象存储的是系统数据字典表。
- SYSCAT 模式下的对象是系统视图，可通过这些视图查看各种数据库对象信息。
- SYSIBMADM 是 V9 引入的系系统管理视图模式。
- SYSSTAT 是统计视图模式，该模式下包含 9 个视图，用来为 DB2 优化器提供统计信息。

7.2 表

表是关系数据库的核心，数据的增/删/改/查、应用的数据模型设计都是围绕着表进行的。

表是二维结构，由行和列构成，列也叫字段，每个字段都有名字和数据类型，选择适当的数据库类型是提升性能的前提。DB2 支持的内置数据类型如图 7.1 所示。

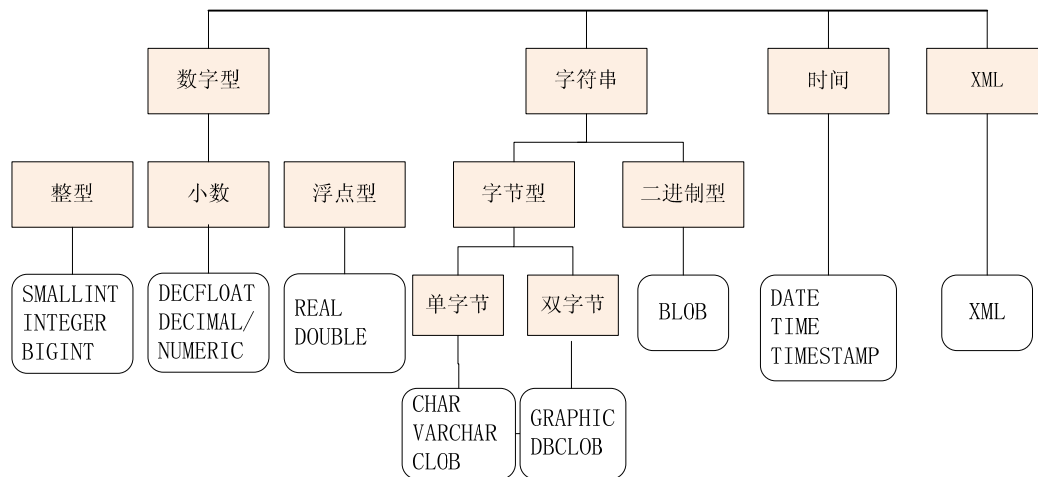


图 7.1 DB2 表数据类型

常用的数据类型包括数字型、字符串型和日期型，XML 是 V9 引入的用来支持 XML 数据的字段类型。

数字型包括小整数型（smallint），整数型（integer）和大整数型（bigint），smallint 支持的数据范围是-32768~32767，占 2 个字节存储空间；integer 最常用，支持的长度可以满足大部分应用需求，占用 4 个字节存储空间，支持的数据范围是-2147483648~2147483647（4 个字节是 32 位，2 的 32 次方）；bigint 支持超大数据，占 8 个字节，支持的数据长度是 19 位。

decimal 用于存储包含小数点的数字，如货币金额等，是我们经常用的数据类型。decimal 以压缩格式存放，表示为 decimal(p,s)，p 表示精度，是数据的总位数，必须小于 32，s 表示小数位，如 51236.324，用 decimal 表示为 decimal(8,3)。如果不指定精度和小数位，默认按 decimal(5,0) 表示。

/float 和 double/float 是浮点类型，表示数据的近似值，一般用于科学计算。

Char 是定长字符类型，最大长度为 255 个字节。varchar 是变长字符类型，最大长度依赖于表空间数据页大小，如果数据页是 4K，则 varchar 字段最长长度是 4005（有 91 个字节用做页头）。varchar 一般用于字段数据长度有很大差异时。

LOB 字段是大对象类型，我们将在 7.8 节重点介绍。

关于日期类型，DB2 支持 3 种：date 日期型、time 时间型和 timestamp 时间戳型。

以下创建了一张 employee 表，id 是主键，int 类型，非空；resume 是 clob 大对象类型；comment 是变长类型。In 指定了数据表空间，index in 指定索引存储的表空间，long in 指定大对象数据存储的表空间。当然，这些表空间需要事先规划并建好。

```
create table employee ( id int not null primary key,
name char(10),
age int,
edulevel int,
resume clob,
comment varchar(100) )
in data_ts
index in index_ts
long in long_ts
```

可通过 describe table 命令查看表字段及数据类型。

```
db2inst1@dpf1:~> db2 describe table db2inst1.employee
```

Column name	Data type schema	Data type name	Column Length	Scale	Nulls
ID	SYSIBM	INTEGER	4	0	No
NAME	SYSIBM	CHARACTER	10	0	Yes
AGE	SYSIBM	INTEGER	4	0	Yes
EDULEVEL	SYSIBM	INTEGER	4	0	Yes
RESUME	SYSIBM	CLOB	1048576	0	Yes
COMMENT	SYSIBM	VARCHAR	100	0	Yes

6 record(s) selected.

如果不知道表名时，可以通过 list tables 命令查看：

通过 list tables for schema <schema_name> 可查看某个模式下的表名、视图和 alias 别名。

通过 list tables for all 可查看所有模式下的表名、视图和 alias 别名。

通过 list tables 可查看以当前连接用户作为模式名下的表名，视图等。

或者通过 SYSCAT.TABLES 系统视图查看表的定义、所属的表空间等。

```
db2inst1@dpf1:~> db2 "select substr(tabschema,1,32) as tabschema,
substr(tabname,1,32) as tabname, tbspaceid from syscat.tables where
tabschema='INST20' "
```

TABSCHEMA	TABNAME	TBSPACEID
DB2INST1	DEPARTMENT	3
DB2INST1	EMPLOYEE	3
DB2INST1	T10	3

3 record(s) selected.

7.2.1 表约束 ■ ■ ■

表约束是在数据库级别限定数据是否可以插入到表中。表约束一般在数据模型设计时，结合业务需求进行定义。

DB2 支持的表约束包括非空约束 (not null)、唯一性约束 (unique)、主键约束 (primary key)、外键约束 (foreign key) 和检查约束 (check)。

- 非空约束：定义了非空约束的字段不能插入 NULL 值。
- 唯一性约束：定义了唯一性约束的字段，其数据必须是唯一的，不能有重复。
- 主键约束和外键约束：定义两个表之间的参考完整性关系，引用的表叫子表，被引用的表叫主表，主表上被引用的字段叫主键，子表上引用的字段叫外键，外键字段的内容在主表中必须存在，否则会报约束违例。
- 检查约束：指定某个字段内容必须在约束的范围内，如定义 age（年龄）字段不能超过 200，sex（性别）只能是 ‘M’ 或 ‘F’。

以下通过部门表 (department) 和员工表 (employee) 两张表的定义演示了这几种约束。

```
CREATE TABLE DEPARTMENT                                -- 创建部门表
    (DEPT CHAR (3) NOT NULL,                             -- NOT NULL 约束
     DEPTNAME CHAR(20) NOT NULL,
     CONSTRAINT DEPT_NAME
     UNIQUE (DEPTNAME),                                   -- 在 DEPTNAME 字段创建唯一性约束
     PRIMARY KEY (DEPT) )                                -- 在 DEPT 字段创建主键约束

CREATE TABLE EMPLOYEE                                  -- 创建员工表
    (EMPNO CHAR(6) NOT NULL,
     NAME CHAR(30),
     AGE INT,
     WKDEPT CHAR(3) NOT NULL,
     CONSTRAINT DEPT
     FOREIGN KEY (WKDEPT)                                -- 在 WKDEPT 字段创建外键约束
     REFERENCES DEPARTMENT (DEPT)                       -- 外键引用 DEPARTMENT 表的 DEPT 字段
     ON DELETE CASCADE ,                                 -- 定义主外键约束的删除规则
     CONSTRAINT AGE CHECK( AGE<200 ) )                   -- 在 AGE 字段定义检查约束，年龄不能超过 200
```

Department 表上有两个字段，dept 字段是主键，deptname 字段是唯一键，均为非空；Employee 表有 4 个字段，empno 是非空约束，wkdept 字段是外键，引用 department 表的 dept 字段，age 上有检查约束。

其中在定义外键约束时指定了 ON DELETE CASCADE 规则。这表示迭代删除，即删除主表数据时，如果子表有数据引用它，那么将子表引用的数据行也删除。

可以在建表时指定约束，也可以在建表后通过 alter table 选项添加或删除约束。以下例子仅供参考，如需查看详细语法，请参考 DB2 信息中心。

```
--添加主键约束
db2inst1@dpf1:~> db2 "alter table t1 add primary key(c4)"
--添加外键约束
db2inst1@dpf1:~> db2 "alter table albums primary key (itemno) add constraint fkartno
foreign key (artno) references artists on delete cascade"

--添加唯一性约束
db2inst1@dpf1:~> db2 "alter table t1 add unique(c4) "

--添加检查约束
db2inst1@dpf1:~> db2 "alter table stock add constraint cctype check (type in ('D',
'C', 'R'))"
```

可通过系统视图查看数据库中定义的约束。

```
--通过 syscat.references 视图查看数据库中定义的所有参考完整性约束
db2inst1@dpf1:~> db2 "select substr(constname,1,18) as constname,
substr(tabname,1,18) as tabname, substr(fk_colnames,1,14) as fk_colnames,
substr(reftabschema,1,14) as ref_schema ,substr(reftabname,1,14) as ref_tabname,
substr(pk_colnames,1,14) as k_colnames ,deleterule from syscat.references where
tabschema= user "

--通过 syscat.checks 视图查看数据库中定义的检查约束
db2inst1@dpf1:~> db2 "select tabname, substr(text,1,500) from syscat.checks"

--删除约束
db2inst1@dpf1:~> db2 "alter table empl drop constraint <const_name>"
```

需要注意的是，尽管 DB2 提供了这几种约束的支持，但是在设计时一定要考虑这些约束在数据库层面实现还是在应用程序层实现。在数据库层实现的好处是减少了很多开发工作量，对应用透明。但缺点是如果业务逻辑复杂的话，放在应用层实现逻辑更清晰，而且也会增加运维的负担，比如在使用 LOAD 进行数据迁移时，如果表上有约束需要执行特别检查。

7.2.2 表状态 ■ ■ ■

前面我们介绍了表空间的状态，当表空间状态异常时，数据是无法访问的。同理，在执行一些操作时，表也可能处于某张状态，比如 Load 操作，可能导致表处于 Load pending、not restartable 等，将在第 8 章重点介绍。

还有，当对表结构进行更改时，也可能导致表状态异常。比如，以下操作可能会导致表处于 reorg-pending 状态。

```
alter table <tablename> alter <colname> set data type <new data type>
alter table <tablename> alter <colname> set not null
alter table <tablename> drop column <colname>
...
```

出现 reorg pending 的根源是当表结构变化后影响了数据行中的数据格式，这时需要对表做 reorg。可能的错误号是：

```
SQL0668N Operation not allowed for reason code "7" on table "SDD.ST_INCRE008".
SQLSTATE=57016
```

```
SQL20054N The table "<table-name>" is in an invalid state for the operation. Reason
code="7".
```

关于 Reorg 的更详细说明请参看第 10 章

7.2.3 表压缩 ■ ■ ■

数据库高级表压缩（Table Compression）是 DB2 9 版本新增的非常强大的功能，其原理是为表、索引、大对象等数据库对象提供基于字典的压缩功能。经过表压缩，相同的数据占用很少的数据页空间，不仅大大节省磁盘存储空间，而且减少了读取 I/O 和内存的使用。

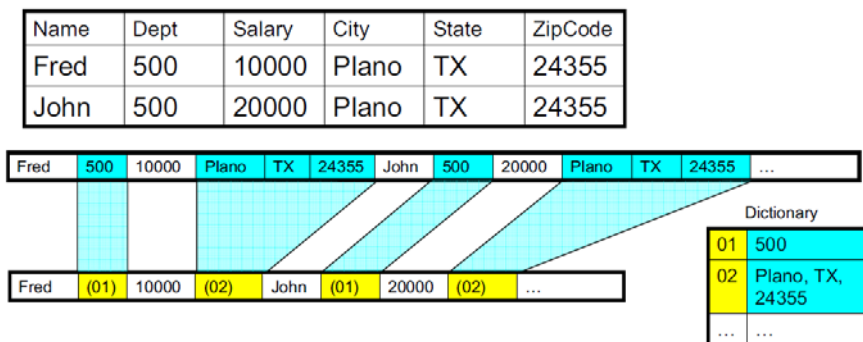


图 7.2 表压缩原理

表压缩采用的是基于字典的压缩算法，而不是采用类似 WinZip、WinRAR 的复杂算法。拿图 7.2 为例，该表中包含一些数据，其中频率高的数据是“500”和“Plano TX 24355”。在压缩时，DB2 首先创建一个压缩字典表，对频率高的数据用简单的字符表示。然后基于这个压缩字典对表数据进行替换，这样就大大节省了存储空间。如果数据库对象包含很多重复值，压缩效果更明显，甚至可以达到 80% 的压缩比，即 100GB 数据压缩后只有 20GB。

表压缩的命令很简单: `Alter table <tablename> compress yes.`

以下举例说明：

(1) 压缩前先向 employee 插入 100 万行数据，观察表数据大小是 95488KB，即大概 95MB 大小。

```
inst20@db2server:~> db2 "select substr(tabname,1,32) as tabname,
DATA_OBJECT_L_SIZE,DATA_OBJECT_P_SIZE, DICTIONARY_SIZE from sysibmadm.admintabinfo
where tabname='EMPLOYEE' "
```

TABNAME	DATA_OBJECT_L_SIZE	DATA_OBJECT_P_SIZE	DICTIONARY_SIZE
EMPLOYEE	95488	95488	0

1 record(s) selected.

(2) 然后对表数据进行压缩，并进行 Reorg 重组

```
inst20@db2server:~> db2 alter table db2inst1.employee compress yes
DB20000I The SQL command completed successfully.

inst20@db2server:~> db2 reorg table db2inst1.employee
DB20000I The REORG command completed successfully.
```

(3) 压缩后，检查表大小，发现只有 31744KB，即 31MB。压缩比为 $(1-31/95)*100=67\%$ 。压缩字典大小是 64640 个字节，即 64KB。

```
inst20@db2server:~> db2 "select substr(tabname,1,32) as tabname,
DATA_OBJECT_L_SIZE,DATA_OBJECT_P_SIZE, DICTIONARY_SIZE from sysibmadm.admintabinfo
where tabname='EMPLOYEE' "
```

TABNAME	DATA_OBJECT_L_SIZE	DATA_OBJECT_P_SIZE	DICTIONARY_SIZE
EMPLOYEE	31744	31744	64640

1 record(s) selected.

表压缩虽然减少存储空间，也节省了 I/O 和内存占用，但压缩和解压需要一定的 CPU 资源，因此在 CPU 利用率较高的系统中不建议使用。通常情况下，表压缩特性在数据量较大的仓库或经营分析系统中应用更广泛，我们将在《DB2 数据库高级管理》一书中详细介绍。

7.2.4 表分区 ■ ■ ■

表分区 (Table Partition)，也叫范围分区，是 DB2 9 版本引入的功能强大的特性。表分区是一种新的数据组织模式，在这种模式中，数据将以一个或多个表列的值为依据，分割到多个称为数据分区 (或范围) 的存储对象中，每一个分区数据独立存储。实际上，DB2 在内部实现上把每个分区作为一个独立的表，但对应用来说是透明的，应用看到的还是一张表。

举例来说，一张事实表 (大表) 存储了 5 年的数据，查询时一般统计最近几个月的数据，如果采用传统的表结构存储，那么仍然要访问 5 年的数据才能获得结果。如果按照月份对表进行分区，那么访问时，DB2 优化器会过滤其他月份，而只计算这几个月的分区数据。因此会带来性能上的巨大提升。另外，采用分区表对数据的归档管理变得简单。

表分区一般用于数据量较大的数据仓库或分析系统中，由于篇幅关系，我们不做进一步的案例演示，感兴趣的朋友请关注《DB2 数据库高级管理》一书。

7.3 索引

对于很多用户来说，当问起他们 DB2 中对提升应用程序性能最有效的武器是什么，得到的回答很多都是“加索引”。这个答案理论上没有错，很多时候合适的索引可以成千上万倍地提高系统的性能。但是，为什么加上了索引，系统的性能就奇迹般地提高了？怎么知道一个索引是不是最合适？索引是不是越多越好？本节我们将为您揭开这些谜团。

从数据结构上，DB2 中的索引采用的是 B+树结构。学过《数据结构与算法》课程的读者，对 B+树一定不会陌生，如果没学过也不要紧，我们在这里简单地介绍一下这种数据结构。

什么是 B+树？它的好处是什么？我们先从二叉搜索树谈起。抛开数据库系统不谈，当想要提高一个算法的效率时，很大的一个方面就是尽量在访问判定数据之前就将部分数据排除在外，减少搜索范围，二叉搜索树就是这样一种算法，树中的每一个节点，其左子树一定比右子树小，如图 7.3 所示。

在图 7.3 中，假如我们要搜索 7 这个数字。那么当进入树的时候，首先判断 7 与 8 的关系，如果 7 比 8 小，那么继续搜索左子树。当发现左子树为 3，而 7 大于 3，则搜索 3 的右子树，直到找到该节点，或者遇到叶节点。

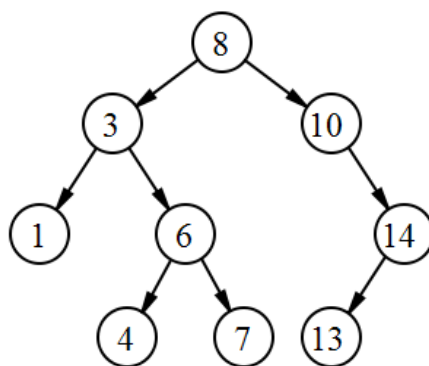


图 7.3 二叉搜索树

通过这个方式，我们可以“越过”绝大部分节点，只搜索树中有限的几个节点，就可以判断出一个数值是否在这个集合之内。

二叉搜索树在数据检索方面与普通的线性检索有着无可比拟的优势。但是，对于数据库系统来说，二叉搜索树又有着自己的不足。其中最重要的一点就是，二叉搜索树假设所有的数据都存在内存中，也就是说，访问每一个节点的效率都是等价的。但是在真实生活当中，操作系统的物理内存是有限的，用户的数据量可能是近乎无限大的，那么不可能将所有需要检索的数据都放入内存。因此，在磁盘上的一部分数据，其访问效率将会慢于内存检索几万倍。这样，我们必须有一种方式能够尽量减少磁盘的访问，而不是像二叉搜索树一样可能会使用到任何一个节点。于是 B+树便诞生了。实际上 B+树是 B 树的一个变形，我们在这里不讨论 B 树，只关注 B+树。

相比起二叉搜索树，B+树的每一个节点存放着超过一组数据，如图 7.4 所示。所有的最终数据只存放于叶节点，中间的节点存放的该其子节点的上限或者下限。

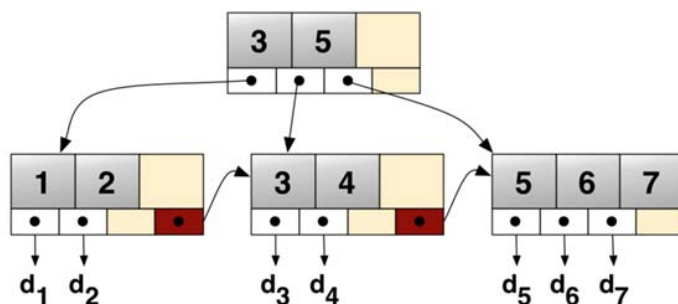


图 7.4 B+树结构

在图 7.4 中，有 1,2,3,4,5,6,7 几个数据。在这个 B+ 树中，每一个节点存放 3 组数据，而根节点下存在 3 个指针，分别指向 3 个存放着真实数据的叶节点。

中间节点（也就是根节点）包含了两个数值：3 和 5。所有比 3 小的数据会存放在 3 左边指针所指向的叶节点中，介于 3 与 5 之间的存放于 3 与 5 之间的指针所指向的叶节点中，而 5 及 5 以上的数据存放在 5 右边的指针所指向的叶节点中。每一个叶节点的末尾都有一个指针，指向下一个叶节点的位置。

通过这种数据结构，当想要访问一个数据，譬如 4 的时候，首先判断这个数值与根节点中数据的关系。当我们发现 4 大于 3 且小于 5 时，就会使用 3 与 5 之间的指针，得到叶节点，然后在这个叶节点中判断 4 是否存在。有人可能会说，这个方法不如二叉搜索树，因为在检索一个节点时，对于其中的多个数据，需要使用线性搜索方式。这样理解没错，但是这个结构有一个二叉搜索树无法比拟的优势，就是每一个节点可以包含多个数据，而节点的大小与可包含数据的数量是不固定的。当使用磁盘时，我们可以定义每一个节点的大小为一个或多个数据页，这样每次我们只需要调用一个 I/O 命令，就可以将若干个相近（对于叶节点），或者可搜索（中间节点）的数据装载入内存。这种 B+ 树结构便是索引的雏形。

可能读者还是不理解，怎么使用这种 B+ 树快速访问数据呢？答案便在于叶节点中每一个数据所包含的信息。

一般来说，搜索树除了为了判定某些数值是否在数据集合之内，更重要的一个功能便是通过该集合中的一个数值访问更多的一些东西。譬如一个仓库管理员，手里中有一个库存序列号的名单。这个库存序列号除了可以判定一个给定的序列号是否在货仓之内处，还能够通过该序列号下面的一系列信息，快速地找到该货物存在哪个货柜、哪个货架之上。而数据库的索引便是同样的原理。比如用户在一个数字列上建立索引，然后想要查找包含 3 的行。那么通过索引，数据库首先可以判断出 3 是否存在，然后还需要某种机制，能够把 3 所对应的所有行都找到，然后在表中取得这些行的其他列信息并返回给用户。

怎样将索引中的数据对应到相应的行？在仓库管理员的角度，他的名单中一定有序列号货物所对应的货柜和货架。那么数据库同样也有类似的东西，需要通过某种数据结构找到对应

一个索引数值的所有数据，包括它们所在的数据页与在数据页中的位置。这个数据结构叫做 Record ID，简称 RID。

每一个 RID 包含着数据页与 slot（也就是数据页中的逻辑位置，每一个数据页中都有一组列表将 slot 转换成真实的偏移地址）。通过访问一个 RID，数据库就可以知道一个给定的 RID 所对应唯一的行。

此时索引的结构就相对清晰了。通过 B+树可以将键值（搜索的数据）迅速定位到叶节点，而在叶节点中，每一个键值对应着一组 RID，每一个 RID 对应表中的一行数据。

现在让我们来举一个例子。

假设我们有一个表 User，如表 7.1 所示。

表 7.1

User ID	Name	Salary	Phone
3	Tom	8500	123456789
80	Jerry	9000	567891234
5	Bob	4800	579812456
7	Nancy	6500	564879124
100	Chad	12000	246789453
38	Tod	7100	423897534
4	Tom	25000	123478942
28	Edward	6000	475789945
75	Nancy	8500	456785222

如果在表 7.1 中我们有一个在 User I 上上面的索引，则可以想象为图 7.5 所示的样式。

通过这个索引，假设用户要查找一个 User ID 为 28 的人的电话，则需要在索引的根节点中找到中间的指针，然后读取叶节点找到 28，接着根据 28 所包含的 RID 找到对应的页面与偏移位置，快速地得到该行用户数据。

想象一下，如果这个表包含 100 万行，在没有索引的情况下，要查找 User ID=28，需要扫描整个表。但是当索引存在时，只需要读取几个索引数据页就可以找到 RID，然后访问相应的数据页面读取数据，大大减少了 I/O 读次数。

那么一个合适的索引是什么概念呢？当判断一个索引是否应该（或者能否）被使用时，一方面应该看这个 B+树所包含的数值，譬如，对应图 7.5 中的表，如果用户想要通过 Name 查找，那么使用 UserID 定义的索引则毫无意义，因为该索引中每一个节点为 User ID，可考虑在 Name 列创建另外一个索引。

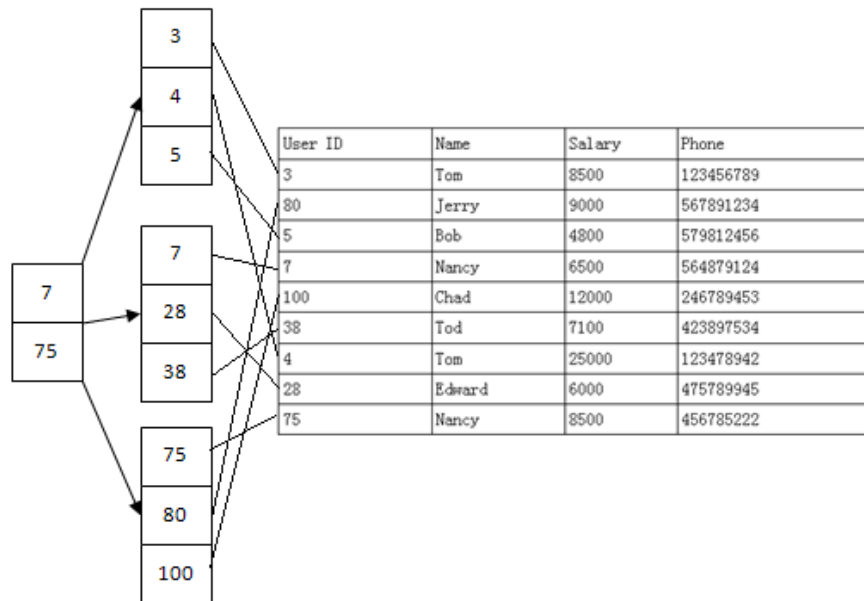


图 7.5 User ID 索引的逻辑结构

如何判断一个索引是否能够过滤大部分数据呢？以表 7.2 为例，同样在 User ID 上面创建索引，该索引中仅包含一个键值，而该键值中包含 4 个 RID 指向这个表中所有的行。这时 DB2 可能不会选择索引，因为索引的代价比表扫描还要大。

表 7.2

User ID	Name	Salary
10	Bob	12345
10	Tom	4555
10	Jerry	8000
10	Tommy	4600

也就是说，用户定义了索引后，并不能够保证这个索引每次访问都会被使用。DB2 的优化器会判定使用索引是否能够提高效率，如果判定索引的效率不如扫描全表，DB2 会使用全表扫描而不是索引扫描（这部分会在优化器章节再次阐述）。

索引键可以包含多个字段，这种索引叫做复合索引。在复合索引中，首先按照第一个键值排序，如果第一个键相同而第二个键不同，则按照第二个键排序，以此类推。

最后，如果一个索引包含多个列，数据结构是什么样的？

譬如表 7.3 所示的二手车的数据。

表 7.3

Make	Model	Year	Color	VIN
Toyota	Camry	1997	Red	12345789
Chrysler	Intrapid	2001	Green	14578942
Toyota	Corolla	2011	Red	43573831
Toyota	Corolla	2010	Gray	45673123
Honda	Civic	1997	Silver	67865432
Chrysler	Intrapid	2011	Silver	13789423
Honda	Accord	2010	Yellow	13786954
Honda	Prelude	2009	Red	34787624
Honda	Accord	2008	Black	15678612

如果在 Make 与 Model 字段创建索引，那么索引的结构如图 7.6 所示。首先按照 Make（制造商）排序，当 Make 相同时，再按照 Model 排序，如 Honda Civic 就排在 Honda Accord 之后。

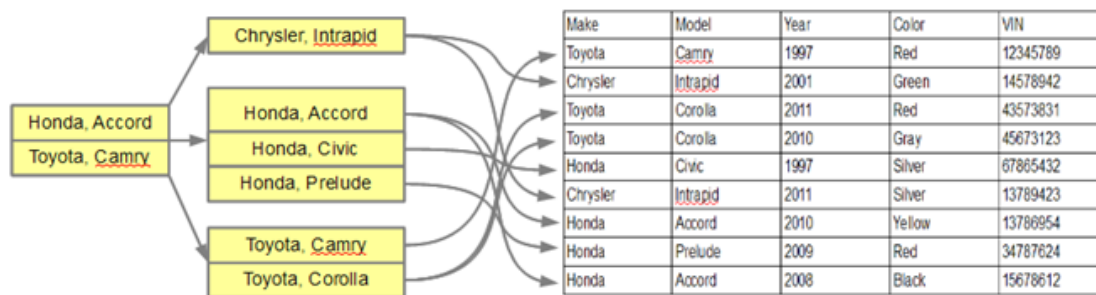


图 7.6 二手车数据表索引

需要注意的是，如果在 (A,B) 字段上创建一个索引，当查询条件包含 A，或者 A 与 B，那么可以有效地使用索引。但是如果条件中仅仅包含 B，那么索引并不能带来效率上的提升。在以上汽车的例子中，如果用户想查询 Model= 的行，DB2 并不会使用在 (Make,Model) 上创建的索引。因此，当创建索引时一定要特别注意索引中的列顺序，在 (Make,Model) 和在 (Model,Make) 列上创建的索引完全不同。

下面我们看一下索引创建的语法：

```
(1) create index idx_album_itemno on albums (itemno)
(2) create unique index dba.empno on dba.employee (empno asc)
(3) create index item on stock (itemno) cluster
(4) create unique index empidx on employee (empno) include (lastname, firstname)
(5) create index name on employee(firstname, lastname)
```

第 (1) 条语句在 albums 表的 itemno 字段创建了一个普通索引。

第（2）条语句在 `employee` 表的 `empno` 字段创建了唯一性索引。Unique 表示唯一性索引，当在表上创建了主键或唯一键时，会自动创建唯一性索引。

第（3）条语句指定了 `Cluster` 选项，`Cluster` 表示集群或簇的意思，目的是尽量保持数据页的物理顺序和索引键顺序保持一致。默认情况下，表数据的物理组织是无序的，数据可能杂乱无章地分散在很多数据页中，这样当按照某个范围查询一组数据的话，就要根据索引叶子节点的 `RID` 映射到很多页中获取数据，如果数据量很大的话，需要的 I/O 是很大的。如果这些数据在物理上是连续的，那么叶子节点的 `RID` 可能指向相邻的数据页，这时数据的获取就更快捷，需要的页数也更少，效果更高。这就是 `Cluster` 索引的目的。

图 7.7 所示是 `Cluster` 索引与 `Non-Cluster` 索引的对比，当根据某个范围获取一组数据的时候，`Cluster` 索引可能带来 I/O 效率上的巨大提升。

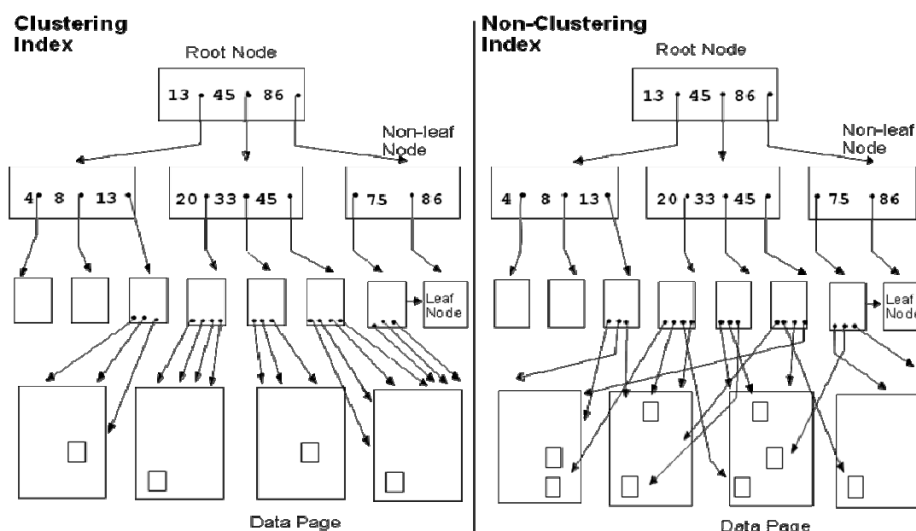


图 7.7 Cluster 索引与 Non-Cluster 索引

注意：对数据的增删改可能会造成数据物理顺序无法和索引键保持一致，可通过 `reorg` 对表和索引进行重组。

第（4）条语句指定了 `include` 选项。有些读者可能不熟悉该选项，其实在某些场景，`include` 对效率的提升也很明显。举例来说，对于 `employee` 表，最经常访问的字段是员工号和姓名，如果在员工号上创建索引，当根据员工号查询姓名时，DB2 会从索引中找到 `RID`，然后从表中获取姓名，这至少需要两次 I/O（索引页和数据页 I/O）。当使用 `include` 选项时，`include` 字段会附加在索引键所指向的每一个 `RID` 上，这样就可以从索引中直接获取 `include` 字段，而不必读取数据页。在本例中，姓名可以直接从索引中获得，而不需要通过 `RID` 读取表，但如果还想获取其他字段，就只能访问数据页了。注意：`include` 也叫 `index-only` 访问，只有唯一性索引才可以使用该选项。

第（5）条语句创建了一个复合索引。

可以通过 `describe indexes for table <tablename> show detail` 命令查看一个表是否有索引、在哪些字段上建的索引，以及索引的类型。也可以通过 `SYSCAT.INDEXES` 字典表查看索引信息。

7.4 视图

视图（View）是一个逻辑表，可以把视图想象成一个复杂 SQL 语句的别名，当查询视图的时候，就是执行其中的 SQL 语句，也就是查询基础表。视图在数据库中是没有存储空间的，基础表的数据更改了，视图的结果也会随之改变。视图的好处体现在：简化用户数据查询和处理操作，隐藏数据复杂性，通过限制基表的行或列提供安全权限控制。但在应用开发设计时，不建议使用过大的视图，特别是嵌套视图，当遇到 SQL 性能问题时，包含大量视图的 SQL 调优是个梦魇。

视图的语法如下：

```
CREATE VIEW <视图名>(<列名组>) AS <子查询>
```

如果视图查询的效率低，应该优化其中的 SQL 语句。对实时性不高的报表或分析型系统，如果实在没有优化的空间，可以考虑做成物化视图（MQT），相当于把视图中的 SQL 语句的查询结果以一个物理表的形式存储起来，即视图的物理化。在仓库或分析类系统中，MQT 是解决性能问题的利器，相关原理和实际案例将在《DB2 数据库高级管理》一书中介绍。。

可通过 `SYSCAT.VIEWS` 系统视图查看视图的信息，包括视图的定义。

DB2 提供了几类非常重要的系统视图：

- SYSCAT 系统视图。
- SYSIBMADM 监控管理视图。
- SYSSTAT 统计视图。

7.5 昵称

昵称（Nickname）是 DB2 联邦数据库中的概念，在正常情况下，在一个数据库中访问另外一个数据库的表是不允许的，但配置了联邦后，在本地为远程表创建一个昵称，就可以访问远端数据库了，我们将在《DB2 数据库高级管理》一书详细介绍。

7.6 序列（Sequence）

对于一些交易系统，经常需要在表中记录操作日志、交易流水等信息，通过日志号、交易

流水号等标识每行记录，那么如何保证记录的唯一性呢？一种通用的设计方法是为每行记录存取一个计数器（Counter），当某个应用需要插入记录时首先获取表中最大的计数器，加 1，然后提交。不幸的是，这种方法在处理多用户并发下存在性能问题。在数据库级别，DB2 提供了如下几种方法解决字段值唯一问题：

- Generate_unique 函数。
- Identity 标识字段。
- Sequence 对象。

Generate_unique 函数是 DB2 最早引入的在表中生成唯一值的方法，该函数的返回值是国际标准时间(UTC)生成的当前时间戳加上当前数据库分区号，包含 13 个字节的数据串(char(13) for bit data)。但需要注意的是，当前时间戳是根据当前系统时钟生成的，当调整了系统日期，可能会出现重复值的情况。

以下是一个简单的例子：

```
CREATE TABLE customers( CUSTNO CHAR(13) FOR BIT DATA, CUSTNAME VARCHAR(16) ) #
创建一张表

INSERT INTO customers VALUES (GENERATE_UNIQUE(),'zhangsan'),(GENERATE_UNIQUE(),'lisi')

SELECT * FROM customers
CUSTNO          CUSTNAME
-----
x'20101021143258122941000000' zhangsan
x'20101021143258122968000000' lisi

SELECT timestamp(custno) as custno, custname FROM customers
CUSTNO          CUSTNAME
-----
2010-10-21-14.32.58.122941 zhangsan
2010-10-21-14.32.58.122968 lisi
```

Sequence 是 DB2 的一个对象，按照一定规则产生自动增加的数字序列。这个序列一般用做主键（因为不会重复），没有其他意义。通过 NEXTVAL FOR <seqname>表达式产生序列的下一个值，通过 PREVVAL 返回序列的前一个值。Sequence 的语法如下：

```
CREATE SEQUENCE db2inst1.my_seq
AS bigint
START WITH 1
INCREMENT BY 1
NO MAXVALUE
CYCLE
CACHE 100;
```

Sequence 的使用相对简单，值得注意的是 cache 选项的使用，cache 的目的是将一组连续值先计算出来，缓存到内存中，每次使用时从内存里直接获取即可，当这组值用完时，再分配一

组。这些缓存值对所有连接是共享的，如果所有连接断开，则缓存的值将丢失，下次启动库的时候会从下一组值分配，因此如果 `cache` 不等于 1 的时候，可能会造成序列的不连续。

`Cache` 选项会对系统的性能产生重大影响，笔者在某银行进行的多并发压力测试中，发现其中一张表的 `insert` 性能比较差，通过 `db2batch` 发现每个 `insert` 差不多要花费 9 毫秒，后来检查发现，该表使用了 `sequence`，但 `cache` 值是默认值 1，将其调整为 100 后，`insert` 的时间降为 3 毫秒。但 `cache` 也不是越大越好，因为每次数据库重启都会导致缓存数据丢失，使得序列值很快达到 `MAXVALUE`。

以下是 `Sequence` 对象的一个简单演示：

```
# 创建 sequence, start with 表示序列从 1 开始, incremet by 1 表示序列增量, cache 100 表示缓存了 100 个值
db2inst1@dpf1:~/db2inst1/NODE0000> db2 "create sequence my_seq start with 1
increment by 1 cache 100"

# 创建一张表, 在表中插入数据, 表的 ID 字段是通过 sequence 产生的
db2inst1@dpf1:~/db2inst1/NODE0000> db2 "create table t1 (id int, name char(20) )"
db2inst1@dpf1:~/db2inst1/NODE0000> db2 "insert into t1 values( nextval for my_seq,
'chendong' ) "
db2inst1@dpf1:~/db2inst1/NODE0000> db2 "insert into t1 values( nextval for my_seq,
'zhangping' ) "
db2inst1@dpf1:~/db2inst1/NODE0000> db2 "select * from t1"
```

ID	NAME
1	chendong
2	zhangping

可通过 `SYSCAT.SEQUENCES` 视图查看序列的定义。

7.7 自增字段

`Sequence` 与 `Identity` 的作用差不多，都用来生成自增数字序列。`Identity` 的定义有两种方法，一种是 `generated always as identity`，表示自增列值由系统自动产生，用户不能指定；另外一种通过 `generated by default as identity`，是指当用户提供了值时，自增字段的值按用户提供值插入，当没有提供时由系统自动生成。

```
CREATE TABLE customers1(
CUSTNO BIGINT NOT NULL GENERATED ALWAYS AS IDENTITY(
    START WITH 500,          -- 自增主键的起始值
    INCREMENT BY 1,          -- 每次主键生成的增量
    MINVALUE 500,            -- 最小值
    MAXVALUE 9223372036854775807, -- 主键最大值
    NO CYCLE,                -- 是否可循环
    NO CACHE,                -- 是否缓存
```

```

NO ORDER
),
CUSTNAME VARCHAR(16) )
INSERT INTO customers1(custname) VALUES('xumingwei')          ---允许插入
INSERT INTO customers1(custno, custname) VALUES( 502, 'xumingwei')---不允许插入

CREATE TABLE customers2(
CUSTNO BIGINT NOT NULL GENERATED BY DEFAULT AS IDENTITY(
        START WITH 500,
        INCREMENT BY 1,
MINVALUE 500,
MAXVALUE 9223372036854775807,
NO CYCLE,
NO CACHE,
NO ORDER
),
CUSTNAME VARCHAR(16) )
INSERT INTO customers2(custname) VALUES('xumingwei')          ---允许插入
INSERT INTO customers2(custno, custname) VALUES( 502, 'xumingwei')---允许插入

```

可通过 SYSCAT.COLUMNS 视图找出数据库中所有带自增列的表和字段，identity 字段用来标识是否为自增列，generated 字段值为 ‘A’ 表示 generated always 类型，‘D’ 表示 generated by default 类型。

```

SELECT substr(tabname,1,20) as tabname, substr(colname,1,20) as colname, identity,
generated FROM SYSCAT.COLUMNS WHERE IDENTITY='Y'

```

TABNAME	COLNAME	IDENTITY	GENERATED
CUSTOMERS2	CUSTNO	Y	D
CUSTOMERS1	CUSTNO	Y	A

Sequence 和 Identity 的差别在于：Sequence 是数据库系统中的一个对象，可以在整个数据库中使用，和表没有任何关系；而 Identity 必须指定在表中某一列，作用范围就是这个表。在应用设计中，要根据情况选择使用哪一种。

在以下情况下中，Identity 很有用：

- 表中只有一列需要自动生成的值。
- 每一行都需要独立的值。
- 用自动生成器来产生表的主键。
- 生成新值的进程与对表的插入操作紧密联系，无论插入操作是如何发生的。

在以下情况下中，Sequence 很有用：

- 要将从一个序列生成的值存储到多个表中。
- 每个表有多个列需要自动生成的值（可能通过使用同一个序列或多个序列为每一行生成多个值）。
- 生成新值的进程与任何对表的引用无关。

7.8 大对象 (LOB)

尽管常规数据类型能够满足大部分应用需求，但随着多媒体技术的发展，对大数据存储的需求越来越多，比如数字图书馆需要存储大量图书、字典的信息，影像系统存储大量音频、视频等内容，DB2 通过大对象类型满足这类应用需求。大对象 (Large Objects)，即 LOBs，包括字符大对象 (Character lob, CLOB) 和二进制大对象 (Binary Lob, BLOB) 类型。从名字上很容易理解，CLOB 存储字符类数据，比如字典、书籍、简历等比较长的文本内容；而 BLOB 一般存储声音、视频等二进制内容。

一张表可以包含多个 LOB 字段，每个 LOB 字段的最大长度为 2GB (DBCLOB 存取双字节内容，每列最大支持 1GB 长度)。

我们知道，每条记录数据不能超过页大小，比如对于 pagesize 是 32K 页的表空间，记录大小不能超过 32K。对于 LOB 字段，在页中存储的并不是实际数据，而是 LOB 描述符，指向 LOB 数据存储的位置，可以通过 db2dart /dd 选项观察数据存储方式：

```
db2 "create table t1 (custno char(10), custname char(20), resume clob(20000), age int)"
```

Slot 4:

```
Offset Location = 6850 (x1AC2)
Record Length = 110 (x6E)
Record Type = Table Data Record (FIXEDVAR) (PUNC)
Record Flags = 0
Fixed part length value = 42
```

```
Column 1:
Fixed offset: 0
Type is Fixed Length Character String
31303030 30202020 2020 10000
```

```
Column 2:
Fixed offset: 11
Type is Fixed Length Character String
78756D69 6E677765 69202020 20202020 xumingwei
20202020
```

```
Column 3:
Fixed offset: 32
Type is Character Large Object String
Descriptor Length = 60
lfd_check = 49
lfd_version = 6
lfd_life_lsn = 0000 BBBE E0CF
lfd_descsize = 60
lfd_size = 43
```

```

        lfd_numsegs = 1
        lfd_mini_numsegs = 0
        lfd_first = 0
        lfd_last_pages = 1
        lfd_last_bytes = 43
        Regular Directory Offsets
        lfd_dir[0] = 0

        Column 4:
        Fixed offset: 37
        Type is Long Integer
        Value = 30

```

对 LOB 数据的处理一般是通程序来实现的，比如通过 Java、C 语言等。限于篇幅原因，我们将不展开讨论，感兴趣的读者可参考 <db2_install_path>/samples/java/jdbc 目录下的 DtLob.java。

需要注意的是，在循环日志模式中，LOB 字段的更改不会记日志；而在归档日志模式下，对于比较长的 LOB 字段可以选择不记日志。

在 DB2 中，对于常规数据的存取，会在 bufferpool 中处理。而 LOB 类型数据的查询和存储则不通过 bufferpool，如果对 CLOB 字段频繁更新和查询，性能往往很差。以下是使用 LOB 的一些最佳实践：

- 长度小于 32KB 的字符字段，建议用 varchar(32k)，而不用 CLOB；只有当长度可能超过 32KB 时，才考虑用 CLOB。
- 长度小于 32KB 的二进制数据，建议用 VARCHAR(32K) for bit data 类型；只有当长度可能超过 32KB 时，才考虑用 BLOB。我们曾经在某银行的一个性能测试中，将一张包含了 7000 万行数据的 BLOB 数据类型改为 VARCHAR(32K) for bit data，结果查询性能提高了几倍，而且这个修改不需要改动源代码。
- 如果 LOB 中包含经常变化的数据，如果可能，将变化的数据抽取为独立字段，而保持 LOB 字段的变更尽量小。
- 为了提升性能和运维管理的便利，建议将 LOB 数据存放在独立的表空间。由于 LOB 不通过 bufferpool，为提高性能，可考虑在文件系统缓存。因此在创建表空间的时候指定 create tablespace...file system caching。

在某些应用中，并不是所有的大对象数据都很大，有一些值只存储十几 KB 大小。如果能够把这些数据按常规数据存储在行记录中，那么将对性能是个很大的提升，这就是内联 LOB（Inline LOB）的思想。

Inline LOB 是 9.7 版本引入的新特性，与传统的 LOB 访问相比，Inline LOB 不需要事先获取 LOB 描述符，减少了 I/O，并且可以利用 bufferpool 缓存，因此提高了查询 LOB 数据的速度。同时，Inline LOB 数据也能利用压缩的功能，减少数据存储空间。

Inline LOB 特性是通过 CREATE TABLE 语句 INLINE LENGTH 选项或 ALTER TABLE 来指

定的。当要插入的 LOB 数据小于 `INLINE LENGTH` 指定的长度时，LOB 数据直接存在行记录中；否则，行记录中仍然存储传统的 LOB 描述符。

如果要使用 Inline LOB，最重要的任务是决定 `INLINE LENGTH` 的长度，这个长度加上其余字段长度的和即是一行数据长度，不能超出一个页面的大小。另外，DB2 也提供了一个表函数 `admin_est_inline_length` 辅助我们估算如何设置，当需要更改 `INLINE LENGTH` 长度时，可以通过 `ALTER TABLE` 语句更改，但为了使 Inline LOB 生效，需要对表数据做 `reorg`。

对于内联 LOB 功能，DB2 9.7 中表重组功能也相应做了扩充，`reorg` 命令多了一个 `LONGLOBDATA` 参数。`LONGLOBDATA` 参数只对 `long` 和 `LOB` 列有效。默认情况下是不启用 `LONGLOBDATA` 的，因为对 `long` 和 `LOB` 列的重组很消耗时间。但是在转换分开存放的 LOB 到内联的 LOB 时就需要这个参数了。`admin_is_inlined` 表函数用来判断某个大对象值是否发生了内联，可通过 `syscat.columns` 视图的 `inline_length` 字段查询内联 LOB 长度。

以下是使用 Inline LOB 的例子：

```
db2inst1@dpf1:~> db2 "create table t2 (custno char(10), custname char(20), resume
clob(20000) inline length 3500, age int)"
db2inst1@dpf1:~> db2 "insert into t2 values('10000', 'xumingwei', 'aaaaaaaaaaaa
aaaaaaaaabbbbbbbbbb', 30) "
db2inst1@dpf1:~> db2 "select max( admin_est_inline_length(resume) ) as
max_inline_length from t2"

MAX_INLINE_LENGTH
-----
30000
1 record(s) selected.
db2inst1@dpf1:~> db2 "alter table t2 alter column resume set inline length 30000"
db2inst1@dpf1:~> db2 "reorg table t2 longlobdata"
db2inst1@dpf1:~> db2 "select count(admin_is_inlined (resume)) as t from t2 "
```

```
T
-----
1

1 record(s) selected.
db2inst1@dpf1:~> db2 "select substr(tabname,1,18) as table, substr(colname,1,20) as
column, inline_length from syscat.columns where tabname='T2' "
```

TABLE	COLUMN	INLINE_LENGTH
T1	AGE	0
T1	CUSTNAME	0
T1	CUSTNO	0
T1	RESUME	30000

```
4 record(s) selected.
```

通过 db2dart /dd 选项可以观察 Inline LOB 的存储方式:

```
Slot 4:

Offset Location = 6875 (x1ADB)
Record Length = 85 (x55)
Record Type = Table Data Record (FIXEDVAR) (PUNC)
Record Flags = 0
Fixed part length value = 42

Column 1:
Fixed offset: 0
Type is Fixed Length Character String
31303030 30202020 2020 10000

Column 2:
Fixed offset: 11
Type is Fixed Length Character String
78756D69 6E677765 69202020 20202020 xumingwei
20202020

Column 3:
Fixed offset: 32
Type is Character Large Object String
Inlined LOB data. Length = 31
61616161 61616161 61616161 61616161 aaaaaaaaaaaaaaaaaa
61616161 61616262 62626262 62626262 aaaaaabbbbbbbbbb

Column 4:
Fixed offset: 37
Type is Long Integer
Value = 30
```

采用 Inline LOB 的好处是提高 LOB 数据查询的性能,也减少了 LOB 数据的存储空间,但在使用过程中还需要特别注意,因为 LOB 数据被存放在表行中,原先的行记录大小增加了,一个数据页中存放的行数就相应减少了,需要的数据页就增加了。这样很可能会影响非 LOB 查询的性能,因为一个查询可能需要读入更多的数据页才能完成检索。所以在实际应用过程中需要根据业务特点和数据特点仔细权衡,如果应用程序对非 LOB 查询的性能要求很高,或对 LOB 访问很少,则慎重使用该特性。

7.9 函数



学过程序设计的人对函数 (Function) 都不会陌生,在日常工作中,我们几乎每天都在接触函数,比如计算一天的销售量, `select sum(sales_amout) from sales`, 这里 `sum` 就是一个内置函数, DB2 提供了大量内置函数方便数据处理,限于篇幅原因,我们就不介绍这些内置函数了,感兴

趣的读者请参考 IBM 信息中心。本节将重点介绍用户自定义函数，即 UDF。

为了直观，我们先给大家演示一个例子。假定有这样一个场景，根据员工号查找员工所属的部门，需要从 `employee` 和 `department` 表做 join。如果有多个地方查询的话，每次都要写这样的语句，因此可考虑创建一个函数。

```
CREATE FUNCTION deptname(p_empid VARCHAR(6))          --①
RETURNS VARCHAR(30)                                   --②
LANGUAGE SQL
SPECIFIC deptname
BEGIN ATOMIC                                           --③
    DECLARE v_department_name VARCHAR(30);
    DECLARE v_err VARCHAR(70);
    SET v_department_name = (
        SELECT d.deptname FROM department d, employee e --④
        WHERE e.workdept=d.deptno AND e.empno= p_empid);
    SET v_err = 'Error: employee ' || p_empid || ' was not found';
    IF v_department_name IS NULL THEN
        SIGNAL SQLSTATE '80000' SET MESSAGE_TEXT=v_err; --⑤
    END IF;
    RETURN v_department_name;
END                                                     --⑥
@
```

本例中，①中定义一个函数，括号内是函数的参数，②是函数返回值，③是将整个函数体中的所有语句作为一个原子（Atomic），Atomic 也是 UDF 唯一支持的形式。在函数体中，先声明变量，然后④将 SQL 查询结果赋给变量，函数中只能将结果通过 SET 赋给变量，不支持 SELECT INTO 语句，并且 SQL 查询结果最多只能返回一行。⑤是错误捕获语句。⑥用 END 结束一个函数。在函数中每条语句以“;”结尾，为了创建该函数，需要将上述语句保存到一个文本文件，并在 END 结尾以一个其他符号结束，如@等，通过 `db2 -td@ -f <deptname>`，函数可以在 SQL 语句中调用。

创建函数的 SQL 语言，Oracle 中是 PL/SQL，DB2 中是 SQL PL（Procedure Language），但函数和触发器用的并不是 SQL PL，而是 SQL PL 的子集，也就是说，并不是 SQL PL 的所有语法都在函数中支持。SQL PL 的全集用于支持存储过程。

7.10 触发器

触发器（Trigger）是一个针对表定义的数据库对象，在这个表中插入、更新或删除行时会激活它。触发器本身使用 DB2 SQL PL 语言编写逻辑，这些逻辑将在触发器被激活时执行。触发器可以用于数据检验、实施完整性约束、审计和其他事件驱动的应用。以下是一个触发器例子：

```
CREATE TRIGGER reorder
AFTER UPDATE OF qty ON stock
```

```
REFERENCING NEW AS n
for each row mode db2sql
WHEN (n.qty <=5)
INSERT REORDER VALUES (n.itemno, current timestamp)
```

这个触发器的逻辑是：当在库存表(stock)中更新了产品数量(qty)后，如果更改后的数量小于等于 5，将会激活该触发器，并在 Reorder 表中插入一行数据，提醒库存管理员该进货了。这个触发器定义为 AFTER，意味着它在 UPDATE 之后激活。REFERENCING NEW AS n 子句允许设置转换变量，可以通过它访问旧的列值（Update 操作之前的值）和新的列值（UPDATE 之后的值）。另外，因为该触发器定义为 for each row，它的逻辑将在更新受影响的每一行之后执行。WHEN 子句用于限制激活触发器的条件。

由于性能的原因，再加上有些触发器本身包含一些业务逻辑，因此并不建议使用太多触发器。

在查询编译阶段，DB2 的查询重写模块会自动将所使用的所有触发器的逻辑编译进一个单独的查询，然后进行解析得到访问计划。因此，执行触发器并不是在插入一行后执行多个触发器语句，而是在插入或者更新数据时，在同一条 SQL 语句中执行。因此，过于复杂和大量的触发器会造成编译执行性能的大幅度下降，在大规模使用触发器时一定要谨慎。

7.11 存储过程



存储过程（Stored Procedure）属于开发范畴，对于 DBA 来说，掌握这部分内容能够使我们的运维工作如虎添翼，同时可以更好地为开发部门提供建议。

要实现一些数据处理和分析的逻辑，可以在应用层通过高级语言实现实现，如 C 和 Java 等，也可以在数据库端通过存储过程实现。存储过程是一组为了完成特定功能的 SQL 语句集，经编译后存储在数据库中，用户通过指定存储过程的名字并给出参数（如果有的话）来执行它。存储过程是数据库中的一个重要对象，当前国内很多系统都或多或少使用它完成一些数据处理任务。

DB2 的存储过程依赖于使用的语言分为 SQL 存储过程和外部存储过程（external），如基于 Java/C 高级语言的过程。DB2 的 SQL 存储过程是通过 SQL PL 语言实现的，SQL PL 包含了条件判断、循环控制、游标、异常处理捕获等，比较强大。

某公司老板年底大发善心，给员工派发奖金，奖金数额根据员工工资比例，低收入员工比例更高。但奖金总额不能超出某个范围。我们通过一个存储过程来实现，其中工资比例和奖金总额通过输入参数提供，返回值是派发的奖金总额，代码如下：

```
CREATE PROCEDURE total_raise ( IN p_min DEC(4,2),
IN p_max DEC(4,2),
OUT p_total DEC(9,2) )
LANGUAGE SQL
```

```

SPECIFIC total_raise
BEGIN
  -- Declare variables
  DECLARE v_salary DEC(9,2);
  DECLARE v_bonus DEC(9,2);
  DECLARE v_raise DEC(4,2);
  DECLARE v_job VARCHAR(15) DEFAULT 'PRES';
  -- Declare returncode
  DECLARE SQLSTATE CHAR(5);

  -- Procedure logic
  DECLARE c_emp CURSOR FOR
    SELECT salary, bonus
    FROM employee
    WHERE job != v_job;                                -- ①

    OPEN c_emp;                                        -- ②
    SET p_total = 0;
    FETCH FROM c_emp INTO v_salary, v_bonus;          -- ③
    WHILE ( SQLSTATE = '00000' ) DO
      SET v_raise = p_min;
    IF ( v_salary < 30000 ) THEN
      SET v_raise = v_raise + 0.04;
    END IF;

    IF ( v_raise > p_max ) THEN
      SET v_raise = p_max;
    END IF;

    SET p_total = p_total + v_salary * v_raise;
    FETCH FROM c_emp INTO v_salary, v_bonus;          -- ④
  END WHILE;

  CLOSE c_emp;                                        -- ⑤
END
@

```

第一个 BEGIN 和最后一个 END 之间是存储过程体，首先声明一些变量，然后①声明一个游标（cursor），②打开游标，③取得第一行记录，并将结果保存到两个变量，然后进入 WHILE 循环，遍历每条记录，⑤关闭游标。

将以上语句存储到文本文件 total.sp 中，通过 db2 -td@ -f total.sp 创建存储过程，通过 call 调用存储过程，代码如下：

```

db2inst1@dpf1:~> db2 -td@ -f total.sp
DB20000I The SQL command completed successfully.
db2inst1@dpf1:~> db2 "call total_raise(0.05,0.1,?) "    # 调用时，“?”表示输出参数，
即返回结果

```

Value of output parameters

```

-----
Parameter Name : P_TOTAL
Parameter Value : 114488.75

Return Status = 0

```

以上展示了存储过程的创建、调用等，接下来的例子对我们更实用，特别是在某些情况下需要 DBA 模拟大量测试数据，数据的生成可以通过存储过程实现。

假设有这样一张用户表 **customer**，需要插入 10 万行数据，要求产生的测试数据尽量与生产类似。以下设计的存储过程，基本包含了常用的数据类型，读者可根据自己的需要进行修改，并且每隔 1000 行提交一次，避免出现日志满错误，代码如下：

```

CREATE PROCEDURE sp_customer (IN count int)
LANGUAGE SQL
SPECIFIC sp_customer
BEGIN not atomic
    DECLARE i INTEGER DEFAULT 1;
    while i<count
    do
        insert into customer values
        ( i,
          translate(char( int(rand()*100000)), 'abcdefghijklmnopqrstuvwxyz',
'012345678901234567890123456' ),
          -- ①
          case
            when mod(int(100*rand()),2)=1
            then 'F'
            else 'M'
          end,
          -- ②
          mod(int(rand()*99999),90)+10,
          -- ③
          date( char(1950+mod( int(rand()*99999),60)) || '-' || char(mod( int(rand()*999),12)) ||
'-' || char(mod( int(rand()*999),31)) ),
          -- ④
          DECIMAL(rand()*99999,7,2) );
          -- ⑤

        set i=i+1;

        if( mod(i,1000)=0 ) then
            commit;
            -- ⑥
        end if;

    end while;
    commit;
END
@

```

①生成随机字符串，对于产生的每位随机数字，通过 **translate** 函数根据后面的数字转换成字符。

②通过 **case when** 表达式，随机生成性别。

- ③生成 10~100 之间的随机数字，表示年龄。
- ④随机生成 1950~2010-12-31 之间的日期。
- ⑤随机生成一个 decimal 类型。
- ⑥每 1000 行提交一次，避免出现日志满错误。

目前国内很多 DB2 应用使用存储过程完成一些逻辑处理，特别是一些数据分析系统。由于存储过程语句是静态语句，无法通过 dynamic sql snapshot 监控，只有通过 application snapshot 的 package 和 section number 抓取当时执行的语句。以下是插入 10 万行数据时抓到的 application snapshot 快照信息，我们会发现正在执行的 Package name 是 P5492935，Section number 是 1，代码如下：

```
Statement type           = Static SQL Statement
Statement                = Execute
Section number           = 1
Application creator      = DB2INST1
Package name             = P5492935
Consistency Token       =
Package Version ID      =
Cursor name             =
Statement database partition number = 0
Statement start timestamp = 02/24/2011 06:29:46.121241
Statement stop timestamp =
Elapsed time of last completed stmt(sec.ms)= 0.000000
Total Statement user CPU time = 0.000253
Total Statement system CPU time = 0.000000
SQL compiler cost estimate in timerons = 8
```

根据 Package name 和 Section number，可以找到对应的存储过程语句：

```
db2inst1@dpf1:~> db2 "select substr(pkgname, 1,20) as pkgname, substr(text,1,1000)
as text from syscat.statements where pkgname='P5492935' and sectno=1"
```

```
PKGNAME          TEXT
-----
-----
P5492935          insert into CUSTOMER values ...(限于篇幅原因，省略详细 SQL 语句)
1 record(s) selected.
```

那么，如何根据 Package 名字找到对应的存储过程名字呢？代码如下：

```
select deps.bschemaname SCHEMA,
       procs.routinename PROCEDURE,
       deps.bname PACKAGE,
       procs.valid VALID
from   sysibm.sysdependencies deps,
       sysibm.sysroutines procs
where  deps.dtype = 'F'and
```

```

deps.btype = 'K'and
procs.specificname = deps.dname and
procs.routineschema = deps.dschem
and deps.bname = ' P5492935'
order by 1,2

```

由于存储过程在创建时会为每条语句生成访问计划，并将存储过程绑定到 DB2 Package 中，当 SQL 语句对应的表数据有重大变化或增加了索引后，需要重新生成访问计划，可通过如下语句进行存储过程的重新绑定（注意：db2inst1.sp_customer 是数据过程名）：

```
CALL SYSPROC.REBIND_ROUTINE_PACKAGE('P', 'db2inst1.sp_customer ', 'ANY')
```

在 V8.2 之后，SQL 存储过程总是作为受信任的进程/线程在 DB2 引擎中运行，因此速度比非 SQL 存储过程快。对于非 SQL SP（如 C/Java）等的调用，在后端表现为一个 db2fmp 进程，这个进程是多线程的，支持若干个 SP 同时执行。还记得我们在创建实例的时候提到的 fenced 用户吗？SP 的执行就是通过 fenced 用户实现的。那么怎么查看这个用户是什么呢？一种方法是通过 `ps -ef | grep -i db2fmp`，另一种方法是通过 `db2pd -fmp | grep -i "Fenced User"`。

如果想要修改 SP 执行的用户，即 fenced 用户，可以通过修改 `<inst_home>/sqlib/adm/.fenced` 文件的属主（owner）。

当执行逻辑比较简单时，建议用 SQL 存储过程实现，对于复杂的逻辑，可考虑用外部存储过程实现，如 Java/C 等。本节给大家演示一下 Java SP 的创建方法和部署。在 Java 开发环境中，如 Eclipse，创建一个 TotalPay class，并实现一个方法 raiseSalary，代码如下：

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class TotalPay
{
    public static void raiseSalary()
    {
        double v_min;
        double v_max;
        Connection con = null;
        v_min=0.01;
        v_max=0.04;

        try
        {
            con = DriverManager.getConnection("jdbc:default:connection");
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT salary, bonus FROM employee WHERE
job!='PRES' ");
            double v_total=0;
            double v_raise;

```

```

double v_salary, v_bonus;

while ( rs.next() )
{
    v_raise = v_min;
    v_salary = rs.getDouble(1);
    v_bonus = rs.getDouble(2);
    if ( v_bonus < 3000 )
        { v_raise += 0.04; }
    if ( v_raise > v_max )
        { v_raise = v_max; }
    v_total += v_salary * v_raise;
}
System.out.println(v_total);
rs.close();
con.close ();
System.out.println("Complete(0).");
}
catch (Exception e)
{
    e.printStackTrace();
}
}
}

```

然后对该 Java 程序进行编译，生成 TotalPay.class，将其复制到<inst_name>/sqllib/function 目录下，接着就可以创建 Java 存储过程了，注意 EXTERNAL NAME，单引号中使用 Java 类名! 方法名，代码如下：

```

CREATE PROCEDURE TotalPay()
SPECIFIC INSERT
DYNAMIC RESULT SETS 0
DETERMINISTIC
LANGUAGE JAVA
PARAMETER STYLE JAVA
NO DBINFO
FENCED
THREADSAFE
MODIFIES SQL DATA
PROGRAM TYPE SUB
EXTERNAL NAME 'TotalPay!raiseSalary'
;

```

注意：创建上述存储过程时有一个比较重要的选项，FENCED/NOT FUNCED，在 Java 中，由于系统没有办法在一个进程空间里面直接运行 Java 的代码，必须要启动一个虚拟机才行，因此 Java 只支持 fenced 选项。在有些语言中支持 NOT FENCED，但使用前需要仔细测试，一定要确保不破坏 DB2 的引擎。

FENCED 与 NOT FENCED 相比，性能要慢一些，主要原因是需要启动 fmp 进程（如果已

经启动了则好一些), 然后将 fenced 存储过程所使用的库装载到 db2fmp 进程的寻址空间。

7.12 小结



本章我们主要讨论了 DB2 中的各种数据对象。熟练使用各种类型的对象, 以及了解它们的结构和原理, 在数据库维护中会给用户带来极大的方便。现在很多的 DBA 都已经或多或少地参与了应用程序的开发, 拥有数据对象原理的知识, DBA 可以在应用程序的开发阶段对开发人员提出很多宝贵意见, 减少后期修改的时间, 也减少系统的维护成本。

7.13 判断题



(1) 表、索引、视图、别名、序列均属于数据库对象。

T: 正确

F: 错误

(2) 视图中包含真实用户数据。

T: 正确

F: 错误

(3) 序列的 Cache 设置得越大越好。

T: 正确

F: 错误

(4) 大对象 (非内联) 数据的读取 I/O 开销一般来说超过 VARCHAR 类型。

T: 正确

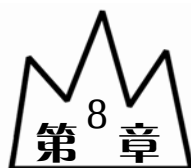
F: 错误

(5) 创建触发器不会对现存的 INSERT/UPDATE/DELETE 语句性能造成影响。

T: 正确

F: 错误

第三部分 DB2 运维管理



数据迁移

数据迁移是 DBA 重要的工作内容之一。广义的数据迁移，既包括平台间数据库迁移，也包括同一数据库不同表间的数据迁移，还包括数据表与文件之间的数据导入/导出。针对不同的场景，DBA 要在众多的方案中选择最优的解决方案：

同构平台间的数据迁移，如从生产库系统到测试库，最简单的迁移方法是通过对生产库做备份，然后在测试机上进行数据库恢复。这部分内容我们将在第 9 章重点讲述。

异构平台间的数据迁移，如从 Windows 平台到 Linux 系统的数据迁移，这种场景不能采用数据库备份恢复，只能先用 db2look 导出表结构，并将表数据导出来，然后用导入到目标库。

灾难或故障情况下的数据挽回，DB2 提供了 db2dart 工具，可以在实例都无法启动的情况下将数据导出。该工具一般用于没有进行数据库备份，但发生了日志故障或其他操作故障，而没有其他机制挽回数据的场景。

本章主要介绍表与文件间的数据导入/导出、表间数据迁移，内容安排如下：

- 数据迁移概述
- 常见的数据导入/导出文件格式。
- Export、Import、Load 等数据迁移工具。
- 常见的导入/导出场景。
- db2look+db2move 进行数据库迁移。
- db2dart 的使用。

8.1 数据迁移概述

在日常工作中，经常有数据的导入/导出需求，比如将某些表的数据保存成报表形式，或将文件数据加载到数据库等。为此，DB2 提供了很多工具供大家选择，如 `export`、`import`、`load`、`db2look`、`db2move` 和 `db2dart`。图 8.1 直观地展现了这几个工具的使用，最上面虚线框部分是工具支持的导入/导出文件格式，中间虚线框是 DB2 提供的导入/导出工具，最下面是 DB2 数据表。

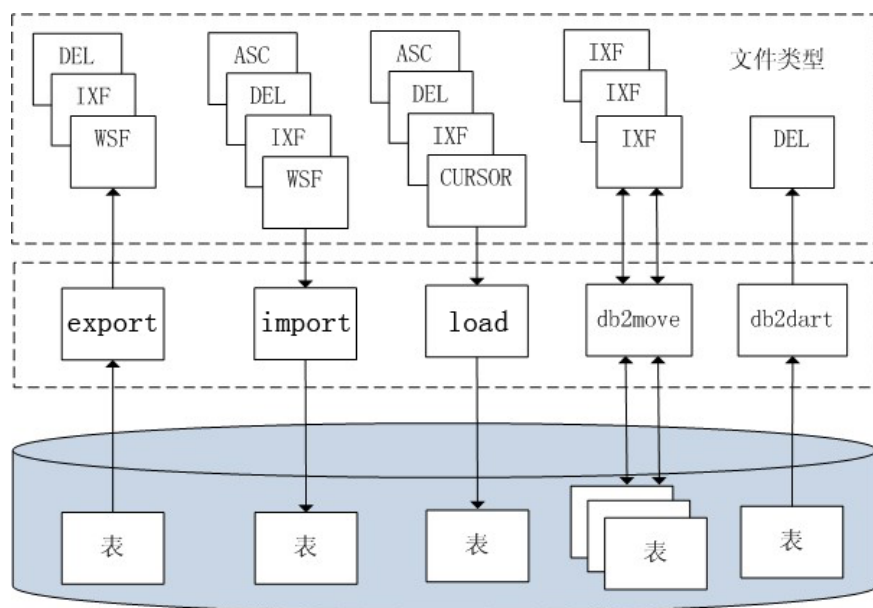


图 8.1 数据导入/导出工具

8.2 文件格式

首先我们介绍 DB2 支持的文件格式：DEL、ASC、PC/IXF 和 WSF 格式，其中 DEL 和 ASC 格式是文本格式，PC/IXF 格式是 IBM 特有的二进制格式，WSF 格式主要用于和 Lotus 1-2-3 进行数据导入/导出，新版本中将不再支持。

8.2.1 DEL 格式

定界 ASCII 格式（DEL）是 DB2 用于数据交换的最常用格式。这种格式包含 ASCII 数据，使用字符分隔符分隔列值，分隔符用来标识数据元素的起始和结束，最主要的分隔符有以下几种。

- 字符分隔符：界定字符字段的起始。在默认情况下，用双引号(“”)作为字符分隔符。
- 列分隔符：界定列的结束。默认用逗号(,)作为列分隔符。
- 行分隔符：用来标识一行或一个记录的结束。默认用换行符作为分隔符。

以下是 DEL 格式的例子：

```
330,"Burke",66,"Clerk",1,+49988.00,+00055.50
340,"Edwards",84,"Sales",7,+67844.00,+01285.00
350,"Gafney",84,"Clerk",5,+43030.50,+00188.00
```

8.2.2 ASC 格式 ■ ■ ■

定长 ASCII (ASC) 格式，顾名思义，这种文件类型包含定长 ASCII 数据，每个数据长度与列定义相同，不足的用空格补齐，行与行之间通过换行符分隔，例子如下：

```
330 Burke    66 Clerk1  +49988.00 +00055.50
340 Edwards  84 Sales7   +67844.00 +01285.00
350 Gafney   84 Clerk5   +43030.50 +00188.00
```

8.2.3 PC/IXF ■ ■ ■

PC/IXF (IXF) 是 IBM 特定的二进制格式，适用于在异构平台间进行数据迁移，IXF 的优点是数据占用空间小，而且包含表结构的定义，可以通过 IXF 文件重建表。

8.2.4 Cursor ■ ■ ■

游标 (Cursor) 是非常重要的一个概念，它提供了一种对从表中检索出的数据进行操作的灵活手段。可以把游标想象为一个指针，刚开始指向结果集中的第一条数据，当第一条数据读取完成后，游标会自动跳转到下一条数据。

如果在两张表之间进行数据迁移，最容易想到的方法是先将数据从一张表导出来，存到一个文件中，然后将这个文件的数据导入到另外一张表。而采用游标，数据不需落地，效率比较高，因此比较适于表间数据迁移。只有 LOAD 支持游标，其余几种工具不支持。

下面是使用 Cursor 进行表间迁移的例子：

```
DECLARE mycursor CURSOR FOR SELECT col1, col2, col3 FROM tab1;
LOAD FROM mycursor OF CURSOR INSERT INTO newtab;
```

8.3 export



export 用于将表里的数据导出到文件中，在这里我们只介绍常用的几个选项，命令的详细语

法请参考 IBM 信息中心。

`export to filename` 用来指定将导出的数据放在 `<filename>` 文件中, `of filetype` 指定导出文件的格式, 如 IXF、DEL 或 WSF 等。`messages` 用来保存导出过程中发生的错误或警告信息。`select-statement` 通过 SQL 语句指定导出的数据。

`lobs to <path>` 指定大对象数据存放的目录, 该目录下生成的文件将保存大对象数据, 在默认情况下, 该文件的命名规则是: `<filename>.<00x>.lob`。如果该表只有一个 LOB 字段, 则 `x=1`, 如果有多个 LOB 字段, 则顺序编号, 为每个 LOB 字段生成一个文件。如果不想按默认的文件命名, 可通过 `lobfile <filename>` 指定 LOB 文件名字。

如果希望控制导出数据的格式, 比如日期的格式、字符串、分隔符等格式, 这时可通过 `modified by` 文件修饰符定制文件格式。比如, `CHARDELx` 用于指定字符串定界符, `COLDELx` 用于指定字段定界符, `x` 是定界符符号。`TIMESTAMPFORMAT` 用于指定日期字段的时间表达式。用户需要确保导出的数据中不包含分隔符, 否则在导入时会出现异常错误。

export 例子:

- 将 `employee` 表数据导出到 `employee.del` 文件中, 并记录消息, 代码如下:

```
db2 "export to employee.del of del messages emp.msg select * from employee"
```

- 导出 `employee` 表数据, 导出字符串用 “'” 分隔, 而不是默认的 “ ””, 代码如下:

```
db2 "export to employee.del of del modified by CHARDEL'" messages emp.msg select * from employee"
```

- 导出 `emp_resume` 表数据, 并将大对象数据存放到 `lobs to` 指定的目录下, 通过指定 `modified by lobsinfile`, 大对象字段的所有值都保存到一个文件中, 代码如下:

```
db2 "export to emp_resume.del of del lobs to d:\temp\lobs modified by lobsinfile select * from emp_resume"
```

- 导出 `emp_resume` 表数据, 并将大对象数据存放到 `lobs to` 指定的目录下, 通过指定 `modified by lobsinsefiles`, 大对象字段的每个值都存在一个独立的文件中, 代码如下:

```
db2 "export to emp_resume.del of del lobs to d:\temp\lobs modified by lobsinsefiles select * from emp_resume"
```

8.4 import

□ -----

`Import` 用来将文件里的数据导入到表中, 是 `export` 操作的逆向过程。在这里我们只介绍常用的几个选项, 详细命令语法请参考 IBM 信息中心。

`import from <filename>` 指定导入文件的位置和名字, `of <filetype>` 指定输入文件的类型, 如

DEL、IXF 或 ASC 等。messages 用来记录导入过程中发生的错误或警告信息。

<Action> INTO table-name 用来指定导入数据的几种方式，目前支持 insert、insert_update、replace、replace_create 和 create。9.7 版本以后 replace_create 和 create 将不再支持。

- insert 用于追加，不改变表中已有数据。
- insert_update 用于表中有主键的情况，如果导入数据与表中数据主键匹配，则 update，否则 insert 追加。
- replace 首先删除表数据，然后插入输入文件数据。由于 replace 会首先清空数据，所以建议先做好备份。

import 命令还包含很多有趣的选项，如 commitcount 和 restartcount。

commitcount 参数的主要目的是避免事务日志满和锁升级。commitcount <N/automatic>表示每导入 N 行数据后提交一次，而不是等所有数据都导入才提交。automatic 表示 DB2 内部会自动计算什么时候提交，默认情况下，DB2 会自动使用 automatic 选项。

restartcount/skipcount N 表示跳过文件前 N 行数据，而从第 N+1 开始继续导入。该选项一般用于导入过程中出现了错误，有些数据已经提交入库了，重新导入时就可以忽略这些已经入库的数据。

在默认情况下，import 会在目标表加 X 锁（排他锁），不允许其他应用访问（allow no access）。如果 import 允许其他应用读和写，可指定 allow write access 选项，这时 import 会在目标表上加 IX 锁，但该选项只能用于 insert 或 insert_update 操作。

import 例子：

- 将 employee.del 文件数据导入到 employee 表中，并记录消息，代码如下：

```
db2 "import from employee.del of del messages emp.msg insert into employee"
```

- 将 employee.del 文件数据导入到 employee 表中，在文件中，字符串用 “' ’” 分隔，而不是默认的 “" ””。allow write access 允许其他应用读或写，commitcount automatic 允许 import 命令自动选择何时提交，代码如下：

```
db2 "import from employee.del of del modified by CHARDEL'" allow write access
commitcount automatic insert into employee"
```

- 将大对象数据导入到表中 lob from 指定大对象数据的位置，代码如下：

```
db2 "import from emp_resume.del of del lob from d:\temp\lobs modified by lobsinfile
insert into emp_resume"
```

8.5 load

8.5.1 load 步骤及原理

从原理上说, `import` 实际上还是会执行 `insert`、`update`、`delete` 等操作, 处理每一行数据都要通过 DB2 引擎, 验证各种约束和触发器, 并通过事务日志记录变化。

对于大数量的导入, 比如一些仓库或分析类系统, `import` 就无法满足性能需求, 这时 `load` 就可以大显身手了。`load` 不是一行一行地处理数据, 而是对输入数据按照 DB2 物理存储方式进行格式化, 并将格式化的数据页直接写到数据库中, 记录的日志很少, 也不会检查 `check` 约束和参照完整性约束, 不触发触发器, 因此特别适合大数据量的导入。

`load` 将文件或游标数据导入到表中, `load` 命令包含很多参数, 在这里我们只介绍常用的几个选项, 详细命令语法请参考 IBM 信息中心。

```
LOAD FROM input_source OF input_type
  MESSAGES message_file
  [ INSERT | REPLACE | TERMINATE | RESTART ]
  INTO target_tablename
```

`load from <input_source>` 指定输入源, 可能是文件或游标。`of input_type` 指定输入格式, 如 `DEL`、`ASC`、`PC/IXF` 或 `Cursor`。在开始 `load` 之前, 目标表必须已经存在。使用 `messages` 选项可以捕获 `load` 期间遇到的错误、警告和相关信息。

`load` 支持以下 4 种动作:

- `insert` 用于追加, 不改变表中已有数据。
- `replace` 首先删除表数据, 然后插入输入文件数据。由于 `replace` 会首先清空数据, 所以建议先做好备份。
- `terminate` 将终止 `load` 操作, 并将数据恢复到 `load` 开始时的状态。注意: 如果在 `load ... replace` 时出现了 `load pending`, 采用 `load ... terminate` 会清空表。
- `restart` 用于重启被中断的 `load` 命令。`restart` 会使用之前 `load` 时产生的临时文件, 并从最近的一点开始加载。因此, 千万不要手动删除 `load` 所生成的任何临时文件。一旦 `load` 成功完成, 这些临时文件将自动被删除。在默认情况下, 这些临时文件是在当前的工作目录中创建的, 可以使用 `TEMPFILES PATH` 选项指定存放临时文件的目录。

注意: `replace` 动作会先删除表中已有数据, 如果在 `load` 过程中出现异常, 则通过 `load ... terminate` 将表清空。因此, 对于重要的表, 建议在 `replace` 之前先做好备份。

一个 `load` 操作包括装载、构建、删除和索引复制 4 个阶段。如果目标表不包含唯一性索引, 则仅会进行装载阶段; 否则装载阶段后, 会进行构建、删除与索引复制的另外 3 个阶段。可通过 `db2diag.log` 观察每个阶段的执行情况。

1) 装载阶段 (load)

装载阶段将源文件解析成数据物理存储的格式，直接装入到页中，而不通过 DB2 引擎。在装入过程中，会收集索引键和表统计信息，并记录一致点。当数据不符合表定义时，这些数据就被当做无效数据，无效数据是不会装载到表中的，但可以放在转储文件 (dump file) 中，并记录在 message 消息文件中。可以使用 `modified by dumpfile` 修饰符来指定转储文件名和路径。

2) 构建索引阶段 (build)

如果加载的表上有索引的话，构建阶段基于装载阶段收集到的键创建索引。

3) 删除重复值阶段 (delete)

如果表上有主键或唯一性索引，此阶段将删除违反唯一键的行。注意，此阶段只检查违背了唯一性约束的行，而不会检查 `check` 约束和参考完整性约束。可以创建一个异常表 (exception table) 来存储被删除的行，这样在 `load` 结束后可以查看异常表，并决定如何处理它们。如果没有指定异常表，那么重复行将被删除。后面将更详细地讨论异常表。

4) 索引复制阶段 (index copy)

如果 `load` 指定了 `allow read access` 和 `use tablespace` 选项，那么此阶段会将索引数据从系统临时表空间中复制到索引表空间中。

异常表需要用户事先创建，它的前 N 个字段具有与目标表相同的列定义。可在表的最后增加两个列，一个是用于记录一个行何时被插入的时间戳列，另一个是用于存放一个行之所以被当做异常的原因的 CLOB 列。异常表的定义如下：

```
CREATE EMPEXP LIKE EMPLOYEE;
ALTER TABLE EMPEXP ADD COLUMN TS TIMESTAMP ADD COLUMN MSG CLOB(32K)
```

我们通过一个例子，演示 `load` 加载过程，执行过程如图 8.2 所示。

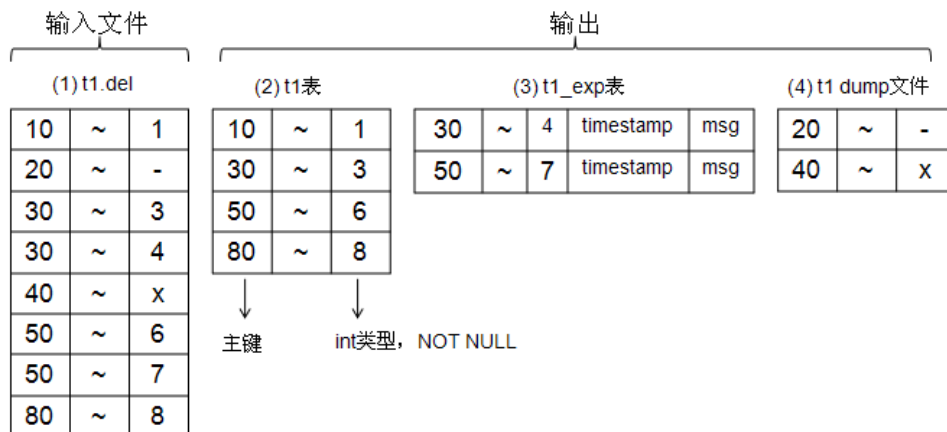


图 8.2 load 过程

(1) 目标表 **t1** 结构定义如下, **empno** 字段上定义了主键, **seqno** 为数值型, 非空。并为 **t1** 表创建异常表 **t1_exp**。

```
db2inst1@dpf1:~> db2 "create table t1 (empno int not null primary key, name char(10),
seqno int not null )"
db2inst1@dpf1:~> db2 "create table t1_exp like t1"
db2inst1@dpf1:~> db2 "alter table t1_exp ADD COLUMN TS TIMESTAMP ADD COLUMN MSG
CLOB(32K) "
```

(2) 创建如下数据文件 **t1.del**, 一共 8 行数据, 代码如下:

```
db2inst1@dpf1:~> cat t1.del
10,"wang qi",1
20,"zhang san",
30,"xu xin",3
30,"chen yan",4
40,"aaaaa",x
50,"bbbb",6
50,"cccc",7
80,"li si",8
```

(3) 将 **t1.del** 数据加载到 **t1** 表中, 分析输出结果发现一共读取了 8 行数据, 拒绝了 2 行, 删除了 2 行, 代码如下:

```
db2inst1@dpf1:~> db2 "load from t1.del OF DEL MODIFIED BY DUMPFILE=/data1/t1.dmp
MESSAGES t1.msg
INSERT INTO t1 FOR EXCEPTION t1_exp"

Number of rows read          = 8
Number of rows skipped       = 0
Number of rows loaded        = 6
Number of rows rejected      = 2
Number of rows deleted       = 2
Number of rows committed    = 8
```

(4) 加载完成后, 检查目标表、转储文件、异常表和消息文件, 然后决定如何处理被拒绝的行。

首先检查目标表, 发现有 4 条数据插入到表中:

```
db2inst1@dpf1:~> db2 "select * from t1"
EMPNO      NAME      SEQNO
-----
      10  wang qi      1
      30  xu xin      3
      50  bbbb      6
      80  li si      8

4 record(s) selected.
```

然后检查 dump file，发现 2 行数据违背了表定义。empno=20 的行最后一列为 NULL，违背了非空定义；empno=40 的最后一列为 x，违背了数值型定义。这个转储文件是在第(1)步装入阶段完成的：

```
db2inst1@dpf1:/data1> cat t1.dmp.load.000
20,"zhang san",
40,"aaaaa",x
```

最后检查异常表 t1_exp,发现 empno=30 和 empno=50 这两行违背了唯一性约束。异常表数据是在第（3）步完成的：

```
db2inst1@dpf1:/data1> db2 "select empno, name, seqno, ts, substr(msg,1,10) as msg
from t1_exp"
EMPNO      NAME      SEQNO      TS                      MSG
-----
      30 chen   yan      4      2011-01-11-20.40.35.426691  00001I0000
      50 cccc           7      2011-01-11-20.40.35.426691  00001I0000

2 record(s) selected.
```

以上通过一个实例为大家演示了 load 的全过程，目的是让大家了解内部执行细节，以便在出现问题的时候能够诊断错误发生在哪个步骤。在实际操作中，如果能保证数据不违背唯一性约束，则不需要使用异常表，否则建议使用。

8.5.2 load 表状态 ■ ■ ■

DBA 在运维时，一般都会考虑所做的操作对业务的影响，对于 load 也是如此。默认情况下，当 load 表时使用 ALLOW NO ACCESS 模式，即 load 完成之前，不允许其他应用访问该表。

为了提高可用性，load 提供了 ALLOW READ ACCESS 模式，在 load 的时候，允许其它应用访问 load 之前的原有数据，但不能访问新加载的数据。需要注意的是，load ... replace 不支持 ALLOW READ ACCESS，因为 replace 操作会先删除已有表数据。

可以通过 load query 命令检查表状态，在某一时刻一张表可能会同时处于几种状态，只有当表处于 normal 状态时，才能对表进行正常的读写操作，当表处于其他状态时，我们要分析可能的原因和解决方案。表 8.1 总结了 load 时可能出现的几种状态及解决方法。

表 8.1

LOAD 表状态	出现原因	解决方法
normal	正常状态	
set integrity pending	如果目标表有 check 约束或 reference 约束，那么 load 后该表将处于 set integrity pending 状态，表明表有约束还未检查	执行完整性检查： set integrity for <tablename> immediate checked

续 表

LOAD 表状态	出现原因	解决方法
load in progress	正在数据加载过程中	
load pending	数据提交前出现了故障，如表空间没有足够的空间等	通过 load.terminate、load.replace 或 load.restart 解除挂起状态
read access only	目标表数据是可以读的，当 load 时指定了 allow read access，那表就会处于 read access only 状态	
unavailable	表可能被删除了或从 backup 中恢复了	
unknown	通过 load query 命令无法得知表的状态	

8.5.3 load 的 copy 选项 ■ ■ ■

在 load 命令中，有一个 copy 选项，很多 DBA 都吃过这个选项的苦头，有的甚至付出了惨重代价，使用时必须要小心。copy 可以理解为备份，我们前面介绍过，load 记录的日志很少（在 build 索引、delete 重复值阶段会记录日志），那么在归档日志模式下(第 9 章会介绍循环日志、归档日志)，当恢复数据库并进行前滚时，如何恢复 load 加载的数据呢？如果能将 load 数据备份，那问题就简单了，这就是 COPY 选项的目的所在。

前面我们介绍的一些命令都没有带 COPY 选项，那是假设我们的数据库是循环日志，COPY 选项只适用于归档日志模式，在循环日志模式下没有意义。

copy 支持 3 种方式：copy no、copy yes 和 nonrecoverable。

copy no 是默认方式，对于可恢复数据库来说，copy no 会将 load 表所属的表空间置于 backup pending 状态，意思就是提示 DBA 要在 load 后对表空间做备份，否则出了问题无法恢复。目标表可以读，但不能进行增/删/改。而且 load 命令一旦发起，表空间立即处于 backup pending 状态，即使终止 load 操作，也不能脱离此状态。

举例：

(1)创建 sample 数据库，然后创建一个表空间 ts1，并在 ts1 中创建一张表 t3。接着将 sample 设为归档日志模式。

```
inst20@db2server:~> db2sampl
inst20@db2server:~> db2 connect to sample
inst20@db2server:~> db2 "create tablespace ts1 "
DB20000I The SQL command completed successfully.
inst20@db2server:~> db2 "create table t3 (id int, name char(20)) in ts1"
DB20000I The SQL command completed successfully.
```

```
inst20@db2server:~> db2 update db cfg for sample using logarchmeth1
disk:/home/db2inst1/archlog
DB20000I The UPDATE DATABASE CONFIGURATION command completed successfully.
inst20@db2server:~> db2 backup db sample to /data1 compress
Backup successful. The timestamp for this backup image is : 20110519005636
```

(2) 将一组测试数据 Load 到 t3 表中。

```
inst20@db2server:~> more t3.del                                # 生成测试数据
1,'aaa'
2,'bbb'
3,'ccc'

inst20@db2server:~> db2 "load from t3.del of del insert into t3"    # 将这几条数据
加载到 t3 表
```

(3) T3 数据可以查看，但无法增删改。

```
inst20@db2server:~> db2 "select * from t3"

ID          NAME
-----
1 'aaa'
2 'bbb'
3 'ccc'

3 record(s) selected.

inst20@db2server:~> db2 "update t3 set name='newccc' where id=3"
DB21034E The command was processed as an SQL statement because it was not a
valid Command Line Processor command. During SQL processing it returned:
SQL0290N Table space access is not allowed. SQLSTATE=55039
```

(4) 检查表空间状态，发现 ts1 处于 backup-pending。

```
inst20@db2server:~> db2 list tablespaces show detail

Tablespaces for Current Database
...
Tablespace ID          = 6
Name                   = TS1
Type                   = Database managed space
Contents               = All permanent data. Large table space.
State                  = 0x0020
Detailed explanation:
Backup pending
```

(5) 对 ts1 或整个数据库做备份，问题解决。

```
inst20@db2server:~> db2 backup db sample to /data1 compress
Backup successful. The timestamp for this backup image is : 20110519022834
```

copy yes 选项会在 load 结束时，自动对表所属的表空间做一次备份，load 结束后，表所在的表空间不会处于 backup pending 状态，而为正常状态，但由于要备份，所需时间要长些。在前滚恢复阶段，DB2 会使用这个备份文件恢复 load 过程中加载的数据。

举例：

(1) 仍然在 sample 数据库，t3 表上 Load 数据，使用 COPY YES 选项。

```
inst20@db2server:~> more t3.del                                # 生成测试数据
1, 'aaa'
2, 'bbb'
3, 'ccc'

inst20@db2server:~> db2 "load from t3.del of del insert into t3 copy yes to /data1"
SQL3109N The utility is beginning to load data from file "/home/inst20/t3.del"
```

(2) 使用 Copy yes 选项时，ts1 表空间不会处于 backup-pending 状态。

```
inst20@db2server:/data1> db2 "select * from t3"

ID          NAME
-----
1 'aaa'
2 'bbb'
3 'ccc'
1 'aaa'
2 'bbb'
3 'ccc'

6 record(s) selected.
inst20@db2server:/data1/range> db2 list tablespaces

Tablespaces for Current Database

Tablespace ID          = 6
Name                    = TS1
Type                    = Database managed space
Contents                = All permanent data. Large table space.
State                   = 0x0000
Detailed explanation:
Normal
```

(3) Load 会产生如下的备份介质，其中备份文件中第二个字段“4”表示 Load 产生的备份。这份备份将在数据库进行前滚恢复操作时用于重新创建 Load 操作对数据库的修改。以下通过实例演示了前滚后，T3 表保留了 Load 的数据。

```
inst20@db2server:~> ls
SAMPLE.0.inst20.NODE0000.CATN0000.20110519022834.001
SAMPLE.4.inst20.NODE0000.CATN0000.20110519022955.001
```

```

inst20@db2server:/data1> db2 restore db sample taken at 20110519022834
SQL2539W Warning! Restoring to an existing database that is the same as the
backup image database. The database files will be deleted.
Do you want to continue ? (y/n) y
DB20000I The RESTORE DATABASE command completed successfully.
inst20@db2server:/data1>

inst20@db2server:/data1> db2 rollforward db sample to end of logs and stop

                                Rollforward Status

Input database alias              = sample
Number of nodes have returned status = 1

Node number                      = 0
Rollforward status               = not pending
Next log file to be read         =
Log files processed               = S0000002.LOG - S0000002.LOG
Last committed transaction       = 2011-05-19-02.48.56.000000 Local
DB20000I The ROLLFORWARD command completed successfully.

inst20@db2server:/data1>
inst20@db2server:/data1> db2 connect to sample
Database Connection Information

Database server      = DB2/LINUX 9.5.5
SQL authorization ID = INST20
Local database alias = SAMPLE

inst20@db2server:/data1> db2 "select * from t3"

ID          NAME
-----
1 'aaa'
2 'bbb'
3 'ccc'
1 'aaa'
2 'bbb'
3 'ccc'

6 record(s) selected.

```

如果想避免在 **LOAD** 完成后立即备份表空间，又不阻止查询和修改表数据，可采用 **nonrecoverable** 选项。但要注意的是，**nonrecoverable** 将表标记为不可恢复，如果以后需要恢复表空间并且回滚到 **nonrecoverable load** 选项之后的某个时间点，这个表是不可恢复的，所有与该表相关的日志会被忽略，您只能删除并重新创建表。因此，这个方法一般用在表可以被重建的场景中。

举例：

(1) 仍然在 `sample` 数据库，`t3` 表上 Load 数据，使用 `NONRECOVERABLE` 选项。

```
inst20@db2server:~> more t3.del                                # 生成测试数据
1, 'aaa'
2, 'bbb'
3, 'ccc'

inst20@db2server:~> db2 "load from t3.del of del insert into t3 nonrecoverable "
SQL3109N The utility is beginning to load data from file "/home/inst20/t3.del"
```

(2) 当使用 `nonrecoverable` 选项时，表和 `ts1` 表空间都可以使用。

```
inst20@db2server:~> db2 "select * from t3"

ID          NAME
-----
1 'aaa'
2 'bbb'
3 'ccc'
1 'aaa'
2 'bbb'
3 'ccc'
1 'aaa'
2 'bbb'
3 'ccc'
9 record(s) selected.

inst20@db2server:/data1/range> db2 list tablespaces

Tablespaces for Current Database

Tablespace ID          = 6
Name                    = TS1
Type                    = Database managed space
Contents                = All permanent data. Large table space.
State                   = 0x0000
Detailed explanation:
Normal
```

(3) 如果执行前滚命令恢复数据库时，前滚操作将跳过 `Load` 的处理，将 `Load` 的表标记为无效，对该表的任何操作都不能进行。此时只有将表删除、重新构建，或使用 `Load` 操作时间点之后所做的数据库全备份或表空间备份来恢复该表。

```
inst20@db2server:~> cd /data1
inst20@db2server:/data1> db2 restore db sample taken at 20110519022834
SQL2539W Warning! Restoring to an existing database that is the same as the
backup image database. The database files will be deleted.
Do you want to continue ? (y/n) y
```

```

DB20000I  The RESTORE DATABASE command completed successfully.

inst20@db2server:/data1> db2 rollforward db sample to end of logs and stop

                                Rollforward Status

Input database alias              = sample
Number of nodes have returned status  = 1

Node number                      = 0
Rollforward status                = not pending
Next log file to be read         =
Log files processed               = S0000002.LOG - S0000003.LOG
Last committed transaction       = 2011-05-19-04.20.15.000000 Local
DB20000I  The ROLLFORWARD command completed successfully.

inst20@db2server:/data1> db2 connect to sample
      Database Connection Information

Database server      = DB2/LINUX 9.5.5
SQL authorization ID = INST20
Local database alias = SAMPLE

inst20@db2server:/data1> db2 "select * from t3"

ID          NAME
-----
SQL1477N  For table "INST20.T3" an object "4" in table space "6" cannot be
accessed.  SQLSTATE=55019

```

在归档日志模式下，各位在选择这几种方法前，需要深入理解它们的优缺点。**COPY NO** 选项会将表所在的表空间置于 **backup pending**，在这个状态中，只能查数据，而无法更改、删除，因此，不建议在日常交易期间使用该方法；**NONRECOVERABLE** 选项的使用更要小心，当前滚时，目标表的数据无法恢复，而必须删除。对于有些系统，用户在做 **load** 时业务应用程序必须停止。为了快速完成 **load**，用户可以考虑选择 **NONRECOVERABLE**，但是在使用 **NONRECOVERABLE** 方式装载数据后，最好在启动业务应用程序以后对数据库进行在线备份，确保被加载的表可以被恢复；**COPY YES** 选项会导致加载的时间变长。

举一个比较典型的例子。笔者的一个朋友在 2009 年底的某一天下午 1 点半左右，突然打电话给我，说他们的交易系统几乎所有表只能查，不能增、删、改，应用几乎处于停滞。然后我问他这边表做了什么操作没有，他说刚做了 **Load**，让他去查表空间状态，已经处于了 **backup pending**。然后继续问他，**load** 的命令怎么写的，是否有 **copy Yes** 等，他说使用的缺省 **copy no** 选项，并解释说他要加载的表其实是一张与业务无关的表，并且已经在测试机上做过测试，没有任何问题，才到生产上 **load** 的。我问他测试机上和生产库上数据库日志的模式，结果测试机上使用的是循环日志，生产上是归档日志模式。现在问题已经很明显了，他立即终止了 **load** 操作，但已经无济于事，因为 **load** 命令一旦发出，表空间就会立即处于 **backup pending** 状态，必

须对该表空间做备份才能使其脱离此状态。由于表空间很大，大概 150GB，备份花了将近 1 个半小时才完成，这段期间该交易系统处于停滞，造成了很恶劣的影响。对他最直接的影响就是写检查，并扣除年终奖。

对于该问题，有两个解决方案：第一种是使用 `copy yes to <path>` 备份 load 表数据；第二种是使用一个独立的表空间，将要加载的表放在该表空间，即使备份表空间，也不会影响其他表和业务。

最后，还是要提醒大家，对命令的使用一定要理解透彻，否则会造成难以挽回的损失。

提醒：在 load 操作前，一定要先检查当前数据库是否为可恢复数据库，即数据库日志是否归档。

8.5.4 set integrity 完整性检查 ■ ■ ■

相信大家还记得，在第 7 章我们讲了 5 种约束，即 NOT NULL 约束、唯一性约束、主键约束、参考完整性约束和检查约束。前面讲 load 的时候，我们提到，在装载阶段，不符合表定义的输入数据不会被装载到表中，如 NOT NULL 约束。在删除阶段，load 会删除违反唯一性约束的行。那么，对于表中包含了违背参照完整性约束和检查约束的数据，又该如何处理呢？load 程序不会检查这些约束，而是将表置于 set integrity pending 状态，当访问时，会提示以下错误：

```
SQL0668N Operation not allowed for reason code "1" on table "<table-name>".
SQLSTATE=57016
```

为解除这种状态，需要通过 set integrity 命令检查数据完整性，该命令语法如下：

```
>>-SET--INTEGRITY----->
      .-.-.-.-.-.
      V          |
>--+FOR---table-name--+--OFF--| access-mode-clause |--| cascade-clause |--+-----+><
|                                     +-FULL ACCESS-----+
|                                     '-PRUNE-----'
|      .-.-.-.-.-.
|      V          |
+-FOR---table-name--| table-checked-options |--IMMEDIATE CHECKED--+-----+
|                                     '-| check-options |--'
|      .-.-.-.-.-.
|      V          |
'-FOR---table-name--| table-unchecked-options |--IMMEDIATE UNCHECKED-----'
```

以下介绍几个重要选项的用法。

- OFF：将检查关闭，同时将表处于 set integrity pending 状态。
- IMMEDIATE CHECKED：对表立即做完整性检查，并将状态从 set integrity pending 状态中脱离出来。当将主表从 set integrity pending 状态中脱离时，它的依赖表可能会处于此状态，可通过返回的警告信息观察（SQLSTATE 01586）。
- IMMEDIATE UNCHECKED：不对表做检查，但将表从 set integrity pending 状态中脱离

出来。当表数据很大时，做完整性检查需要花费很长时间，因此如果能够确保加载的数据都是正确的，采用此选项将会大大降低执行时间。

对于处在 `set integrity pending` 状态下的表，如果包含异常数据，则必须创建异常表，才能使表脱离此状态。下面我们举例说明。

(1) 创建一张表 `t2`，在该表上增加检查约束，并加载数据，代码如下：

```
db2inst1@dpf1:/data1>$ db2 "create table t2 ( col1 char(10), col2 char(10))"
db2inst1@dpf1:/data1>$ db2 "alter table t2 add constraint check1 check( col2 in ('A',
'B', 'C') ) "
db2inst1@dpf1:~> cat t2.del
"AAA","A"
"BBB","B"
"CCC","CCC"
"DDD","BBB"
db2inst1@dpf1:/data1>$ db2 "load from t2.del of del insert into t2"
```

(2) 查询表 `t2`，发现处于 `SQL0668N`，即 `set integrity` 状态。检查 `t2` 的完整性，如果没有检测到数据异常，则将表脱离 `set integrity pending` 状态。如果检测到有数据违背完整性，则整个操作回滚，表仍处于 `set integrity pending` 状态，代码如下：

```
db2inst1@dpf1:/data1>$ db2 "select * from t2"
COL1          COL2
-----
SQL0668N  Operation not allowed for reason code "1" on table "DB2INST1.T2".
SQLSTATE=57016

db2inst1@dpf1:/data1>$ db2 set integrity for db2inst1.t2 immediate checked
DB21034E  The command was processed as an SQL statement because it was not a valid
Command Line Processor command. During SQL processing it returned:
SQL3603N  Check data processing through the SET INTEGRITY statement has found
integrity violation involving a constraint or a unique index with name
"DB2INST1.T2.CHECK1". SQLSTATE=23514
```

(3) 创建 `t2` 异常表，如果有数据违背完整性，则将其放入 `T2EXP` 异常表中。不管是否有数据异常，`t2` 都将脱离 `set integrity pending` 状态，代码如下：

```
db2inst1@dpf1:/data1>$ db2 "create table t2_exp like t2"
DB20000I  The SQL command completed successfully.

db2inst1@dpf1:/data1>$ db2 "set integrity for db2inst1.t2 immediate checked for
exception in t2 use t2_exp "
SQL3602W  Check data processing found constraint violations and moved them to
exception tables. SQLSTATE=01603

db2inst1@dpf1:/data1>$ db2 "select * from t2_exp"
COL1          COL2
-----
```

CCC	CCC
DDD	BBB

对于数据仓库系统来说，每天晚上都要加载多张表数据，如何查找哪些表处于 `set integrity pending` 状态呢？以下语句可用来检查表的状态、是否允许访问及表的主外键约束、检查约束等：

```
SELECT TABNAME, STATUS, ACCESS_MODE,
       SUBSTR(CONST_CHECKED,1,1) AS FK_CHECKED,
       SUBSTR(CONST_CHECKED,2,1) AS CC_CHECKED
FROM SYSCAT.TABLES WHERE STATUS= 'C'
```

对于参考完整性约束表，在主表上的 `set integrity` 命令会将其依赖表（迭代）置于 `set integrity pending` 状态，但如果该表有父表，则父表不会置于此状态。下例中，a 表是 b 表的子表，b 表是 c 表的子表，c 表是 d 表的子表，d 表是 e 表的子表。在 c 表上加载数据后，c 表会处于 `set integrity pending` 状态，而其他表并不会处于此状态。当在 C 上进行了 `set integrity` 检查后，它的子表 b 和 a 表会处于此状态。在这种主外键约束比较多的情况下，可能需要执行多次 `set integrity` 命令。

```
create table x.a (x int not null primary key);
create table x.b (x int not null primary key);
create table x.c (x int not null primary key);
create table x.d (x int not null primary key);
create table x.e (x int not null primary key);
alter table x.a add foreign key (x) references x.b;
alter table x.b add foreign key (x) references x.c;
alter table x.c add foreign key (x) references x.d;
alter table x.d add foreign key (x) references x.e;

load from /dev/null of ixf replace into x.c;  -- 在 c 表上加载数据，c 表应该处于 check
pending status
set integrity for x.c immediate checked;      -- 对 c 表做完整性检查，c 表的依赖表会处于
check pending
SQL3601W The statement caused one or more tables to automatically be placed in the
check pending state. SQLSTATE=01586

select tabname from syscat.tables where status = 'C'  --可通过此命令验证，A 和 B 都
处于此状态
a
b
```

为此，我们编写了如下迭代脚本 `set.sh`，希望对大家有帮助：

```
db2 connect to db1
db2 -tx +w "with gen(tabname, seq) as( select rtrim(tabschema) || '.' || rtrim(tabname)
as tabname, row_number() over (partition by status) as seq from syscat.tables WHERE
status='C' ), r(a, seq1) as (select CAST(tabname as VARCHAR(3900)), seq from gen
where seq=1 union all select r.a || ',' || rtrim(gen.tabname), gen.seq from gen ,
r where (r.seq1+1)=gen.seq ), r1 as (select a, seq1 from r) select 'SET INTEGRITY
FOR ' || a || ' IMMEDIATE CHECKED;' from r1 where seq1=(select max(seq1) from r1)"
> db2FixCheckPending.sql
```

```
db2 -tvf db2FixCheckPending.sql
```

./set.sh 执行后，会将 set integrity 表命令拼成一个语句，存到 db2FixCheckPending.sql。接连执行几次 set.sh 后，会将所有处于 set integrity pending 的表脱离此状态。

下面通过几个实例详细说明 set integrity 的其他用法。

例子 1:

将 T1 表置于 set integrity pending 状态、不允许访问状态，同时将它的依赖表置于 set integrity pending 状态，代码如下：

```
SET INTEGRITY FOR T1 OFF NO ACCESS CASCADE IMMEDIATE
```

例子 2:

不对 manager 表的外键约束和 employee 的检查约束做检查，并将该表脱离 set integrity pending 状态，代码如下：

```
SET INTEGRITY FOR MANAGER FOREIGN KEY, EMPLOYEE CHECK IMMEDIATE UNCHECKED
```

例子 3:

在表中已经有数据的情况下增加外键或检查约束，但表中存在违背约束的数据，因此创建约束不会成功。这时可通过 set integrity 命令先关闭约束检查，待创建完约束后，再进行数据完整性检查，代码如下：

```
db2inst1@dpf1:/data1>$ db2 "create table emp1 ( empno int not null primary key, empname
char(20), demptno int)"
db2inst1@dpf1:/data1>$ db2 "create table dept1 ( demptno int, deptname char(20) )"

db2inst1@dpf1:/data1>$ db2 "insert into emp1 values(111, 'zhangming', 1)"
db2inst1@dpf1:/data1>$ db2 "insert into emp1 values(222, 'ligang', 2)"
db2inst1@dpf1:/data1>$ db2 "insert into dept1 values(1, 'dev')"          -- dept1 表并
没有包含 deptno=2 的值

db2inst1@dpf1:/data1>$ db2 create table emp1_exp like emp1
db2inst1@dpf1:/data1>$ db2 SET INTEGRITY FOR emp1 OFF
db2inst1@dpf1:/data1>$ db2 "ALTER TABLE emp1 ADD FOREIGN KEY (deptno) REFERENCES dept1
(deptno) ON DELETE CASCADE"
db2inst1@dpf1:/data1>$ db2 SET INTEGRITY FOR emp1 IMMEDIATE CHECKED FOR EXCEPTION
IN emp1 USE emp1_exp
SQL3602W Check data processing found constraint violations and moved them to
exception tables. SQLSTATE=01603
```

做 set integrity 的时候可能会遇到日志满的情况。针对于不同的选项，set integrity 记录日志的方式不同，如 immediate unchecked 选项不会记日志，而 immediate checked 则会检查，当指定异常表，并有异常数据插入到异常表的时候，这些操作就会写日志以便必要的时候回滚。所以，对日志空间的占用依赖于异常数据的多少，如果是少量数据自然不会成为问题，而如果量很大

的情况，就可能出现日志满的情况。

8.6 12 个怎么办

8.6.1 出现了 load pending 了怎么办 ■ ■ ■

如果在提交数据之前，表上的正在执行的 load 操作被异常终止，那么该表就处于 load pending 状态。若要使该表恢复到正常状态，则需要调用 load terminate、load restart 或 load replace 操作。

让我们来模拟一个 load pending。

首先创建一个大小为 256 页的表空间，然后在其中建一张表，代码如下：

```
db2inst1@dpf1:~> db2 "create bufferpool bp8k size automatic"
db2inst1@dpf1:~> db2 "create tablespace ts3 pagesize 8k managed by database using
(file '/data1/ts2' 256) bufferpool bp8k"
db2inst1@dpf1:~> db2 "create table t1 (id int, name char(50), desc char(50) ) in ts1"
```

创建如下的存储过程：

```
db2inst1@dpf1:~> cat sp_insert.sql
CREATE PROCEDURE sp_insert (IN count int)
LANGUAGE SQL
BEGIN
    DECLARE i INTEGER DEFAULT 0;
    while i<count
    do
        insert into t1 values( i, 'abcdefghixxxxxxxxxxxx' || char(i),
'bbbbbbbbbbbbbbbbbbbb' || char(i) );
        set i=i+1;
    end while;
END
@
```

执行存储过程插入数据，然后将数据导出到 t1.del 文件中，代码如下：

```
db2inst1@dpf1:~> db2 -td@ -f sp_insert.sql
db2inst1@dpf1:~> db2 "call sp_insert(6500) "
db2inst1@dpf1:~> db2 "export to t1.del of del select * from t1"
```

重建一个大小为 232 页的表空间，并将 t1.del 数据加载，由于无法分配新页，导致 load 出现异常。通过 load query 命令返回的信息显示，t1 表处于 load pending 状态，查询时报 SQL0668N 错误，reason code 为 3，代码如下：

```
db2inst1@dpf1:~> db2 drop tablespace ts1
```

```
db2inst1@dpf1:~> db2 "create tablespace ts3 pagesize 8k managed by database using
(file '/data1/ts2' 232) bufferpool bp8k"
db2inst1@dpf1:~> db2 "create table t1 (id int, name char(50), desc char(50) ) in ts1"
db2inst1@dpf1:~> db2 "load from t1.del of del insert into t1"
db2inst1@dpf1:~> db2 load query table t1
Tablestate:
  Load Pending

db2inst1@dpf1:~> db2 "select * from t1"
ID          NAME          DESC
-----
SQL0668N  Operation not allowed for reason code "3" on table "DB2INST1.T1".
SQLSTATE=57016
```

在执行 **load terminate** 操作之后，该表就重新处于 **normal** 状态，代码如下：

```
db2inst1@dpf1:~> db2 "load from /dev/null of del terminate into t1"
db2inst1@dpf1:~> db2 "select * from t1"
ID          NAME          DESC
-----
0 record(s) selected.
```

可通过如下语句查找处于 **load pending** 状态的表，代码如下：

```
db2inst1@dpf1:~> db2 "select substr(tabschema,1,10) as tabschema,substr(tabname,
1,20) tabname from sysibmadm.admintabinfo where load_status='PENDING'"
TABSHEMA  TABNAME
-----
DB2INST1  T1
```

8.6.2 在客户端 load 问题 ■ ■ ■

如果要加载的数据在客户端，可通过 **Load client from** 语句加载数据。

当从客户端远程 **load** 数据时，当远程路径无效时，可能会发生 **load in progress** 状态，阻塞其他应用对该表的读/写操作，代码如下：

```
D:\temp>db2 "load client from d:\temp\xxxx of del insert into t1"
SQL2036N  文件或设备 "d:\temp\xxxx" 的路径无效。

D:\temp>db2 "load client from d:\temp\xxxx.txt of del insert into t1"
SQL0668N  不允许对表 "DB2INST1.T1" 执行操作，原因码为 "5"。  SQLSTATE=57016
```

在服务端执行 **load query table** 检查状态，发现处于 **load in progress** 状态，代码如下：

```
db2inst1@dpf1:~> db2 load query table t1
SQL2036N  The path for the file or device "d:\temp\xxxx" is not valid.
SQL3532I  The Load utility is currently in the "UNKNOWN" phase.
Tablestate:
  Load in Progress
```

在客户端执行 `terminate` 终止 `load` 操作，问题解决。

```
D:\temp>db2 "load client from d:\temp\xxxx.txt of del terminate into t1"
SQL3501W 由于禁用数据库正向恢复，因此表所驻留的表空间将不被置于备份暂挂状态。
```

注意：如果是 `insert` 操作，通过 `terminate` 会恢复到初始状态；如果是 `replace` 操作，出现刚才的问题就麻烦了，表会变成空表。

8.6.3 要加载的数据是 Excel 格式怎么办 ■ ■ ■

处理 Excel 数据，可将其另存为“CSV（逗号分隔）”格式，然后按照前述方法导入，如图 8.3 所示。

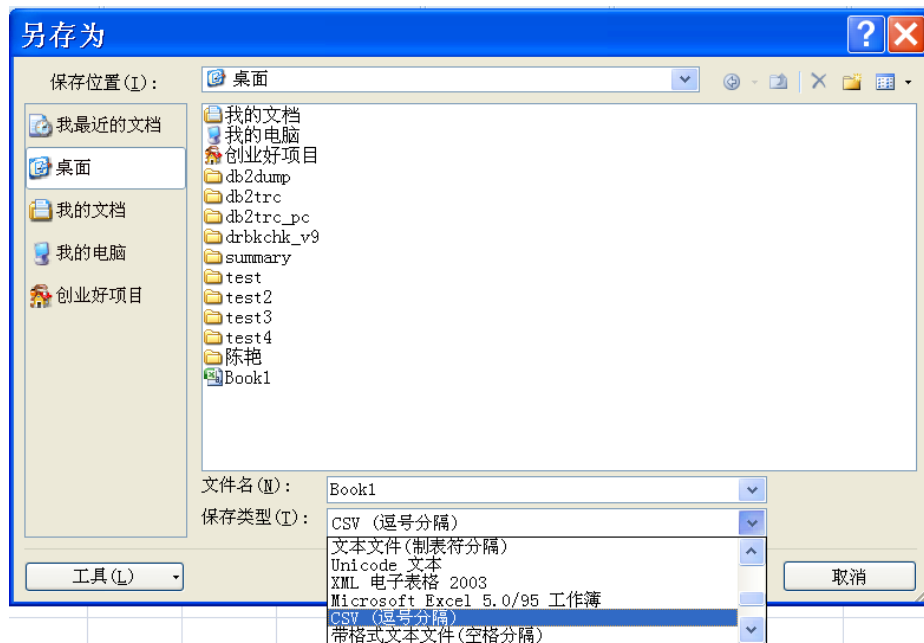


图 8.3 另存为“CSV（逗号分隔）”格式

8.6.4 要导出/加载的数据不是逗号/双引号分隔怎么办 ■ ■ ■

在默认情况下，对于 `DEL` 格式，每行数据字段之间用逗号分隔，字符串数据用双引号分隔。如果提供的不是这种格式，如字段间用分号“;”分隔，字符串用单引号“'”分隔，那么可用 `modified by` 修饰符来指定，`coldelx` 用来指定字段间分隔符，`chardelx` 用来指定字符串分隔符，代码如下：

```
db2inst1@dpf1:~> cat t1.del
10;'wang qi';1
20;'zhang san';2
db2inst1@dpf1:~> db2 "import from t1.del of del modified by coldel; chardel'" insert
```

```
into t1"
```

注意：对于分隔符，有以下限制：

- 用户要确保数据中不能包含分隔符，如果存在，会出现各种异常。
- 分隔符不能是换行符、回车符、0x00 或空格。
- 在 DBCS 环境下（如中文系统），不能用 “|” 符号作为分隔符。
- 如果 DEL 文件通过某个特殊字符分隔，而 import、load、export 命令中通过键盘无法敲入时，可转换为十六进制表示，如 0x7c 代表 “|”。

对于 decimal 值，默认的导出格式是在前面有+号，如果位数不够定义的长度，则用 0 补齐，代码如下：

```
db2inst1@dpf1:~> db2 "create table t2 (col1 char(10), col2 decimal (7,2) )"
db2inst1@dpf1:~> db2 "insert into t2 values('aaa', 23.4 )"
db2inst1@dpf1:~> db2 "insert into t2 values('bbb', 1.2 )"
db2inst1@dpf1:~> db2 "export to t2.del of del select * from t2"
db2inst1@dpf1:~> cat t2.del
"aaa      ",+00023.40
"bbb      ",+00001.20
```

有些情况下，客户不希望保留+号，或者去掉前面的 0，DB2 提供了两个修饰符选项，一个是 decplusblank(del plus blank，即+换成空格)，另一个是 striplzeros (strip zero，去掉空格)。

```
db2inst1@dpf1:~> db2 "export to t21.del of del modified by DECPLUSBLANK STRIPLZEROS
select * from t2"
db2inst1@dpf1:~> cat t21.del
"aaa      ", 23.40
"bbb      ", 1.20
```

对日期格式的支持也比较灵活，如客户指定了导出的时间格式，那么可以通过 modified by timestampformat 来指定。

```
db2inst1@dpf1:~> db2 "create table t3 (col1 char(10), col2 timestamp) "
db2inst1@dpf1:~> db2 "insert into t3 values('aaaa', current timestamp) "
db2inst1@dpf1:~> db2 "export to t3.del of del select * from t3"
db2inst1@dpf1:~> cat t3.del
"aaaa      ", "2011-01-12-09.16.07.630919"
```

如果客户要求导出的时间按照“年/月/日 时:分:秒.毫秒”的格式，可在 export 命令中指定（注意：YYYY 前面的 “\” 用于对 “” 转义），结果如下：

```
db2inst1@dpf1:~> db2 "export to t3.del of del modified by
timestampformat=\"YYYY/MM/DD HH:MM:SS.UUUUUU\" select * from t3"
db2inst1@dpf1:~> cat t3.del
"aaaa      ", "2011/01/12 09:16:07.630919"
```

8.6.5 文件中的列比要导入的表中的字段多怎么办 ■ ■ ■

针对这种情况，一般有以下两种处理方式：

- 修改数据文件，删除多余数据。可以利用 UNIX 的 sed、awk 等工具进行文件内容处理，但如果文件数据量特别大的情况，处理的时间和工作量会比较大。
- 对于 DEL 格式，用 import、load 的 method P 选项；对于 ASC 格式，用 Method L 选项；对于 IXF 格式，用 Method N 选项。我们以 DEL 格式为例，代码如下：

```
db2inst1@dpf1:~> db2 "create table t4 (col1 char(10), col2 int ) "
```

```
db2inst1@dpf1:~> cat t4.del
```

```
"aaaa","bbbb",2,"abc",3
```

```
"cccc","dddf",4,"aaa",4
```

```
"dddd","dda",5,"abcd",5
```

```
db2inst1@dpf1:~> db2 "import from t4.del of del method P(1,3) insert into t4"
```

```
db2inst1@dpf1:~> db2 "select * from t4"
```

COL1	COL2
aaaa	2
cccc	4
dddd	5

8.6.6 文件中的列比要导入的表中的字段少怎么办 ■ ■ ■

与前一种情况类似，仍然使用 Method P 选项，但不同的是，对于不存在的数据，可以使用一个不存在的字段位置。下例中，Method P(1,99,2)，输入文件中第 1 个和第 3 个字段内容对应表的第 1 和第 3 个字段，而用空值补充第 2 个字段（因为文件中不存在第 99 列，所以 DB2 用 NULL 值替代）。

```
db2inst1@dpf1:~> db2 "create table t3 (c1 int, c2 int, c3 int)"
```

```
db2inst1@dpf1:~> db2 "select * from t3"
```

C1	C2	C3

```
0 record(s) selected.
```

```
db2inst1@dpf1:~> more t3.del
```

```
1,2
```

```
db2inst1@dpf1:~> db2 "load from test.del of del method P(1,99,2) insert into t3"
```

```
SQL3501W The table space(s) in which the table resides will not be placed in
```

```
backup pending state since forward recovery is disabled for the database.
```

```
...
```

```
db2inst1@dpf1:~> db2 "select * from t3"
```

```

C1          C2          C3
-----
          1          -          2

1 record(s) selected.

```

8.6.7 要导入/导出大字段 (LOB) 怎么办 ■ ■ ■

我们发现，很多客户使用大对象 (LOB) 来存储图片、大文本等内容。以下演示了两种大对象数据导出的方法。

第一种方法是通过 `modified by lobsinfile` 将每个大对象字段的所有行数据存放在 `lobs to` 指定的文件中，然后在导出文件(通过 `export to` 指定的文件)中通过指针来定位大对象数据在指定文件中的位置偏移和长度。

在下例中，表 `emp_resume` 的 `RESUME` 字段为 `CLOB` 类型，将数据导出到 `emp_resume.del` 文件中，大对象数据导出到 `/data1/lobs` 目录下。观察 `emp_resume.del` 文件内容可知，大对象数据通过位置偏移和长度来定位在大对象文件的位置，代码如下：

```

db2inst1@dpf1:/data1/lobs> db2 describe table emp_resume
Column name      Data type name    Column Length
-----
EMPNO            CHARACTER          6
RESUME_FORMAT    VARCHAR            10
RESUME           CLOB              5120

db2inst1@dpf1:/data1> db2 "export to emp_resume.del of del lob to /data1/lobs
modified by lobsinfile select * from emp_resume "

db2inst1@dpf1:/data1> cat emp_resume.del
"000130","ascii","emp_resume.del.001.lob.0.1257/"
"000130","html","emp_resume.del.001.lob.1257.2415/"
"000140","ascii","emp_resume.del.001.lob.3672.1261/"
"000140","html","emp_resume.del.001.lob.4933.2440/"
"000150","ascii","emp_resume.del.001.lob.7373.1308/"
"000150","html","emp_resume.del.001.lob.8681.2467/"
"000190","ascii","emp_resume.del.001.lob.11148.1237/"
"000190","html","emp_resume.del.001.lob.12385.2402/"

db2inst1@dpf1:/data1> cd lob
db2inst1@dpf1:/data1/lobs> ls
emp_resume.del.001.lob

```

第二种方法是通过 `modified by lobsinsefiles` 将每一行的大对象数据存放在独立文件中，在导出文件（通过 `export to` 指定的文件）中指定文件名。在下例中，表 `emp_resume` 的 `RESUME` 字段为 `CLOB` 类型，将数据导出到 `emp_resume1.del` 文件中，大对象数据导出到 `/data1/lobs` 目录下，代码如下：

```

db2inst1@dpf1:/data1> db2 "export to emp_resume1.del of del lobs to /data1/lobs
modified by lobsinsepfiles select * from emp_resume "

db2inst1@dpf1:/data1> cat emp_resume1.del
"000130","ascii","emp_resume1.del.001.lob"
"000130","html","emp_resume1.del.002.lob"
"000140","ascii","emp_resume1.del.003.lob"
"000140","html","emp_resume1.del.004.lob"
"000150","ascii","emp_resume1.del.005.lob"
"000150","html","emp_resume1.del.006.lob"
"000190","ascii","emp_resume1.del.007.lob"
"000190","html","emp_resume1.del.008.lob"

db2inst1@dpf1:/data1> cd lobs
db2inst1@dpf1:/data1/lobs> ls
emp_resume1.del.001.lob      emp_resume1.del.003.lob      emp_resume1.del.005.lob
emp_resume1.del.007.lob
emp_resume1.del.002.lob      emp_resume1.del.004.lob      emp_resume1.del.006.lob
emp_resume1.del.008.lob

```

8.6.8 sequence 数据怎么办 ■ ■ ■

如果目标表的值是通过 **sequence** 插入的，而初始数据来源于数据文件。那么加载完数据后，需要对 **sequence** 的初始值进行重置，代码如下：

```

SELECT MAX(<id>) AS counter FROM <Table_name>
ALTER SEQUENCE <Seq_name> RESTART WITH counter+1

db2inst1@dpf1:/data1> db2 "create sequence seq1 as integer start with 1 increment
by 1 minvalue 1 maxvalue 999999999 cycle cache 10"
db2inst1@dpf1:/data1> cat seq.del
1,"AAAA"
2,"bbbb"
3,"ccc"

db2inst1@dpf1:/data1> db2 "create table test2 ( id int, name char(20) )"
db2inst1@dpf1:/data1> db2 "load from seq.del of del insert into test2"
db2inst1@dpf1:/data1> db2 "select * from test2"

ID          NAME
-----
1  AAAA
2  bbbb
3  ccc

record(s) selected.

Db2inst1@dpf1:/data1> db2 "select max(id) as max from test2"

```

```

MAX
3

db2inst1@dpf1:/data1> db2 "alter sequence seq1 restart with 4"
db2inst1@dpf1:/data1> db2 "insert into test2 values(nextval for seq1, 'ddddddd') "
db2inst1@dpf1:/data1> db2 "select * from test2"
ID          NAME
-----
      4 ddddddd
      1 AAAA
      2 bbbb
      3 ccc

4 record(s) selected.

```

8.6.9 导入 identity 数据怎么办 ■ ■ ■

在数据加载过程中，经常遇到 **identity** 标识列数据处理出现问题。比如，某张表含有 **identity** 自增字段，而且表中包含一定数据。这时需要把某文件数据 **import/load** 到这张表中，那么自增字段的值是继续增加呢？还是重置呢？还是忽略呢？针对不同的需求，DB2 提供了两组修饰符来解决。一组为 **identity** 修饰符，包含 **identityoverride**、**identityignore** 和 **identitymissing** 3 个，用于当表字段有 **generated always as identity** 或 **generated by default as identity** 自增列时数据加载处理方法；另外一组为 **generated** 修饰符，包含 **generatedoverride**、**generatedignore** 和 **generatedmissing** 3 个，用于当表字段有 **generated always as <expression>** 时数据加载处理方法。

identityoverride 表示使用数据文件中提供的值，特别适用于系统间数据迁移时，需要保留 **identity** 列值的情况；**identitymissing** 表示当数据文件中不包含自增值时，**import/load** 数据产生自增值；**identityignore** 命令表示当数据文件中包含自增值时，**import/load** 命令忽略该值，并生成自增值。所以可以根据数据文件和用户需求，决定采用哪个修饰符。

注意：**identityoverride** 修饰符只适用于 **generated always**，不支持 **generated by default**。

identityoverride 不能在 **import** 命令中使用，只在 **load** 中支持。

我们举例说明这 3 个修饰符的使用。

例子 1：在 **load** 数据时，指定 **identityoverride** 会使用输入文件中标识列的值；任何标识列上值为空（或为 **NULL** 值）的行都会被拒绝。

```

db2inst1@dpf1:/data1> cat t6.del
3,"Shrek"

db2inst1@dpf1:/data1> db2 "create table t6 (custno smallint not null generated always
as identity (start with 500, increment by 1), custname varchar(16)) "

db2inst1@dpf1:/data1> db2 "load from t6.del of del modified by identityoverride
messages t6.msg insert into t6 "

```

```
db2inst1@dpf1:/data1> db2 "select * from t6"
CUSTNO      CUSTNAME
-----
3           Shrek
```

例子 2: 通过 `import` 或 `load` 加载数据时, 输入文件不包含任何目标表中生成列的值, 加载过程会生成自增值。

```
db2inst1@dpf1:/data1> cat t7.del
MingWei

db2inst1@dpf1:/data1> db2 "create table t7 (custno smallint not null generated always
as identity (start with 500, increment by 1), custname varchar(16)) "

db2inst1@dpf1:/data1> db2 "load from t7.del of del modified by identitymissing
messages t7.msg insert into t7"

db2inst1@dpf1:/data1> db2 "select * from t7"
CUSTNO      CUSTNAME
-----
500         MingWei
```

例子 3: 当通过 `import` 或 `load` 数据时, 输入文件中任何生成列的值都会被忽略, 并且会为每一行生成一个新的值。

```
db2inst1@dpf1:/data1> cat t8.del
1,"WangTao"

db2inst1@dpf1:/data1> db2 "create table t8 (custno smallint not null generated always
as identity (start with 500, increment by 1), custname varchar(16))"

db2inst1@dpf1:/data1> db2 "load from t8.del of del modified by identityignore messages
load.msg insert into t8"

db2inst1@dpf1:/data1> db2 "select * from t8"
CUSTNO      CUSTNAME
-----
500         WangTao
```

例子 4: 对于含有 `generated by default as identity` 自增列的表数据导入, 可能会遇到加载完成后, 再插入数据时产生重复值的情况。举例来说, 表中 `ID` 字段为 `generated by default as identity(start with 500)`, 当通过 `import/load` 加载了 1 万行数据后, 通过 `insert` 插入数据时, 自动产生的自增值可能还是会从 500 开始, 即自增分配值与原值重复 (注意: 这个问题并不是总会出现, 当出现时我们知道怎么做就可以了)。

解决办法: 首先找到表中标识列的最大值, 然后更改表定义, 使得标识列的值从最大值+1 开始, 代码如下:

```
SELECT MAX(<IDENTITY column>) AS maxcounter from <table name>
ALTER TABLE <table name> ALTER COLUMN <IDENTITY column> RESTART WITH < maxcounter
+ 1>
```

举例如下：

```
db2inst1@dpf1:/data1> db2 "create table t9 (custno smallint not null generated by
default as identity (start with 500, increment by 1), custname varchar(16))"

db2inst1@dpf1:/data1> db2 "LOAD FROM t9.del OF DEL modified by identityignore INSERT
INTO t9"

db2inst1@dpf1:/data1> db2 "select * from t9"
CUSTNO CUSTNAME
-----
500 xusi
501 wangsang

db2inst1@dpf1:/data1> db2 "insert into t9(custname) values('afadf') "
db2inst1@dpf1:/data1> db2 "select * from t9"
CUSTNO CUSTNAME
-----
500 afadf
500 xusi
501 wangsang
```

发现新插入的行，custno 仍然从 500 开始，这不是我们希望的。通过 alter table ... restart with 解决问题，代码如下：

```
db2inst1@dpf1:/data1> db2 "SELECT max(custno) as MAXID FROM t9"
MAXID
-----
501

db2inst1@dpf1:/data1> db2 "alter table t9 alter custno restart with 502"
db2inst1@dpf1:/data1> db2 "insert into t9(custname) values('test4') "
db2inst1@dpf1:/data1> db2 "select * from t9"
CUSTNO CUSTNAME
-----
500 afadf
500 xusi
501 wangsang
502 test4
```

例子 5：对于 generated always <expression> 的标识列，就要用 generated 修饰符了。使用方法同前述 identity 修饰符，我们举例说明，代码如下：

```
db2inst1@dpf1:/data1> cat t10.del
"Crystal",102000.00
db2inst1@dpf1:/data1> db2 "create table t10 (name varchar(16) not null, salary
decimal(9,2), bonus decimal(9,2) generated always as (salary/10) )"
```

```
db2inst1@dpf1:/data1> db2 "load from t10.del of del modified by generatedmissing
messages t10.msg insert into t10"

db2inst1@dpf1:/data1> db2 "select * from t10"
NAME          SALARY          BONUS
-----
Crystal       102000.00       10200.00
```

8.6.10 要加载的数据有换行符怎么办 ■ ■ ■

如何处理数据中含有换行符的问题，本来是一条数据，取的时候就变成两条了，造成再往库里 load 的时候就会报错。

举例如下：通过 quest central 插入 2 行数据（在命令行下很难模拟换行插入），如图 8.4 所示。

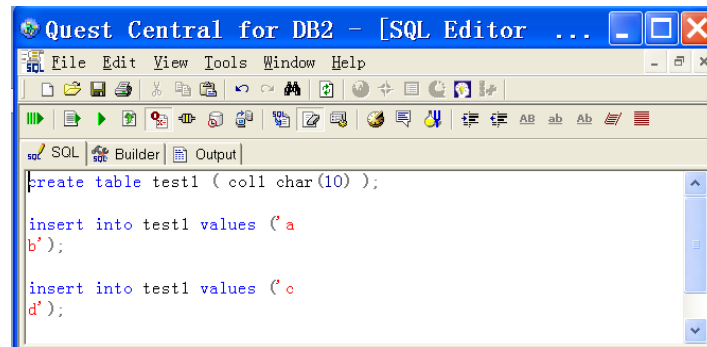


图 8.4

查询结果发现，共有 2 行数据：

```
db2inst1@dpf1:/data1> db2 "select * from test1"
COL1
-----
a
b
c
d

2 record(s) selected.
```

将数据导出到文本文件，结果如下：

```
db2inst1@dpf1:/data1> db2 "export to test.del of del select * from test1"
Number of rows exported: 2

db2inst1@dpf1:/data1> cat test.del
"a
```

```
b      "
"c
d      "
```

如果采用默认的 load 选项加载到表中，那么会出现 4 条记录，这明显是有问题的，代码如下：

```
db2inst1@dpf1:/data1> db2 "create table test2 like test1"
db2inst1@dpf1:/data1> db2 "load from test.del of del insert into test2"
db2inst1@dpf1:/data1> db2 "select * from test2"
COL1
-----
"a
b      "
"c
d      "
4 record(s) selected.
```

为解决此问题，我们提供如下几种解决方案：

(1) 如果允许的话，导出的时候替换换行符，代码如下：

```
db2inst1@dpf1:/data1> db2 "export to test1.del of del select replace(coll,chr(10),'')
from test1"
SQL3104N The Export utility is beginning to export data to file "test1.del".
SQL3105N The Export utility has finished exporting "2" rows.
Number of rows exported: 2

db2inst1@dpf1:/data1> cat test1.del
"ab      "
"cd      "
```

(2) 如果允许的话，通过脚本对输入数据进行清洗，去掉换行符，代码如下：

```
db2inst1@dpf1:/data1> cat test.del
"a
b      "
"c
d      "

db2inst1@dpf1:/data1> sed 'N;s/\n//' test.del
"ab      "
"cd      "
```

(3) 在 load/import 时指定 modified by delprioritychar 修饰符，代码如下：

```
db2inst1@dpf1:/data1> db2 "load from test.del of del modified by DELPRIORITYCHAR
insert into test2"
Number of rows read          = 2
Number of rows skipped      = 0
Number of rows loaded       = 2
Number of rows rejected     = 0
Number of rows deleted      = 0
```

```

Number of rows committed      = 2

db2inst1@dpf1:/data1> db2 "select * from test2"
COL1
-----
a
b
c
d
2 record(s) selected.

```

DB2 默认 import/load 优先级策略为：行分隔符（record delimiter）、字符分隔符（character delimiter）和列分隔符（column delimiter），行分隔符优先级最高。所以如果原始文件有换行的话，load 就认为是新的记录，这时可用 `delprioritychar` 修饰符改变默认的优先级别，确保""之间的数据不管有没有换行符都被认为是同一条记录。

注意：用 IXF 不会出现此问题。

8.6.11 迁移出现乱码怎么办 ■ ■ ■

对于 IXF 格式，一般不会出现编码问题。因为会发生自动编码转换。对于 DEL 格式，当在不同系统间 export/import/load 数据时，由于数据的编码不同，可能会遇到汉字乱码问题，而 import 和 load 对数据文件的编码处理也有很大的差异。数据库编码在建库时指定，可通过 `db2 get db cfg` 查看。

在默认情况下，load 假定输入文件用数据库 codepage 编码，并将文件转换为数据库 codepage 编码来导入。如果输入文件的 codepage 与数据库编码不一样，那么就会将错误的数据装载到数据库中。因此对于数据文件和数据库的 codepage 不同的情况，可以使用 `modified by codepage=<codepage>` 选项，显式地将数据文件 codepage 告诉 load 程序，确保 DB2 能够正确地执行编码页转换。

在默认情况下，import 认为输入文件中的数据是用当前系统的代码页编码（通过 `db2set db2codepage=<codepage>` 设置，如 1386、1208 等）。当将数据文件导入到数据库时，DB2 会自动将数据文件从当前系统代码页转换成数据库代码页。如果输入文件不是当前系统的代码页编码，也可以通过 `codepage` 修饰符来轻松导入正确文件。

下面的例子很清晰地说明了这个过程。

（1）首先通过 `db2 get db cfg` 查看当前数据库编码集是 UTF-8，codepage 是 1208，在当前编码下插入中文数据，代码如下：

```

db2inst1@dpf1:~> db2 "insert into t1 values('10000','徐明伟' )"
db2inst1@dpf1:~> db2 "select * from t1"

EMPNO      EMPNAME

```

```
-----
10000      徐明伟
```

(2) 然后将数据以不同编码导出，在 **export** 时，如果不指定导出数据编码，则按数据库编码导出。如果通过 **modified by codepage** 指定了编码，则按指定的编码导出，这样我们就可以构造几种不同编码的数据，用来测试 **load** 和 **import** 加载过程，代码如下：

```
db2inst1@dpf1:~> db2 "export to t1 of del select * from t1"
db2inst1@dpf1:~> db2 "export to t1_819 of del modified by codepage=819 select * from t1"
db2inst1@dpf1:~> db2 "export to t1_1208 of del modified by codepage=1208 select * from t1"
db2inst1@dpf1:~> db2 "export to t1_1386 of del modified by codepage=1386 select * from t1"
```

(3) 清空当前表数据，然后按顺序以不同编码将数据 **load**，代码如下：

```
db2inst1@dpf1:~> db2 delete from t1
db2 "load from t1_819 of del modified by codepage=819 insert into t1"
db2 "load from t1_1208 of del modified by codepage=1208 insert into t1"
db2 "load from t1_1386 of del modified by codepage=1386 insert into t1"
db2 "load from t1_1386 of del insert into t1"

db2inst1@dpf1:~> db2 "select * from t1"

EMPNO      EMPNAME
-----
10000      -- 按 819 编码的数据无法导入，这是因为导出的文件已经发生了乱码(819
不支持汉字)
10000      徐明伟      -- 按 1208 编码数据导入，正确
10000      徐明伟      -- 指定 1386 编码，正确
10000      徐明伟      -- 1386 编码数据，但没有指定 codepage，出现乱码

4 record(s) selected.
```

(4) **import** 对编码的处理过程如下：

```
db2inst1@dpf1:~> db2set db2codepage=1386      --将系统编码改为 1386
db2inst1@dpf1:~> db2 terminate
db2inst1@dpf1:~> db2 connect to sample
db2inst1@dpf1:~> db2 "import from t1_1386 of del insert into t1" -- 输入文件和系统
编码一致，可导入
db2inst1@dpf1:~> db2 "import from t1_1208 of del insert into t1" -- 输入文件和系统
编码不一致，出现乱码

db2inst1@dpf1:~> db2set db2codepage=1208      --将系统编码改为 1208
db2inst1@dpf1:~> db2 terminate
db2inst1@dpf1:~> db2 connect to sample
db2inst1@dpf1:~> db2 "import from t1_1386 of del insert into t1" -- 输入文件和系统
编码不一致，出现乱码
```

```
db2inst1@dpf1:~> db2 "import from t1_1386 of del modified by codepage=1386 insert
into t1" -- 输入文件和系统编码不一致，通过 codepage 修饰符通知 import 输入文件的编码是 1386
```

我们举这个例子的目的就是要告诉大家，当遇到编码的时候，可通过指定 `modified by codepage` 来告诉 `import` 或 `load` 执行正确的编码转换。

8.6.12 表数据从一个表空间迁移到另外一个表空间怎么办 ■ ■ ■

表数据在不同表空间的迁移是很多客户遇到的问题，原因有以下几个：

- 表空间页大小初始设置是 4K 页，随着业务需求的变化，某张表（假设为 t1）的一行记录大小需要超过 4K，当前表空间页大小无法满足需求。
- 在 V8 版本，4K 页大小的表空间最大限制为 64GB。当数据量达到此限制时是无法插入数据的。
- 性能考虑，需要将某些表从一个表空间迁移到另外一个表空间。

对于此类问题，典型做法是新建一个表空间，在此表空间创建与原表相同的表结构（假设为 t2），然后将原表 t1 数据导出，再导入到 t2 表，完成测试后，将 t1 表改名或删除，将 t2 表改为 t1。如果要迁移的表很多，可考虑写成脚本。注意：如果在 T1 表上有主外键约束等，则需要先删除，否则无法 `rename`。

举例如下：

```
db2inst1@dpf1:/data1/data> db2 "create tablespace ts2"
db2inst1@dpf1:/data1/data> db2 "create table t2 like t1 in ts2"
db2inst1@dpf1:/data1/data> db2 "export to t1.del of del select * from t1"
db2inst1@dpf1:/data1/data> db2 "load from t1.del of del insert into t2"
db2inst1@dpf1:/data1/data> db2 rename t1 to t1_bak
db2inst1@dpf1:/data1/data> db2 rename t2 to t1
```

可以看到，上述解决方案操作比较烦琐，而且要求在操作时 t1 是离线的，即不允许应用进行增、删、改操作，因此只能在特定的运维时间窗口内操作，如果表数据量很大，在指定的时间无法完成的话，则对应用的影响较大。

为增加系统的可用性，9.7 版本引入了在线数据迁移方案，通过 `ADMIN_MOVE_TABLE` 存储过程将表数据从一个表空间迁移到另外一个表空间，迁移过程中保持对原表数据的持续访问。

`ADMIN_MOVE_TABLE` 过程有两种等价的调用方式。第一种是提供目标表的数据、索引、大对象表空间名；第二种事先建一个表，建表时指定数据、索引、大对象表空间，在调用时提供表名而不需要提供表空间名，迁移完成后，表名替换为原表名。

对于在线表迁移操作，需要以下了个表。

- 原表（source table）：要迁移的表名。

- 目标表 (target table): 迁移的目标, 如果是第一种调用, 则这张表会在指定的表空间自动创建, 如果是第二种调用, 目标表已经建好, 作为参数传给存储过程即可。在迁移过程中, 仍然可以在原表上做数据的 insert、update 和 delete 操作, 通过 Trigger 机制来捕获变化数据, 并同步到目标表上, 为此需要创建第三张表。
- 中间表 (staging table): 记录迁移过程中在原表上产生的更新数据, 这些变化数据随后被同步到目标表上。迁移完成后, 这张表会删除。



图 8.5

在线表迁移操作包括 5 个阶段: INIT、COPY、REPLAY、SWAP 和 CLEANUP, 如图 8.5 所示。

(1) INIT 阶段: 初始化阶段将验证调用者是否有权限进行表迁移; 根据调用方法验证或创建目标表; 创建中间表; 在原表上创建 4 个触发器, 这样对原表的 insert、update、delete 操作都将触发触发器, 并将变化记录到中间表中, 对 update 操作, 更新前和更新后的值都将记录。

(2) COPY 阶段: 复制阶段就是从原表复制数据到目标表, 当复制时原表仍然可访问。复制数据的方式有两种, 一种是 insert <Target> select * from <Source>, 另一种是采用 load...from cursor 的方式。复制的最后, 会在目标表上创建和原表一致的索引。复制过程中对原表的变更都将记录到中间表中。

(3) REPLAY 阶段: REPLAY 阶段会遍历中间表, 将数据写到目标表上。根据原数据的变更情况, REPLAY 可能需要做很多次。

(4) SWAP 阶段: 此阶段将交换原表和目标表表名, 然后删除原表。在交换表名之前, 需要确保两边数据完全一致。因此, 在最后一次 REPLAY 时, 要在原表和目标表上加 x 锁, 确保没有新数据产生, 这样保证了两表数据的完全一致。然后删除原表, rename 目标表到原表, 并释放锁。

(5) CLEANUP 阶段: 最后的清理阶段将清除一些临时对象; 删除第一步创建的触发器, 并删除中间表。

以下是一个将 t4 表从 ts1 表空间迁移到 TBS32K 表空间的例子, 根据存储过程结果可以明显地看到各个阶段的起始和完成时间, 代码如下:

```

db2inst1@dpf1:~> db2 create tablespace ts1
db2inst1@dpf1:~> db2 "create table t4 (col1 char(100), col2 int) in ts1"
db2inst1@dpf1:~> db2 create bufferpool bp32k size automatic pagesize 32k
db2inst1@dpf1:~> db2 create tablespace tbs32k pagesize 32k bufferpool bp32k
db2inst1@dpf1:~> db2 "call sysproc.admin_move_table('DB2INST1', 'T4', 'TBS32K',
'TBS32K', 'TBS32K', '','', '','', 'MOVE') "
```

```

Result set 1
-----
KEY                                VALUE
-----
AUTHID                            DB2INST1
CLEANUP_END                        2010-10-30-08.12.50.656272
CLEANUP_START                      2010-10-30-08.12.50.626811
COPY_END                          2010-10-30-08.12.49.560915
COPY_OPTS                         ARRAY_INSERT,NON_CLUSTER
COPY_START                        2010-10-30-08.12.49.281178
COPY_TOTAL_ROWS                   10000
INDEXNAME
INDEXSCHEMA
INIT_END                          2010-10-30-08.12.49.122941
INIT_START                        2010-10-30-08.12.48.219311
REPLAY_END                        2010-10-30-08.12.50.504815
REPLAY_START                      2010-10-30-08.12.49.561244
REPLAY_TOTAL_ROWS                 0
REPLAY_TOTAL_TIME                 0
STATUS                            COMPLETE
SWAP_END                          2010-10-30-08.12.50.596472
SWAP_RETRIES                      0
SWAP_START                        2010-10-30-08.12.50.543877
VERSION                           09.07.0001

20 record(s) selected.
Return Status = 0

```

使用建议：

- 避免在同一个表空间上同时进行多个表迁移，否则在表空间上可能会产生比较多的碎片。
- 在空闲时进行迁移，因为迁移过程产生的大量 I/O，可能会影响对该表的并发访问。
- 使用多步迁移步骤。前面我们讲的 5 个步骤可以分步执行，比如 INIT 和 COPY 阶段可以随时调用，尽量多次执行 REPLAY 保持中间表的数据最少，在表几乎不活动时进行 SWAP。
- 评估是否可以采用离线迁移方法。
- 在线表数据迁移依赖于触发器来捕获迁移过程中变化的数据，所以不触发 Trigger 的一些操作会导致两边数据的不一致，如 load、alter，reorg 等。建议在在线迁移时避免进行以上操作。
- 考虑增加表空间大小，因为迁移过程会产生两份数据和索引副本，中间表还要占用空间。
- 需要创建 SYSTOOLSPACE 表空间，并且对 PUBLIC 有访问权限。

表迁移限制，有些情况下不支持在线表迁移过程，可以根据 reason code 查找错误原因，以下是一些常见的限制：

- 原表上不能有主外键约束，主表或依赖表都不可以，可以通过 db2look 或系统表查找约束，并在迁移前删除，待迁移后再重建约束。
- 如果表上有事件监控器则不能迁移。
- 如果表上有 LOB、XML、LONG 类型字段，则要求表上必须有唯一性索引。
- 处于 set integrity pending 状态的表不能迁移。

关于此存储过程的详细解释，请参看信息中心。

8.7 db2look/db2move

8.7.1 db2move 工具介绍

db2move 程序用来在两个数据库间数据迁移，特别适合于不同平台、表数量比较多的情况。此程序是对 export、import、load 命令的封装，根据系统表获得用户表，将数据导出为 PC/IXF 格式，同时会产生一个 db2move.lst 文件，记录导出表和数据文件名字。然后将这些文件传送到目标系统中，通过 load 或 import 进行导入。

db2move 命令的语法如下：

```

      .------.
      V               |
>>-db2move--dbname--action-----+-----+-----><
      +- -tc--table-definers---+
      +- -tn--table-names-----+
      +- -sn--schema-names-----+
      +- -ts--tablespace-names--+
      +- -tf--filename-----+
      +- -io--import-option----+
      +- -lo--load-option-----+
      +- -co--copy-option-----+
      +- -l--lobpaths-----+
      +- -u--userid-----+
      +- -p--password-----+
      '- -aw-----'
```

Action 可以是 export、import 和 load 中的一种，db2move 执行时会自动进行数据库连接。

8.7.2 db2look 工具介绍

在数据迁移前，需要在目标端建立数据对象定义，这正是 db2look 最擅长的工作。db2look 能产生表、视图、索引、函数、Trigger、存储过程等对象定义语句，可通过 db2look -h 查看帮助。

以下是 db2look 常用的选项，-d 表示数据库名，-e 抽取数据库对象定义，-l 抽取表空间定义，-t 表示抽取的表名（注意：数据库默认创建的 bufferpool、表空间和节点组不会抽取），代码如下：

```
db2look -d db_name -e -l -o db2look.ddl      -- 抽取所有用户对象定义，包括表空间等
db2look -d -t t1 t2 -e -o db2look_tables.out -- 抽取某个或某些表定义信息，t1、t2 为表名
```

8.7.3 db2look+db2move 迁移案例 ■ ■ ■

下面举一个实例为大家演示通过 db2look+db2move 工具从 Linux 到 Win 的迁移过程。我们要迁移的库选用 DB2 自带的 sample 数据库，别看这个库小，麻雀虽小，五脏俱全，sample 库中包含了表、表约束、索引、视图、函数、存储过程等定义，大家很容易在自己的环境上模拟。由于在实际环境中，有些客户表中采用了 identity 标识列，而 db2move 是无法迁移这种类型的表数据的。因此，为了保持完整性，在数据库中定义了一张带有 identity 标识列的表。

(1) 在 Linux 上创建 sample 数据库，并创建一个带有 identity 标识列的 cust1 表，在 cust1 表中插入两行数据，代码如下：

```
db2inst1@dpf1:~> db2sampl
...
'db2sampl' processing complete.

db2inst1@dpf1:~> db2 connect to sample
Database Connection Information
Database server          = DB2/LINUX 9.7.1
SQL authorization ID     = DB2INST1
Local database alias     = SAMPLE

db2inst1@dpf1:/data1/data> db2 "create tablespace ts1 pagesize 8k managed by database
using (file '/data1/ts1/cont0' 10M ) bufferpool ibmdefaultbp "
db2inst1@dpf1:~> db2 "create table cust1 (custno smallint not null generated always
as identity (start with 500, increment by 1), custname varchar(16)) in ts1"
db2inst1@dpf1:~> db2 "insert into cust1(custname) values('suntao') "
db2inst1@dpf1:~> db2 "insert into cust1(custname) values('shiqing') "
db2inst1@dpf1:~> db2 "select * from cust1"
CUSTNO CUSTNAME
-----
500 suntao
501 shiqing
2 record(s) selected.
```

(2) 将 Linux 上 sample 的对象定义通过 db2look 导出，结果保存在 db2look.ddl 中，打开文件就可以看到对象的定义了，内容如下：

```
db2inst1@dpf1:~> db2look -d sample -e -l -o db2look.ddl
-- No userid was specified, db2look tries to use Environment variable USER
-- USER is: DB2INST1
```

```
-- Creating DDL for table(s)
-- Output is sent to file: db2look.ddl
-- Binding package automatically ...
-- Bind is successful
-- Binding package automatically ...
-- Bind is successful
```

(3) 将 Linux 上 sample 数据库表数据通过 db2move 导出，为数据传输方便，建议新建一个目录，存放导出数据（如果数据量大的话，考虑在共享存储上建目录），代码如下：

```
db2inst1@dpf1:~> mkdir data
db2inst1@dpf1:/data1/data> db2move sample export
Application code page not determined, using ANSI codepage 1208
***** DB2MOVE *****

Action: EXPORT
Start time: Thu Jan 13 19:05:57 2011
Connecting to database SAMPLE ... successful! Server : DB2 Common Server V9.7.1
Binding package automatically ... /home/db2inst1/sqllib/bnd/db2common.bnd ...
successful!
Binding package automatically ... /home/db2inst1/sqllib/bnd/db2move.bnd ...
successful!

EXPORT:      18 rows from table "DB2INST1"."ACT"
EXPORT:       0 rows from table "DB2INST1"."CATALOG"
EXPORT:       5 rows from table "DB2INST1"."CL_SCHED"
EXPORT:       2 rows from table "DB2INST1"."CUST1"
EXPORT:       6 rows from table "DB2INST1"."CUSTOMER"
EXPORT:      14 rows from table "DB2INST1"."DEPARTMENT"
EXPORT:      42 rows from table "DB2INST1"."EMPLOYEE"
EXPORT: 10000 rows from table "DB2INST1"."EMPMDC"
EXPORT:      73 rows from table "DB2INST1"."EMPPROJECT"
EXPORT:       8 rows from table "DB2INST1"."EMP_PHOTO"
EXPORT:       8 rows from table "DB2INST1"."EMP_RESUME"
EXPORT:       4 rows from table "DB2INST1"."INVENTORY"
EXPORT:       3 rows from table "DB2INST1"."IN_TRAY"
EXPORT:       8 rows from table "DB2INST1"."ORG"
EXPORT:       5 rows from table "SYSTOOLS"."POLICY"
EXPORT:       4 rows from table "DB2INST1"."PRODUCT"
EXPORT:       2 rows from table "DB2INST1"."PRODUCTSUPPLIER"
EXPORT:      65 rows from table "DB2INST1"."PROJACT"
EXPORT:      20 rows from table "DB2INST1"."PROJECT"
EXPORT:       6 rows from table "DB2INST1"."PURCHASEORDER"
EXPORT:      41 rows from table "DB2INST1"."SALES"
EXPORT:      35 rows from table "DB2INST1"."STAFF"
EXPORT:      35 rows from table "DB2INST1"."STAFFG"
EXPORT:       2 rows from table "DB2INST1"."SUPPLIERS"

Disconnecting from database ... successful!
End time: Thu Jan 13 19:06:01 2011
```

进入/data1/data 目录，发现数据已经导出。其中 db2move.lst 存放导出的表和对应的导出数据及消息文件列表，EXPORT.out 存放导出过程，tabx.ixf 和 tabx.msg 分别存放表数据和消息文件，代码如下：

```
db2inst1@dpf1:/data1/data> ls
db2move.lst  EXPORT.out  tab2.ixf  tab2.msg  ...
```

(4) 将数据通过 FTP 或其他方式传到 Windows 机器中，注意传输编码。对于数据文件通过 binary 格式传输；对于 db2look.ddl 和 db2move.lst 用 ASC11 传输。否则可能会出现乱码问题。

(5) 在 Windows 系统创建数据库，然后重建结构。打开 db2look.ddl，修改表空间路径，将/data1/ts1/cont0 改为 Windows 路径。注意：如果目标库名与源库不一致，需要在 db2look.ddl 中进行相应更改，对象的 schema 也要注意，代码如下：

```
D:\temp\data>db2 "create db sample pagesize 8 k"
DB20000I  CREATE DATABASE 命令成功完成。

CREATE LARGE TABLESPACE "TS1" IN DATABASE PARTITION GROUP IBMDEFAULTGROUP PAGESIZE
8192 MANAGED BY DATABASE
    USING (FILE '/data1/ts1/cont0' 1280)    --比如修改为 (File 'd:\data1\ts1\cont0' 1280)
    EXTENTSIZE 32
    PREFETCHSIZE AUTOMATIC
    BUFFERPOOL IBMDEFAULTBP
    OVERHEAD 7.500000
    TRANSFERRATE 0.060000
    NO FILE SYSTEM CACHING
    DROPPED TABLE RECOVERY ON;

D:\temp\data>db2 -tvf db2look.ddl
内容较多，省略...
```

(6) 然后将数据通过 db2move 导入。导入结束后，注意检查 LOAD.out 文件，其中记录了警告和错误信息，如加载了多少行、拒绝了多少行等。对于拒绝的表，要检查对应表的数据和 MSG 消息输出，代码如下（限于篇幅原因，删除了一些信息）：

```
D:\temp\data>db2move sample load
Application code page not determined, using ANSI codepage 1386
***** DB2MOVE *****
Action:  LOAD
Start time:  Fri Jan 14 11:47:27 2011
Connecting to database SAMPLE ... successful!  Server : DB2 Common Server V9.7.1
Binding package automatically ... G:\IBM\SQLLIB\BND\DB2COMMON.BND ... successful!
Binding package automatically ... G:\IBM\SQLLIB\BND\DB2MOVE.BND ... successful!

* LOAD:  table "DB2INST1"."ACT"
  -Rows read:          18
  -Loaded:             18
  -Rejected:           0
  -Deleted:            0
```

```

-Committed:          18

* LOAD: table "DB2INST1"."CUST1"
*** WARNING 3107. Check message file tab4.msg!
*** SQL Warning! SQLCODE is 3107
*** SQL3107W 消息文件中至少有一条警告消息。

-Rows read:          2
-Loaded:              0
-Rejected:            2
-Deleted:             0
-Committed:          2

* LOAD: table "DB2INST1"."DEPARTMENT"
-Rows read:          14
-Loaded:              14
-Rejected:            0
-Deleted:             0
-Committed:          14

* LOAD: table "DB2INST1"."EMPLOYEE"
-Rows read:          42
-Loaded:              42
-Rejected:            0
-Deleted:             0
-Committed:          42

... ..
Disconnecting from database ... successful!
End time:  Fri Jan 14 11:47:51 2011

```

(7) Identity 标识列处理。还记得我们在第 1 步中创建的 cust1 表吗？通过以上输出发现，这个表的数据并没有插入进去了，因为 db2move 不支持 identity 标识列的导入。现在只能通过 load 命令单独导入了，打开 db2move.lst，找到 cust1 表对应的数据文件，代码如下：

```

!"DB2INST1"."CUST1"!tab4.ixf!tab4.msg!

D:\temp\data>db2 "load from tab4.ixf of ixf modified by IDENTITYOVERRIDE insert into
db2inst1.cust1"

...
读取行数          = 2
跳过行数          = 0
装入行数          = 2
拒绝行数          = 0
删除行数          = 0
落实行数          = 2

D:\temp\data>db2 "select * from db2inst1.cust1"
CUSTNO CUSTNAME
-----
500 suntao

```

```
501 shiqing
2 条记录已选择。
```

是不是和我们在 Linux 上的数据一样呢？

（8）到这里，发现还是没有，别急，最后一步。还记得我们讲过，load 并不会进行检查参考完整性约束和检查约束检查，需要通过 set integrity 命令进行数据完整性检查。执行以下语句找出处于 set integrity pending 状态的表：

```
D:\temp\data>db2 "SELECT substr(TABNAME,1,30) as TAB_NAME, STATUS, ACCESS_MODE,
SUBSTR(CONST_CHECKED,1,1) AS FK_CHECKED, SUBSTR(CONST_CHECKED,2,1) AS CC_CHECKED
FROM SYSCAT.TABLES WHERE STATUS= 'C' "
```

TAB_NAME	STATUS	ACCESS_MODE	FK_CHECKED	CC_CHECKED
DEPARTMENT	C	N	N	Y
EMPLOYEE	C	N	N	N
EMP_PHOTO	C	N	N	Y
EMP_RESUME	C	N	N	Y
PROJECT	C	N	N	Y
PROJACT	C	N	N	Y
EMPPROJACT	C	N	N	Y
ACT	C	N	N	Y
ADEFUSR	C	N	Y	Y
PURCHASEORDER	C	N	N	Y

执行./set.sh 脚本，解除 pending 状态（set.sh 在 8.5.4 节有详细介绍）。

8.8 db2dart

db2dart 的功能非常强大，本节主要介绍数据导出功能。由于事务日志被破坏，或磁盘故障导致数据库无法连接，而又没有数据库备份情况下，db2dart 可能是您的最后一根救命稻草了。

下面是通过 db2dart /ddel 选项导出表数据的例子，该选项需要用户提供表 ID 或表名、表空间 ID、起始页和页数。表 ID 和表空间 ID 可以通过 syscat.tables 系统表查询，起始页可以从 0 开始，页数可以设为 999999999。默认的输出目录是<inst_home>/sqllib/db2dump/DART0000，可以通过/rpt 来指定。默认的导出数据名是 TS<tblspace_id>T<table_id>.DEL，如本例是 TS8T4.DEL。代码如下：

```
db2inst1@dpf1:/data1/data> db2dart sample /ddel
Table object data formatting start.
Please enter
Table ID or name, tablespace ID, first page, num of pages:
4,8,0,999999999
```

```

1 of 1 columns in the table will be dumped.
Column numbers and datatypes of the columns dumped:
    0  CHAR() -FIXED LENGTH CHARACTER STRING
Default filename for output data file is TS8T4.DEL,
do you wish to change filename used? y/n
Filename used for output data file is TS8T4.DEL. If existing file, data will be
appended to it.

Formatted data being dumped ...
Dumping Page 0 ...
Table object data formatting end.
        DB2DART Processing completed with warning(s)!
        Complete DB2DART report found in:
/home/db2inst1/sqllib/db2dump/DART0000/SAMPLE.RPT

```

注意：强烈建议执行 db2dart 的时候 deactivate 数据库，否则可能会出现不一致。

这时，有的读者可能会问，既然数据库都无法连接了，还怎么查询 tableid 和 tablepaceid 呢？问得好，要解决这个问题还得靠 db2dart，不过这次使用/db 选项。

```
db2dart sample /db /rpt /home/db2inst1/dart /rptn chekdb_dart.out
```

db2dart /db 选项会检查数据库的完整性，对库中的每张表进行检查，输出报告中会打印表名、表 ID 和表空间 ID。以下是一个结果片段，结果显示了表名 DBA.SNAP_LOCKWAIT，以及表 ID=29 和表空间 ID=3：

```

Table inspection start: DBA.SNAP_LOCKWAIT
Data inspection phase start. Data obj: 29 In pool: 3
Data inspection phase end.
...

```

对于一个存在很多表的数据库，需要 db2dart 进行表数据导出时，建议写 shell 或批处理脚本。将数据导出后，接下来就要对数据库重建了，重建仍然需要通过 db2look 目录导出表结构，因此，我们强烈建议在平时运维中经常保存一份建库脚本，以便出现问题时能够进行系统恢复。

```
db2look -d sample -e -l -o db2look.ddl
```

建完数据对象后，就可以将 db2dart 导出的数据文件通过 load/import 命令进行数据加载。

注意：db2dart 不会抽取大对象数据，如果表中含有 CLOB/BLOB 字段，并且是最后一个字段，且字段类型允许为空，那么 db2dart 导出的数据是可以插入的，但 CLOB/BLOB 值为 NULL 值。如果不是最后一个字段，load 加载时就会出错，这时需要对 db2dart 导出的 DEL 格式数据进行处理，用 NULL 或空格等补齐大对象所在的字段。

除非在没有别的办法的情况下，否则不建议用 db2dart，因为：

- 如果缓冲池有数据没有写到磁盘，用 db2dart 导出的数据可能是不完整、不准确的。
- db2dart 抽取速度会比较慢，特别是表很多、数据量很大的情况下，需要的时间很长。
- db2dart 会忽略大对象数据的抽取。
- 必须有表、索引、存储过程、函数等的定义，否则无法重建结构。

注意：只有当系统表（catalog tables）有效时，才能用 db2dart；当系统表数据被破坏的情况下，除了使用 restore 没有别的方法。这也从另外一个侧面说明了做好备份的重要性。

8.9 小结



数据迁移也许是数据库运维中最常见的工作。将一些数据从数据库 A 输入数据库 B，如何才能简单有效地完成这样的工作，是很多 DBA 的一大挑战。DB2 提供了多种工具可以选择，例如 import、load，甚至一些第三方的 ETL 工具（底层不外乎 insert、import 和 load）。

当用户手工运行 import 和 load 命令时，一定要注意这两个命令包含多种不同的参数适用于不同类型的数据。用户一定要对自己的数据迁移脚本做好完整测试以后再投入使用，否则如果生产系统的数据与测试数据出入过大时，可能测试良好的脚本在生产系统中会出现问题（例如，生产系统的迁移数据中，字符串包含空行，而测试系统中的数据没有空行）。

8.10 判断题



（1）DB2 可以使用 IMPORT 与 LOAD 命令加载数据。

T：正确

F：错误

（2）DB2 可以使用 EXPORT 命令导出数据。

T：正确

F：错误

（3）使用 db2move 移动数据性能远高于 EXPORT/LOAD。

T：正确

F：错误

(4) 用户可以使用 db2look 与 db2move 工具进行数据的跨平台迁移。

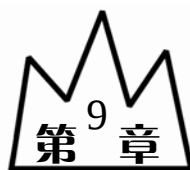
T: 正确

F: 错误

(5) db2dart 命令可以用来导出 LOB 数据。

T: 正确

F: 错误



备份恢复

数据是企业的生命线，现代企业对 IT 系统的依赖超出了以往任何一个时期，任何数据的丢失都可能对企业造成难以挽回的损失。因此，为保证数据的可用性、可恢复性，在存储级、系统级和数据库、应用级都涌现出很多技术，如 RAID 技术通过磁盘数据冗余尽量减少磁盘损坏造成的数据丢失；HA 技术通过双机确保数据的持续可用；在应用层通过集群技术确保其中一台出现故障的时候，其他机器可以继续提供服务。尽管以上技术可以最大程度保证系统的持续可用，但并不能解决数据人为误删、软件故障、电源故障、硬件故障等造成的数据损失。在有限的资源内，制定周密、合理的备份恢复策略，并有效实施，是 DBA 最重要的工作。

本章数据库日志部分深入解析了 DB2 日志的内部原理，澄清了很多 DBA 容易误解的问题。常见的备份和恢复场景基本涵盖了日常运维工作中的各类问题，是作者多年来实战经验的总结。

本章内容安排如下：

- 备份恢复概述。
- 数据库日志。
- 备份命令和操作。
- 各种恢复类型。
- 常见备份和恢复场景。

9.1 备份恢复概述



曾看到这样一句话：唯一能让 DBA 睡不踏实的就是没有做好备份。这有点夸张，但确实体现出了备份的重要性。有经验的 DBA 可能都有过由于没有备份，导致数据丢失的惨痛经历，以

下列举了可能出现问题的场景：

- 由于硬件原因引发的故障。包括电源和其他硬件在内的故障都可能导致数据库处于不一致状态。曾遇到过一个客户，由于电源异常断电导致 DB2 实例目录中的某些文件被破坏，实例无法启动。
- 存储介质引起的故障，如磁盘阵列、磁盘介质故障等都可能造成数据丢失。磁盘有一定的生命周期，而一些老旧磁盘，出问题的概率很大。曾经遇见过一个 RAID 5 同时坏 4 块盘的场景。
- 人为操作故障，如不小心删了数据、移除了表、删除了文件系统等。IT 人员经常处于高压和加班疲惫状态，误操作很常见，但不经意的错误可能造成难以挽回的损失，因此操作前一定要谨慎。
- 战争和自然灾害等。如美国的“911”恐怖袭击导致上千家企业数据丢失；我国的“5.12”汶川大地震造成多家企业数据中心瘫痪。

作为 DBA，需要根据企业应用类型和特点，定制有效的备份恢复策略，从以下 3 个方面确保数据库的可恢复性：

- 将数据库的故障频率降到最低，从而使数据库保持最大的可用性。
- 当数据库发生故障后，尽量减少数据丢失，从而使数据具有最大的可恢复性。
- 当数据库发生故障后，将恢复时间减到最少，从而使恢复的效率达到最高。

为了了解数据库备份/恢复相关的基本概念，我们介绍两个最常见的场景。第一个场景是：某 DBA 晚上 12 点对数据库做了备份，随后进行了一些业务处理；第二天上午 9 点用户不小心破坏了系统数据，那么如何进行数据恢复呢？对于采用了归档模式的数据库，采用的方法是恢复+前滚，即首先恢复到昨天晚上 12 点的备份介质，然后利用日志前滚到 9 点出事前的时间点，如图 9.1 所示。

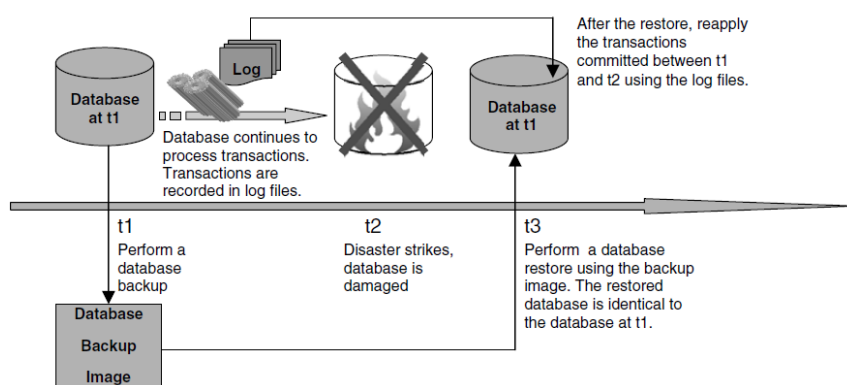


图 9.1 恢复场景

第二个场景是数据库正在执行过程中突然断电。这时某些数据可能还未写到数据盘，那么 DB2 如何保证数据库的一致性呢？答案是通过 DB2 事务日志和崩溃恢复。

这两个场景涉及到几个重要概念：日志、备份、恢复、前滚、崩溃恢复等，以下将分别介绍。

首先让我们再次回顾一下事务的概念。什么是事务？事务由一条或多条 SQL 语句组成，所有语句构成一个工作单元（Unit Of Work, UOW），事务的最后一条是 COMMIT 或 ROLLBACK 语句。事务具有 ACID 4 个特性，即原子性、一致性、隔离性和持久性，简单地讲就是事务包含的 SQL 语句要么全部成功，要么全部失败，不允许一些语句成功，另一些语句失败。

我们举个银行转账的例子，从储蓄账户向信用卡账户划款，假定还款金额为 X，用 SQL 语句表示就是：

```
Update saving set balance=balance-X where acct_id=aaa
Update credit set balance=balance+X where acct_no=bbb
COMMIT
```

在上面还款的过程中，如果钱从储蓄账号扣掉，但没有加到信用卡账户；或者钱没有从储蓄账号扣掉，而信用卡账户已经加上都是不允许的。这两条语句要么都操作成功，要么都不做才能保证账是平的。

每个事务中包含的 SQL 语句执行信息，以及事务是否成功提交或回滚的信息，都将写到日志文件中。DB2 通过对日志进行 Redo 和 Undo 来保证数据库的一致性。关于日志的原理、日志参数配置和维护等，将在下一节详细介绍。

关于恢复，DB2 支持以下几种类型。

- 版本恢复（Restore）：每个备份相当于一个版本，版本恢复是将数据库恢复到某个备份时的状态，这个备份之后的数据将会丢失。在第一个场景中，将数据恢复到昨晚 12 点的操作就是版本恢复。
- 前滚恢复（Rollforward）：前滚恢复是在版本恢复的基础上进行的日志重用。前滚恢复要求日志必须是归档类型，可以前滚日志到某个时间点，也可以到日志文件的结尾。如果需要追回 12 点以后的操作数据，则需要在版本恢复的基础上，将日志前滚到 12 点以后的某个时间点，前滚操作的机制是利用日志尽量减少数据的丢失。注意：前滚和回滚是完全不同的概念，不要混淆。
- 崩溃恢复（Crash Recovery）：崩溃恢复的目的是保证数据库的一致性。在第二个场景中，假定执行完事务的第一条语句后，机器突然异常停机，这时操作还没有结束，事务就被中断了，这时数据库会处于不一致状态。当重新启动系统后，必须对数据库做崩溃恢复，即回滚（Undo）还没有提交的数据，并重做（Redo）已经提交但还没有写入磁盘的数据，将数据库恢复到一致状态。缺省情况下，崩溃恢复是自动执行的，不需要人工干预。

提醒：在前滚和崩溃恢复时，DB2 怎么知道从哪个日志开始 Redo 和 Undo 呢？9.2.1 节将会揭开谜团。

9.2 DB2 日志



9.2.1 日志机制和原理

根据前面两个场景，我们发现日志扮演了非常重要的角色。没有日志的支持，DB2 就无法保证数据库的一致性和可恢复性。

备注：本节的内容相对高级，初学者可以先跳过本节内容，当积累了一些实战经验后再来学习。

1. 为什么需要日志？

首先，假设您是一个数据库系统的设计师，为了维护数据库的一致性，您将会用什么机制来避免下面几个问题呢？

（1）当数据库崩溃恢复时，原先写在缓冲池中的数据可能并没有来得及写入磁盘，这样磁盘上的数据还是修改之前的记录。

（2）当操作回滚的时候，我们需要知道被修改前的数据。

（3）数据库被版本恢复之后，我们需要全部历史操作将数据库前滚到一个给定的时间点。

从这几个问题中我们明白，至少数据库需要一种机制，能够记录所有的数据更新操作及被更新之前的旧数据。同时，该机制必须保证记录在真正的数据到达磁盘前被写入永久存储系统，以便在系统意外崩溃时能够恢复到一致的状态。

在 DB2 中，该机制使用日志功能实现。所谓日志，可以被认为是在一条事务被落实之前，能够保证其记录被写入永久存储系统的一种方法。

可能有读者问，为什么需要日志呢？直接把变化的数据写入磁盘不是更好？之所以需要日志，主要是从性能考虑。通常情况下，每一个事务包含若干条数据更改语句，每个更改可能需要操作大量的数据。如果将这些数据直接写入磁盘才返回给应用端，那么将严重影响写的性能。因此，DB2 采用写日志优先算法，即先写日志，再写数据，而写数据的过程是异步的。

图 9.2 所示是 DB2 日志原理图，当插入/更改/删除数据时，该条记录并不会直接写到数据盘，而是首先写到日志文件。而缓存池数据什么时候写到数据盘，是个异步过程。这样即使某时刻突然宕机，数据没有从缓存池内存写到数据盘也没有关系，因为数据已经记到日志文件里了。当重新开机进行恢复的时候，DB2 将日志文件的内容重写到数据盘，保证数据库的一致性。

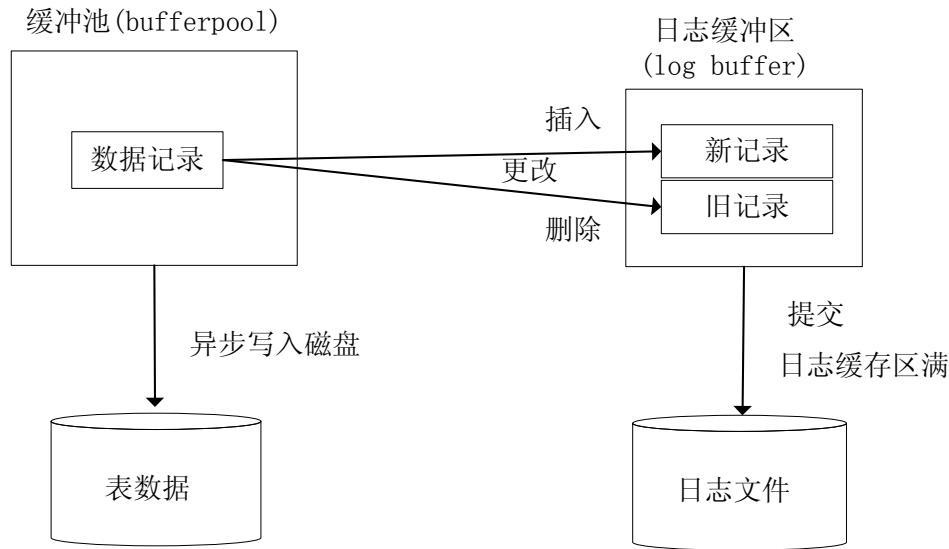


图 9.2 DB2 日志原理

2. 日志中都包含了什么？

既然我们知道了为什么需要日志，那么数据库日志里面都包含了什么呢？根据我们刚才所提到的 3 点需求，日志不需要记录任何不进行物理数据更新的操作。譬如一条查询，不进行任何数据更改，就不需要写入日志。而数据增、删、改操作，则必须将更改前后的数据同时写入日志，以保证能够回滚、前滚及崩溃恢复。

在日志中，每一个数据变化的操作，被记录下来的只有在什么地方发生变化、变化前后的数据（commit/rollback 操作也会记录时间戳），其他的信息（如所使用的 SQL 和命令发起的用户）都不会记录日志。

不过既然这些记录是一条一条的，总要有一个方法记住每条记录的“位置”。譬如说如果一条数据更新请求在 3 点被执行，1 秒钟之后写入磁盘，然后在 2 秒钟之后系统崩溃，那么数据库如何知道，该记录已经被写入磁盘了呢？也就是怎样记录这些日志记录的逻辑顺序。答案就是日志数据流，数据流可以用两种方式来计算：LSN 和 LSO。

LSN 代表着物理数据流，全称为 Log Sequence Number。物理流就是数据在磁盘上的真实位置。譬如说，一个日志文件大小是 10MB，也就是 10485760 字节，那么在第 5 个日志中第 10 个页面所在的 page lsn 就是 $5 \times 10485760 + 10 \times 4096 = 0x320A000$ （可通过 Windows 附件的计算器换算为十六进制）。每个日志页中第一个 LSN 就是代表该日志页中第一个日志记录的相对字节数。而 LSO 表示逻辑数据流，全称为 Logical Stream Offset。逻辑数据流只包含真实日志数据，不包含日志文件的各种头信息。

LSN 与 LSO 两者代表着同样的东西，在新的数据库版本中，内部数据跟踪的方法主要使用

LSO，由于 LSN 与 LSO 可以自由转化，很多外部的日志命令与控制文件中，仍然普遍使用 LSN 记录数据流位置。因此，本书中重点介绍 LSN。

根据 LSN 的定义，数据库中每一个数据变化，都可以找到一条 LSN 与之相配。如果一条记录在日志中的位置比另一条靠前，那么其 LSN 一定低于另外一条记录，而且不可能有两条记录包含相同的 LSN。

当我们通过日志恢复数据的时候，我们不希望用旧的日志数据取代已经被刷入磁盘的数据页。譬如说，一个数据页包含了 1 和 2 两条记录。在 3 点的时候，一个 update 请求将 1 更新成 3，3 点半的时候另外一个 update 请求将 2 更新成 4，紧接着数据被刷新到磁盘。

当崩溃发生的时候，DB2 会读取日志来重新写入这些更新，但是我们不希望重写已经写入磁盘的数据。DB2 如果做到呢？

答案就是 pageLSN。DB2 在每一个数据页的页头，记录该页最后一条记录的 LSN。假定某数据页的 pageLSN 是 0x12345，恢复的时候读到的某条日志记录 LSN 是 0x12300，DB2 日志管理器进程会将当前日志 LSN 与 pageLSN 对比，如果发现 pageLSN 更新，就会忽略这条记录。这样可以保证，日志不会覆盖已经刷新的磁盘数据。

注意：如果想要看每一个数据页的 pageLSN 怎么办呢？可以用 db2dart /Dd 选项，其中的输出数据就会包含 pageLSN 的信息。

现在 pageLSN 介绍完毕，我们该开始提出另一个问题了。

当数据库崩溃的时候，有些数据还没有来得及从缓存池写入磁盘，有些数据还没有提交但已经写入了磁盘，DB2 怎么知道从哪个日志文件恢复呢？难道每次都从 10 分钟之前，或者 1 个小时之前的日志文件恢复？如果一个事务从 30 个小时之前就开始执行，崩溃前 10 秒钟刚刚执行完，很多数据还来不及刷入磁盘，我们应该怎样去恢复它？

这个问题直接引出了两个最重要的 LSN：MinbuffLSN 与 LowtranLSN。其中，MinbuffLSN 是指系统中，最古老的还没有被刷入磁盘的日志记录所对应的 pageLSN。而 LowtranLSN 是指系统中，最古老的还没有被提交的事务中第一条日志记录所对应的 LSN。

这两个定义听上去不太好理解，让我们来看一个例子：

张三在 3 点的时候首次连接数据库，并对数据页 A 和 B 进行了修改。这两个修改所对应的日志记录为 a 和 b。这时张三并没有提交这些更改。

过了一会，三点一刻的时候李四从另一个地方连接了数据库，并且对另一个表的一百万个数据页进行了删除，然后立刻提交了操作。

这时我们可以看到，既然更新 A 和 B 的操作比删除一百万数据的操作更早发生，并且还没有被提交，那么 lowtranLSN 就会指向 a。

可是 minbuffLSN 呢？这个比较难以回答。因为数据库控制着什么时候一个数据页会被刷入

磁盘。因此，我们无法预测当前的 minbufLSN 是什么，有可能是 a 也有可能是 b，或者是一百万行数据删除中所对应的任何一个操作。

这样看来，lowtranLSN 就是一个系统中最早的还没有被提交的事务中，第一个操作所对应的 LSN；而 minbufLSN 就是系统中还没有被刷入磁盘中的所有脏页中，最早的那个数据页的 pageLSN。

这两个 LSN 有什么重要的？这就是我们刚才提出的问题 DB2 怎么知道应该从哪里开始恢复数据库 D 的答案：B2 会选择两者中更小的 LSN 作为崩溃恢复起始点：

```
min(minbufLSN, lowtranLSN)
```

如果 LowtranLSN 小于 MinbuffLSN，也就是前面张三的那种情况，那么就从 LowtranLSN 开始恢复。如果 MinbuffLSN 小于 LowtranLSN，就是说有些事务即使已经被提交，它们还是驻留在内存中，还没有来得及被刷入磁盘。

当我们知道了崩溃恢复的起始时间点，如何计算应该从哪个日志文件恢复呢？这个好办，既然 LSN 是物理数据流，那么通过 LSN 自然可以很轻易地计算出该 LSN 所属的日志文件。这个日志之前的所有日志在恢复过程中不需要，我们把它们叫做非活动日志；而这个日志及后面的所有日志则被称为活动日志。

活动日志为包含 min(minbufLSN, lowtranLSN) 及所有更高 LSN 的日志的统称。

这时，有的读者可能还有一个疑问：MinbuffLSN 和 LowtranLSN 存在哪里？答案就是 SQLOGCTL.LFH 控制文件。当数据库运行时，所有重要 LSN 的更改都会物理写入该控制文件，确保系统崩溃时有最新的信息来恢复。这个控制文件相当于整个数据库的中枢。数据库的状态，譬如 backup pending、rollforward pending 等，都是由这个控制文件所掌握的。SQLOGCTL.LFH 是二进制文件，不允许以任何方式修改，否则会造成数据库的不可用。

接下来，我们顺便介绍其他几个比较重要的 LSN，大家了解即可，不必深究。

- **baseLSN**：这个 LSN 是指最早活动日志中的第一个字节所对应的物理流所在的位置。譬如我们现在的日志是 S001000.LOG，而每一个日志的大小都是 10MB，那么 S001000.LOG 所对应的 baseLSN 就是 $1000 \times 10 \times 1024 \times 1024 = 0x271000000$ 。
- **tailLSN**：这个 LSN 代表着最后一个日志中 DB2 所写到的最后一个字节。一般来说 tailLSN+1 就是下一条 LSN 的起始点。
- **lifeLSN**：这个 LSN 是每一个数据库对象在创建时所对应的 LSN。譬如说张三的数据库中曾经有一个表叫做 A，其创建的 LSN 为 a。然后某一时刻这个表被 drop 后创建了一个新的 B，其对应的数据库对象 ID 与原来的 A 一模一样，但是这个表的 LSN 为 b。而当崩溃恢复的时候，如果 min(minbufLSN, lowtranLSN) 所对应的时刻表 A 依然存在，那么对表 A 的所有操作不就会写入表 B 吗？因此在每次通过日志恢复的时候，都要检查数据库对象的 lifeLSN 与当前 LSN。如果当前 LSN 小于数据库对象 LSN，则说明数据库对象的创建时间在当前 LSN 之后，因此当前 LSN 会被忽略掉。

3. 日志缓冲区

在本节的开始，我们介绍了日志的概念。不过当时我们有一点说得比较含糊，就是日志写入磁盘的方式。如果对每一个数据的更改都要操作磁盘的话，就算是不需要移动磁头，每一个 I/O 请求都要经过这样的过程：文件系统到达内核，内核传达给驱动，驱动传达给硬件，硬件等待盘片旋转到相应位置再写入。这一过程所需的时间也是非常可观的。因此，为了提高性能，每一次日志的写入都是先写到日志缓冲区，再通过一定的机制写到日志文件，如图 9.1 所示。

那么，日志记录什么时候从日志缓冲区写入磁盘呢？

(1) 交易提交。如果设置了 `mincommit`，则每个 `mincommit` 提交的时候将日志缓冲区刷入磁盘

(2) 日志缓冲区满。

(3) 当 `mincommit` 被设置为非 1 时，每秒钟刷新一次。

(4) 其余的一些 DB2 内部事件。

这几个机制可以确保即使使用日志缓存区，也可以完全满足我们刚开始提到的数据库一致性。

这里，我们给出数据库一致性的定义。所谓数据库一致性，就是当崩溃恢复，或者前滚到某一个时间点时，该时间点所有已经被提交的事务都会被写入磁盘，而所有未提交或者已经回滚的事务都被撤销。

对于那些回滚到最末尾 LSN 依然没有被提交，或者已经被回滚的事务，实际上所改变的数据是否真的写入日志并不重要。最后阶段依然没有提交的事务将与回滚的事务同等对待。

对于第 (1) 种情况，在事务提交时，该事务对应的 LSN 之前的事务全部刷入日志文件。

第 (2) 种情况，事务还没提交，但日志缓冲区满了，没关系，DB2 会写到日志文件，并清空缓冲区，继续处理其他数据。

第 (3) 种情况，使用了 `miniccommit`，这意味着并不是每次 `commit` 都会刷新日志缓冲区。如果这时数据库崩溃，那么没有写入磁盘的日志，其所对应的事务会被当做无效事务回滚。为了减少损失，当 `mincommit` 被使用时，DB2 每隔一秒钟将日志缓冲区写入磁盘。

第 (4) 种情况，主要是一些 DB2 内部事件需要明确地将日志缓冲区刷入磁盘，确保没有任何未写入磁盘的事务，保证数据库的一致性。比如 `online backup include logs` 时，在备份结束时，DB2 会刷新日志缓冲池。

4. 小结

本节介绍了 DB2 日志的原理，写日志优先算法，即先写日志，再写数据。当崩溃恢复或前滚恢复的时候，需要将日志文件写回到数据盘，保证 DB2 数据库的一致性。DB2 日志最终都存到日志文件，在内部结构上，每个日志文件由一些日志页构成，每个日志页包含一些日志记录，

每个日志记录通过 LSN 标识。当恢复时, DB2 需要知道从哪个 LSN 开始, 这就是 $\min(\text{MinbuffLSN}, \text{LowtranLSN})$ 。

同时, 我们介绍了活动日志这个非常重要的概念, 简单地说, 活动日志是包含 $\min(\text{MinbuffLSN}, \text{LowtranLSN})$, 以及所有更高 LSN 的日志文件。活动日志一定不要删除, 否则会造成数据库的宕机。

另外, 我们也介绍了数据库一致性的概念。所谓数据库一致性, 就是当崩溃恢复或者前滚到某一个时间点时, 该时间点所有已经被提交的事务都会被写入磁盘, 而所有未提交或者已经回滚的事务都被撤销。

9.2.2 日志参数配置最佳实践 ■ ■ ■

DB2 提供了很多与日志相关的参数, 如日志空间设置、日志文件大小、日志个数、日志存放目录、循环日志还是归档日志的选择、日志的存储和管理等。很多参数的默认设置都不能满足生产环境需求, 因此需要 DBA 认真规划和配置。

通过以下命令可查看某个数据库的日志参数:

```
inst20@db2server:~> db2 get db cfg for sample |grep -i "log"
Log retain for recovery status                = NO
User exit for logging status                  = YES
Catalog cache size (4KB)                     (CATALOGCACHE_SZ) = (MAXAPPLS*5)
Log buffer size (4KB)                         (LOGBUFSZ) = 8
Log file size (4KB)                          (LOGFILSIZ) = 1000
Number of primary log files                   (LOGPRIMARY) = 3
Number of secondary log files                 (LOGSECOND) = 2
Changed path to log files                     (NEWLOGPATH) =
Path to log files                            = /home/inst20/inst20/NODE0000/SQL00001/SQLLOGDIR/
Overflow log path                            (OVERFLOWLOGPATH) =
Mirror log path                              (MIRRORLOGPATH) =
First active log file                        = S00000016.LOG
Block log on disk full                       (BLK_LOG_DSK_FUL) = NO
Block non logged operations                   (BLOCKNONLOGGED) = NO
Percent max primary log space by transaction (MAX_LOG) = 0
Num. of active log files for 1 active UOW(NUM_LOG_SPAN) = 0
Percent log file reclaimed before soft ckcpt (SOFTMAX) = 100
Log retain for recovery enabled               (LOGRETAIN) = OFF
User exit for logging enabled                 (USEREXIT) = OFF
HADR log write synchronization mode           (HADR_SYNCMODE) = NEARSYNC
First log archive method                     (LOGARCHMETH1) = DISK:/data1/archlog/
Options for logarchmeth1                     (LOGARCHOPT1) =
Second log archive method                     (LOGARCHMETH2) = OFF
Options for logarchmeth2                     (LOGARCHOPT2) =
Failover log archive path                     (FAILARCHPATH) =
Number of log archive retries on error        (NUMARCHRETRY) = 5
Log archive retry Delay (secs)                (ARCHRETRYDELAY) = 20
```

Log pages during index build	(LOGINDEXBUILD) = OFF
------------------------------	-----------------------

1. 日志大小、路径设置

首先看日志空间大小设置,日志空间大小由 3 个参数控制:logprimary、logsecond 和 logfilsiz。Logprimary 用来设置主日志文件个数, logsecond 参数用来设置辅助日志文件个数, logfilsiz 用来指定每个日志文件的页数, 乘以 4K 就是每个日志文件的大小 (单位是 Byte)。因此, 日志空间的限制就等于 (logprimary+logsecond) *logfilsiz*4K。

什么是主日志文件和辅助日志文件呢? 主日志文件是数据库首次连接或激活时直接分配的, 辅助日志是在主日志文件写满时按需分配的。假设数据库参数如下配置:

Log file size (4KB)	(LOGFILSIZ) = 1000
Number of primary log files	(LOGPRIMARY) = 3
Number of secondary log files	(LOGSECOND) = 2
Path to log files	= /data1/activelog/NODE0000/

那么首次连接时会分配 3 个主日志:

```
db2inst1@dpf1:/data1/activelog/NODE0000> ls -alt
total 12048
-rw----- 1 db2inst1 db2iadm1 4104192 2011-01-16 23:59 S00000000.LOG
-rw----- 1 db2inst1 db2iadm1 4104192 2011-01-16 23:59 S00000001.LOG
-rw----- 1 db2inst1 db2iadm1 4104192 2011-01-16 23:59 S00000002.LOG
```

这时我们执行一个大事务操作, 比如在 t1 表中插入 5 万行数据, 这些数据都将被记录在日志中:

```
db2inst1@dpf1:/data1/activelog/NODE0000> db2 "create table t1 (col1 char(100), col2 char(10) )"
DB20000I The SQL command completed successfully.
db2inst1@dpf1:/data1/activelog/NODE0000> db2 "begin atomic declare i int default 0; while (i<50000) do insert into t1 values('aaaa' || char(i), 'bbbb' || char(i) ); set i=i+1; end while; end "
DB20000I The SQL command completed successfully.
```

打开另外一个窗口观察日志生成情况:

```
db2inst1@dpf1:/data1/activelog/NODE0000> ls -alt
total 16060
-rw----- 1 db2inst1 db2iadm1 4104192 2011-01-17 00:20 S00000003.LOG
-rw----- 1 db2inst1 db2iadm1 4104192 2011-01-17 00:20 S00000001.LOG
-rw----- 1 db2inst1 db2iadm1 4104192 2011-01-17 00:20 S00000002.LOG
-rw----- 1 db2inst1 db2iadm1 4104192 2011-01-17 00:20 S00000000.LOG
```

DB2 首先按编号顺序填充那 3 个主日志文件, 当没有更多的主日志文件分配的时候, 就开始动态分配第一个辅助日志文件, 一旦这个日志文件被填满, 就将继续分配下一个辅助日志文件, 以此类推, 直到所有的日志文件填满还无法容纳所有变化的数据时, 就会报事务日志满的 SQL0964C 错误, 然后事务将自动回滚, 代码如下:

```
db2inst1@dpf1:/data1/active/LOG/NODE0000> db2 "begin atomic declare i int default 0;
while (i<100000) do insert into t1 values('aaaa' || char(i), 'bbbb' || char(i) );
set i=i+1; end while; end "
DB21034E The command was processed as an SQL statement because it was not a valid
Command Line Processor command. During SQL processing it returned:
SQL0964C The transaction log for the database is full. SQLSTATE=57011
```

事务日志满不仅会丢失所做的操作，还可能需很长的回滚时间，因此要尽量分配足够多的日志空间减少此类问题出现的几率。那么这个空间分配多少才合适呢？日志空间与数据量有无必然联系？这几个参数设成多大？

首先，可以确定的是日志空间大小与数据量大小无必然联系，数据量更新很少，产生的日志就少；数据量很小，但变化频繁，产生的日志也会很大。根据最佳实践，我们一般将日志空间初始大小设置为数据库大小的 10%~20%之间，然后根据业务需求和监控结果进行相应调整。

确定了日志空间后，就要考虑每个日志文件的大小，即 `logfilsiz`。一般情况下，每个日志文件不要设得很大，因为文件创建和初始化需要很长时间。对于 `Logprimary` 个数，一般情况下，也不要分配太多，因为它们会在数据库初始化时完全分配，如果每天的工作负载很小，可能会造成空间的浪费。可设置足够的 `Logsecond` 大小应对峰值的情况，比如月初或月末可能有比较大的事务处理。

注意：`Logprimary+logsecond` 不能超过 255，日志空间大小不能超过 256GB。对 `logprimary` 和 `logfilsiz` 参数的更改需要断开连接，重新连接数据库才会生效，`logsecond` 参数的修改立即生效。`logsecond` 日志文件使用完后并不会立即删除，而是在所有连接断开并重新连接的时候才会删除。

2. 日志路径设置

在默认情况下，数据库日志文件存放在数据库目录下的 `SQLLOGDIR` 下，可通过 `NEWLOGPATH` 参数更改，此参数需要断开连接并重新连接时才会生效。日志文件的最佳实践如下：

- 日志是 I/O 写频繁类型的操作，为减少 I/O 竞争，建议将其与数据文件分离。
- 日志文件一定要放在速度快的共享存储上，提高性能。
- 鉴于日志的重要性，且多为写操作，建议采用 RAID10 阵列存放日志目录。

```
Changed path to log files (NEWLOGPATH) =
Path to log files = /home/inst20/inst20/NODE0000/SQL00001/SQLLOGDIR/

db2inst1@dpf1:/data1/active/LOG/NODE0000> db2 update db cfg for sample using newlogpath
/data1/active/LOG
DB20000I The UPDATE DATABASE CONFIGURATION command completed successfully.
SQL1363W One or more of the parameters submitted for immediate modification were not changed
dynamically. For these configuration parameters, all applications must disconnect from this
database before the changes become effective.
```

需要注意的是，一定要确保日志路径有足够的空间，否则会出现文件系统满而无法进行交易的情况。

如果是非常关键的系统，为降低日志出问题的风险，可考虑通过 `MIRRORLOGPATH` 参数设置日志镜像路径，也就是将日志存在两个位置。

注意：如果配置了 `MIRRORLOGPATH`，建议将路径放置在与 `NEWLOGPATH` 不同的物理存储上，否则无法应对磁盘的损坏。

3. 日志模式设置

DB2 包含两类日志模式：循环日志和归档日志。本节我们首先介绍两种日志模式的原理，然后介绍配置方法最佳实践，最后说明两种日志模式适用的场景。

1) 循环日志

循环日志模式是以循环的方式使用日志，当一个日志文件中包含的所有事务都已经提交或回滚，并且变化的数据已经写到数据磁盘，这个日志文件就可以被重用。否则，就只能按我们前面介绍的分配辅助日志文件。

下面举两个例子进行说明：

(1) 某数据库有 4 个主日志文件和 3 个辅助日志，DB2 将按照以下顺序使用日志：Log #1, Log #2, Log #3, Log #4, Log #1, Log #2，以此类推。这种日志使用的前提是当写完 Log #4 后，主日志文件虽然已用完，但 Log #1 可以重用。

(2) 同样是以上配置，但 DB2 可能按如下方式使用日志：Log #1, Log #2, Log #3, Log #4, Log #5, Log #6, Log #1...。当 Log #4 写完后，按照 DB2 优先使用主日志的原则，接下来应该使用 Log #1，但 Log #1 包含未提交数据，或包含提交的但还没有写入数据磁盘的数据，因此无法重用，这时只能按顺序使用辅助日志文件 Log #5。当写完 Log #5 后，发现 Log #1 还是无法重用，继续用 Log #6，如果这时发现 Log #1 可用，那么就会重用 Log #1。但如果 Log #1 仍然无法用，就会使用 Log #7，当 Log #7 也使用完后，如果发现 Log #1 仍然无法重用，就会报 SQL0964C 错误了。

有没有可能跳过 Log #1，而直接使用 Log #2 呢？答案是 NO。

2) 归档日志

归档日志模式是将所有日志保留，而不会重用。我们发现很多书籍在讨论这个知识点时有问题，特此说明。

举例说明，如果有 4 个主日志文件，DB2 将按照以下顺序使用它们：Log #1, Log #2, Log #3, Log #4。只要 Log #1 写满，就可以认为是可归档的日志（不管它是否已经 commit 或者是否已经将变化数据写到磁盘），在归档的时候会对它进行复制，以此类推。当 Log #4 写满，需要新日志时，它会检查 Log #1 是否是活动的，如果是活动的，DB2 会创建辅助日志文件 Log #5。如果 Log #1 是非活动的，DB2 就会重命名 Log #1 到 Log #5，并重用它（而不是创建一个新的 Log #5），之所以这样做是因为省去了重建创建并格式化文件的时间。

根据以上说明，就澄清了一个让很多人都迷惑的问题，日志什么时候归档？答案就是日志文件一旦写满就会被归档，日志归档时，会将该日志复制到归档日志目录，注意是复制，而不是移动，该日志并不会从活动日志目录中删除，归档并不意味着删除。归档的日志也可能还是活动日志，仍然包含未提交的数据。

3) 参数设置

如何判断和设置一个数据库的日志模式呢？答案是通过 LOGARCHMETH1 参数。当参数值等于 OFF 时，表示循环日志模式，当参数值为 OFF 以外的其他值时，表示归档模式。归档模式的方法如表 9.1 所示。

表 9.1

LOGARCHMETH1 参数值	用法解释
LOGRETAIN	日志文件保留在活动日志目录中。一般不建议采用这种设置，因为对于以后的维护很不方便，经常出现误删日志导致数据库被破坏的情况
USEREXIT	日志归档通过用户出口（userexit）程序来管理，这个出口程序需要 DBA 编译、配置，8.2 之前版本采用。8.2 后不推荐使用
disk:/dir_name	DB2 将日志归档到指定的磁盘目录中，原理与 USEREXIT 类似，但比它简单、易管理。是我们推荐的配置方式，如： db2 update db cfg x using LOGARCHMETH1 disk:/archlogs
TSM	与 IBM TSM 软件提供了很好的集成
VENDOR	通过第三方产品提供的归档类包。不同的产品提供的类包名字不同，如： Db2 update db cfg for x using LOGARCHMETH1 VENDOR:/home/dba/venlib/libLogArch.a

在表 9.1 提供的几种归档方法中，具体该选择哪一种呢？根据经验，建议采用 DISK:/dir_PATH 选项配置，即首先归档到共享磁盘的某个目录，然后通过操作系统复制命令或存储管理软件定期对归档日志目录备份。如果采用 TSM 或 VENDOR 方式，当日志出现异常问题时，很难定位是存储管理软件引起的还是 DB2 自身的问题，曾经有几个客户遇到过类似问题，最终还是改为 DISK 模式。

注意：循环日志模式是 DB2 数据库创建时默认的日志模式，当改为归档模式时，需要做离线完全备份，否则连接时会报 backup-pending 错误。

举例说明配置归档的步骤如下：

(1) 设置归档路径：

```
db2inst1@dpf1:/data1/archlog> db2 update db cfg for sample using logarchmeth1
disk:/data1/archlog
DB20000I The UPDATE DATABASE CONFIGURATION command completed successfully.
SQL1363W One or more of the parameters submitted for immediate modification were
```

not changed dynamically. For these configuration parameters, all applications must disconnect from this database before the changes become effective.

(2) 断开连接，使设置生效：

```
db2inst1@dpf1:/data1/archlog> db2 force applications all
DB20000I The FORCE APPLICATION command completed successfully.
DB21024I This command is asynchronous and may not be effective immediately.
```

(3) 对数据库做离线完全备份：

```
db2inst1@dpf1:/data1/archlog> db2 connect to sample
SQL1116N A connection to or activation of database "SAMPLE" cannot be made because
of BACKUP PENDING. SQLSTATE=57019
db2inst1@dpf1:/data1/archlog> db2 backup db sample to /data1/dbbak
```

在设置归档日志时，发现很多人对 LOGRETAIN 和 USEREXIT 参数比较疑惑。这两个参数是 DB2 8.2 (DB2 8.1 fp7) 以前版本提供的，为保持兼容性，在以后版本中仍然保留，但强烈不推荐使用。8.2 后的版本我们推荐用 LOGARCHMETH1 参数。

Log retain for recovery status	= NO
User exit for logging status	= YES
Log retain for recovery enabled	(LOGRETAIN) = OFF
User exit for logging enabled	(USEREXIT) = OFF

注意：

8.1 或更低版本，如果 LOGRTAIN 和 USEREXIT 参数值都为 OFF，则为循环日志。

8.2 或更高版本，如果 LOGARCHMETH1 和 LOGARCHMETH2 参数值都为 OFF，则为循环日志。

注意：8.2 版本后，归档日志的设置通过 LOGARCHMETH1 和 LOGARCHMETH2 参数设置，不建议使用 LOGRETAIN 和 USEREXIT。

4) 循环和归档日志选择

循环日志是默认的日志模式，支持 rollback、崩溃恢复和 backup/restore，但无法保留日志，因此不支持前滚恢复。一般用于 OLAP、数据仓库等查询类多的系统。

对于交易型系统，建议采用归档日志。尽管运维相对复杂，但能够确保当出现问题需要恢复的时候，可以利用日志进行数据前滚，减少数据丢失。

9.2.3 日志监控和维护管理 ■ ■ ■

1. 日志空间使用监控

通过数据库快照可以监控日志空间的使用。Log space available to database for SAMPLE 表示配置的数据库日志空间大小，Log space used by the database (Bytes)表示当前使用的日志空间，

Maximum total log space used (Bytes)表示曾经使用的最大日志空间。根据这些值可以判断当前日志空间是否可用。

```
$ db2 get snapshot for database on SAMPLE
Log space available to the database (Bytes)      = 20023994
Log space used by the database (Bytes)           = 376006
Maximum secondary log space used (Bytes)         = 2988883
Maximum total log space used (Bytes)             = 15228883
Secondary logs allocated currently               = 1
Log pages read                                  = 858
Log read time (sec.ns)                         = 0.000000004
Log pages written                               = 17809
Log write time (sec.ns)                        = 61.000000004
Number write log I/Os                          = 11002
Number read log I/Os                           = 858
Number partial page log I/Os                   = 6153
Number log buffer full                         = 1875
Log data found in buffer                       = 4832
Log to be redone for recovery (Bytes)           = 3105648
Log accounted for by dirty pages (Bytes)        = 3105648
```

2. 查看归档日志历史

有些时候，我们需要查看日志归档的频度，即多长时间归档一次。这时，数据库恢复历史文件（recovery history file）就派上了用场，该文件会记录备份、恢复、创建表空间、LOAD 等操作信息，当然也包括活动日志和归档日志的位置、时间等信息。每次当第一条日志记录写到日志文件时，就会在恢复历史文件中插入一个条目，当该日志归档后就会更新它的归档目录和归档结束时间。

以下是查看归档日志历史的例子，开始归档的时间戳是 20110117052856，归档共用了 (20110117055653-20110117052856) = 0.28 秒。

```
db2inst1@dpf1:/data1> db2 list history archive log all for sample
Op Obj Timestamp+Sequence Type Dev Earliest Log Current Log Backup ID
-----
X D 20110117052856      1   D S0000003.LOG C0000007
-----
Comment:
Start Time: 20110117052856
End Time: 20110117055653
Status: A
-----
EID: 75 Location: /data1/archlog/db2inst1/SAMPLE/NODE0000/C0000007/S0000003.LOG
```

通过 Type 值，可识别日志历史的类型：

- 'P'表示创建主日志文件（PRIMARY log）。
- 'M'表示镜像日志（MIRROR log）。

- 'F'表示归档失败（ARCHFAILURE）。
- '1'表示通过 Logarchmeth1 指定归档日志方法（ARCHIVE 1）。
- '2'表示通过 Logarchmeth2 指定归档日志方法（ARCHIVE 2）。

3. 日志的维护

作为 DBA，要每天检查文件系统空间。如果空间不够，可能无法创建活动日志，导致业务无法进行。即便如此，也不建议手工处理日志文件，因为日志文件丢失或破坏都会引起前滚和崩溃恢复失败，进而造成数据库不可访问。如果由于某些原因需要处理日志，也要特别小心，坚决不允许根据日期时间戳删除日志。

当主日志文件创建的时候，它们都具有相同的时间戳，当有事务写入日志文件时，该日志文件的时间戳才会改变。举例来说，logprimary 设置为 10，那么数据库激活的时候，会按照激活的时间创建这 10 个文件，假设这个时间是 T1。随后开始一些事务处理，并将事务日志记录在日志文件中，假设先写到 Log #1 到 Log #5。那么 Log #1~Log #5 的时间戳肯定会高于 T1，但 Log #6~Log #10 的时间戳仍然为 T1。您可能会认为 Log #6~Log #10 的时间戳比较早，可以删除。事实上，这些日志文件仍然是活动日志。如果删除，以后 DB2 需要的时候，数据库就会宕机。

为了确定哪些日志文件是活动的，哪些是不活动的，可查看 first active log file 参数。这个参数之前的所有文件是 inactive，之后的所有文件都是活动日志一定不要动。

例如，如果 first active log 是 S000005.LOG，那么 S01、S02，S03 和 S04 都是 inactive，而从 S05 开始的文件是活动的。

初始连接的日志文件，时间戳都为 2011-01-17 23:40:

```
db2inst1@dpf1:/data1/activelog/NODE0000> ls -alt
total 932
drwxr-x--- 2 db2inst1 db2iadm1 4096 2011-01-17 23:40 .
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:40 S0000000.LOG
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:40 S0000001.LOG
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:40 S0000002.LOG
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:40 S0000003.LOG
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:40 S0000004.LOG
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:40 S0000005.LOG
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:40 S0000006.LOG
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:40 S0000007.LOG
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:40 S0000008.LOG
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:40 S0000009.LOG
```

进行了一些事务处理后，S0000000~S0000004 的时间戳变为 2011-01-17 23:44，而 S0000005~S0000009 的时间戳仍为初始时的时间点 2011-01-17 23:40:

```
db2inst1@dpf1:/data1/activelog/NODE0000> ls -alt
total 1208
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:44 S0000000.LOG
```

```
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:44 S00000001.LOG
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:44 S00000002.LOG
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:44 S00000003.LOG
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:44 S00000004.LOG
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:40 S00000005.LOG
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:40 S00000006.LOG
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:40 S00000007.LOG
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:40 S00000008.LOG
-rw----- 1 db2inst1 db2iadm1 90112 2011-01-17 23:40 S00000009.LOG
```

查找最早的活动日志，即为 S00000005.LOG：

```
db2inst1@dpf1:/data1/active/log/NODE0000 > db2 get db cfg for sample |grep -i first
First active log file = S00000005.LOG
```

归档日志一般不建议删除，因为这些日志有可能在前滚的时候用到。如果有条件，可以将较早的归档日志备份到磁带或磁盘上，以便以后需要的时候可以找到。如果存储空间有限制，一定要删除的话，可考虑以下方法：首先使用 `list history backup` 找到需要保留的最早的备份条目，在条目中找到 first log，first log 之前的归档日志对于该备份来说，就是不需要的，可以删除。

如果 LOGARCHMETH1 的值是 LOGRETAIN，删除的时候务必要小心，因为活动日志也在这个目录，误删除可能会导致数据库无法启动，数据丢失。对于 LOGRETAIN 值，可通过 `PRUNE LOGFILE PRIOR TO log-file-name` 删除日志名之前的所有文件。

注意：活动日志一定不要删除，归档日志删除的时候也要务必小心。

4. ARCHIVE LOG 命令

如果用户希望在某一个时间点确定所有的日志不包含其他的内容（比如说，用户可能在某一时刻完成了一个重要的任务。出于安全考虑用户希望如果在需要恢复和回滚的情况下，回滚的数据不包含任何这个任务之后所发生的交易），那么用户可以使用 ARCHIVE LOG 命令。

简单地说，ARCHIVE LOG 命令一方面截断当前日志（不管当前日志是否写满，重新产生一个新的日志，所有其他的交易都写到新的日志中），同时将这个被截断日志之前的所有未归档的日志都归档。

这样，如果用户知道了当前被截断的日志号，在恢复回滚的过程中，只需要将该日志及其以前的所有日志复制到目标系统恢复即可。

9.2.4 其他日志相关的考虑 ■ ■ ■

如果用户希望对某些操作不进行日志记录（比如对于大表的删除或者更新），可以在 UOW 中使用 `activate not logged initially`：

```
db2 +c "alter table MYTABLE activate not logged initially"
```

```
db2 +c "delete from MYTABLE"
db2 +c "commit"
```

当 commit 发生后, 表状态会自动转回日志记录状态。因此, 如果用户希望对多个操作不记录日志, 可以将所有这些操作写入同一个 UOW 中。但需要注意的是, 配置了该选项后, 如果以后发生了回滚等操作, 将导致该表不可用, 必须 drop 重建。

在循环日志模式下, 大对象 (LOB) 和长字段 (LONG VARCHAR) 不记入日志。

9.2.5 经常遇到的日志问题 ■ ■ ■

在日常运维中, 经常遇到与日志相关的问题, 以下将对这些问题进行场景模拟, 并给出解决方案。

1. 事务日志满的处理

在日常操作中, 相信很多人遇到过事务日志满的情况, 即 SQL0964N 错误。出现这个错误的可能原因包括:

(1) 事务太大, 而日志空间设置太小。我们前面讲过, 日志空间=(logprimary+logsecond)*logfilsiz*4K, 当执行较大的事务时, 比如批量增删改操作, 默认的日志参数就很难满足需求, 这时, 可首先考虑对日志参数进行调整。。

调整的顺序, 为减少对应用的影响, 建议先从 logsecond 参数着手, 因为 logsecond 按需分配, 当不需要的时候 DB2 会考虑回收, 不会造成空间的浪费, 而且 logsecond 参数修改会立即生效, 不需要断开连接, 不会对业务造成中断影响。当然, logprimary + logsecond 的值不能超过 255 个, 如果增加了 logsecond 仍然不能解决问题, 可考虑修改 logprimary 和 logfilsiz 大小, 但这两个参数的更改需要重启数据库才会生效。

参数更改语句:

```
db2inst1@dpf1:~>db2 udparm db cfg for sample using logsecond 20
db2inst1@dpf1:~>db2 udparm db cfg for sample using logprimary 20
db2inst1@dpf1:~>db2 udparm db cfg for sample using logprimary 10000
```

当修改参数仍然无法解决问题的时候, 就应该从应用逻辑上进行优化, 尽量减少每个事务占用的日志空间。可供选择的方案如下:

- 如果需要删除一张含有海量数据的表, 可以根据某些过滤条件分多次删除, 而不是一次操作完成。
- 如果是清空一张大表, 尽量不要用 delete, 可考虑用 load from /dev/null of dev replace into <tablename>。
- 如果用户不想对某些操作记日志, 可以使用 ACTIVATE NOT LOGGED INITIALLY 选项。当 COMMIT 提交事务后, 表会自动转为记日志状态。使用该选项需要特别注意, 因为一旦随后出现问题, 该表必须删除 (这里的+c 代表关闭自动提交)。

```
db2 +c "alter table MYTABLE activate not logged initially"
db2 +c "delete from MYTABLE"
db2 +c "commit"
```

(2) 操作数据量很小，但仍然出现事务日志满。经常遇到客户抱怨说，日志空间已经改的很大了，为什么还会出现日志满呢？这种情况原因是某个长事务一直没提交，导致这个事务所在的活动日志一直没法使用。当所有的主日志和辅助日志都分配完后，就会报日志满错误。

对于该类问题，DBA 需要协助应用或开发人员找到引起问题的根源，并采取相应的措施。从应用程序设计的角度，建议每个事务尽可能快的提交，这样既会减少日志占有的时间，也会减少锁持有时间，提高并发效率。

以下通过实例模拟，在一个窗口中模拟一个事务，但不提交（+c 表示取消自动提交）。

```
db2inst1@dpf1:/data1/active/LOG/NODE0000> db2 +c "insert into t1 values('aaaa',
'aaaa') "
DB20000I The SQL command completed successfully.
```

然后打开另外一个窗口，批量插入一些数据。执行多次，直到出现日志满：

```
db2inst1@dpf1:~> db2 "begin atomic declare i int default 0; while (i<1000) do insert
into t1 values('aaaa' || char(i), 'bbbb' || char(i) ); set i=i+1; end while; end "
db2inst1@dpf1:~> db2 commit

db2inst1@dpf1:~> db2 "begin atomic declare i int default 0; while (i<1000) do insert
into t1 values('aaaa' || char(i), 'bbbb' || char(i) ); set i=i+1; end while; end "
DB21034E The command was processed as an SQL statement because it was not a valid
Command Line Processor command. During SQL processing it returned:
SQL0964C The transaction log for the database is full. SQLSTATE=57011
```

这时，我们打开第三个窗口，通过数据库快照（database snapshot）中的“Appl id holding the oldest transaction”找到最早未提交的事务所属的应用。本例中，228 就是我们要找的应用句柄。

```
Log space available to the database (Bytes)= 13923
Log space used by the database (Bytes)      = 964317
Maximum secondary log space used (Bytes)    = 149167
Maximum total log space used (Bytes)        = 964367
Secondary logs allocated currently          = 2
Log pages read                             = 0
Log read time (sec.ns)                     = 0.000000004
Log pages written                           = 613
Log write time (sec.ns)                     = 0.000000004
Number write log I/Os                       = 325
Number read log I/Os                       = 0
Number partial page log I/Os                = 200
Number log buffer full                      = 0
Log data found in buffer                    = 0
Appl id holding the oldest transaction      = 228
Log to be redone for recovery (Bytes)       = 938699
Log accounted for by dirty pages (Bytes)    = 103516
```

然后抓取 228 的应用快照(application snapshot), 找出这个应用正在执行的操作:

```
db2inst1@dpf1:~> db2 get snapshot for application agentid 228
...
Dynamic SQL statement text:
insert into t1 values('aaaa','aaaa')
```

接着需要与业务部门确认该操作的执行逻辑, 以及是否可以杀掉此应用。如果确认可以杀掉, 可执行以下操作:

```
db2inst1@dpf1:~> db2 "force application (228) "
DB20000I The FORCE APPLICATION command completed successfully.
DB21024I This command is asynchronous and may not be effective immediately.
```

为了运维方便, 也可通过以下脚本找出最早未提交的应用:

```
db2inst1@dpf1:~> db2 -tvf a.sql
SELECT      AI.APPL_STATUS      as      Status,      AI.AGENT_ID      as      AGENT_ID,
SUBSTR(AI.PRIMARY_AUTH_ID,1,10) AS "Authid", SUBSTR(AI.APPL_NAME,1,15) AS "Appl
Name", INT(AP.UOW_LOG_SPACE_USED/1024/1024) AS "Log Used (M)", AP.APPL_IDLE_TIME,
AP.APPL_CON_TIME AS "Connected Since" FROM SYSIBMADM.SNAPDB DB, SYSIBMADM.SNAPAPPL
AP,SYSIBMADM.SNAPAPPL_INFO AI WHERE AI.AGENT_ID = DB.APPL_ID_OLDEST_XACT AND
AI.AGENT_ID = AP.AGENT_ID

STATUS      AGENT_ID  Authid      Appl Name  Log Used (M)  APPL_IDLE_TIME  Connected
Since
-----
-----
-----
UOWWAIT      228      DB2INST1  db2bp      0      28      2011-01-18-00.53.47.914170
1 record(s) selected.
```

(3) 对于需要大量日志的事务, 考虑能够减少日志的方法。比如, 如果需要删除一张含有 50 万行数据的表, 就不要用 `delete`, 而尽量用 `load from /dev/null of dev replace into <tablename>` 这种不需要记日志的方法。如果要从 1000 万行的表中删除 50 万行, 这时可以考虑换个思路, 不要一次性删除 50 万行, 看能否根据过滤条件把这个任务分成多次, 每次删除更少的数据, 这样就减少了每次占用的日志空间。

2. 活动日志异常删除后的处理

活动日志用来保证数据的一致性, 如果日志丢失或破坏, 可能会造成数据库宕机, 这类问题在实际运维中出现的频率特别多, 每次的后果都是比较严重的, 因此一定要特别注意。当然也有非人为原因造成的日志损坏, 比如存放日志的磁盘或阵列出现问题等。

当出现此类问题时, 可采取如下几种解决方法:

- 如果存在数据库备份, 则可通过数据库恢复的方法。在数据库循环日志模式下, 只能恢复到最近一次备份时间点, 此时间点后的数据会丢失; 在归档日志模式下, 通过最后一次备份恢复后, 利用归档日志和尚存的活动日志进行前滚, 删除的日志则无法恢复。

- 如果购买了 IBM 800 电话支持服务，IBM 技术支持会尝试修复 SQLOGCTL.LFH 控制文件，然后重建数据库。这种方法比恢复数据库要复杂得多，而且会有一定程度的数据丢失。
- 通过 db2dart 将表数据导出，然后通过 db2look 重建表结构，再将数据 load 到表中，细节请参看第 8.7 章节。

既然活动日志这么重要，为保证日志的安全可靠，我们建议：

- 对于交易系统的活动日志目录配置，尽量使用 RAID 10。RAID 10 相对于 RAID 5 来说，提供了更好的冗余，从理论上说只要不是处于同一对的两个磁盘都坏掉，就可以修复。而一个 RAID 5 阵列，只要同时坏两块盘，修复的可能性就很小了。并且，由于日志是写频繁操作，RAID 10 的写性能优势更明显。但 RAID 10 的缺点是需要的磁盘比 RAID 5 要多。
- 对于非常关键的交易系统，可考虑用 MIRRORLOGPATH，即将日志保存在两个不同目录下，这样即使一组日志出现故障，还有一组作为后备。

3. 日志什么时候归档问题

前面我们提到过，日志一旦写满就会归档，而不是某些材料中写的“事务已经提交，并且变化数据写到磁盘才会归档”。通过下例进行模拟，在一个操作中插入 1000 条数据，即使没有提交事务，也会在归档日志目录下产生归档日志文件，这就验证了日志文件写满即归档的说法。

我们通过一个例子模拟一下：

```
db2inst1@dpf1:/data1/archlog/db2inst1/SAMPLE/NODE0000/C0000001>ls
S0000250.LOG
db2inst1@dpf1:/data1/archlog/db2inst1/SAMPLE/NODE0000/C0000001> db2 +c "begin
atomic declare i int default 0; while (i<1000) do insert into t1 values('aaaa' ||
char(i), 'bbbb' || char(i) ); set i=i+1; end while; end "
db2inst1@dpf1:/data1/archlog/db2inst1/SAMPLE/NODE0000/C0000001> ls
S0000250.LOG S0000251.LOG S0000252.LOG
```

4. 日志频繁归档问题

问题描述：某客户数据库每隔几十秒就会产生一个归档日志，查看日志大小，最大只有几 KB。检查数据库日志参数配置，没有发现任何异常，但通过 list applications 命令发现每隔一段时间所有数据库连接会断开。

解决方案：先用 activate 命令激活数据库，再去连接，之后问题消失。

原因分析：在归档日志模式下，有些场景，如数据库 deactivate、archive log 命令和 online backup 结束时，都会触发日志管理器对日志进行归档。如果没有使用 Activate <db> 激活数据库，那么在第一次连接时，DB2 会分配所有资源，当所有对数据库的连接断开后，数据库就会自动 deactivate，然后对日志归档；如果采用 Activate <db> 激活数据库，当所有连接断开后，DB2 并不会 deactivate 数据库，也就不会触发日志归档。本例的原因就是没有使用 Activate <db> 激活数

数据库，当所有数据库连接断开后，导致了日志归档。因此，我们建议在数据库连接前先通过 `Activate <db>` 激活数据库，这样也避免了第一次连接时需要的资源分配时间。

5. 双机环境下，系统时间不一致会有影响吗？

DB2 比较依赖系统时间，每条日志记录都有时间信息，如果双机的系统时间不一致，比如 A 机比 B 机快，当切到 B 机后，有可能会出现插入的日志时间比已有的日志时间早，在日志前滚的时候就可能出现异常。强烈建议系统管理员在做双机的时候确保时间一致，并且不要随意修改时间。

9.3 备份

做好数据备份是对一个 DBA 的最基本要求，如果没有备份，那么当出现各类故障或人为操作失误的话，企业数据就将面临很大风险。我们曾经遇到过很多客户由于没有备份而造成了重大损失。

因此，DBA 必须根据应用特点，制定有效的备份策略，并付诸实施。备份策略需要考虑很多方面，如成本、业务类型、对数据可用性的需求、对恢复时间长短的要求、运维管理的复杂度等。对于交易系统，为了保证数据的持续可用性，建议采用在线备份方式，如果数据库不大，可考虑数据库全备，相对于增量备份来说，恢复起来简单，如果数据库很大，可考虑每周做一次全备，每天做增量。对于仓库类系统，由于数据量巨大，很多客户无法提供如此庞大的存储空间，也无法忍受长时间的作业中断，这时可考虑采用数据导出的方法将一些重要的表数据备份。

以下是从各种角度，对备份进行的分类：

- 根据备份时是否允许业务访问可分为离线备份和在线备份。离线备份（offline backup）要求备份时没有任何应用连接，而在线备份（online backup）则允许在不中断业务的情况下进行备份，备份时允许读写数据。
- 根据备份的对象可分为数据库备份和表空间备份。数据库备份是对整个数据库所有数据所做的备份，表空间备份则是针对数据库中某个或某些表空间做的备份，一般是经常变化而又比较重要的表空间。
- 根据备份的规模可分为完全备份和增量备份。完全备份是对库或表空间所有数据所做的备份，而增量备份只备份前一次备份以来变化的数据。

DB2 的备份采用的是逻辑备份方式，也就是基于页备份，DB2 会备份每个数据页和索引页，而不是备份物理文件，好处是在恢复时可以更改磁盘布局，更加灵活方便。

DB2 备份是高度并行的，通过几组进程协调作业：db2bm 将数据从磁盘读入备份缓冲区（有时会使用 db2pfchr 异步读取），一旦缓冲区页面满，db2med 进程就负责将缓冲区数据写到备份介质。每个 db2med 进程可以将数据写到一个独立的备份介质上，DB2 支持多个 db2med 线程同

Incremental delta 备份是在最近一次备份（任何类型）以来变化数据库的备份。

- **With、Buffer、Parallelism**。这几个参数是用来提升性能的，9.5 版本 DB2 会自动选择优化这些参数。
- **Compress**。压缩备份介质。对于大数据库来说，指定 **Compress** 选项可以大大降低备份介质空间，但 **Compress** 选项需要消耗额外的 CPU 时间。
- **Include logs**。这个选项只适用于在线备份（online backup）时，当数据库备份结束时，会将当前的活动日志归档，并将其打包到备份介质中。

9.3.1 离线备份 ■ ■ ■

离线备份（offline backup）要求对数据库的所有连接断开，数据库是不可用的。由于备份过程中没有日志产生，因此离线备份数据和数据库总是一致的。

离线备份的语法如下：

```
backup db <db> [to <path>]
```

9.3.2 在线备份 ■ ■ ■

在线备份（online backup）是备份过程中数据库仍然是可读可写的。为了保持一致性，在线备份要求日志模式必须是归档的，这样可以保证在备份过程中，变化的数据通过日志保留。

在线备份的语法如下：

```
backup db <db> online [to <path>] [include logs]
```

Include logs 代表什么含义呢？发现很多人对此有疑问。对于在线备份，从开始备份到备份结束期间，对数据库所做的增删改操作都将记录到日志中。假如这时要在另外一台机器上进行恢复，那么除了要拷贝备份介质外，还要将备份期间产生的日志复制到目标端并进行前滚，才能确保数据库恢复成功。但是这种方法比较烦琐，因为 **DBA** 需要单独传送日志文件，并且需要找出哪些是必须传送的日志文件。**Include logs** 选项就是这个目的，当在线备份结束时，DB2 会将关闭当前活动的日志，并进行归档，然后将其打包到存到备份介质中，这样只需要传送一个备份介质就可以满足最小恢复的要求。这个选项是可选的，但强烈建议加上。

在线备份举例：

```
inst20@db2server:~> db2 backup db sample online to /dbbak include logs
```

提醒：在线备份要求日志是归档模式。

注意：在线备份时最好不要执行其他的运维任务，比如 **runstats** 等。否则可能会出现锁竞争而回滚的情况。

9.3.3 表空间备份 ■ ■ ■

对于较小的数据库，建议采用数据库备份，因为恢复比较简单。但对于大容量数据库来说，全库备份需要的存储空间太大，备份时间也较长，这时可选择对某些重要的表空间做备份。

对表空间备份时，建议将相关联的表空间同时备份，比如表 T1 在 TS1 表空间，表 T2 在 TS2 表空间，假如表 T1 和 T2 相关，则最好将 TS1 和 TS2 同时备份，否则可能在恢复时遇到数据不一致的情况，我们将在表空间恢复章节重点介绍。

表空间备份的语法如下：

```
backup db <db> tablespace (<tbs1>[,<tbs2>] ) online [to <path>]
```

表空间备份举例：

```
db2inst1@dpf1:/data1/dbbak> db2 "backup db sample tablespace (SYSCATSPACE,
IBMDB2SAMPLEREL) online to /data1/dbbak"
Backup successful. The timestamp for this backup image is : 20110118230057
```

9.3.4 增量备份 ■ ■ ■

对于大容量数据库，每次全备需要占用大量时间和空间，这时可考虑做增量备份，增量备份的优点表现在：备份介质占用空间更小，备份时间更快，对系统的影响更小，恢复时需要前滚的日志可能更少，但增量备份在恢复时相对复杂。

DB2 支持两种增量备份：累积备份（cumulative backup）和迭代增量备份（delta backup）。累积备份是在全备的基础上做的备份，而迭代增量则是在上一次备份的基础上做的备份，上一次备份可以是任意形式的备份。图 9.3 演示了累积备份策略，在这个策略中，每周日做全备，从周一到周六做增量备份，每次备份都是在周日全备的基础上累积的。

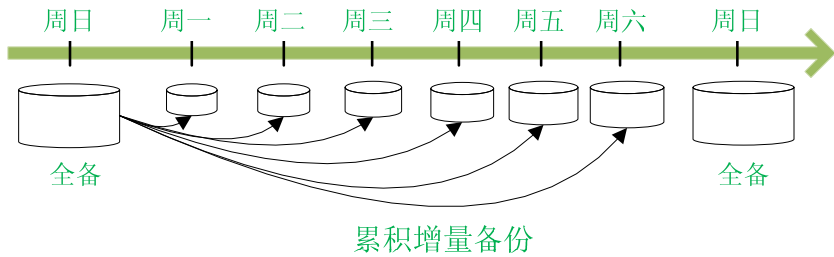


图 9.3 累积增量备份

图 9.4 所示是迭代增量备份，在这个策略中，每周日做全备，从周一到周六做迭代增量，每次都是在前一天的基础上的增量。

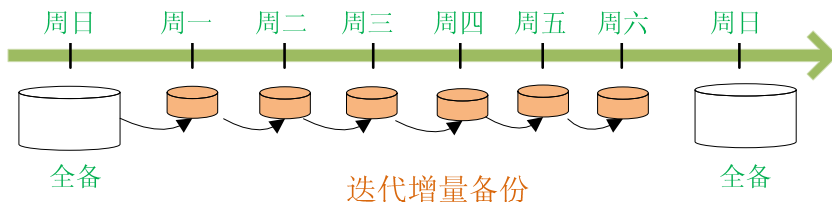


图 9.4 迭代增量备份

提醒：要启用增量备份，需将数据库参数 trackmod 改为 ON。

以下是增量备份语法，incremental 表示累积增量备份，incremental delta 表示迭代增量备份。

```
backup db <db> [tablespace (<tbs1>[,<tbs2>] )] [online] incremental [delta] [to
<path>]
```

增量备份举例：

```
inst20@db2server:~> db2 update db cfg for sample using trackmod on
DB20000I The UPDATE DATABASE CONFIGURATION command completed successfully.
SQL1363W One or more of the parameters submitted for immediate modification were
not changed dynamically. For these configuration parameters, all applications must
disconnect from this database before the changes become effective.
inst20@db2server:~> db2 force applications all
...
inst20@db2server:~> db2 backup db sample -- 做离线完全备份
inst20@db2server:~> db2 backup db sample online incremental delta -- 在线增量迭代备份
inst20@db2server:~> db2 backup db sample online incremental -- 在线增量备份
inst20@db2server:~> db2 backup db sample online incremental -- 增量
```

9.3.5 备份介质检查 ■ ■ ■

从 DB2 9 版本开始，不管在 Windows 还是 UNIX/Linux 平台，备份文件有统一的命名规则，备份文件名包含的字段含义如图 9.5 所示

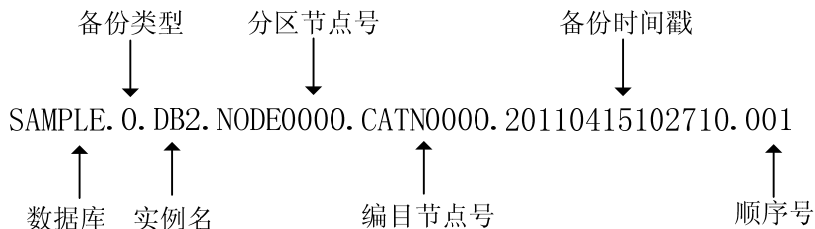


图 9.5 备份文件命名规则

其中备份类型字段，0 代表全备，3 代表表空间备份，4 代表 Load COPY 备份。单分区环境中，分区节点号固定为 NODE0000，编目节点号固定为 CATN0000。备份时间戳表示备份开始的时间，以年月日时分秒表示。

如果有一个备份介质需要恢复，如何检查该介质的正确性呢？答案是通过 `db2ckbkp` 命令。`db2ckbkp` 不仅可以检查备份介质的完整性、决定是否可以恢复外，还可以显示备份文件的元数据信息，如是否是在线备份、是否带日志、是否为压缩备份等，命令如下：

```
db2inst1@dpf1:/data1> db2ckbkp -h SAMPLE.0.db2inst1.NODE0000.CATN0000.20110118032318.001
=====
MEDIA HEADER REACHED:
=====
      Server Database Name      -- SAMPLE
      Server Database Alias     -- SAMPLE
      Client Database Alias     -- SAMPLE
      Timestamp                 -- 20110118032318
      Database Partition Number -- 0
      Instance                  -- db2inst1
      Sequence Number           -- 1
      Release ID                 -- D00
      Database Seed              -- 8568E614
      DB Comment's Codepage (Volume) -- 0
      DB Comment (Volume)       --
      DB Comment's Codepage (System) -- 0
      DB Comment (System)       --
      Authentication Value       -- -1
      Backup Mode                -- 1
      Includes Logs              -- 1
      Compression                -- 1
      Backup Type                -- 0
      Backup Gran.               -- 0
      Status Flags               -- 21
      System Cats inc            -- 1
      Catalog Partition Number   -- 0
      DB Codeset                 -- UTF-8
      DB Territory               --
      LogID                      -- 1295336020
      LogPath                    -- /data1/activelog/NODE0000/
      Backup Buffer Size          -- 1572864
      Number of Sessions         -- 1
      Platform                   -- 12

The proper image file name would be:
SAMPLE.0.db2inst1.NODE0000.CATN0000.20110118032318.001

[1] Buffers processed: #####

Image Verification Complete - successful.
```

表 9.2 所示是对以上结果中重要字段值的解释。对照着看，我们就很容易知道，当前给定的备份介质 `SAMPLE.0.db2inst1.NODE0000.CATN0000.20110118032318.001` 是带日志的在线数据库压缩完全备份，数据库版本是 9.7。

表 9.2

字 段 名	值 的 含 义
Backup Mode	0 - offline 1 - online
Includes Logs	0 - no 1 - yes
Compression	0 - no 1 - yes
Backup Type	0 - full 1 - tablespace
Backup Gran	0 - normal 16 - incremental 48 - delta
Release ID	900 - v7 A00 - V8 B00 - V9.1 C00 - V9.5 D00 - V9.7

db2ckbkp 还包含其他有用的选项，如 -a 可以检查需要前滚的日志文件。根据结果，S0000046.LOG – SQL0000055.LOG 是需要前滚的日志，代码如下：

```
db2inst1@dpf1:/data1> db2ckbkp -a SAMPLE.0.db2inst1.NODE0000.CATN0000.
20110118032318.001 | grep -i "File Number"
File Number [000] = 46
File Number [001] = 47
File Number [002] = 48
File Number [003] = 49
File Number [004] = 50
File Number [005] = 51
File Number [006] = 52
File Number [007] = 53
File Number [008] = 54
File Number [009] = 55
```

有些时候，DBA 需要检查过去 1 天内数据库是否进行过备份，以下脚本可以作为参考：

```
db2inst1@dpf1:/data1> db2 "select substr(comment,1,30) as comment ,timestamp(start_time)
as start_time,timestamp(end_time) as end_time,substr(firstlog,1,25) as firstlog,
substr(lastlog,1,25) as lastlog,seqnum,location from sysibmadm.db_history where
operation = 'B' and timestamp(start_time) > current_timestamp - 24 hours and sqlcode is
null "
```

COMMENT	START_TIME	END_TIME	FIRSTLOG	LASTLOG	SEQNUM	LOCATION


```
DB2 BACKUP SAMPLE ONLINE 2011-01-18-03.23.18.000000 2011-01-18-03.23.26.000000 S0000046.LOG S0000046.LOG 1
/data1
```

9.3.6 备份监控 ■ ■ ■

DB2 提供了 `list utilities` 命令检查一些工具的执行情况，这对于估算执行时间和进度有很大的帮助。下例中，监控时大概完成了整个备份的 15%，根据执行时间和监控时间，就可以大致估算中完成整个备份需要的时间：

```
$ db2 list utilities show detail

ID                        = 618
Type                      = BACKUP
Database Name             = BCDL
Partition Number          = 0
Description               = offline db
Start Time                = 2010-11-13 10:42:21.453322
State                     = Executing
Invocation Type           = User
Throttling:
  Priority                 = Unthrottled
Progress Monitoring:
  Estimated Percentage Complete = 15
  Total Work               = 2606158793 bytes
  Completed Work           = 381781055 bytes
  Start Time               = 2010-11-13 10:42:21.453358
```

9.4 恢复



DB2 支持 3 种类型的恢复：崩溃恢复、版本恢复和前滚恢复。相对于备份而言，恢复就没有那么轻松了，因为需要恢复的场合往往是出现重大故障或事故的时候，这也是 DBA 大显身手的好时机。作为一名优秀的 DBA，在应对和处理这种问题的时候，一定要权衡各种方案，在最大程度的减少数据丢失的基础上，减少恢复时间，提高数据库的可用性。接下来为大家介绍这几种恢复。

9.4.1 崩溃恢复 ■ ■ ■

在日志机制和原理章节，我们介绍过 DB2 的写日志优先算法，即事务先写到日志，再写到数据盘。那么假定数据库突然异常或者机器宕机，DB2 如何保证一致性呢？答案是通过 DB2 提供的崩溃恢复。

首先，我们看一下事务中断前数据可能出现的 4 种情况：

- (1) 事务已经提交，日志缓冲区数据已经写到日志文件，数据也写入磁盘。
- (2) 事务已经提交，日志缓冲区数据已经写到日志文件，但是缓冲池数据还没有写入磁盘。
- (3) 事务还未提交，日志缓冲区数据已经写到日志文件，缓冲池也写入数据文件。
- (4) 事务还未提交，日志缓冲区数据还没有写到日志文件，缓冲池也没有写入数据文件。

有没有可能出现数据已经写入数据文件，但还没有写日志的情况呢？这是不可能出现的，因为 DB2 采用的是写日志优先算法。

对于第（1）种情况，重启系统后，并不会做处理，因为日志和数据文件是一致的；对于第（4）种情况，事务还未提交，数据也未写入，所以也不会做处理；对于第（2）种，就要将日志的内容重写到数据文件里，即 redo；对于第（3）种，则要将写入磁盘的数据 undo。

那么崩溃恢复从哪个日志开始 redo 和 undo 呢？在日志原理部分曾经讲过两个最关键的 LSN，即 MinbuffLSN 与 LowtranLSN，这两个 LSN 记录在 SQLOGCTL.LFH 控制文件中。当崩溃后，再进行数据库连接时，DB2 会读取 SQLOGCTL.LFH 文件获得这两个 LSN 进行崩溃恢复。

默认情况下，崩溃恢复是自动发生的，不需用户干预，因为 `AUTORESTART` 数据库参数默认是 `ON`。崩溃恢复的过程可通过 `db2diag.log` 诊断日志和 `db2 list utilities show detail` 命令监控。

如何减少崩溃恢复的时间？建议在应用中频繁提交，并且使用合理的 `chpgp_thresh` 数据库参数使得缓冲池数据尽快写到磁盘。

9.4.2 版本恢复

DB2 的每个备份都是一个版本，版本恢复就是将数据恢复到某个备份时的状态，是备份的逆向过程。版本恢复通过 `restore` 命令实现，我们浏览一下 `restore` 命令语法：

```
>>RESTORE--++DATABASE--+source-database-alias-->>
      '-DB-----'

>--| restore-options |-----><
      +-CONTINUE-----+
      '-ABORT-----'

restore-options

|----->
      '-USER--username--+
              '-USING--password-'

>-----+----->
+-REBUILD WITH--+--ALL TABLESPACES IN DATABASE--+--++
|               |'-ALL TABLESPACES IN IMAGE-----' '-EXCEPT--| rebuild-tablespace-clause |-' |
|               |'-| rebuild-tablespace-clause |-----' |
|+-TABLESPACE--+-----+
|               |'.',-----|
|               |v-----|
|               |'-(---tablespace-name+---)' |
+-HISTORY FILE-----+
+-COMPRESSION LIBRARY-----+
'-LOGS-----'

>-----+----->
      '-INCREMENTAL--+
              +-AUTO-----+
              +-AUTOMATIC--+
              '-ABORT-----'
```


例子 2：表空间恢复

可以从数据库备份（需启用了归档日志模式）或表空间备份中恢复表空间。恢复时，正在恢复的表空间不可访问，指定 `online` 选项后，用户可以连接数据库，并访问其余表空间，代码如下：

```
db2inst1@dpf1:/data1> db2 "restore db sample tablespace (IBMDB2SAMPLEREL) online
from /data1 taken at 20110119084458 "
```

DB20000I The RESTORE DATABASE command completed successfully.

表空间恢复后，会处于 `rollforward-pending` 状态，需要前滚到表空间最小恢复时间点，细节请参看 9.4.3 节。

例子 3：增量备份和恢复

以下是一个增量部分的例子，备份策略为：周末做离线完全备份，然后周一、周二做迭代增量（`delta`）备份，周三做增量备份，周四、周五做迭代增量，周六做增量备份（以下示例用来模拟）：

```
--周日做离线完全备份
db2inst1@dpf1:/data1> db2 backup db sample
Backup successful. The timestamp for this backup image is : 20110119090610
--周一做迭代增量
db2inst1@dpf1:/data1> db2 backup db sample online incremental delta
Backup successful. The timestamp for this backup image is : 20110119090628
--周二做迭代增量
db2inst1@dpf1:/data1> db2 backup db sample online incremental delta
Backup successful. The timestamp for this backup image is : 20110119090633
--周三做增量
db2inst1@dpf1:/data1> db2 backup db sample online incremental
Backup successful. The timestamp for this backup image is : 20110119090700
--周四做迭代增量
db2inst1@dpf1:/data1> db2 backup db sample online incremental delta
Backup successful. The timestamp for this backup image is : 20110119090706
--周五做迭代增量
db2inst1@dpf1:/data1> db2 backup db sample online incremental delta
Backup successful. The timestamp for this backup image is : 20110119090710
--周六做增量
db2inst1@dpf1:/data1> db2 backup db sample online incremental
Backup successful. The timestamp for this backup image is : 20110119090719
```

如果需要恢复到周五，可以通过 `incremental automatic taken at <周五时间戳>`，`automatic` 选项自动按照顺序进行恢复，大大简化了操作，代码如下：

```
db2inst1@dpf1:/data1> db2 restore db sample incremental automatic taken at
20110119090710
```

如果希望了解恢复的顺序，并手工完成操作，DB2 也提供一个工具 `db2ckrst`。`db2ckrst` 命令根据恢复历史文件产生恢复顺序，代码如下：

```
db2inst1@dpf1:/data1> db2ckrst -d sample -t 20110119090710 -r database
```

```
Suggested restore order of images using timestamp 20110119090710 for
database sample.
```

```
=====
restore db sample incremental taken at 20110119090710
restore db sample incremental taken at 20110119090610
restore db sample incremental taken at 20110119090700
restore db sample incremental taken at 20110119090706
restore db sample incremental taken at 20110119090710
=====
```

例子 4: 恢复日志

对于带有 include logs 的在线备份，如果需要在另外一台机器上恢复，可在恢复过程中将备份介质中的日志恢复到某个位置，然后利用这些日志进行前滚。

首先对原机数据库做备份，带 include logs 选项，代码如下：

```
db2inst1@dpf1:/data1> db2 backup db sample online include logs
Backup successful. The timestamp for this backup image is : 20110119101132
```

将备份介质传到另外一台机器，通过 logtarget 将日志恢复到一个指定的目录，代码如下：

```
db2inst1@dpf1:/data1> db2 restore db sample from /data1 taken at 20110119101132
logtarget /data1/logs
db2inst1@dpf1:/data1> cd /data1/logs
db2inst1@dpf1:/data1/logs> ls
S0000000.LOG
```

利用恢复出的日志进行前滚，overflow log path 指定前滚获取日志的目标，代码如下：

```
db2inst1@dpf1:/data1/logs> db2 "rollforward db sample to end of logs and stop overflow
log path (/data1/logs) "
```

Rollforward Status

Input database alias	= sample
Number of nodes have returned status	= 1
Node number	= 0
Rollforward status	= not pending
Next log file to be read	=
Log files processed	= S0000000.LOG - S0000000.LOG
Last committed transaction	= 2011-01-19-18.11.46.000000 UTC

```
DB20000I The ROLLFORWARD command completed successfully.
```

如果不想恢复数据库，而只想恢复备份介质中的日志，应该怎么办呢？还是通过 logtarget，但是需要先指定 Logs 选项，代码如下：

```
db2inst1@dpf1:/data1/logs> ls
```

```
db2inst1@dpf1:/data1/logs> db2 restore db sample logs from /data1 logtarget
/data1/logs
DB20000I The RESTORE DATABASE command completed successfully.
db2inst1@dpf1:/data1/logs> ls
S00000000.LOG
```

例子 5：恢复到另外一个数据库目录或存储路径

经常有这样的需求，将数据库恢复到另外一台机器上，但恢复的数据库目录或自动存储路径不同。为解决这个问题，restore 提供了三个选项：To target-directory、DBPATH ON target-directory 和 ON path-list。

- **To target-directory:** 这个参数指定了目标数据库目录，如果是恢复到已存在的数据库，将忽略该参数值。如果备份介质包含的数据库启用了自动存储，那么目标数据库目录将改变到此参数指定的路径，而数据库的自动存储路径并不会改变。
- **DBPATH ON target-directory:** 这个参数指定了目标数据库目录，如果是恢复到已存在的数据库，将忽略该参数值。如果备份介质包含的数据库启用了自动存储，并且 ON 参数没有指定的话，那么这个参数与 TO 参数等同。
- **ON path-list:** 这个参数重新定义自动存储路径。对于没有启用自动存储的数据库，使用这个参数会报错（SQL20321N）。备份介质中已有的自动存储路径将不再使用，而是重定向到这个新指定的路径。如果没有指定该参数，那么自动存储路径仍保留已有的路径。如果数据库不存在，而且 DBPATH ON 参数没有指定，那么第一个自动存储路径将作为目标数据库路径。

看起来是不是很麻烦，其实很简单，如果目标库不存在，则 restore 的 TO 选项和 DBPATH ON 选项用来指定目标库的数据库目录，ON 选项用来指定自动存储路径的位置，如果只有 ON，则数据库目录放在 ON 指定的第一个目录；如果目标库存在，则忽略 TO 和 DBPATH ON 选项，如果指定了 ON 选项，则改变自动存储路径。

在下例中，指定了 DBPATH ON/data1/dbdir，如果目标库不存在，那么会将数据库目录存在 /data1/dbdir，但自动存储路径不变。如果目标库存在，则忽略 DBPATH ON。

```
db2inst1@dpf1:/data1> db2 "restore db sample dbpath on /data1/dbdir "
DB20000I The RESTORE DATABASE command completed successfully.
```

在下例中，指定了 ON 选项，如果目标库不存在，那么会将数据库创建在 /data1 目录下，包括数据库目录和存储路径：

```
db2inst1@dpf1:/data1> db2 "restore db sample on /data1 "
DB20000I The RESTORE DATABASE command completed successfully.
```

例子 6：重定向恢复

数据库在备份时，有关表空间容器的信息会保存到介质中。在恢复过程中，将在目标端检查这些表空间容器是否存在，以及是否可用，如果不存在或无法使用，那么数据将无法恢复。举个最简单的例子，A 机数据库表空间容器存在 D 盘，需要将 A 机数据库恢复到 B 机，但 B 机

D 盘是光驱，对于这种情况，恢复就会报错。DB2 通过重定向恢复（redirect restore）解决这类问题。所谓重定向恢复，就是在恢复过程中，重新定义表空间容器，包括增加、删除、更改等。

我们通过一个实例演示重定向恢复的步骤。

(1) 在 restore 命令中指定 redirect 选项。

当发出 restore ... redirect 命令后，DB2 提示 SQL1277W 警告，这个警告说明：现在正在执行重定向恢复，可以查看表空间配置信息，对于没有使用自动存储的表空间可以重新定义容器。言外之意，对于使用了自动存储的表空间是不能重新定义容器的，代码如下：

```
db2inst1@dpf1:/data1> db2 restore db sample into mydb redirect
SQL1277W  A redirected restore operation is being performed.  Table space
configuration can now be viewed and table spaces that do not use automatic storage
can have their containers reconfigured.
```

(2) 查看目标库表空间状态。

通过 list tablespaces show detail 命令查看表空间状态，发现 ts1 表空间的状态是：

```
Restore pending
Storage must be defined
Storage may be defined

Tablespace ID          = 3
Name                   = TS1
Type                   = Database managed space
Contents               = All permanent data. Large table space.
State                  = 0x2001100
Detailed explanation:
  Restore pending
  Storage must be defined
  Storage may be defined
```

可继续查看表空间容器的路径、大小等信息：

```
db2inst1@dpf1:/data1> db2 list tablespace containers for 3 show detail
Tablespace Containers for Tablespace 3
Container ID           = 0
Name                   = /data1/ts1/cont0
Type                   = File
Total pages            = 1280
Useable pages          = 1248
Accessible             = No
```

(3) 修改表空间容器定义。对每一个需要修改的表空间重新定义容器，代码如下：

```
db2inst1@dpf1:/data1> db2 "set tablespace containers for 3 using (file
'/tbs/ts1/cont0' 3000 )" "
```

(4) 通过 restore...continue 开始正式恢复，代码如下：

```
db2inst1@dpf1:/data1> db2 restore db sample continue
DB20000I The RESTORE DATABASE command completed successfully.
```

对于存在多个表空间容器需要修改的情况，采用命令行很麻烦，而且容易出错，因此，DB2 9 版本提供了一个新的选项，即 `redirect generate script`，这样就可以在恢复前，把我们前面手工操作的 4 个步骤保存到一个文件中，代码如下：

```
db2inst1@dpf1:/data1> db2 restore db sample redirect generate script redirect.ddl
DB20000I The RESTORE DATABASE command completed successfully.
db2inst1@dpf1:/data1>
```

产生的重定向恢复内容如下，可以根据需要重新定义表空间容器（对于自动存储表空间，没有办法修改。前面加注释的行，不要修改）：

```
db2inst1@dpf1:/data1> cat redirect.ddl
-- *****
-- ** automatically created redirect restore script
-- *****
UPDATE COMMAND OPTIONS USING S ON Z ON SAMPLE_NODE0000.out V ON;
SET CLIENT ATTACH_DBPARTITIONNUM 0;
SET CLIENT CONNECT_DBPARTITIONNUM 0;
-- *****
-- ** automatically created redirect restore script
-- *****
RESTORE DATABASE SAMPLE
-- USER <username>
-- USING '<password>'
FROM '/data1'
TAKEN AT 20110119182531
-- ON '/data1'
-- DBPATH ON '<target-directory>'
INTO SAMPLE
-- LOGTARGET '<directory>'
-- NEWLOGPATH '/data1/dbdir/db2inst1/NODE0000/SQL00001/SQLLOGDIR/'
-- WITH <num-buff> BUFFERS
-- BUFFER <buffer-size>
-- REPLACE HISTORY FILE
-- REPLACE EXISTING
REDIRECT
-- PARALLELISM <n>
-- WITHOUT ROLLING FORWARD
-- WITHOUT PROMPTING
;
-- *****
-- ** table space definition
-- *****
-- ** Tablespace name                = SYSCATSPACE
-- ** Tablespace ID                  = 0
-- ** Tablespace Type                = Database managed space
```

```

-- ** Tablespace Content Type          = All permanent data. Regular table space.
-- ** Tablespace Page size (bytes)      = 8192
-- ** Tablespace Extent size (pages)    = 4
-- ** Using automatic storage           = Yes
-- ** Auto-resize enabled                = Yes
-- ** Total number of pages              = 12288
-- ** Number of usable pages             = 12284
-- ** High water mark (pages)           = 11720
-- *****
-- *****
-- ** Tablespace name                   = USERSPACE1
-- ** Tablespace ID                     = 2
-- ** Tablespace Type                   = Database managed space
-- ** Tablespace Content Type           = All permanent data. Large table space.
-- ** Tablespace Page size (bytes)      = 8192
-- ** Tablespace Extent size (pages)    = 32
-- ** Using automatic storage           = Yes
-- ** Auto-resize enabled                = Yes
-- ** Total number of pages              = 4096
-- ** Number of usable pages             = 4064
-- ** High water mark (pages)           = 1824
-- *****
-- *****
-- ** Tablespace name                   = TS1
-- ** Tablespace ID                     = 3
-- ** Tablespace Type                   = Database managed space
-- ** Tablespace Content Type           = All permanent data. Large table space.
-- ** Tablespace Page size (bytes)      = 8192
-- ** Tablespace Extent size (pages)    = 32
-- ** Using automatic storage           = No
-- ** Auto-resize enabled                = No
-- ** Total number of pages              = 1280
-- ** Number of usable pages             = 1248
-- ** High water mark (pages)           = 96
-- *****
SET TABLESPACE CONTAINERS FOR 3
-- IGNORE ROLLFORWARD CONTAINER OPERATIONS
USING (
    FILE  '/data1/ts1/cont0'              1280
);
-- *****
-- ** start redirected restore
-- *****
RESTORE DATABASE SAMPLE CONTINUE;
-- *****
-- ** end of file
-- *****

```

然后通过 `db2 -tvf redirect.ddl` 进行重定向恢复。

注意：如果原库是可恢复数据库（即启用了归档日志），重定向恢复结束后，还需做前滚恢复，请参看 9.4.3 节。

提醒：重定向恢复对于源端和目标端表空间容器不匹配的情况特别适用。

注意: 重定向恢复不支持自动存储表空间, 可在恢复命令中通过 ON path-list 指定存储路径。

9.4.3 前滚恢复

前面我们几次提到前滚恢复，本节将深入讨论它。为什么需要前滚恢复呢？回想我们本章开始的第一个场景，晚上 12 点做了备份，第二天出故障，如果只是恢复到 12 点，而不执行前滚恢复，那么 12 点备份之后的数据都将丢失，这对绝大多数关键业务来说是不可接受的。

前滚恢复需要在版本恢复的基础上执行。在前滚恢复过程中，日志再次发挥了它的重要作用，所有事务操作都记录在日志中，前滚恢复会遍历并重做时间戳之前归档日志中的所有事务，满足用户对数据库恢复操作的要求。

哪些情况必须要做前滚恢复呢？以及最小前滚到什么时候呢？

- 对数据库在线备份进行版本恢复后，数据库会处于 `rollforward pending` 状态，前滚恢复最小要前滚到在线备份结束的时间点才可以。这是因为在线备份过程中，还会产生一些事务日志，只有将备份过程中产生的这些日志重做，才能保证数据库的一致性。
- 对数据库离线备份在（归档日志模式下），可以在 `restore` 命令中指定 `without rollingforward`，这样就不需要再做前滚了。
- 对表空间恢复，恢复的表空间会处于 `rollforward pending` 状态。表空间恢复有个最小时间点概念（`minimum restore time`），本节后面会详细解释。
- 写数据时磁盘介质损坏可能会使数据库处于 `rollforward pending` 状态。

从应用的角度看，如果在进行了版本恢复后，进行数据库连接时，会出现 `rollforward pending` 错误，那就表示需要做前滚了，一个库只有脱离了 `rollforward pending` 状态，才可以正常访问。

前滚命令的语法如下:

```
>>--ROLLFORWARD--+-DATABASE-+--database-alias----->
      '-DB-----'

>--+-----+----->
      '-USER--username-+-'
      '-USING--password-'

>--+-----+----->
      | .ON ALL DBPARTITIONNUMS-. .-USING UTC TIME---. |
      +-TO--+-isotime-+-----+-----+-----+-----+
      | |                                     '-USING LOCAL TIME-' | +-AND COMPLETE-+ |
      | |                                     .-ON ALL DBPARTITIONNUMS-. | '-AND STOP-----' |
```

```

|      +-END OF BACKUP--+-----+-----+-----+-----+
|      '-END OF LOGS--+-----+-----+-----+-----+
|      '-| On Database Partition clause |-|
|      +-COMPLETE-----+-----+-----+-----+-----+
|      +-STOP-----+-----+-----+-----+-----+
|      +-CANCEL-----+-----+-----+-----+-----+
|      |      .-USING UTC TIME--.|
|      '-QUERY STATUS--+-----+-----+-----+-----+
|      |      '-USING LOCAL TIME-'
|
>--+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|      '-TABLESPACE--+-----+-----+-----+-----+-----+
|      |      .-|
|      |      V
|      |      '-(---tablespace-name---)---+-----+-----+
|      |      '-ONLINE-'
|
>--+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|      '-OVERFLOW LOG PATH--(--log-directory--+-----+-----+-----+-----+
|      |      '-|--| Log Overflow clause |-|
|
>--+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|      '-NORETRIEVE-'
|
>--+-----+-----+-----+-----+-----+-----+-----+-----+-----+<
|      '-RECOVER DROPPED TABLE--drop-table-id--TO--export-directory-'

```

通过以上语法可以看到，Rollforward TO 选项支持 3 种：

(1) 前滚到某个时间点。在默认情况下，前滚时间是格林尼治标准时间，即 UTC；也可以通过 using local time 选项指定本地时间。我们建议使用本地时间，因为比较直观，不需要换算成 UTC 时间。

以下是几种比较常见的时区。

- CST: China Standard Time, 中国标准时间，比 UTC 快 8 个小时。
- UTC: Coordinated Universal Time, 格林尼治标准时间。
- PST: Pacific Standard Time (美国冬天采用的时间)，比 UTC 慢 8 个小时。
- PDT: Pacific Daylight Time, 美国夏天采用的时间，比 UTC 慢 7 个小时。

在下例中，当地时间是北京时间，比 UTC 快 8 个小时：

```

db2inst1@dpf1:/data1> db2 "select current timestamp as local_time, (current
timestamp - current timezone) as UTC_time from sysibm.sysdummy1"

LOCAL_TIME                UTC_TIME
-----
2011-01-20-12.34.18.890729 2011-01-20-4.34.18.890729

```

在 rollforward 命令中，以下两种方式是等价的（注意前滚的日期格式）：

```

db2inst1@dpf1:/data1> db2 rollforward db sample to 2011-01-20-04.34.18.890729
db2inst1@dpf1:/data1> db2 rollforward db sample to 2011-01-20-12.34.18.890729 using
local time

```

(2) 前滚到日志结尾 (to end of logs)。会遍历需要的归档日志和活动日志，能够最大程度的恢复数据，使用也比较简单。

(3) 前滚到备份结尾 (to end of backup)。是 DB2 9.5 版本引进的，其目的是能够快速方便的前滚到备份结束的时间点，也就是前滚操作结束要求的最小恢复时间。前面我们讲到，DB2 会在 online backup 结束的时候将备份过程中产生的日志打包到备份介质中，在恢复时，必须至少要前滚到备份结束的时间点，这个点就叫前滚操作结束要求的最小恢复时间。在某些场景下，比如灾难发生或创建测试环境时，只需要将数据库恢复到可用状态即可，这时通过 rollforward to end of backup 就可以完成恢复。这个选项对于多分区数据库的恢复尤其方便。

举例如下：

```
db2inst1@dpf1:/data1> db2 restore db sample
DB20000I The RESTORE DATABASE command completed successfully.
db2inst1@dpf1:/data1> db2 rollforward db sample to end of backup and stop
DB20000I The ROLLFORWARD command completed successfully.
```

9.5 版本前，怎么查看前滚操作要求的最小恢复时间？有一种方法是通过 list history backup all for <db_name> 查看备份历史，其中有个字段是 End Time，表示备份结束的时间，这也是前滚要求的最小恢复时间点，代码如下：

```
Op Obj Timestamp+Sequence Type Dev Earliest Log Current Log Backup ID
-----
B D 20110120100404001 N D S0000006.LOG S0000006.LOG
-----
Contains 6 tablespace(s):

00001 SYSCATSPACE
00002 USERSPACE1
00003 IBMDB2SAMPLEREL
00004 IBMDB2SAMPLEXML
00005 TS1
00006 SYSTOOLSPACE
-----
Comment: DB2 BACKUP SAMPLE ONLINE
Start Time: 20110120100404
End Time: 20110120100413
Status: A
-----
EID: 11 Location: /data1
```

另外一个小技巧是前滚时指定一个早于备份结束点的时间，这时 DB2 会报 SQL1275N 错误，并且在错误信息里提示前滚操作结束要求的最小恢复时间。本例中会提示最小恢复的时间点是 2011-01-20-10.04.12.000000 Local。代码如下：

```
db2inst1@dpf1:/data1> db2 rollforward db sample to 2011-01-20-10.04.11.000000 using
local time
SQL1275N The stoptime passed to roll-forward must be greater than or equal to
"2011-01-20-10.04.12.000000 Local", because database "SAMPLE" on node(s) "0"
contains information later than the specified time.
```

在结束前滚前，可以多次执行前滚命令，但后一次指定的前滚时间一定要大于前一次，也就是前滚操作只能往前，不能回退，即使中途出错中断了，也不能够指定比前一次时间点靠前的时间戳。在结束前滚前，数据库处于 rollforward pending 状态，当发出 STOP 或 COMPLETE 的时候，结束前滚操作，数据库脱离 rollforward pending 状态（注：rollforward 的 STOP 和 COMPLETE 选项没有任何差别）。

注意：前滚的时间点不能比前一次靠前。前滚以后，如果中途出错中断了，也不能够指定比前一次指定的时间点靠前的时间戳。

前滚时，DB2 获取日志的顺序如图 9.6 所示。

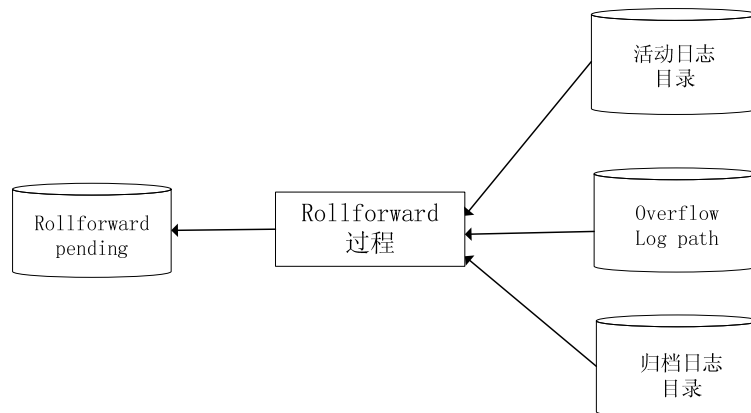


图 9.6 前滚获取日志的顺序

（1）首先从活动日志目录查找日志文件，如果找到，则使用该目录下的日志。

（2）如果找不到，检查 rollforward 命令中是否指定了 **OVERFLOW LOG PATH** 选项。如果指定了，则从 **OVERFLOW LOG PATH** 指定的目录中查找日志。

（3）最后检查 **LOGARCHMETH1** 数据库参数指定的归档日志位置。

对于从 2) 和 3) 路径获取的日志将要先放入 DB2 活动日志目录，当前滚完成后，DB2 将从活动目录删除这些日志文件。

下面讲解表空间前滚恢复。

为保证表空间前滚操作的成功，前滚的时间点必须大于或等于表空间前滚操作所需的最小恢复时间点。最小恢复时间点是系统目录表对表空间或其中表的最后一次更新操作的时间戳，也就是任何 **ROLLFORWARD** 命令要成功执行（不一定是前滚结束或完成）必须恢复到的时间戳。

举例来说，我们在 T1 时间点对数据库做了一次备份，然后 T2 时间点在 TS2 表空间中新建了一张表 TEST2。接着我们要恢复 T2 表空间，那么是否可以前滚到 T1 备份的时间点就结束呢？答案是不可以的。因为系统表中已经包含了 TEST2 表的定义，如果前滚 TS2 表空间到 T1 时间

点，这时 TEST2 表还没有创建，就会出现不一致了，因此建表的那一时刻就是该表所在表空间的最小恢复时间点，恢复到该点之前的任何一点都是不允许的。可通过以下表函数或 list tablespaces 查看表空间的最小恢复时间点：

```
db2inst1@dpf1:/data1> db2 "select substr(tablespace_name,1,15) as tablespace_name,
min_recovery_time from table (snapshot_tbs_cfg('SAMPLE',-1) ) as a"

TABLESPACE_NAME MIN_RECOVERY_TIME
-----
SYSCATSPACE      -
TEMPSPACE1       -
USERSPACE1       -
IBMDB2SAMPLEREL  2011-01-20-10.03.53.000000
IBMDB2SAMPLEXML  -
TS1              2011-01-19-18.17.06.000000
6 record(s) selected.
```

可通过 list utilities show detail 监控前滚进度，代码如下：

```
db2inst1@dpf1:~/sqlllib/db2dump> db2 list utilities show detail

ID                                = 44
Type                              = ROLLFORWARD RECOVERY
Database Name                     = SAMPLE
Partition Number                  = 0
Description                       = Database Rollforward Recovery
Start Time                       = 01/19/2011 08:25:31.392040
State                             = Executing
Invocation Type                   = User
Progress Monitoring:
  Phase Number [Current]          = 1
    Description                   = Forward
    Completed Work                 = 23233132 bytes
    Start Time                    = 01/19/2011 08:25:31.392050

  Phase Number                    = 2
    Description                   = Backward
    Completed Work                 = 0 bytes
    Start Time                    = Not Started
```

9.4.4 删除表恢复 (dropped table recovery) ■ ■ ■

在实际场景中，经常遇到不小心删除表的场景，在抓狂的同时，肇事者一定在盘算这样几个问题：这个表还能恢复吗？怎么恢复？需要多长时间？会不会影响我的年终奖啊？公司会不会开除我啊？问题已经出现了，想别的都没用，先把问题解决才是正事。

对于表误删除这个问题，依赖于备份情况，有以下几种解决方案：

(1) 如果该表的表结构可以重建, 而且数据可以从其它渠道获得, 那可以考虑重建。

(2) 从来没有做过备份, 也没办法从其它渠道获取数据。这是最糟糕的情况, 我们曾经在几个客户的老旧系统上遇到过。

(3) 如果做过数据库备份并且是循环日志模式, 那么, 考虑可以恢复最近的一次备份, 但这个备份之后的数据将全部丢失, 这种方案需要用户需要仔细权衡。我们曾经遇见过的客户几个月才备份一次, 那就意味着需要通过渠道回补丢失的数据。

(4) 如果做过数据库备份并且是归档日志模式, 这种情况下, 有经验的 DBA 可能都会想到通过对数据库做 restore+rollforward, 前滚到表删除之前的状态, 但表删除之后所做的数据变化都将丢失, 而且数据量很大的话, 恢复的时间会比较长, 影响系统的正常使用。

对于第 2 和第 3 种情况, 我们应该尽量避免, 作为 DBA, 保证数据可用是基本前提。

对于第 4 种情况, 是否有更好的解决方案呢? 有没有可能只恢复这个表所属的表空间呢? 回想上一节我们给大家介绍表空间前滚时, 提到的最小恢复时间点概念: 当有 drop、create、alter 等 DDL 操作发生时, DB2 会在系统表中更新定义, 并更新表空间的最小恢复时间点, 只有前滚到最小恢复时间之后才能才能保证系统表和数据库对象保持一致。本例中, 直接通过前滚表空间的方式来复原删除表是不可能的, 因为该表删除后, 系统表空间中已经不存在这张表的定义。

如果我们能够得到这张表的定义, 并且取得删除前的数据, 那就可以重建了, 这正是 dropped table 恢复原理。简单地说, dropped table recovery 利用删除表恢复历史文件和表空间前滚机制(仅在归档模式下可用)分别获得删除表的定义和数据。该机制要求表空间的 dropped table recovery 选项必须打开, 缺省情况下建表空间时该选项是打开的, 如果没有打开, 也可以使用 ALTER TABLESPACE <TBSPACENAME> DROPPED TABLE RECOVERY ON 进行更改。

以下我们为大家演示一个表删除恢复过程。

(1) 首先创建一张 t2 表, 并插入了 1 条数据, 然后做一个在线备份, 代码如下:

```
db2inst1@dpf1:/data1> db2 "create table t2 (id int, name char(20)) "
```

```
db2inst1@dpf1:/data1> db2 "insert into t2 values(10, 'zhang' ) "
```

```
db2inst1@dpf1:/data1> db2 backup db sample online
```

```
Backup successful. The timestamp for this backup image is : 20110120233036
```

(2) 删除表 T2。当启动了 dropped table recovery 选项后, 删除表时, DB2 会在日志中记录一些额外的信息, 如 table name、timestamp、TID 和 FID 等信息, 并在历史文件中记录 table name、timestamp、TID、FID 和建表语句的 DDL, 这些信息将在随后的恢复中用到。

删除表 t2, 代码如下:

```
db2inst1@dpf1:/data1> db2 drop table t2
```

在历史文件查找删除表 t2 的定义, 代码如下:

```
db2inst1@dpf1:/data1> db2 list history dropped table all for sample

List History File for sample

Number of matching file entries = 1

Op Obj Timestamp+Sequence Type Dev Earliest Log Current Log Backup ID
-----
D T 20110120233237 0000000000000c32000030006
-----
"DB2INST1"."T2" resides in 1 tablespace(s):

00001 IBMDB2SAMPLEREL
-----
Comment: DROP TABLE
Start Time: 20110120233237
End Time: 20110120233237
Status: A
-----
EID: 43

DDL: CREATE TABLE "DB2INST1"."T2" ( "ID" INTEGER , "NAME" CHAR(20) ) IN
"IBMDB2SAMPLEREL" ;
-----
```

(3) 恢复 t2 表所属的表空间，并将表空间前滚，在前滚过程中，指定 RECOVER DROPPED TABLE <Backup Id>选项，并指定 T2 数据的导出存放目录。Backup Id 可以从第(2)步的恢复历史文件中找到，每个 backup id 对应一张删除表。:

```
db2inst1@dpf1:/data1> db2 " restore db sample tablespace(IBMDB2SAMPLEREL) taken at
20110120233036 "
```

前滚时指定要恢复的表 backup id (对应 list dropped table 中的 backup id，每个 backup id 对应一张删除表)。在前滚的过程将表数据导出到一个目录里，通过 to 指定，代码如下:

```
db2inst1@dpf1:/data1> mkdir data
db2inst1@dpf1:/data1> db2 "ROLLFORWARD DATABASE sample TO END OF LOGS
TABLESPACE(IBMDB2SAMPLEREL) RECOVER DROPPED TABLE 000000000000c32000030006 to
/data1/data"

db2inst1@dpf1:/data1/data/NODE00000> pwd
/data1/data/NODE00000
db2inst1@dpf1:/data1/data/NODE00000> ls
data
```

切换到数据目录，查看 data 文件内容。发现这正是备份前插入的内容:

```
db2inst1@dpf1:/data1/data/NODE00000> cat data
10,"zhang"
```

(4) 重建表结构，并导入数据。

第(2)步中，通过恢复历史文件可以取得删除表的定义，将其存入脚本文件，重建即可，代码如下：

```
db2inst1@dpf1:/data1/data/NODE0000> cat createt2.ddl
CREATE TABLE "DB2INST1"."T2" ( "ID" INTEGER , "NAME" CHAR(20) ) IN "IBMDB2SAMPLEREL";

db2inst1@dpf1:/data1/data/NODE0000> db2 -tvf createt2.ddl
CREATE TABLE "DB2INST1"."T2" ( "ID" INTEGER , "NAME" CHAR(20) ) IN "IBMDB2SAMPLEREL"
DB20000I The SQL command completed successfully.
```

第(3)步中，通过表空间前滚可以获取删除表的数据，通过 load/import 加载即可，代码如下：

```
db2inst1@dpf1:/data1/data/NODE0000> db2 "load from data of del insert into t2 copy
yes to /data1/data"
db2inst1@dpf1:/data1/data/NODE0000> db2 "select * from t2"
ID          NAME
-----
10 zhang
```

9.5 常见备份恢复场景及遇到的问题

本部分总结了常用的备份恢复场景，读者可根据自己的需求选择对应的解决方案，希望本部分能够对您选择恢复方案及解决恢复过程中遇到的问题有所帮助。

9.5.1 宕机后数据库连接 hang 的处理 ■ ■ ■

问题描述：

某客户异常宕机，开机后连接数据库，执行了 10 分钟还没结束，以为机器问题，因此关机，重新连，连续几次都未果，打电话求助。

解决方案：

查看 db2diag.log，并通过 db2 list utilities show detail 查看当前正在执行的操作，发现在做 crash recovery。耐心等待，大概 30 分钟后，连接成功。

原因分析：

当实例出现异常或电源故障，重启连接数据库后 DB2 会自动做崩溃恢复。

举个例子说明，注意：本实验具有很大的杀伤力，一定不要在生产上测试。有条件的可在自己的机器上实验一下，了解一下原理即可。

在 A 窗口执行一个事务，B 窗口找到 db2sysc 进程（kill -9 <db2sys_pid>）。

然后启动实例，连接数据库。以下是观察的 crash recovery 进度：

```
db2inst1@dpf1:/data1/active/active.log/NODE0000> db2 list utilities show detail

ID                                = 1
Type                              = CRASH RECOVERY
Database Name                     = SAMPLE
Partition Number                  = 0
Description                       = Crash Recovery
Start Time                       = 01/18/2011 02:44:11.004073
State                            = Executing
Invocation Type                   = User
Progress Monitoring:
  Estimated Percentage Complete    = 10
  Phase Number [Current]          = 1
    Description                    = Forward
    Total Work                     = 8179684 bytes
    Completed Work                 = 1663068 bytes
    Start Time                    = 01/18/2011 02:44:11.004078

  Phase Number                    = 2
    Description                    = Backward
    Total Work                     = 8179684 bytes
    Completed Work                 = 0 bytes
    Start Time                    = Not Started
```

db2diag.log 信息如下（LowtranLSN 和 MinbufferLSN 的含义请参看日志原理部分）：

```
2011-01-18-02.44.11.002927-480 I1198014G500 LEVEL: Warning
PID      : 22004          TID   : 2946493344 PROC  : db2sysc 0
INSTANCE: db2inst1       NODE   : 000      DB    : SAMPLE
APPHDL   : 0-7          APPID: *LOCAL.db2inst1.110118104410
AUTHID   : DB2INST1
EDUID    : 19           EDUNAME: db2agent (SAMPLE) 0
FUNCTION: DB2 UDB, recovery manager, sqlpresr, probe:410
MESSAGE  : Crash recovery started. LowtranLSN 0000000002A83FB8 MinbuffLSN
          000000000268702C
```

9.5.2 循环日志模式下的离线备份恢复 ■ ■ ■

某客户每天做离线备份，突然某天出现故障，由于是循环日志模式，只能恢复到最近一个成功备份的时间点，这个时间点之后的数据将会丢失。因此，为保证数据可恢复，不建议交易系统用循环日志。

模拟：

（1）首先检查日志参数，确认是循环日志模式，代码如下：

```
inst20@db2server:~> db2 get db cfg for sample |grep -i status
Log retain for recovery status          = NO
User exit for logging status            = NO

inst20@db2server:~> db2 get db cfg for sample |grep -i log
Log retain for recovery enabled          (LOGRETAIN) = OFF
User exit for logging enabled            (USEREXIT) = OFF
First log archive method                 (LOGARCHMETH1) = OFF
```

(2) 创建测试表 t1，断开对数据库的连接，对数据库做离线备份，然后插入新记录'bbb'，代码如下：

```
inst20@db2server:/data1> db2 "create table t1 (col1 char(10) )"
inst20@db2server:/data1> db2 "insert into t1 values('aaa') "
inst20@db2server:~> db2 force applications all
inst20@db2server:/data1> db2 backup db sample to /data1/dbbak compress
Backup successful. The timestamp for this backup image is : 20101214001512

inst20@db2server:/data1> db2 connect to sample
inst20@db2server:/data1> db2 "insert into t1 values('bbb') "
```

(3) 出现故障，进行数据库恢复。恢复后，bbb 数据将丢失，代码如下：

```
inst20@db2server:/data1/dbbak> db2 force applications all
inst20@db2server:/data1/dbbak> db2 restore db sample from /data1/dbbak taken at
20101214001512
inst20@db2server:/data1/dbbak> db2 connect to sample
inst20@db2server:/data1/dbbak> db2 "select * from t1"
aaa
```

9.5.3 归档日志模式下的备份恢复

某客户每天做在线备份，突然某天出现故障，客户希望恢复到某个时间点。

模拟：

(1) 首先对数据库做备份，然后插入 2 行记录，并记录插入时间，代码如下：

```
db2inst1@dpf1:/data1> db2 backup db sample online
Backup successful. The timestamp for this backup image is : 20110120100404

db2inst1@dpf1:/data1> db2 "insert into t1 values(111,'aaaa') "
DB20000I The SQL command completed successfully.
db2inst1@dpf1:/data1> date
Thu Jan 20 10:04:50 PST 2011

db2inst1@dpf1:/data1> db2 "insert into t1 values(222,'bbbb') "
DB20000I The SQL command completed successfully.
db2inst1@dpf1:/data1> date
Thu Jan 20 10:06:13 PST 2011
```

(2) 进行恢复并前滚到第 1 条记录插入后、第 2 条记录插入前的时间点，这样便于验证我们第 1 条记录是否插入成功，代码如下：

```
db2inst1@dpf1:/data1> db2 restore db sample taken at 20110120100404
DB20000I The RESTORE DATABASE command completed successfully.

db2inst1@dpf1:/data1> db2 rollforward db sample to 2011-01-20-10.04.50.000000 using
local time

                                Rollforward Status

Input database alias              = sample
Number of nodes have returned status = 1

Node number                       = 0
Rollforward status                 = DB working
Next log file to be read           = S0000007.LOG
Log files processed                 = S0000006.LOG - S0000006.LOG
Last committed transaction         = 2011-01-20-10.04.46.000000 Local

DB20000I The ROLLFORWARD command completed successfully.

db2inst1@dpf1:/data1> db2 rollforward db sample stop
db2 "

                                Rollforward Status

Input database alias              = sample
Number of nodes have returned status = 1

Node number                       = 0
Rollforward status                 = not pending
Next log file to be read           =
Log files processed                 = S0000006.LOG - S0000007.LOG
Last committed transaction         = 2011-01-20-10.04.46.000000 Local

DB20000I The ROLLFORWARD command completed successfully.
```

(3) 当前滚恢复结束后，连接数据库，查看表数据，发现只有第 1 条记录插入，恢复成功，代码如下：

```
db2inst1@dpf1:/data1> db2 connect to sample
db2 "select
Database Connection Information

Database server      = DB2/LINUX 9.7.1
SQL authorization ID = DB2INST1
Local database alias = SAMPLE

db2inst1@dpf1:/data1> db2 "select * from t1"
```

```
ID          NAME
-----
111 aaaa
1 record(s) selected.
```

(4) 一旦恢复完成, 就会生成新的日志链, 第 2 条记录将会永久地丢失, 再也无法找回。我们尝试重新恢复, 并前滚到日志结尾, 观察结果, 只有第 1 条记录。这就提示我们, 前滚操作的时间点一定要谨慎, 一旦完成前滚, 之后的数据再也无法使用, 代码如下:

```
db2inst1@dpf1:/data1> db2 restore db sample taken at 20110120100404
DB20000I The RESTORE DATABASE command completed successfully

db2inst1@dpf1:/data1> db2 rollforward db sample to end of logs and stop
DB20000I The ROLLFORWARD command completed successfully

db2inst1@dpf1:/data1> db2 connect to sample
db2inst1@dpf1:/data1> db2 "select * from t1"
ID          NAME
-----
111 aaaa
1 record(s) selected
```

9.5.4 归档日志模式下前滚恢复的几个时间戳 ■ ■ ■

前滚恢复时的时间戳特别关键, 也很容易出错, 本节对此进行了汇总, 并给出了每种问题的解决方案。

前滚的时间要在最小恢复时间点之后, last committed 事务时间点之前。如果前滚到最小恢复时间点前, 则报 SQL1275N 错误; 如果超过了事务日志最后提交的时间点, 则报 SQL4970N 错误, 只能 rollforward ... stop。

模拟如下:

(1) 首先对数据库做备份, 然后插入 3 行记录, 并记录插入时间, 代码如下:

```
-- 首先做离线全备, 备份时间戳为 20101214014104
inst20@db2server:/data1/dbbak> db2 backup db sample online to /data1/dbbak compress
Backup successful. The timestamp for this backup image is : 20101214014104

inst20@db2server:/data1/dbbak> db2 connect to sample
inst20@db2server:/data1/dbbak> db2 "create table t2 (col1 char(10) )"
inst20@db2server:/data1/dbbak> db2 "insert into t2 values('111') "
inst20@db2server:/data1/dbbak> db2 "values current timestamp"
2010-12-14-01.42.32.158244
inst20@db2server:/data1/dbbak> db2 "insert into t2 values('222') "
inst20@db2server:/data1/dbbak> db2 "values current timestamp"
```

```
2010-12-14-01.42.47.923480
inst20@db2server:/data1/dbbak> db2 "insert into t2 values('333') "
inst20@db2server:/data1/dbbak> db2 "values current timestamp"
2010-12-14-01.43.03.368174
```

(2) 查看恢复历史，找到备份结束（End Time）的时间点，代码如下：

```
inst20@db2server:/data1/dbbak> db2 list history backup all for sample
Op Obj Timestamp+Sequence Type Dev Earliest Log Current Log Backup ID
-----
B D 20101214014104001 N D S0000003.LOG S0000003.LOG
-----
Contains 4 tablespace(s):

00001 SYSCATSPACE
00002 USERSPACE1
00003 IBMDB2SAMPLEREL
00004 SYSTOOLSPACE
-----
Comment: DB2 BACKUP SAMPLE ONLINE
Start Time: 20101214014104
End Time: 20101214014114
Status: A
-----
EID: 61 Location: /data1/dbbak
```

(3) 恢复数据库，并前滚到时间戳“2010-12-14-01.41.04.000000”，由于该时间小于备份结束的最小恢复时间点，因此报 SQL1275N 错误。该错误也提示了最小恢复的时间点是“2010-12-14-01.41.14.000000 Local”，代码如下：

```
--恢复数据库
inst20@db2server:/data1/dbbak> db2 restore db sample taken at 20101214014104

inst20@db2server:/data1/dbbak> db2 rollforward db sample to
2010-12-14-01.41.04.000000 using local time
SQL1275N The stoptime passed to roll-forward must be greater than or equal to
"2010-12-14-01.41.14.000000 Local", because database "SAMPLE" on node(s) "0"
contains information later than the specified time.
```

(4) 接着前滚到一个较新的时间点（该时间点超出了最后一个日志记录的时间），这时 DB2 报 SQL4970N 错误，该错误的意思是前滚无法达到指定的时间点，代码如下：

```
--接着，我们前滚到当前时间之后的某个点，如 2010-12-14-02.32.18.000000，这时会报 SQL4970N 错误。
inst20@db2server:/data1/dbbak> db2 rollforward db sample to 2010-12-14-02.32.18.
000000 using local time
SQL4970N Roll-forward recovery on database "SAMPLE" cannot reach the specified stop
point (end-of-log or point-in-time) on database partition(s) "0". Roll-forward
recovery processing has halted on log file "S0000005.LOG"
```

(5) 既然前滚时间超出了范围, 我们尝试向早一点的时间恢复。这时 DB2 报 SQL1226N 错误, 验证了我们之前说的只能向后 rollforward, 而无法回退, 代码如下:

```
-- 出现以上 SQL4970N 错误后, 再前滚到靠前一点的时间戳, 就会报 SQL1266N, 也就是前滚只能靠后不能往前
inst20@db2server:/data1/dbbak> db2 rollforward db sample to
2010-12-14-01.42.32.158244 using local time
SQL1266N Database "SAMPLE" has been rolled forward to "2010-12-14-02.32.18.000000
Local", which is past the specified point-in-time
```

(6) 对于 SQL1226N 错误, 只能通过 rollforward ... stop 完成前滚, 恢复的时间点类似 to end of logs, 代码如下:

```
inst20@db2server:/data1/dbbak> db2 rollforward db sample stop
DB20000I The ROLLFORWARD command completed successfully
```

(7) 连接数据库, 数据前滚成功。

```
inst20@db2server:/data1/dbbak> db2 connect to sample
inst20@db2server:/data1/dbbak> db2 "select * from t2"
111
222
333
```

9.5.5 同版本不同实例下的数据库备份恢复 (表空间是自动存储管理)

我们经常遇到的一个场景是同机器下不同实例的数据库恢复, 比如在 inst20 上对数据库做了一个备份, 恢复到 inst21 实例。在 UNIX/Linux 平台下, 这会遇到如下的 SQL0970N 权限问题, 相当于一个用户在另外一个用户下写文件, 这是没有权限的。

```
db2server:/data2 # chown -R inst21:db2iadm2 /data2
db2server:/data2 # cp /data1/dbbak/SAMPLE.0.inst20.NODE0000.CATN0000.
20101214014104.001 /data2
db2server:/data2 # su - inst21
inst21@db2server:/data2> db2 restore db sample from /data2 taken at 20101214014104
SQL0970N The system attempted to write to a read-only file. SQLSTATE=55009
```

如果数据库表空间都是自动存储管理的, 那么解决方案很简单, 只需在恢复时通过 ON 指定目录, 这样这个库的数据库目录和自动存储路径都将存在 ON 指定的目录下:

```
inst21@db2server:/data1> db2 restore db sample from /data2 taken at 20101214014104
on /home/inst21
DB20000I The RESTORE DATABASE command completed successfully
```

如果是在线备份介质, 恢复时需要指定 logtarget 将介质中的文件抽取, 前滚时指定 overflow log path 选项进行日志的前滚:

```
inst21@db2server:/data2> db2 restore db sample from /data2 taken at 20101214014104
on /home/inst21 logtarget /data2/log
```

```
inst21@db2server:/data2> db2 "rollforward db sample to end of logs overflow log path
('/data2/log' )"
inst21@db2server:/data2> db2 "rollforward db sample stop overflow log path
('/data2/log' )"

```

9.5.6 同版本不同实例下的数据库备份恢复（表空间是非自动存储管理） ■ ■ ■

在这种场景下，表空间是非自动存储管理的，即表空间的路径是独立创建的，如果还按 9.5.5 节的恢复方法，就会出现表空间冲突问题（一个容器只能归属于一个表空间），在这种情况下，我们只能用重定向恢复了（ON 的路径同 9.5.5 节）。

```
inst21@db2server:/data1> db2 restore db sample from /data2 taken at 20101214014104
on /home/inst21 redirect generate script /home/inst411/redirect.ddl
DB20000I The RESTORE DATABASE command completed successfully.

```

在 redirect.ddl 脚本中更改表空间路径即可，比如将其改为 /inst21/ts1/cont0：

```
SET TABLESPACE CONTAINERS FOR 5
-- IGNORE ROLLFORWARD CONTAINER OPERATIONS
USING (
  FILE  '/data1/ts1/cont0'                                1280
);

```

然后执行 db2 -tvf redirect.ddl 进行重定向恢复。

如果是在线备份介质，仍然需要进行 rollforward。

9.5.7 不同版本不同实例下的数据库恢复 ■ ■ ■

DB2 支持低版本的备份在高版本上恢复，如从 v8.1→v9.1、v8.1→v9.5、v9.1→v9.5、v9.1→v9.7 等，高版本到低版本恢复则不可以。注意：在不同版本恢复时，必须对原库做离线完全备份。

以下是从 9.1 版本到 9.7 版本的恢复案例（自动存储管理表空间）：

从 DB2 9.1 版本 inst20 实例下的 sample 恢复到 DB2 9.7 版本 db2inst1 实例下，报如下错误：

```
db2inst1@dpf1:~> db2 restore db sample taken at 20101019024413 on /home/db2inst1
SQL2519N The database was restored but the restored database was not upgraded to
the current release. Error "-1704" with tokens "3" is returned

```

查看错误号代表的含义，是日志满了。原来是低版本从高版本恢复过程中，实际发生了一次 upgrade，upgrade 操作需要记录日志，而当前日志大小设置偏小导致了这个错误：

```
inst411@dpf1:/data1> db2 ? sql1704
SQL1704N Database upgrade failed. Reason code "<reason-code>"
Explanation:
3      database logs are full

```

在当前库修改日志参数，并执行数据库升级，成功执行：

```
inst411@dpf1:~> db2 update db cfg for sample using logfilesiz 4096 logprimary 5
logsecond 10
inst411@dpf1:~> db2 upgrade db sample
SQL1103W The UPGRADE DATABASE command was completed successfully
```

备份介质可以恢复为一个新库，也可以用来替换一个已有的数据库。

如果恢复到新库，则新库参数来自原库的配置，日志路径来自原库或采用默认值。

如果恢复到已有库，则已有库的表、索引、大对象数据被删除，并用备份介质的数据替换，并保留恢复历史文件（除非此文件损坏）。如果当前备份介质来自当前库，则恢复时保留当前库配置文件，否则从备份介质中复制配置文件。

如果表空间包含自定义路径，那么还是需要用重定向恢复。

9.5.8 从生产库到测试库恢复的案例分析 ■ ■ ■

本案例来自某移动公司从生产库到测试机的恢复。客户的生产库跑在一个比较 powerful 的机器上，32C/64GB 内存，而测试机配置比较弱，只有本地存储，内存只有 4GB。客户希望将生产库恢复到测试机上，进行一些功能验证。在恢复过程中遇到几个问题，下面与大家分享一下解决过程。

（1）对生产库做在线备份，并将备份介质传到目标端进行恢复，在恢复过程发现测试机的存储不够，第一次恢复失败。

（2）在生产库上检查表空间使用页（used pages）大小，发现实际使用的页数较小，决定采用重定向恢复，这样可以修改目标端表空间容器大小，减小需要的存储空间。

```
Tablespace ID          = 5
Name                   = APPSD
Type                   = Database managed space
Contents               = All permanent data. Regular table space.
State                  = 0x0000
  Detailed explanation:
    Normal
Total pages            = 1500000
Useable pages          = 1499968
Used pages             = 27136
Free pages             = 1472832
High water mark (pages) = 80256
Page size (bytes)      = 32768
Extent size (pages)    = 32
Prefetch size (pages)  = 32
Number of containers   = 1
Minimum recovery time   = 2009-12-16-04.03.44.000000
Tablespace ID          = 6
```

图 9.6 表空间使用页大小

(3) 在重定向脚本中, 修改表空间容器大小为 30 000 页, 但恢复过程中一直提示 SQL1277N 错误, 说明表空间容器设置有问题。

```
RESTORE DATABASE TDEASDB CONTINUE
SQL1277W  A redirected restore operation is being performed.  Table space
configuration can now be viewed and table spaces that do not use automatic storage
can have their containers reconfigured
```

(4) 回想起表空间 High Water Mark 的概念, 恢复的时候, 必须确保目标容器大于 HWM 值。该表空间 High Water Mark 是 80 256 页, 改成 100 000 页后, 重定向恢复成功。

(5) 由于是在线备份, 恢复成功后, 还需要做日志前滚。前滚时报 SQL1218N 错误。

```
$ db2 rollforward db tdeasdb to end of logs
SQL1218N There are no pages currently available in bufferpool "".
SQLSTATE=57011
```

查看 db2diag.log, 报 “no available buffer pool pages”。

```
2010-12-17-16.23.31.636603+480 I743454A502      LEVEL: Error
PID      : 3684          TID : 59          PROC : db2sysc 0
INSTANCE: db2inst1      NODE : 000        DB   : TDEASDB
APPHDL   : 0-46         APPID: *LOCAL.db2inst1.101217082325
AUTHID   : DB2INST1
EDUID    : 59           EDUNAME: db2agent (TDEASDB) 0
FUNCTION: DB2 UDB, recovery manager, sqlprDoForwardPhase, probe:330
RETCODE  : ZRC=0x8502002C=-2063466452=SQLB_BPFULL
          "no available buffer pool pages"
```

回想一下, 我们开篇提到的测试机只有 4GB 内存, 而生产库内存较大, 导致 rollforward 时出现 bufferpool 内存不足。

(6) 正常情况下, bufferpool 在数据库激活时就会分配, 当机器没有足够的内存分配给 bufferpool 时, 就会出现数据库无法启动的情况。为解决这个问题, DB2 提供了 4 个隐含的 bufferpool (每种页大小有一个银行 bufferpool), 当设置了 DB2_OVERRIDE_BPF 注册变量后, DB2 就使用隐含 bufferpool, 而不会分配其他 bufferpool。因此, 更改该参数。

```
db2set DB2_OVERRIDE_BPF=1000
db2stop force
db2start
```

(7) 继续前滚, 恢复成功。然后连接数据库, 将数据库 bufferpool 大小降到测试机可以承受的范围。最后将 DB2_OVERRIDE_BPF 参数修改为初始状态。

```
db2 connect to TDEASDB
db2 alter bufferpool bp4k size 200000
db2set DB2_OVERRIDE_BPF=
db2stop force
db2start
```

本例演示了在资源不对等条件下恢复可能出现的问题及解决方案。

9.5.9 历史文件过大造成数据库停止响应案例分析 ■ ■ ■

某客户数据库不定时停止响应，大概 20 分钟以后自动恢复正常。通过在挂起时所抓到的数据显示，存在很多代理有类似如下的堆栈：

```
<StackTrace>
-----Frame----- -----Function + Offset-----
0xD044144C semop + 0x88
0xD16F6974 sqloNLCKLock + 0x1B8
0xD27C1490 sqluhLock__FPP12SQLUH_HANDLE + 0xEC
0xD27C2904
sqluhAllocateHandleVerifyHistoryFilesAndOptToRecover__FPP12SQLUH_HANDLEP
cCb + 0x148
0xD27C2EE0 sqluhOpen__FPP12SQLUH_HANDLEPcUl + 0x4C
0xD27C3484 sqluhBeginBatch + 0x44
0xD27C0854 sqluhInsertSingle__FP8sqledbcbP11SQLUH_ENTRY + 0x58
0xD2B02598 sqlpInsertLogToHistFile__FP9SQLP_DBCBULcPcT4T3T2PUl + 0x34C
0xD2B020B4 sqlpAddNewLogToHistFile__FP9SQLP_DBCBP10SQLPG_XHDR + 0x50
0xD1DA0348 sqlpgWriteToDisk__FP9SQLP_DBCBP9SQLP_LFPBULN23 + 0x354
0xD1DA1B94 sqlpgPingPong__FP9SQLP_DBCBP9SQLP_LFPBULN23 + 0x144
0xD1DA1628 sqlpgwlp__FP9SQLP_DBCBULUl + 0xAAC
0xD1DA7E80 sqlpgasn2__FPcUl + 0x20AC
0xD21F66D4 sqloCreateEDU__FPFPcUl_vPcUlP13SQLO_EDU_INFOPl + 0x1A8
0xD21F6294 sqloRunGDS__Fv + 0x78
0xD21F6D24 sqloInitEDUServices + 0x14C
0xD21FA640 sqloRunInstance + 0x48C
0x10002910 DB2main + 0x888
0x100035A0 main + 0xC
</StackTrace>
```

这个堆栈说明该进程尝试在向历史文件中写入一条记录：`sqlpInsertLogToHistFile`，但是被锁住了。那么我们需要找到，到底是谁正在使用历史文件。通过排查所有的进程，发现其中一个进程持续读取历史文件：

```
<StackTrace>
-----Frame----- -----Function + Offset-----
0xD1C62F4C strncpy + 0x16C
0xD27C6038
sqluhReadEntry__F12SQLO_FHANDLEP11SQLUH_ENTRYP14SQLUH_WORKAREAPcN34PUl +
0x11D8
0xD2B5D27C sqluh_get_next_history_file_entry + 0x498
0xD2EB6B54 sqluhGetEntryDRDA__FP5sqldaT1P13sqle_agent_cbP5sqlca + 0x940
0xD19F54F4
sqlerKnownProcedure__F1PcPlP5sqldaT4P13sqlerFmpTableP13sqle_agent_cbP5sq
lca + 0xC4C
0xD19E94C4 sqlerCallDL__FP7UCintfcP9UCstpInfo + 0x8B0
```

```

0xD1CB0630 sqljs_ddm_excsqlstt__FP7UCintfcP14sqljsDDMObject + 0x480
0xD1CA1CB4
sqljsParseRdbAccessed__FP13sqljsDrdaAsCbP14sqljsDDMObjectP7UCintfc +
0x58
0xD1CA1B54 sqljsParse__FP13sqljsDrdaAsCbP7UCintfc + 0x284
0xD1CA65C4 sqljsSqlam__FP7UCintfcP13sqle_agent_cbb + 0xA8
0xD1CA6B68 sqljsDriveRequests__FP13sqle_agent_cbP11UCconHandle + 0x88
0xD1CA6A0C sqljsDrdaAsInnerDriver__FP17sqlcc_init_structb + 0xB0
0xD1CA67BC sqljsDrdaAsDriver__FP17sqlcc_init_struct + 0x84

```

从历史文件本身来说，每一条记录都是很小的，而当 DB2 需要从中查找一条特定的记录时，DB2 需要每次从其中读取一条记录，然后进行一些判断。尽管文件本身可能只有几十兆字节，但是由于每一条记录很短小，一条条地读取记录依然是非常耗费时间的。尤其是当多个进程同时需要读取或插入记录的时候，其造成互相等待的结果会使得整个系统暂时停顿，所有的执行进程都在等待系统完成历史文件的读/写。

因此在日常工作中，我们一定要随时留意历史文件的大小。一般建议不要超过 10MB。当历史文件过大时，可以使用 `prune history` 命令删除不需要的数据，在极端情况下可以直接删除 `db2rhist.asc` 和 `db2rhist.bak` 文件。

Prune 历史文件的命令如下：

```
db2 prune history <timestamp> [with force application option] [and delete]
```

在数据仓库类系统中，由于每天都要进行大量的批量数据 LOAD 操作，而 LOAD 操作会记录在历史文件，这会导致这个文件增长特别快。我们曾经遇到过一个客户，以前 LOAD 操作很快，但从某天开始变得很慢，检查系统 CPU/IO 资源占用都很少，查看 `db2diag.log` 也无任何报错和有价值的信息，后经诊断发现历史文件很大，清空历史文件后，LOAD 加载恢复正常。

9.5.10 恢复时解压类包问题 ■ ■ ■

以下是恢复过程中可能遇到的解压包无法找到或格式不匹配的情况：

```

db2 restore db gamestat
SQL2071N An error occurred while accessing the shared library
"/home/db2inst/db2inst/NODE0000/SQL00001/libdb2compr.so". Reason code: "1".

2005-11-23-13.34.23.144168-300 E22440C563      LEVEL: Error (OS)
PID      : 364680      TID   : 1      PROC  : db2bm.630934.0 0
INSTANCE: uatweb      NODE   : 000
FUNCTION: DB2 UDB, oper system services, sqloLoadModule, probe:130      CALLED  : OS,
-, dlopen
OSERR    : ENOEXEC (8) "Exec format error"
MESSAGE  : Attempt to load specified library failed.
DATA #1 : Library name or path, /uatweb/uatweb/NODE0000/SQL00001/libdb2compr.a
DATA #2 : shared library load flags, PD_TYPE_LOAD_FLAGS, 4 bytes
DATA #3 : 2 String, 11 bytes Bad address

```

问题原因:

如果执行了带有 `compress` 选项的备份, 恢复的时候, DB2 会尝试使用保存在备份介质中的类包 `libdb2compr.so` 解压数据库, 如果这个类不能加载, 就会报错。这个类包默认会在 `<inst_home>/sqllib/lib` 目录下。比如 32 位备份在 64 位环境恢复或 64 位向 32 位平台恢复都可能遇到格式不匹配, 或类无法加载的情况。

解决方案:

验证原库所在的操作系统版本、`db2level`, 检查格式是否在两端兼容。

可以在恢复的时候指定类包的位置来解决:

```
db2 restore db db_name from /backup/path into db_name COMPRLIB instance_path/sqllib/
lib/libdb2compr.so
```

9.5.11 备份失败问题 ■ ■ ■

当备份时, 如果表空间容器文件正在被第三方软件锁定, 备份操作无法获得文件时就会发生共享违例问题, 导致备份失败。常见的如安全软件、反病毒和操作系统备份等。所以, 确保备份时不要让第三方软件扫描 DB2 文件。

```
2010-07-07-03.05.06.698000+120 E11582566F467      LEVEL: Info
PID      : 1848                TID : 3168                PROC : db2syscs.
INSTANCE: DB2                 NODE : 000                DB   : SAMPLE
APPHDL   : 0-26152            APPID: *LOCAL.DB2.100707010504
AUTHID   : db2inst1
EDUID    : 3168                EDUNAME: db2agent (SAMPLE) 0
FUNCTION: DB2 UDB, database utilities, sqlubSetupJobControl, probe
MESSAGE  : Starting an online db backup.
...
2010-07-07-03.14.07.906000+120 I11590917F627      LEVEL: Severe
PID      : 1848                TID : 2980                PROC : db2syscs.exe
INSTANCE: DB2                 NODE : 000                DB   : SAMPLE
APPHDL   : 0-26152            APPID: *LOCAL.DB2.100707010504
AUTHID   : db2inst1
EDUID    : 2980                EDUNAME: db2bm.3168.1 (SAMPLE) 0
FUNCTION: DB2 UDB, database utilities, sqlubValidateOrReReadSinglePag
probe:2344
MESSAGE  : ZRC=0x870F0009=-2029060087=SQL0_EOF "the data does not exist"
          DIA8506C Unexpected end of file was reached.
DATA #1 : Page ID, PD_TYPE_SQLB_PAGE_ID, 4 bytes
256
```

还有一种情况是由于锁超时而导致备份失败, 曾经在一个客户遇到, 当时的情况是客户在晚上 12 点同时启动备份和 `runstats` 运维任务, 结果 `runstats` 占有了锁, 导致备份锁超时失败。

所以，建议将一些运维工具安排在不同的时间窗口执行。

9.6 小结

备份与恢复是保证企业数据生命线的最后一道防线。当所有的可用调错机制都无法顺利解决问题的时候，唯一能够让系统重新恢复正常的就是恢复与前滚操作。因此，作为一名 DBA，一定要保证系统随时处于可以被恢复和前滚的状态。

本节详细介绍了日志的原理和机制，备份、恢复的场景和命令演示，每一个重要的知识点都有案例供大家理解和模拟学习。最后，总结了数 10 个作者亲身经历的备份恢复场景，具有很强的实战型和指导意义，相信会对大家有所帮助和启发。

9.7 判断题

(1) DB2 日志中记录着数据库的每一次数据更改。

T: 正确

F: 错误

(2) 只有当事务已经提交后才会进行日志归档。

T: 正确

F: 错误

(3) DB2 的备份主要分为离线与在线两种。

T: 正确

F: 错误

(4) 前滚恢复可以在不使用版本恢复的前提下直接进行。

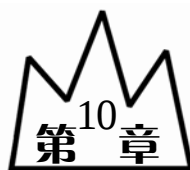
T: 正确

F: 错误

(5) 表被意外删除后可以使用 TABLE RECOVERY 功能进行恢复。

T: 正确

F: 错误



DB2 日常运维

随着数据量的累积，原本执行的很好的语句，性能突然变得很慢，这是很多客户都曾经遇到的问题。究其原因，发现这些客户很少对数据库做常规运维管理。对于 DBA 来说，数据库就是我们的孩子，需要精心呵护，你不关心它，它时不时就会给你制造点麻烦。DB2 提供了几个常用的运维工具：runstats、reorgchk、reorg 和 rebind。

本章的内容组织如下：

- 日常运维工具概述。
- Runstats 原理和用法。
- Reorg 原理和用法。
- Rebind 原理和用法。
- 计算数据库空间的方法。
- 计算表空间的方法。
- 计算表/索引空间的方法。

10.1 日常运维工具概述



对于 DBA 来说，为了确保数据库的性能，需要定期对数据库做维护。DB2 提供了几个重要的运维工具：runstats、reorgchk、reorg 和 rebind 等。Runstats 是 run statistics 的缩写，意思是收集统计信息，目的是为 DB2 优化器提供最佳路径选择；Reorg 是重组的意思，目的是减少表和索引在物理存储上的碎片，提高性能；Reorgchk 是重组前的检查；Rebind 是对一些包、存储过程或静态程序进行重新绑定。

图 10.1 描述了这几个工具的执行流程。首先通过 **Runstats** 收集表和索引的统计信息，然后执行 **Reorgchk** 检查是否有必要执行 **Reorg** 重组，如果有必要则执行，然后再次收集统计信息。最后，对于静态语句、存储过程等，执行 **Rebind** 绑定。

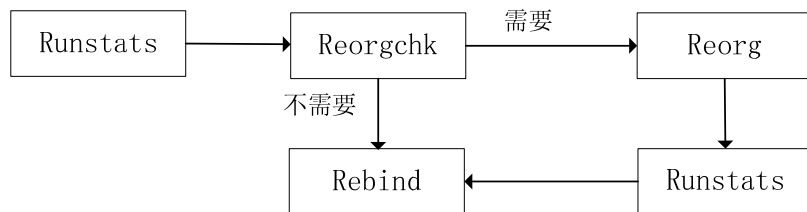


图 10.1 运维工具执行流程

10.2 Runstats

10.2.1 Runstats 原理

在系统运行一个查询的时候，优化器需要决定用某种方式来访问数据。

譬如说，如果 3 个表做关联，可以表 A 和表 B 关联，以后的结果集关联表 C；或者表 A 和表 C 关联，然后结果集关联表 B。不同的访问方式可以得到同样的结果集，但是其中所需要访问的数据量可能是截然不同的。

例如，如果表 A 有 10 条数据，表 B 有 10 000 条数据，表 C 有 100 条数据；而表 A 与表 B 的关联会返回 1000 条；表 A 与表 C 的关联返回 1 条；那么如果我们选择表 A 关联表 B，然后关联表 C，则会需要用表 A、表 B 关联的 10000 条结果关联表 C 的 100 条。但是如果我们首先执行表 A 与表 C 的关联，然后用一条结果关联表 B 的 10000 条数据，其总体性能想必会比前一种方式更有优势。

但是 DB2 基于什么原理来判断如何选择访问数据的顺序呢？答案就是统计信息。只有当 DB2 对表中的数据有一个大概的了解，才能知道每一步操作大约需要处理多少数据，返回多少行。当优化器了解了这些信息后，就会根据一系列运算，判定出各种访问途径所需要消耗的资源，然后从中选择一个消耗资源最少的方法。

DB2 收集统计信息的命令就是 **Runstats**。最普通的 **Runstats** 就是统计表和索引中有多少行数据，有多少不同的数值（比如 100 行的数据全部包含 1，那么 DB2 会知道表中所有的数值几乎都是相等的）。

有的时候用户会在查询中加入谓词判定，比如 **WHERE C1 = 10**，这样就需要一种方法，能够估算出有多少行的数据符合这个谓词。

一般来说, DB2 假设用户的数据均匀分布, 也就是说, 如果表中有 10 000 行, 其中包含 100 个不同的数值, 那么选择到符合一个特定的数值的几率为 1%, 也就是估计有大约 100 行符合 $C1=10$ 的条件。但是, 并不是所有的系统都有着均匀的数据分布, 可能在这 10 000 行数据中, 9000 行都包含 $C1=10$ 的数据, 那么我们需要另一种更精确的方法得到估算数值。这个方法就是数据分布。

Runstats 命令使用 DISTRIBUTION 参数收集数据分布。数据分布分为两种, 一种叫做频率采样 (Frequency), 一种叫做百分比采样 (Quantile)。

当收集数据分布时, 两种采样方式都会被收集。其中频率采样是收集表中拥有相同数值最多的几行, 比如 10 000 行数据中 9000 行为 10, 然后 500 行为 9, 然后 100 行为 8, 剩下的部分平均分布。如果我们指定 Frequency 为 3 的话, 那么系统就会记录下来有 9000 行 10, 500 行 9, 然后还有 100 行 8, 剩下的部分在估算时则假定平均分布。而百分比采样则是将整个 10 000 行数据分成相等大小的若干段, 然后记录每一段的段首与段尾的数值, 当需要查询一个数据段时 (比如 $C1 > 10 \text{ AND } C1 < 15$), 就可以根据每一个数据段的起始与终结的数值, 加上段落的大小, 估算出符合查询条件的记录数量。

理论上, 数据分布收集得越细致越好。但是过于细致的数据分布信息可能会导致 DB2 在优化 SQL 时需要处理更多的信息, 并占用更多的系统存储空间, 可能会导致性能的下降。因此, 一般情况下我们建议使用默认的数据分布采样设置, 也就是频率采样为 10, 百分比采样为 20。但是, 有些情况下, 则需要根据实际情况调整分布参数。。

10.2.2 Runstats 用法 ■ ■ ■

Runstats 语法如下:

```
>>-RUNSTATS--ON TABLE--object-name----->
>--+-----+----->
+-USE PROFILE-----+
+-UNSET PROFILE-----+
'| Statistics Options |-|
>--+-----+----->
'-UTIL_IMPACT_PRIORITY--+-+-----+-'
'-priority-'
```

Runstats 的语法比较复杂, 在实际应用中, 最常用的几种使用方法如下。

(1) 为表和索引收集统计信息, 包括数据分布, 代码如下:

```
RUNSTATS ON TABLE <表模式>.<表名> ON ALL COLUMNS WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

(2) 收集索引统计信息, 如果表上没有统计信息, 该选项会同时对表做统计。但该选项并不会收集数据分布信息, 代码如下:

```
RUNSTATS ON TABLE <表模式>.<表名> for indexes all
```

(3) 使用伯努利算法抽样统计。DB2 会扫描每一行数据，但只对一定比例的抽样数据进行统计。这种方法一般用于数据仓库中的大表，如果收集全表数据统计，将需要很长时间，并占用 CPU 资源，会应用性能造成影响。下例采用了伯努利 10% 抽样统计，代码如下：

```
RUNSTATS ON TABLE <表模式>.<表名> TABLESAMPLE bernoulli(10)
```

如何查看一个表是否收集了统计信息？一个比较有效的办法是查看 SYSCAT.TABLES 的 STATS_TIME 字段，如果该字段值为空，则表示没有收集过统计信息，否则会显示统计信息的时间：

```
db2 "select char(tabname,20) as tabname, stats_time from syscat.tables where STATS_TIME is NULL"
```

DB2 runstats 命令只能针对单表执行，而无法对整个数据库做运行时统计（虽然可以使用 reorgchk update statistics 对所有表收集统计信息，但 reorgchk 并不会收集分布统计）。可考虑将需要执行 runstats 的表写入一个脚本，以下是一个脚本范例：

```
#!/bin/ksh

if [ "$#" < 3 ] ; then
echo "USAGE: $0 DB_NAME DB_USER_NAME DB_PASSWORD"
exit
fi

DB=$1
DB_USER=$2
DB_PWD=$3

db2 connect to $DB user $DB_USER using $DB_PWD
db2 "select rtrim('RUNSTATS ON TABLE ' || rtrim(tabschema) || '.' || tabname || '
ON ALL COLUMNS WITH DISTRIBUTION ON ALL COLUMNS AND SAMPLED DETAILED INDEXES ALL ALLOW
WRITE ACCESS;') from syscat.tables where type = 'T'" > createRunstats.txt
grep RUNSTATS createRunstats.txt > runstats_detailed.sql
db2 -tvf runstats_detailed.sql
```

Runstats 是否会影响正常的数据增、删、改操作？runstats 是否会锁表？runstats 需要运行多长时间？对原系统有什么影响？这是实践中很多 DBA 经常问的问题。

Runstats 命令有 allow write access 和 allow read access 选项，allow write access 选项是默认行为，表示 runstats 表时，其他应用可以读取和修改该表数据；allow read access 选项则表示在 runstats 表时，其他应用只能读取该表数据，而无法修改。当指定 allow write access 选项时，DB2 会在 runstats 的表上加 IN 锁，而指定 allow read access 时，会在 runstats 表上加 S 锁。

当运行 Runstats 时，如果出现表和索引统计信息不一致，将会导致 Runstats 报警而影响优化器路径选择。出现该种情况时，需要同时收集表和索引统计信息。

```
db2inst1@dpf1:~> db2 delete from t1
db2inst1@dpf1:~> db2 runstats on table db2inst1.t1
SQL2314W Some statistics are in an inconsistent state. The newly collected
```

```
"TABLE" statistics are inconsistent with the existing "INDEX" statistics.
SQLSTATE=01650
```

```
db2inst1@dpf1:~> db2 runstats on table db2inst1.t1 and detailed indexes all
DB20000I The RUNSTATS command completed successfully.
```

当执行大数据量的统计信息收集时，可能出现“sql2310N 使用程序不能生成统计信息，返回错误“-930” ”错误，这时可考虑采用抽样统计。

Runstats 统计结果存在系统表中，如 SYSSTAT.TABLES 保存了表的统计信息，SYSSTAT.INDEXES 保存了索引统计信息，可以查看这些统计信息，但不建议手动更改。

在生成环境中，当遇到性能问题时，通常的做法是在测试机上搭建环境，模拟实际场景，但如果生产环境数据量太大、太敏感时，就无法创建相同的数据环境了，这样统计信息就不会一致，从而无法保证两边的执行计划是一致的。这时，可以采用的做法是将生产数据库的统计信息抽取出来，在测试库上进行更新，以此来“欺骗”优化器。

db2look 提供了 mimic 选项用于保存统计数据。

```
db2look -d sample -m > db2look_stat.out
```

以下是 Runstats 命令的最佳实践：

- 当表的数据量发生了很大变化，如通过 load 加载了大量数据，或 reorg 后，或新增了索引等，建议为相应对象收集 runstats 信息，为优化器提供最准确的依据。
- 为减小对应用的影响，尽可能地在空闲时执行 runstats。
- 当表很大，或运维窗口很小，或表数据频繁变动时，可考虑在某些关键字段上执行 runstats，而不是在所有字段。
- 当表很大、统计的时间很长时，可考虑采用抽样统计。
- 为提高可用性，推荐使用 allow write access 选项，但使用该选项时不能有太多的增删改操作，否则可能会造成数据和索引统计的不一致。
- 系统表也需要经常做 runstats。
- 为减少统计信息不一致的情况，考虑在表和索引上同时进行统计信息收集。
- 当执行完 runstats 后，要发出 commit 命令以释放锁，对于静态语句，还需要对 package 重新绑定，以便生成新的访问计划。

10.3 Reorg



10.3.1 为什么需要 Reorg



在 Windows 系统中，如果磁盘上的文件数据经常更改、迁移或删除，导致数据在磁盘上的

存储不连续，就会形成碎片，影响文件处理性能。因此，Windows 系统提供了磁盘碎片整理，目的是清除碎片，将不连续的数据变得连续，提高文件读/写速度。

在 DB2 中，如果经常对数据进行增、删、改操作，可能会造成表和索引数据的物理组织不连续，出现空页和溢出等情况。同磁盘碎片整理工具类似，DB2 提供了 Reorg 工具进行表和索引重组，使数据在物理存储上连续，提高页使用效率，减少 I/O 次数，提高查询性能。Reorg 是日常运维的重要工具，如图 10.2 所示。

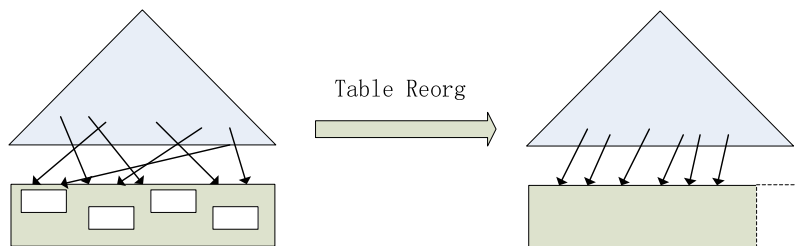


图 10.2 Reorg 原理

当出现以下问题时，Reorg 表可能会大大提高性能：

(1) 如果表中有许多行被删除，可能会导致某些数据页只包含一部分数据，甚至有些页变成空页，但空间并不会释放。通过 Reorg 后表占用的空间会大大降低。

(2) 当发生行溢出时。所谓行溢出 (overflow)，主要发生在表中包含 varchar 变长字段的情况，对变长字段值进行更新后，记录的长度比以前更长，使得原有的页空间不足以放下更新的数据，DB2 就会将这行数据存放到另外一页，而在原有位置通过指针指向新行的 RID，这样对该行的访问需要两次 I/O。如果行溢出的发生得很频繁，会导致大量不必要的 I/O 开销。这时可 Reorg 表数据，重新组织数据的存储顺序，减少不必要的 I/O。

(3) 按照某个索引重新组织表数据的物理顺序。前面我们讲过聚集索引 (cluster index) 的概念，就是表数据在物理上的存储顺序与索引的顺序相匹配，但随着数据的频繁操作，有些表数据的物理顺序可能无法匹配索引顺序，这时可通过 Reorg 操作重新按索引物理排序。

(4) 当启用了表数据压缩功能时，可通过 Reorg 建立字典表，并对表数据进行压缩。

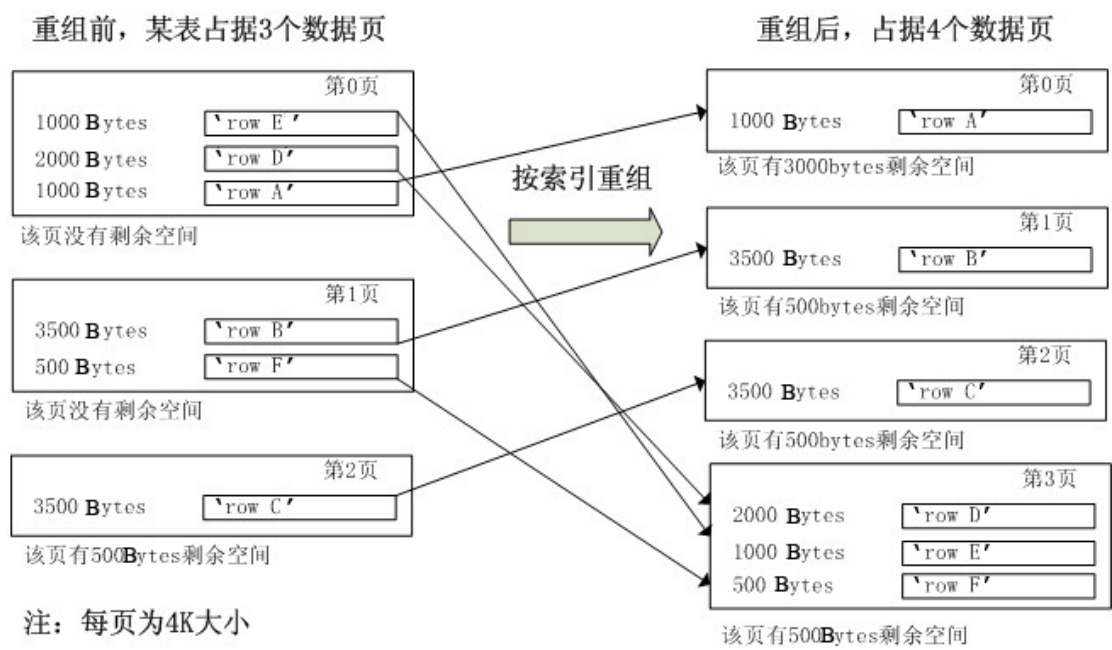
当出现以下问题时，Reorg 索引可能会大大提高性能：

(1) 当表数据删除后，可能导致很多索引页变为空页，可通过 Reorg 索引减小索引页空间。

(2) 减少索引的层次。索引是 B+树结构，包含树根、树枝和叶子节点，根据索引数据的大小，索引可能会包含几层，层数越低，索引查询需要的 I/O 就越少。通过 Reorg 可能降低索引的层次。

(3) 去除伪删除的行和页。当删除数据行后，索引中指向这些行的指针被标记为 pseudo deleted，而不是物理上删除。这时可通过 Reorg 删除这些指针，减少索引叶子节点的数量。

当然，Reorg 并不总是会降低数据页大小。以图 10.3 为例，某张表数据存在 3 个页面中，页大小为 4K，其中第 1 页存了 3 行数据，第 2 页存了 2 行数据，第 3 页存了 1 行数据。这时，按照索引对表进行重组，数据将按由小到大的顺序排列，row A 放在第 1 页，row B 的长度是 3500 字节，超出了 1 页的限制，只能放在第 2 页，以此类推，row C 放在第 3 页，其余数据放在第 4 页。最终，重组前是表数据是 3 页，重组后变为 4 页。



SCHEMA.NAME	CARD	OV	NP	FP	ACTBLK	TSIZE	F1	F2	F3

表: DB2ADMIN.ACT	18	0	1	1	-	648	0	- 100	---
表: DB2ADMIN.ADEFUSR	8	0	1	1	-	144	0	- 100	---
表: DB2ADMIN.CATALOG	-	-	-	-	-	-	-	- -	---
表: DB2ADMIN.CL_SCHED	5	0	1	1	-	145	0	- 100	---
表: DB2ADMIN.CUSTOMER	6	0	1	1	-	648	0	- 100	---
表: DB2ADMIN.DEPARTMENT	14	0	1	1	-	840	0	- 100	---
表: DB2ADMIN.EMPLOYEE	42	0	1	1	-	3696	0	- 100	---
索引统计信息:									
F4: CLUSTERRATIO 或正常化的 CLUSTERFACTOR > 80									
F5: 100 * (叶子页的已用空间/非空叶子页的可用空间) > MIN(50, (100 - PCTFREE))									
F6: (100 - PCTFREE) * (在一个较小层索引中的可用空间数量/所有键所需的空间数量) < 100									
F7: 100 * (伪删除的 RID 数/RID 总数) < 20									
F8: 100 * (伪空叶子页数/叶子页总数) < 20									
SCHEMA.NAME	INDCARD	LEAF	ELEAF	LVLS	NDEL	KEYS	LEAF_RECSize		

表: DB2ADMIN.ACT									
索引: DB2ADMIN.PK_ACT	18	1	0	1	0	18			2
2									
索引: DB2ADMIN.XACT2	18	1	0	1	0	18			8
8									
LEAF_PAGE_OVERHEAD	NLEAF_PAGE_OVERHEAD	PCT_PAGES_SAVED	F4	F5	F6	F7	F8	REORG	

1568	1568		0 100	-	-	0	0	-----	
1048	1048		0 100	-	-	0	0	-----	

那么如何判断某张表或索引是否需要重组呢？有两种方法，一种是通过 Reorgchk 工具，另外一种是通过 sysibmadm.snaptab 管理视图。

当表很多时，对 Reorgchk 结果的解析会比较麻烦，这时可考虑用 sysibmadm.snaptab 管理视图，如果发现 overflow_accesses 与 rows_read 比例高于 3%，则需要对表进行重组。注意：采用此方法需要将实例监控器开关打开（update dbm cfg using dft_mon_table on）：

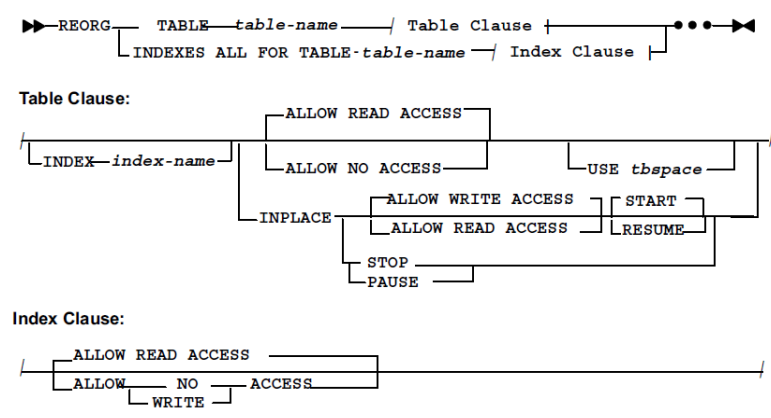
```
db2inst1@dpf1:~/sqlllib/db2dump> db2 "SELECT substr(TABNAME,1,18) as TABNAME,
ROWS_READ, OVERFLOW_ACCESSES from SYSIBMADM.SNAPTAB where (ROWS_READ > 999) and
((OVERFLOW_ACCESSES * 100) / (ROWS_READ + 1) > 3)"
```

TABNAME	ROWS_READ	OVERFLOW_ACCESSES
T1	10004	3864

对索引重组的检查仍然需要 Reorgchk。

10.3.2 Reorg 用法

Reorg 重组分为表重组和索引重组，表重组又支持两种方式：离线重组和在线重组。以下是 Reorg 命令语法，详细的解释说明请参看信息中心。



1. 离线 Reorg 表

离线表 Reorg 也叫 classic reorg, 支持 allow read access 和 allow no access 两个选项, allow read access 是默认的选项，表示重组时其他应用可以读取数据，allow no access 表示重组过程中不允许访问该表。

离线 Reorg 采用影子拷贝（shadow copy）方法，即创建一份原始数据的副本，在影子拷贝中进行数据 Reorg, Reorg 结束后替换原表数据，这样这样可以保持原表数据可读。默认情况下，数据拷贝会在原表空间进行，如果指定了 USE 选项，则会在 USE 指定的临时表空间进行。Reorg 过程中，会记录事务日志，当出现异常时可通过日志进行恢复。

离线 Reorg 最多可包含 4 个阶段：

- Scan-sort: 根据 reorg 指定的索引对表数据进行扫描、排序。
- Build: 根据第一阶段的结果进行表数据构建。
- Replace(copy): 用新数据替换原有数据。
- Index rebuild: 基于新数据，重建索引。

对于离线 Reorg，可以根据 index index-name 选项指定根据哪个索引进行表重组，DB2 会按该索引的顺序重新组织表数据的物理存储。如果没有指定索引，表数据重组时不关心顺序。如果该表定义了聚集索引（cluster index），即使 Reorg 没有指定索引，默认也会按聚集索引顺序重

组表。当离线表重组结束后，会重建表上的所有索引。

离线 Reorg 举例：

```
# 离线 Reorg, 默认是 Allow Read Access
db2inst1@dpf1:~ > db2 reorg table inst20.t1

db2inst1@dpf1:~ > db2 create temporary tablespace tempts1
db2inst1@dpf1:~ > db2 "create index i1 on inst20.t1(id) "
# 按索引重组表, 在临时表空间里进行重组
db2inst1@dpf1:~ > db2 reorg table inst20.t1 index i1 allow read access use tempts1
```

DB2 提供了多种方式检查 Reorg 是否完成：

- 通过 sysibmadm.snaptab 检查（要求打开 DFT_MON_TABLE 实例监控器开关）：

```
db2inst1@dpf1:~/tablepart> db2 "SELECT SUBSTR(TABNAME, 1, 15) AS TAB_NAME,
SUBSTR(TABSHEMA, 1, 15) AS TAB_SCHEMA, REORG_PHASE, SUBSTR(REORG_TYPE, 1, 20) AS
REORG_TYPE, REORG_STATUS, REORG_COMPLETION, DBPARTITIONNUM FROM
SYSIBMADM.SNAPTAB_REORG "
TAB_NAME TAB_SCHEMA REORG_PHASE REORG_TYPE REORG_STATUS REORG_COMPLETION
DBPARTITIONNUM
-----
HISTORYPART PARTTAB INDEX_RECREATE RECLAIM+OFFLINE+ALLO COMPLETED SUCCESS
0
```

- 通过 db2 get snapshot for tables on SAMPLE 进行 reorg 监控，如图 10.4 所示。下例中，根据监控结果，我们可以知道 Reorg 类型、启/停时间、当前状态及完成进度（Current Counter/ Max Counter 来预测完成的百分比）。

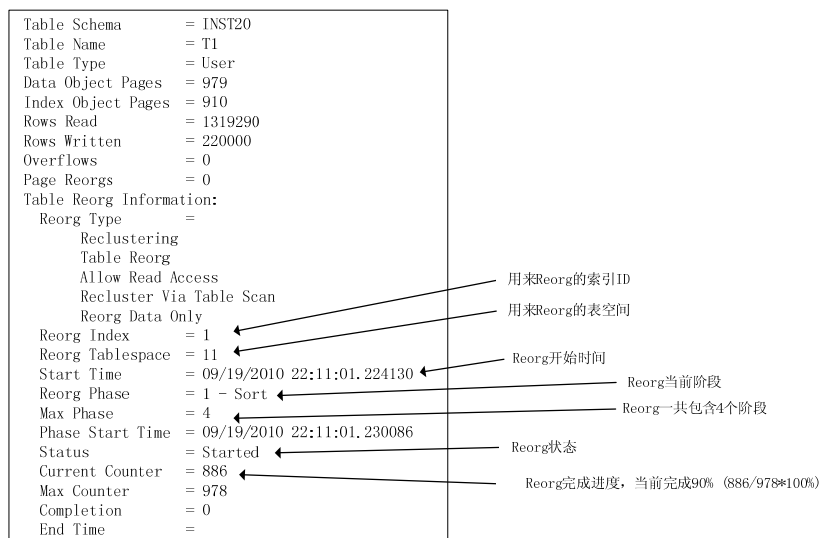


图 10.4 通过 snapshot 进行 reorg 监控

- 通过 db2pd reorg 选项可以获得当前正在执行的和近期完成的重组信息，代码如下：

```
inst20@db2server:~> db2pd -d sample -reorg

Database Partition 4294967295 -- Database SAMPLE -- Active -- Up 0 days 00:13:54

Table Reorg Information:
Address      TbspaceID TableID PartID MasterTbs MasterTab TableName      Type      IndexID
TempSpaceID
0xA7657BE0   3          10      n/a      n/a      n/a      T1        Offline   1         11

Table Reorg Stats:
Address      TableName      Start              End                PhaseStart
MaxPhase    Phase          CurCount    MaxCount    Status Completion
0xA7657BE0 T1              09/20/2010 00:00:57 09/20/2010 00:01:00 09/20/2010 00:00:58
4           IdxRecreat 0              0              Done      0
```

- 通过 list history reorg all for sample 获得表或索引重组信息，代码如图 10.5 所示。

Op	Obj	Timestamp+Sequence	Type	Dev	Earliest Log	Current Log	Backup ID
G	T	20100920000056	F		S0000018.LOG	S0000018.LOG	
Table: "INST20"."T1"							
Comment: REORG INDEX 1 USE 11							
Start Time: 20100920000056							
End Time: 20100920000100							
Status: A							
EID: 184							

图 10.5 通过 list history reorg 查看 Reorg 信息

2. 在线 Reorg 表

与离线 Reorg 相比，在线 Reorg 对资源的占用很少，对应用的影响也很小。在线 Reorg 也叫 inplace reorg，可以保证在重组过程中，其他应用对数据的不间断访问。在线表重组并不会创建数据副本，而是在原空间中进行，表数据的重组是分批次的，每批次只处理一部分数据，因此它的速度比离线 Reorg 要慢很多。

在线 Reorg 可随时启动和终止，为了保证可恢复性，在线 Reorg 会记录大量的日志，需要的日志空间依赖于要移动的行数、表上索引的个数和索引键大小，因此可能是表大小的几倍。

在线 Reorg 命令举例：

```
db2inst1@dpf1:~ > db2 reorg table db2inst1.employee inplace allow write access
DB20000I The REORG command completed successfully.
DB21024I This command is asynchronous and may not be effective immediately
```

在线 Reorg 是在后台异步执行的，因此即使我们看到命令成功返回，实际上仍然在后台执行。如果要重组的表很多，通常的做法是写成脚本，需要注意的是，在线重组是异步过程，这可能会造成脚本里的多个 reorg 命令同时执行，导致 I/O 和 CPU 资源占用很多，并且可能会消耗所有的活动日志，影响应用系统正常运行。

为解决该问题，在编写脚本时采用以下方法控制多个表的在线 Reorg 执行顺序：每个在线 Reorg 执行时，都有一个对应的 db2reorg 应用程序，通过 `db2 list applications show detail | grep -i db2reorg` 判断，如果有 Reorg 正在执行，则等待该 Reorg 执行完毕，否则执行脚本里的下一个表重组。

```
inst20@dpf1:~> db2 list applications show detail
CONNECT Auth Id      Application Name      Appl.      Application Id Handle
-----
INST20              db2reorg             296        *LOCAL.DB2.101013101156

Seq#  Number of  Coordinating DB  Coordinator  Status          Status Change Time
DB Name DB Path
      Agents    partition number pid/thread
-----
00002 1           0                28083        Connect Completed Not Collected
TESTDB /database
```

除此之外，还可以通过其他方法监控在线表重组，参考前面的离线表重组。

3. Reorg 索引

当离线表重组结束后，会重建表上的所有索引。在线表重组仅仅维护索引，而不会重建索引(聚集索引除外)，当在线重组结束后，如果需要进行索引单独重组，可通过 `reorg indexes all for <tablename>`。

```
db2inst1@dpf1:~/sqlllib/db2dump> db2 reorg indexes all for table t1
DB20000I The REORG command completed successfully.
```

对于在线索引重组的监控，可以通过 `list history reorg all` 或查看 `db2diag.log` 文件。

```
inst20@db2server:~> db2 list history reorg all for sample

Op Obj Timestamp+Sequence Type Dev Earliest Log Current Log Backup ID
---
G I 20110521013633      N      S0000019.LOG S0000019.LOG
-----
Table: "INST20"."T1"

-----
Comment: REORG INDEXES
Start Time: 20100921013633
End Time: 20100921013636
Status: A
```

EID: 203

10.3.3 Reorg 最佳实践 ■ ■ ■

对于 DBA 来说，在 reorg 时有几个问题需要注意：

1. 离线重组还是在线重组

表 10.1 汇总了离线重组和在线重组的差别，每种方法的优缺点可以作为 DBA 运维选择的重要依据。一般来说，如果应用有一定的运维时间窗口，建议选择离线 Reorg 加快执行速度；对于要求 7*24 小时持续运行、具有很小运维窗口的应用，建议在业务相对空闲的时候选择在线 Reorg，虽然速度稍慢，但可以减小对应用的影响。在线重组在设计时考虑的更多的是如何减小对应用的影响，而不是速度。

表 10.1

Reorg 类型	离 线 重 组	在 线 重 组
支持的 访问模式	Allow No Access Allow Read Access（默认）	Allow Read Access Allow Write Access（默认）
优点	提供最快的表重组 原始数据在最终替换前是只读的 一旦表重组结束会进行索引重建	执行重组时允许应用访问 可以随时终止和恢复 异步运行 因为增量处理，所以占用的空间小
缺点	占用数据空间大，大概是原表的 2 倍 访问受限，只读 如果重组过程中失败，则需要重新执行	要求更多日志空间 比离线 reorg 速度慢很多，可能慢 10~20 倍 不能重组大对象 维护索引，但不是重建索引，可能需要随后对索引重组
选择建议	如果有运维时间窗口，并且要求 Reorg 执行速度，则选择离线 Reorg	一般来说，对于 7×24 小时持续运行并具有很小运维窗口的应用，建议选用在线重组，确保高可用，但建议在交易较小的时候使用

2. 如果在重组的过程中出现意外（比如断电），对正在重组的表或索引有何影响？

依赖于重组的类型和进度，DB2 内部有不同的处理机制。对于离线重组，如果断电时 Reorg 正在进行 scan 或 build，当崩溃恢复时该表将恢复到初始状态；如果 Reorg 正在进行 replace(copy)，崩溃恢复将重新执行该阶段；如果 Reorg 正在进行 index create，崩溃恢复阶段将把索引失效，并延迟索引的重建。

对于在线重组，如果断电时 **Reorg** 正在进行，那么正在执行 **Reorg** 的那个事务将回滚（因为在线 **Reorg** 采用增量的方式进行数据处理，已完成的迁移并不会受到影响）。但 **reorg** 的状态是终止（**paused**），可以重新恢复。

对于在线索引重组，断电后，崩溃恢复将回滚索引创建，并延迟索引重建。

以下是关于 **Reorg** 的几点最佳实践：

- 如果表上有大量的增、删、改操作，产生碎片的几率会很大，建议重组。
- 当 **Reorgchk** 命令提示需要重组时，建议重组。
- 创建表压缩时，通过 **Reorg** 建立压缩字典表，并进行表压缩。
- 对于离线重组，需要确保有足够的表空间存取影子拷贝，否则重组会失败。
- 对于在线重组，由于每次数据迁移都需要记录日志，因此必须确保有足够的日志空间。同时，需要确保 **util_heap_sz** 数据库参数的值足够大。
- 在同一时刻，同一张表上只能有一个重组。但只要资源允许，可以同时执行几个表重组。
- 对于普通表来说，当执行在线索引重组时，必须为一张表的所有索引执行，而无法选择为某个索引执行。
- 对系统表也要经常做 **Reorg**。
- 重组最好不要和其他工具一起执行，因为可能会产生锁竞争，导致回滚。

10.4 Rebind

■ -----

讲到 **Rebind**，则不能不提 **bind**。开发过 **SQLC** 的工程师对 **bind** 应该不会陌生，包含在 **C** 程序中的 **DB2 SQL** 语句经过预编译后（**prep**），会绑定（**bind**）到 **DB2** 的 **package** 中，**package** 里包含了每条 **SQL** 语句的访问计划。以后执行这个 **C** 程序时，就会按照保存在 **package** 的访问计划执行 **SQL** 语句。

那么 **Rebind** 有什么用呢？假设上述 **C** 程序中的 **SQL** 语句所在的表上新增了索引，如果不做 **rebind**，那么 **DB2** 仍然会使用 **package** 里的已有的访问计划，而不会使用新增索引。**Rebind** 工具会根据当前的统计信息为 **package** 里的 **SQL** 语句重新生成新的访问计划，对性能可能有比较好的提升。

Rebind 一般用于嵌入式 **SQL**，如嵌入 **SQL C**、嵌入 **SQL Java**、嵌入 **SQL Cobol** 等，这些嵌入式语言的 **SQL** 如果是静态语句时，当表的统计信息发生重大变化，或新增索引等可能造成执行计划发生变化时，建议用 **Rebind** 命令重新绑定。对于存储过程，本质上也是存储到 **package** 中，**Rebind** 同样适合。

使用 **Rebind** 命令时，需要提供 **package** 名。对于 **SQC**（嵌入 **C**）等程序，在 **bind** 的时候会创建一个 **package**。对于存储过程，在 **DB2** 内部，每个存储过程对应一个 **package**。可以通过 **db2**

list packages for all (或 schema xx) 列出相应的 package 名。

例子如下:

```
inst20@db2server:~> db2 list packages for schema inst20
```

Package	Schema	Version	Bound by	Total sections	Valid	Isolation Format	level
P2012233	INST20		INST20	1 Y	0	CS	U
P2432259	INST20		INST20	10 Y	0	CS	U
P6045027	INST20		INST20	1 Y	0	CS	U

```

3 record(s) selected.

inst20@db2server:~> db2 rebind package P6045027
DB20000I The REBIND PACKAGE command completed successfully

```

Rebind 命令只能针对每个 package, 如果需要对所有 package 重新绑定, 可以考虑用 db2rbind 命令。

```
inst20@db2server:~> db2rbind sample -l db2rbind.log all
```

```
Rebind done successfully for database 'SAMPLE'.
```

动态 SQL 是在执行时才编译, 并存储到 package cache 中。如果更新了统计信息, 可以通过 flush package cache dynamic 更新 package cache。

10.5 获取数据库占用空间的大小

从 v9 开始, DB2 提供了 SYSSPROC.GET_DBSIZE_INFO 存储过程来计算当前数据库大小和最大容量大小:

```
db2 call GET_DBSIZE_INFO(?,?,?,<refresh-window>)
```

其中, 前 3 个参数为输出参数, 第 4 个参数 refresh-window 为输入参数, 表示在该时间后进行数据库大小和容量大小的刷新, 单位为分钟, 默认值为 30 分钟, 如果为 0, 则会马上进行刷新。如果需要统计每天数据大小的增长情况, 可考虑将此更新窗口设为 24 小时, 即 24×60=1440 分钟。例如:

```
G:\Documents and Settings\db2admin>db2 "call GET_DBSIZE_INFO(?,?,?,0) "
```

输出参数的值

```

参数名: SNAPSHOTTIMESTAMP
参数值: 2010-08-14-08.31.27.731000

参数名: DATABASESIZE
参数值: 180142080

参数名: DATABASECAPACITY
参数值: 39617922048 （操作系统剩余空间很大）

返回状态 = 0

```

其中: $180142080/1024=175920K$

返回值如下。

- **SNAPSHOTTIMESTAMP**: 表示执行快照的时间戳。
- **DATABASESIZE**: 返回数据库的大小（单位是 bytes），大小的计算如下: $dbsize = \sum (used_pages * page_size)$ 每个表空间（SMS 和 DMS）的已使用页数乘以页大小。
- **DATABASECAPACITY**: 返回数据库的容量大小（单位是 Bytes），在分区数据库上这个值不可用。计算如下: $dbcapacity = \sum (DMS\ usable_pages * page\ size) + \sum (SMS\ container\ size + file\ system\ free\ size\ per\ container)$ 。如果多个 SMS 表空间容器在同一个文件系统上定义，那么文件系统剩余空间只计算一次。

注意:

- 如果存在多个 SMS 表空间的情况，db 容量的计算结果不一定准确。
- 在多分区环境下，此存储过程只能看 dbsize 大小，无法看 db 容量。

10.6 获取某个表空间占用空间大小



DB2 表空间根据管理方式分为 SMS、DMS 和自动存储表空间。

对于 DMS 表空间，如果用户想要知道 DMS 容器所占用的磁盘空间，则可以简单地检查容器文件的大小。不过如果用户想要知道容器中所包含的数据占用了多少数据页，就可以使用 `db2 list tablespaces show detail` 命令，其中的 `Used pages` 是使用的数据页数，而 `Free pages` 则是空闲数据页数量：

```

Tablespace ID      = 3
Name               = TS1
Type               = Database managed space
Contents           = All permanent data. Large table space.
State              = 0x0000
  Detailed explanation:
    Normal
Total pages        = 45056
Useable pages      = 44928

```

```

Used pages                = 320
Free pages                = 44608
High water mark (pages)  = 320
Page size (bytes)        = 8192
Extent size (pages)      = 32
Prefetch size (pages)    = 896
Number of containers      = 4
Minimum recovery time     = 2011-03-10-12.14.35.000000

```

在 9.1 版本中，可通过 SYSIBMADM.TBSP_UTILIZATION 查看表空间使用大小：

```

db2inst1@dpf1:~> db2 -tvf aa.xx
select substr(tbsp_name,1,30) as TABLESPACE_NAME, substr(TBSP_TYPE,1,10) as
TBSP_TYPE,
          substr(tbsp_content_type,1,10) as
TABLESPACE_TYPE,sum(tbsp_total_size_kb)/1024 as
TOTAL_MB,sum(tbsp_used_size_kb)/1024 as USED_MB, sum(tbsp_free_size_kb)/1024 as
FREE_MB, tbsp_page_size AS PAGE_SIZE from SYSIBMADM.TBSP_UTILIZATION where
TBSP_TYPE!='SMS' group by tbsp_name,tbsp_type, tbsp_content_type,tbsp_page_size
order by 1

```

TABLESPACE_NAME	TBSP_TYPE	TABLESPACE_TYPE	TOTAL_MB	USED_MB	FREE_MB	PAGE_SIZE
-----	-----	-----	-----	-----	-----	-----
----- IBMDB2SAMPLEREL	DMS	LARGE	64	48	15	8192
IBMDB2SAMPLEXML	DMS	LARGE	32	9	22	8192
SYSCATSPACE	DMS	ANY	96	93	2	8192
SYSTOOLSPACE	DMS	LARGE	32	0	1	8192
USERSPACE1	DMS	LARGE	32	20	11	8192
5 record(s) selected.						

对于 SMS 表空间，无法通过 list tablespace 或管理视图查看使用大小，由于每个对象都占用一个文件，可考虑计算 SMS 表空间容器中文件的总大小，只需确保表空间容器所属的文件系统有足够空间即可。

对于自动存储管理的表空间，只要自动存储路径有足够的空间，当表空间满了以后，DB2 就会自动扩展。

10.7 获取某个表/索引占用空间的大小



计算某个表占用空间有 db2pd -tcstats、Admin_get_tab_info 表函数和 SYSIBMADM.ADMINTABINFO 系统管理视图 3 种方法。

(1) db2pd 的 tcstats 可以查看表的 TCB 信息，其中 DataSize 字段用来表示表的页数，乘以页大小即为表的大小。使用该方法时，只有该表被访问过才会显示出来：

```

db2inst1@dpf1:~> db2pd -d sample -tcstats

```

```
Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 14:55:15

TCB Table Information:
Address    TbspaceID TableID PartID MasterTbs MasterTab TableName      SchemaNm
ObjClass DataSize  LfSize  LobSize  XMLSize
0xA6663A44 2        6      n/a     2        6        EMPLOYEE      DB2INST1 Perm
1          0        0        0
```

(2) Reorgchk 结果中, NPAGES 代表页数, 乘以页大小就是表的大小, 但 Reorgchk 需要执行 runstats, 对于大表来说, 需要的时间较长:

```
db2inst1@dpf1:~> db2 reorgchk update statistics on table db2inst1.employee

Doing RUNSTATS...

Table statistics:

F1: 100 * OVERFLOW / CARD < 5
F2: 100 * (Effective Space Utilization of Data Pages) > 70
F3: 100 * (Required Pages / Total Pages) > 80

SCHEMA.NAME          CARD    OV    NP    FP ACTBLK    TSIZE  F1  F2  F3 REORG
-----
Table: DB2INST1.EMPLOYEE      42     0     1     1     -    3696  0  - 100 ---
-----
```

(3) SYSIBMADM.ADMINTABINFO 管理视图, DB2 9 版本引入, 可获得表的大小和状态信息, 以 KB 为计量单位:

```
db2 => describe table SYSIBMADM.ADMINTABINFO

Column          Type    Type
name            schema  name          Length  Scale Nulls
-----
TABSCHEMA       SYSIBM  VARCHAR       128     0 Yes
TABNAME         SYSIBM  VARCHAR       128     0 Yes
TABTYPE         SYSIBM  CHARACTER      1     0 Yes
DBPARTITIONNUM  SYSIBM  SMALLINT       2     0 Yes
DATA_PARTITION_ID SYSIBM  INTEGER        4     0 Yes
AVAILABLE       SYSIBM  CHARACTER      1     0 Yes
DATA_OBJECT_L_SIZE SYSIBM  BIGINT         8     0 Yes
DATA_OBJECT_P_SIZE SYSIBM  BIGINT         8     0 Yes
INDEX_OBJECT_L_SIZE SYSIBM  BIGINT         8     0 Yes
INDEX_OBJECT_P_SIZE SYSIBM  BIGINT         8     0 Yes
LONG_OBJECT_L_SIZE SYSIBM  BIGINT         8     0 Yes
LONG_OBJECT_P_SIZE SYSIBM  BIGINT         8     0 Yes
LOB_OBJECT_L_SIZE SYSIBM  BIGINT         8     0 Yes
LOB_OBJECT_P_SIZE SYSIBM  BIGINT         8     0 Yes
```

XML_OBJECT_L_SIZE	SYSIBM	BIGINT	8	0	Yes
XML_OBJECT_P_SIZE	SYSIBM	BIGINT	8	0	Yes
INDEX_TYPE	SYSIBM	SMALLINT	2	0	Yes
REORG_PENDING	SYSIBM	CHARACTER	1	0	Yes
INPLACE_REORG_STATUS	SYSIBM	VARCHAR	10	0	Yes
LOAD_STATUS	SYSIBM	VARCHAR	12	0	Yes
READ_ACCESS_ONLY	SYSIBM	CHARACTER	1	0	Yes
NO_LOAD_RESTART	SYSIBM	CHARACTER	1	0	Yes
NUM_REORG_REC_ALTERS	SYSIBM	SMALLINT	2	0	Yes
INDEXES_REQUIRE_REBUILD	SYSIBM	CHARACTER	1	0	Yes
LARGE_RIDS	SYSIBM	CHARACTER	1	0	Yes
LARGE_SLOTS	SYSIBM	CHARACTER	1	0	Yes
DICTIONARY_SIZE	SYSIBM	BIGINT	8	0	Yes

27 record(s) selected.

以下用来统计 EMPLOYEE 表所占的物理空间的大小，包括数据、索引、大对象和 XML 空间大小。

```
db2inst1@dpf1:~> db2 "SELECT (data_object_p_size + index_object_p_size +
long_object_p_size +
lob_object_p_size + xml_object_p_size) as total_p_size
FROM sysibmadm.admintabinfo where tabname='EMPLOYEE' "
```

```
TOTAL_P_SIZE
-----
1024
1 record(s) selected.
```

SYSIBMADM.ADMINTABINFO 有几个字段值得注意：DATA_OBJECT_L_SIZE 和 DATA_OBJECT_P_SIZE。其中 DATA_OBJECT_L_SIZE 代表表的逻辑大小（KB），DATA_OBJECT_P_SIZE 为表的物理大小（KB）。逻辑大小和物理大小有什么差别呢？想象一下，我们往某张表中插入很多数据，然后删除一些数据，但表占据的空间并不会释放，当新的数据插入时，仍然会使用这些空间。实际占用的空间叫做逻辑空间，分配过的空间叫物理空间，逻辑空间可能会小于物理空间，这两者的差异可以通过 reorg 来消除。

除此以外，SYSIBMADM.ADMINTABINFO 视图提供了其他几个有用的字段，如 REORG_PENDING、INPLACE_REORG_STATUS、LOAD_STATUS 等。详细解释，请参看信息中心。

(4)Admin_get_tab_info 表函数返回结果与 SYSIBMADM.ADMINTABINFO 管理视图类似。

10.8 小结

一般来说，DBA 的日常运维包括定期收集统计信息，整理表和表空间，监测磁盘使用情况，当然还有定期的备份与检查备份介质等。对于一个较为成熟的系统，大部分这些运维操作早已

经存在规范化的脚本，每天或者每周定期运行。但是对于一个新建立的系统，作为 DBA 就要建立这一套规范化的运维体系。

10.9 判断题



(1) RUNSTATS 用来收集表与索引的统计信息。

T: 正确

F: 错误

(2) REORGCHK 工具可以用来检查一个表和索引是否需要重组。

T: 正确

F: 错误

(3) SYSPROC.GET_DBSIZE_INFO 返回的信息在任何情况下都是精确的。

T: 正确

F: 错误

(4) 用户只能通过 LIST TABLESPACES SHOW DETAIL 命令得到表空间的占用率。

T: 正确

F: 错误

(5) REORGCHK 结果中的 NP 与 db2pd -tcostat 中的结果应当吻合。

T: 正确

F: 错误



锁和并发

现在的 IT 系统对多用户并发性能提出了很高的要求，那么如何保证多用户并发情况下数据的一致性呢？答案就是事务、日志、锁和隔离级别。相对于其他数据库而言，在 DB2 中遇到锁的问题比较多，特别是习惯了其他数据库的工程师，刚转到 DB2 时会很不适应，比如经常会遇到死锁、锁等、超时、锁升级等令人困惑的问题，加上 DB2 自身提供的工具相对有限，使锁问题的诊断、分析和调优变得步履维艰。不过在最近几个版本中，IBM 顺应市场需求，在 DB2 锁特性和工具方面做了持续改进，相信以后的监控和诊断会方便很多。

本章将针对不同的 DB2 版本，结合一些实际案例，分享各种锁问题的诊断和优化技巧。本章内容安排如下：

- 锁概述。
- 锁的模式、锁兼容性。
- 锁的各种问题。
- 锁的各种现象监控。
- 锁和并发调优。
- 9.7 版本提供的 Currently Committed。

11.1 锁和隔离级别概述



锁是什么？为什么需要锁？在回答这个问题之前，我们首先回顾一下之前介绍过的事务概念，事务是一个完整的单元，包含一条或多条语句，这些语句要么都成功，要么都失败，具有

ACID（原子性、一致性、隔离性和持久性）特性。

假设您是一个数据库系统的设计师，为了保证多个事务并发执行，您将会用什么机制来解决下面几个场景出现的问题呢？

（1）A 事务更改了某张表的一条记录，B 事务去读这条未提交的记录，随后 A 事务被回滚了，那么 B 读到的值就是错的。

（2）A 事务更改了某张表的一条记录，B 事务也尝试更改这条记录，A 事务再去读时就会发现之前自己更改的数据被别人更改了。

（3）A 事务根据某个过滤条件查询某张表，返回一个结果集。这时 B 事务更改了这个结果集的某些行，A 事务再去读时会发现满足条件的结果集变小了。

（4）A 事务根据某个过滤条件查询某张表，返回一个结果集。这时 B 事务更改了这个结果集之外的某些行，而更改的这些行正好满足 A 的查询条件，这时 A 再去查询时会发现结果集变大了。

从这几个问题中我们明白，DB2 需要一种机制来解决多个事务并发处理时数据的一致性和完整性。这种机制就是锁，锁是一种资源，这个资源是和事务关联在一起的，当某个事务获取了锁，在提交或回滚之前，就一直持有该锁，DB2 管理器会根据这个锁的类型来决定其他事务是否可以获取该锁。当该事务完成后，就会释放锁资源，其他事务就可以使用这个资源。

锁主要分两类：读锁和写锁，读锁也叫共享锁（S），写锁也叫排他锁（X）。共享锁的意思就是，不同的任务间都可以访问数据，因此我们将这种锁叫做共享的；而排他的意思就是排除他人，只让这个请求锁单独处理数据。

如果一条记录已经加了共享锁，那么申请排它锁的应用就要等待这个锁释放。同理，如果一条记录上已经加了排他锁，那么其他所有新进的排他锁与共享锁都要等待当前的排他锁释放，才能访问该条记录。

对于第 1 个场景，当 A 事务更改这张表的记录时，DB2 会在对应行上加排他锁，当 B 事务读取该行时，要申请读锁，在默认的隔离级别下，读锁和写锁是不兼容的（我们将在 11.2 章介绍锁的兼容性），因此就不会出现 B 读到未提交数据的问题。

对于第 2 个场景，A 事务更改这张表的记录，DB2 会在对应行上加排他锁，当 B 事务去更新时，也要申请排他锁，而排他锁之间是不兼容的，因此不会出现 B 事务更改记录的情况。

对于第 3 个和第 4 个场景，DB2 会有以下几种选择：

- 将满足 A 事务查询条件的某一行记录加读锁，那么 B 事务就不能更改这行，但可能还会更改其他行。
- 将满足 A 事务查询条件的所有行记录加读锁，那么 B 事务就不能更改这些加锁的行，但仍然可以更改其他没有满足条件的行，A 事务再去查询时可能会读取更多满

足条件的行（B 事务更改后，满足了 A 的查询条件）。

- 将 A 事务查询过程中读到的所有行加读锁，这样 B 事务就无法更改了，A 事务再去查询时返回的结果与第一次相同。

那么对于这几种选择，DB2 会选择哪种呢？这是由隔离级别来决定的。隔离级别（Isolation Level）决定了锁的粒度，级别越低，锁的行数就少。DB 的隔离级别分以下 4 种。

- UR（Uncommitted Read）：未提交读隔离级别，即读的时候不加锁，可以读到未提交的数据。
- CS（Cursor Stability）：游标稳定性隔离级别，即读到哪一行就在这一行加 S 锁，读完就释放，类似游标一样。CS 是默认的隔离级别。
- RS（Read Stability）：读稳定性隔离级别，即把查询的结果集都加 S 锁。
- RR（Repeatable Stability）：可重复读隔离级别，即把读过的行都加 S 锁。

由此可知，隔离级别只适用于读，在查询的时候用来通知 DB2 管理器，决定不加锁、加一行锁、在结果集上加锁还是把读过的行都加锁。对于增、删、改操作，不管采用什么隔离级别，都要加写锁。

另外，我们可以看到，隔离级别越高，锁的粒度就越大，并发性就越低，因此，在目前的大多数系统中，大家都选择默认的隔离级别，即 CS。隔离级别的设置方法也比较简单，可以在程序中，也可以在语句级，还可以在事务级。

小结一下：锁的模式主要分为读锁和写锁两类，读锁的粒度和范围是由隔离级别决定的，写锁不受隔离级别影响

提示：隔离级别是 SQL ANSI-99 国际标准，但不同的数据库厂商有不同的实现。隔离级别只适于读，控制读锁的范围和粒度。

11.2 锁的模式和兼容性



锁都有一些基本属性：锁的模式、锁的对象、锁的持久性等。

锁的对象就是锁在什么目标上，DB2 支持的目标实例、数据库、表空间、表和行，行锁是 DB2 默认的锁对象，我们平时工作遇见最多的是行锁和表锁，这也是本书主要介绍的。

锁的模式定义了锁的拥有者和其他用户所允许的访问类型。锁的持久性就是定义锁什么时候释放，一般是事务提交就会释放。

本节我们重点讲述表锁和行锁的模式，以及表锁模式和行锁模式的兼容性。本节的内容比较枯燥，也难以理解，但却非常关键，如果不能深入理解和掌握，则无法对锁问题进行准确的诊断和优化。

11.2.1 表锁模式 ■ ■ ■

表锁模式主要分两类：强类型表锁和弱类型表锁，强类型表锁适用于表里的所有行，主要包括 S、U、X 和 Z 锁四种；弱类型表锁又叫意向锁（intent），它的目的是配合行锁，在获得行锁之前必须先获得表锁，主要包括：IN、IS、IX 和 SIX。

默认情况下，DB2 不会实施强类型表锁，只有通过 lock table 锁表或发生锁升级的时候才会在表上加强类型锁模式。

表 11.1 是 DB2 支持的表锁模式及其说明，我们将通过几个例子为大家模拟强类型锁，关于弱类型表锁，将在行锁模式一节中演示。

表 11.1

表锁模式（mode）	解 释 说 明
弱类型锁	
IN 意向无（Intent None）	锁的拥有者可以读表中的任何数据，包括未提交的数据，但不能更改任何一行；其他并发应用可以读，也可以写表中任意行
IS 意向共享（Intent Share）	锁的拥有者可以读表中的任何数据，但不能更改 其他并发应用可以读，也可以写表中任意行 当要读表中的行时，需要先获取表上的 IS 锁
IX 意向排它（Intent eXclusive）	锁的拥有者和其他并发应用都可以读和写数据。一般当要更改表中的行（如 insert、update、delete）时，需要先获取表上的 IX 锁
SIX Share with Intent eXclusive	SIX 锁是比较特殊的锁，是 S+IX 或 IX+S 形成的 锁的拥有者可以读取表中所有的行，在获得 X 锁的行上可以修改
强类型锁	
S 共享锁（Share）	锁的拥有者和其他并发应用都可以读表中的任何数据，但不能更改
U 更新锁（Update）	一种处于 S 锁和 X 锁中间状态的锁。主要是为了避免两个拥有 S 锁的应用同时申请 X 锁时发生死锁
X 排他锁（eXclusive）	锁的拥有者可以读，也可以更改表中的任何数据 其他并发应用不能读也不能写表中的任何数据（UR 隔离级别的应用可以读未提交的数据）
Z 超级排他锁 （Super eXclusive）	当 create、alter、drop 表或 create、drop 索引对象时，需要 Z 锁

例子 1: X 锁模拟

打开两个 DB2 命令行窗口，模拟两个事务。第一个窗口执行锁表操作，+c 表示取消自动提交，默认是 auto-commit。lock table in exclusive mode 表示以排他的方式锁表，这样其他应用就不能访问该表的数据了。如果希望以共享的模式锁表，就用 lock table in share mode。通过 lock table 方式锁表，一般用于运维任务，排除别人干扰。

```
db2inst1@dpf1:~> db2 +c "lock table t1 in exclusive mode"
DB20000I The SQL command completed successfully.
```

第二个窗口观察锁情况。db2pd 锁监控的好工具，因为不需要锁定引擎资源，对系统的开销小，而且格式看起来比较直观。通过 db2pd 的输出结果可知，当前有一个 X 锁，锁的类型是表锁，Sts 字段是 G (grant, 授予的意思)。还有 P 锁和 CatalogCache 锁，是内部锁。

```
db2inst1@dpf1:~> db2pd -d sample -locks

Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:00:38

Locks:
Address      TranHdl      Lockname                                     Type          Mode Sts Owner      Dur
HoldCount Att      ReleaseFlg rrIID
0x9A50A580 2          41414141415A425A7F4760B841 Internal P ..S G    2          1 0
0x00000000 0x40000000 0
0x9A50A980 2          00000500091C000000A7979C43 CatCache      ..S G    2          1 0
0x00000000 0x40000000 0
0x9A50A680 2          03000500000000000000000000054 Table         ..X G    2          255 0
0x00000200 0x40000000 0
```

当在第一个窗口中将事务提交后，锁资源释放，再观察第二个窗口，没有发现锁。

例子 2: 表的超级排他锁 (Z 锁) 模拟

当创建/删除/更改表或索引对象时，DB2 会在相应的表上加 Z 锁，即超级排它锁。同样开两个命令行窗口，模拟两个事务。

窗口 1: 删除一张表。

```
db2inst1@dpf1:~> db2 +c drop table t1
DB20000I The SQL command completed successfully.
```

窗口 2: 观察锁的情况，通过 db2pd 结果会看到一个 Z 锁，除了 Z 锁，为什么还有很多其他类型的锁呢？这是因为 drop 是 DDL 操作，DDL 定义会存到在系统表 (catalog table) 中，因此会在系统表上加表锁和行锁。

```
db2inst1@dpf1:~> db2pd -d sample -locks

Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:24:53

Locks:
```

Address	TranHdl	Lockname	Type	Mode	Sts	Owner	Dur
HoldCount	Att	ReleaseFlg rrIID					
0x9A511D00	2	00000C001800060000000000052	Row	..S	G	2	1 0
0x00000010	0x40000000	1					
0x9A50FE00	2	00001E004C00FE0000000000052	Row	..X	G	2	1 0
0x00000020	0x40000000	0					
..... (省略 N 行)							
0x9A512180	2	030000000000000000000000004F	ObjTab	.IN	G	2	255 0
0x00000100	0x40000000	0					
0x9A50D180	2	0100000000000000000000A7979C43	CatCache	..X	G	2	255 0
0x00000000	0x40000000	0					
0x9A511F00	2	0300050000000000000000000054	Table	..Z	G	2	255 0
0x00002000	0x40000000	0					
0x9A50CF80	2	00000A0000000000000000000054	Table	.IX	G	2	1 0
0x00002000	0x40000000	0					
..... (省略 N 行)							
0x9A511C00	2	00000C0000000000000000000054	Table	.IS	G	2	1 0
0x00002010	0x40000000	0					

例子 3: U 锁的模拟

有朋友经常问 U 锁是什么锁, 其实 U 锁是一种中间状态的锁, 一般用于 select ... for update, 目的是为了保持和 S 锁的兼容, 但两个 U 锁是不兼容的。同样使用两个窗口模拟:

第一个窗口中, 执行 select .. for update with rr, with rr 是在语句级别实现 rr 隔离级别:

```
db2inst1@dpf1:~> db2 +c "select * from t1 for update with rr "
```

```
COL1          COL2
-----
ddd           bbb
bbbb          ccc
2 record(s) selected.
```

第二个窗口观察锁情况, 可以看到 U 锁:

```
db2inst1@dpf1:~> db2pd -d sample -locks
```

```
Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 10:47:16
```

Locks:

Address	TranHdl	Lockname	Type	Mode	Sts	Owner	Dur
HoldCount	Att	ReleaseFlg rrIID					
0x9A50AB00	9	41414141415A425A7F4760B841	Internal P	..S	G	9	1 0
0x00000000	0x40000000	0					
0x9A517780	2	41414141415A425A7F4760B841	Internal P	..S	G	2	1 0
0x00000000	0x40000000	0					
0x9A510100	2	0300050000000000000000000054	Table	..U	G	2	1 0
0x00003010	0x00000001	0					

例子 4: SIX 锁的模拟

SIX 锁是比较特殊的锁，是 S+IX 或 IX+S 形成的。同样使用两个窗口模拟：

在第一个窗口，执行 `lock table in share mode` 在 t1 表加 S 锁。然后继续在 t1 表插入一行数据，这需要在 t1 上获得 IX 锁。由于事先申请 S 锁，再去申请 IX 锁的时候，DB2 会申请 S+IX=SIX 锁。

```
db2inst1@dpf1:~> db2 +c "lock table t1 in share mode"
DB20000I The SQL command completed successfully.

db2inst1@dpf1:~> db2 +c "insert into t1 values('ddd', 'ddd')"
DB20000I The SQL command completed successfully.
```

在第二个窗口观察锁情况，通过 `db2pd` 的输出，会发现表上有一个 SIX 锁。

```
db2inst1@dpf1:~> db2pd -d sample -locks
Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:25:12

Locks:
Address      TranHdl      Lockname      Type      Mode Sts Owner      Dur
HoldCount Att      ReleaseFlg rrIID
0x7FC70B80 231          41414141415A425A7F4760B841 Internal P ..S G 231      1 0
0x00000000 0x40000000 0
0x7FC72C00 231          03000600040000000000000052 Row      ..X G 231      1 0
0x00000008 0x40000000 0
0x7FC73C00 231          000005000A1C000020FA757C43 CatCache ..S G 231      4 0
0x00000000 0x40000000 0
0x7FC72E00 231          03000600000000000000000054 Table     SIX G 231     255 0
0x00002000 0x40000000 0
```

11.2.2 行锁模式 ■ ■ ■

行锁模式主要分为读锁和写锁，在获得行锁前，必须先获得表锁。表 11.2 列出了 DB2 支持的行锁模式，以及为获得该行锁需要的表锁模式。

表 11.2

行锁模式（mode）	解 释 说 明	要求表锁的最低模式
S 共享锁（share）	锁的拥有者和其他并发应用都只能读该行，而不能写	IS
U 更新锁（update）	锁的拥有者可以读该行，并有可以写该行，其他并发应用只能读，U 锁是一种中间状态	IX
X 排他锁（eXclusive）	锁的拥有者可以写该行，其他并发应用不能读，也不能写。UR 隔离可以读	IX

续 表

行锁模式 (mode)	解 释 说 明	要求表锁的最低模式
W 弱排他锁 (Weak eXclusive)	类似 X 锁，但与 NW 兼容	IX
NS 下一键共享锁 (next key share)	在 CS 或 RS 隔离级别上替代 S 锁	IS
NW 下一键弱排他锁 (next key weak share)	与 X 和 W 类似。当插入索引时，该行下一键获得该锁	IX

注意：当有索引的时候才会用到 W、NW，这里我们重点关注 S、U 和 X。NS 和 S 锁是不同隔离级别下读锁的模式，在 CS 和 RS 隔离级别下读锁是 NS，在 RR 隔离级别读锁是 S

在表 11.2 中，在获得行锁前要获得最小要求的表锁，您可能会问：为什么获得行锁之前要先获得表锁呢？直接获得行锁不是更简单？在回答该问题之前，我们先做个类比，把房子比做表，房间比做行。要进房间是不是要先拿到房子的钥匙？

我们知道，现在全中国都在炒房，精明的老王当然不会放过这个机会。趁着行情好，他准备将投资的房子迅速出手，但老王是外地的，没时间亲自打理，因此交给了房地产中介。但这套房子大毛病没有，小毛病不断，比如卫生间会偶尔漏水等，从安全考虑，老王跟中介约定，如果有房间在修理，则这个房间不允许看房人进去。

房地产市场非常火爆，中介刚把房源广告贴到网上，就有一堆人要求看房。小张是第一个客人，他首先在中介那做了登记（有看房的意图，相当于表的 IS 锁），小张比较关注卧室，在里面来回观察（相当于行锁 S 锁）。

这时中介发现卫生间漏水，立即打电话请物业来修理，物业的小李在中介那做了登记（有修房的意图，相当于表的 IX 锁），开始修理卫生间（相当于行锁 X 锁）。根据老王和中介的约定，如果有房间在修理，则这个房间不允许别人进去，因此小张不能进卫生间（可理解为 S 锁和 X 锁是不兼容的）。看房的人越来越多（表的 IS 锁是兼容的），客厅里挤满了人（行锁是共享锁 S）。

中介将这些信息告诉老王，老王得知后，既兴奋又着急，着急的是他还不知道这个房子可以卖多少钱，他立即拨通了房产评估咨询师老黑的电话，老黑说现在就可以过去对房子做评估，但房间里不能有工人维修，看房的可以。

老黑在中介那查看登记信息，实际上查里面有没有在修理，假如没有登记信息的话，老黑就需要到每个房间里才能确认有没有工人维修，假设有几万个房间（几万行），那是不是需要查

完这几万个房间才可以知道？而有了登记信息，当有物业修理的时候，房子上已经做了标记（IX 锁），通过标记就知道是不是有房间在维修，这样是不是省事多了？

类比一下就是，老黑要获得这个房子的 S 锁，如果发现这个表上已经有 IX 锁，就无法评估，因为表的 S 锁和 IX 是不兼容的。

隔壁邻居家的小孩听到了敲敲打打的声音，觉得好奇，透过窗子看个究竟。中介发现了他，在房子登记表上写下了 IN（Intent None，无意图）。

总结：在获得行锁之前，需要先获得最低要求的表锁。如要查询表中某行，就是对表有读的意图，需要先得到表的意向读（IS）；如果要增、删、改某行，就是对表有写的意图，需要先得到表的意向写（IX）。

注意：DB2 中读也是要加锁的，在表上加 IS 锁，行上加 S 锁（UR 隔离级别除外）。通过 db2pd 监控时，我们会发现 NS 锁，在 RS 或 CS 隔离级别下，该锁用来替代 S 锁。

以下将通过两个实例演示行锁模式。

例子 1：S 行锁和 X 行锁模式模拟

当对某行数据增、删、改时，需要获得对应表的 IX 锁和该行的 X 锁。这时，其他应用可以对该表其他行进行增删改操作。打开两个窗口，进行模拟：

（1）在第 1 个窗口插入一行数据，不提交。

```
db2inst1@dpf1:~> db2 +c "insert into t1 values('bbb','bbb') "
```

DB20000I The SQL command completed successfully.

（2）在第 2 个窗口通过 db2pd 观察锁的情况。根据输出结果，发现有一个 IX 的表锁和一个 X 的行锁。

```
db2inst1@dpf1:~> db2pd -d sample -locks
```

Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:16:45

Locks:

Address	TranHdl	Lockname	Type	Mode	Sts	Owner	Dur
0x7FC70280	2	41414141415A425A7F4760B841	Internal	P ..S	G	2	1 0
0x00000000	0x40000000	0					
0x7FC70600	2	030006000700000000000000052	Row	..X	G	2	1 0
0x00000008	0x40000000	0					
0x7FC70800	2	030006000000000000000000054	Table	.IX	G	2	1 0
0x00002000	0x40000000	0					

（3）这时打开第 3 个窗口，并查询数据，会发现该查询处于等待状态。

```
db2inst1@dpf1:~> db2 "select * from t1"
```

(4) 在第 2 个窗口，观察锁的情况。这时会发现新增了一个 IS 表锁和一个 NS 行锁，但 NS 行锁的状态是 W，表示锁等待。

```
db2inst1@dpf1:~> db2pd -d sample -locks

Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:17:26

Locks:
Address   TranHdl   Lockname                                     Type      Mode Sts Owner
Dur HoldCount Att      ReleaseFlg rrIID
0x7FC70280 2          41414141415A425A7F4760B841 Internal P ..S G 2
1 0          0x00000000 0x40000000 0
0x7FC72380 10         41414141415A425A7F4760B841 Internal P ..S G 10
1 0          0x00000000 0x40000000 0
0x7FC70700 10         01000000010000000100007656 Internal V ..S G 10
1 0          0x00000000 0x40000000 0
0x7FC70600 2          03000600070000000000000052 Row        ..X G 2
1 0          0x00000000 0x40000000 0
0x7FC70200 10         03000600070000000000000052 Row        .NS W 0
1 0          0x00000000 0x40000000 0
0x7FC70800 2          03000600000000000000000054 Table       .IX G 2
1 0          0x00002000 0x40000000 0
0x7FC72480 10         03000600000000000000000054 Table       .IS G 10
1 0          0x00003000 0x40000000 0
```

例子 2：U 锁模拟

U 锁是一种特殊状态的锁，当通过游标执行 `select ... for update` 时，DB2 会在满足条件的行上加 U 锁，然后执行 `Update`，U 锁变为 X 锁。因为 U 锁和 S 锁是兼容的，所以当执行 `select ... for update` 时，其他应用仍然可以查该行数据，但无法更改。U 锁的目的是提高并发性。

(1) 创建 T2 表，并在里面插入 1000 条数据，然后为 `select ... for update` 语句创建一个游标，并 fetch 结果。

```
db2inst1@dpf1:~> db2 "create table t2 (id int, name char(20))"
DB20000I The SQL command completed successfully.
db2inst1@dpf1:~> db2 "begin atomic declare i int default 0; while (i<1000) do insert
into t2 values(i, 'name' || char(i) ); set i=i+1; end while; end "
DB20000I The SQL command completed successfully.

db2inst1@dpf1:~> db2 +c declare c1 cursor for select * from t2 where id=200 for update
DB20000I The SQL command completed successfully.

db2inst1@dpf1:~> db2 +c open c1
DB20000I The SQL command completed successfully.

db2inst1@dpf1:~> db2 +c fetch c1

ID          NAME
```

```
-----
200 name200

DB20000I The SQL command completed successfully.
```

(2) 在第 2 个窗口, 观察锁的情况。会发现有一个 U 锁。

```
db2inst1@dpf1:~> db2pd -d sample -locks

Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:42:09

Locks:
Address   TranHdl   Lockname                               Type      Mode Sts Owner
Dur HoldCount Att      ReleaseFlg rrIID
0x7FC74080 2          41414141415A425A7F4760B841 Internal P ..S G 2
1 0          0x00000000 0x40000000 0
0x7F5A3600 2          030007000700010000000000052 Row        ..U G 2
1 0          0x00000000 0x40000000 0
0x7F5A3400 2          01000000010000000100C00E56 Internal V ..S G 2
1 0          0x00000000 0x40000000 0
0x7F5A3300 2          030007000000000000000000054 Table        .IX G 2
1 0          0x00002000 0x40000000 0
```

(3) 这时继续在第 1 个窗口 update 刚刚获取的数据。

```
db2inst1@dpf1:~> db2 +c update t2 set name='newname' where id=200
DB20000I The SQL command completed successfully.
```

(4) 在第 2 个窗口观察锁的情况, 发现 U 锁变为了 X 锁。

```
db2inst1@dpf1:~> db2pd -d sample -locks

Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 03:00:11

Locks:
Address   TranHdl   Lockname                               Type      Mode Sts Owner
Dur HoldCount Att      ReleaseFlg rrIID
0x7FC74080 2          41414141415A425A7F4760B841 Internal P ..S G 2
1 0          0x00000000 0x40000000 0
0x7F5A3600 2          030007000700010000000000052 Row        ..X G 2
2 0          0x00000000 0x40000000 0
0x7F5A3400 2          01000000010000000100C00E56 Internal V ..S G 2
1 0          0x00000000 0x40000000 0
          0x7F5A3300 2          030007000000000000000000054 Table        .IX G 2
2 0          0x00002000 0x40000000 0
```

11.2.3 表锁和行锁兼容性 ■ ■ ■

所谓锁的兼容性, 就是指两个或多个事务是否可以同时获取锁, 如果可以, 则表示兼容;

如果不可以，则表示不兼容，其中一个事务只能等待锁释放。图 11.1 所示是表锁的兼容性表，图 11.2 所示是行锁的兼容性表（YES 表示兼容，NO 为不兼容）。

A事务的 锁模式	B事务锁模式							
	IN	IS	S	IX	SIX	U	X	Z
IN	YES	YES	YES	YES	YES	YES	YES	NO
IS	YES	YES	YES	YES	YES	YES	NO	NO
S	YES	YES	YES	NO	NO	YES	NO	NO
IX	YES	YES	NO	YES	NO	NO	NO	NO
SIX	YES	YES	NO	NO	NO	NO	NO	NO
U	YES	YES	YES	NO	NO	NO	NO	NO
X	YES	NO	NO	NO	NO	NO	NO	NO
Z	NO	NO	NO	NO	NO	NO	NO	NO

图 11.1 表锁兼容性

A事务 锁模式	B事务锁模式					
	S	U	X	W	NS	NW
S	YES	YES	NO	NO	YES	NO
U	YES	NO	NO	NO	YES	NO
X	NO	NO	NO	NO	NO	NO
W	NO	NO	NO	NO	NO	YES
NS	YES	YES	NO	NO	YES	YES
NW	NO	NO	NO	YES	YES	NO

图 11.2 行锁兼容性

举例来说，A 事务在 T1 表中插入一行数据，B 事务在 T1 插入另一行数据，那么，A,B 两个事务在 T1 表锁是否兼容呢？插入数据时，A 事务在表上需要获取 IX，B 事务也需要 IX 锁，通过图 11.1 可知，IX 与 IX 锁是兼容的。这就说明数据库支持多个应用同时对一张表进行数据增、删、改，只要操作的不是同一行即可。

另外一个例子，假设在默认的隔离级别（CS）下，A 事务插入一行数据，B 事务读取该行数据，两者是否兼容呢？插入数据需要在行上获取 X 锁，读取数据需要获取 NS 锁，通过图 11.2 可知，X 锁和 NS 锁是不兼容的。在这种情况下，B 事务只能等待 A 事务释放 X 锁才能继续处理，否则只能等待。

11.3 锁的各种问题

对很多应用来说，锁机制对最终用户来说是完全透明的。但在程序开发和 DBA 运维过程中，可能会遇到一些锁等待、锁超时、死锁等问题，本节我们将介绍这些问题出现的原因，以及监控方法。

11.3.1 锁等

前面我们说过，锁与事务关联在一起，在事务完成之前，会一直占有锁。当其他事务需要该锁时，如果申请的锁与现有的锁不兼容，就只能等待，直到其他事务释放锁，我们将这种现象叫做锁等（lock wait）。举个简单的例子：在默认的隔离级别下，A 事务在表 t1 上插入了一行数据，当 B 事务去查询此行数据时就要等待，直到 A 事务提交，释放了锁，B 事务才能继续查询。

通过命令行演示如下：

打开 3 个窗口，第一个和第二个窗口用来模拟插入和查询操作，第三个窗口用来监控。首先在第一个窗口中插入一行数据：

```
db2inst1@dpf1:~> db2 +c "insert into t1 values('dddd', 'dddd') "
```

DB20000I The SQL command completed successfully.

在第三个窗口通过 db2pd 观察锁，可以发现在表上有 IX 锁，在插入的行上有 X 锁：

```
db2inst1@dpf1:~> db2pd -d sample -locks
```

Database Partition 0 -- Database SAMPLE -- Active -- Up 4 days 04:56:47

Locks:

Address	TranHdl	Lockname	Type	Mode	Sts	Owner	Dur	HoldCount
0x9A514E00	11	41414141415A425A7F4760B841	Internal P	..S	G	11	1	0
0x9A50FF80	11	0300050009000000000000000052	Row	..X	G	11	1	0
0x9A512080	11	0300050000000000000000000054	Table	.IX	G	11	1	0

然后在第二个窗口查询表 t1 的数据，会发现查询处于等待，结果无法返回：

```
db2inst1@dpf1:~> db2 "select * from t1"
```

--等待，无法返回结果

这时，在第三个窗口中再通过 db2pd 观察锁。可以发现有个行锁处于“W”状态，W=wait，就是我们说的锁等。在本例中，查询操作需要在表上获取 IS 锁，行上需要 S 锁（NS），而之前的插入操作已经在表上有 IX 锁，插入的行上有 X 锁。表锁 IS 锁和 IX 锁是兼容的，而 S 锁和 X 锁不兼容，只能等待：

```
db2inst1@dpf1:~> db2pd -d sample -locks
```

```
Database Partition 0 -- Database SAMPLE -- Active -- Up 4 days 05:03:41
```

Locks:

Address	TranHdl	Lockname	Type	Mode	Sts	Owner	Dur	HoldCount
0x9A514E00	11	41414141415A425A7F4760B841	Internal	P	..S	G	11	1 0
0x9A514C80	9	41414141415A425A7F4760B841	Internal	P	..S	G	9	1 0
0x9A50FF80	11	030005000900000000000000052	Row		..X	G	11	1 0
0x9A50CA00	9	030005000900000000000000052	Row		.NS	W	11	1 0
0x9A50A680	9	05000000010000000100007656	Internal	V	..S	G	9	1 0
0x9A512080	11	030005000000000000000000054	Table		.IX	G	11	1 0
0x9A50C180	9	030005000000000000000000054	Table		.IS	G	9	1 0

11.3.2 锁超时 ■ ■ ■

继续上面的例子，如果 A 事务一直不提交，B 事务难道要一直等下去吗？为解决此问题，DB2 提供了 `locktimeout` 数据库参数，即设置了一个锁超时时间，当在这个参数指定的时间内无法获取锁时，就发生事务回滚。这个参数的默认值是 -1，表示无限期等待，建议根据系统的特点修改该参数。对于交易系统，一般处理比较快，可将该参数设成 30~60s，对于仓库系统，可考虑长一点。锁超时出现时，会报一个比较著名的 911 错误，即 SQL0911，Reason Code=68。

锁超时模拟如下：

首先将 SAMPLE 数据库的锁超时参数设置为 15s：

```
db2inst1@dpf1:~> db2 update db cfg for sample using locktimeout 15
DB20000I The UPDATE DATABASE CONFIGURATION command completed successfully.
SQL1363W One or more of the parameters submitted for immediate modification were
not changed dynamically. For these configuration parameters, all applications must
disconnect from this database before the changes become effective.
```

然后，在第一个窗口执行插入：

```
db2inst1@dpf1:~> db2 +c "insert into t1 values('ddaaaa') "
DB20000I The SQL command completed successfully.
```

第二个窗口执行查询，当等待 15s 后，出现锁超时，事务回滚：

```
db2inst1@dpf1:~> db2 "select * from t1"
SQL0911N The current transaction has been rolled back because of a deadlock or timeout.
Reason code "68". SQLSTATE=40001
```

11.3.3 死锁 ■ ■ ■

锁超时可以通过增加 `locktimeout` 参数值减小其发生几率，但死锁是无法通过修改数据库参数避免的。

所谓死锁，就是在两个应用中，应用 A 在等待应用 B，同时应用 B 也在等待应用 A，这样两者互相等待，造成谁也不能动。一般来说，死锁 99% 都是由于应用程序逻辑设计的问题造成的。

我们举个例子说明两个应用程序如何互相等待。面包+黄油是 TOM 和 MIKE 兄弟两个最喜欢的早餐食品，某天早晨兄弟俩吵架，哥哥 TOM 抢占了当天的面包，弟弟 MIKE 则霸占了黄油，结果两人都无法进餐。两人僵持了 10min，妈妈过来和解，TOM 分给弟弟面包，MIKE 则贡献了黄油，二人各自得到了需要的材料才得以进餐。

这是一个现实生活中死锁的例子。在 DB2 中，当若干个应用（或事务）各自占有一定资源又去申请对方资源而无法得到时，就会发生死锁。当死锁发生时，各参与方都等着对方释放资源而无法继续进行，对各参与方都没有任何益处，因此 DB2 提供了一个名为 db2dlock 的死锁检测器进程/线程，该进程每隔一定时间（默认是 10s）唤醒，它会随机找到一个参与方将其杀掉，这样这个参与方事务将回滚并释放锁资源，而其他事务继续进行。

死锁对各参与方都没有益处，应该尽量避免。在 DB2 中，死锁的发生最少要求 4 条 SQL 语句。当死锁发生时，会报 SQL0911N，Reason Code=2 错误。

死锁的模拟如下：

在第一个窗口中创建 2 张表，T1 和 T2，然后在 T1 表中插入 1 行数据，但不提交：

```
db2inst1@dpf1:~> db2 drop table t1
db2inst1@dpf1:~> db2 drop table t2
db2inst1@dpf1:~> db2 "create table t1 (col1 char(10) )"
db2inst1@dpf1:~> db2 "create table t2 (col1 char(10) )"

db2inst1@dpf1:~> db2 +c "insert into t1 values('aaa') "
DB20000I The SQL command completed successfully.
```

在第二个窗口中，在 T2 表中插入 1 行数据，不提交：

```
db2inst1@dpf1:~> db2 +c "insert into t2 values('bbb') "
DB20000I The SQL command completed successfully.
```

然后在第一个窗口中查询 T2 表，在第二个窗口中查询 T1。注意：在第一个窗口输入完命令后，先不要回车执行，当第 2 个窗口也输入完命令后再同时按 Enter 键，否则会出现锁超时而不是死锁。命令如下：

第一个窗口中访问 T2 表，

```
db2inst1@dpf1:~> db2 +c "select * from t2"
```

在第二个窗口访问 T1 表，

```
db2inst1@dpf1:~> db2 +c "select * from t1"
```

当两个窗口都输入后，同时 Enter（或很短的切换时间），等一段时间，会发现其中一个事

务由于死锁而回滚。

```
db2inst1@dpf1:~> db2 +c "select * from t1"
SQL0911N The current transaction has been rolled back because of a deadlock
or timeout. Reason code "2". SQLSTATE=40001
```

11.3.4 锁升级 ■ ■ ■

锁是一种内存资源，每个锁都要占用一定的内存空间，所谓锁升级（lock escalation），就是由行锁替换为表锁，释放内存资源的过程。举个例子，某个操作要删除 10 万行数据，在删除过程中，每行记录都要加一个行锁，当这些行锁所占内存空间超出一定大小时 DB2 就会自动将行锁替换为一个表锁，发生锁升级。

在 64bit 平台下，如果一个对象上没有锁，那么初次加锁需要 128B，如果已经有锁，则需要 64B。对于上述的 10 万行数据，行锁占用的空间大概是 $64 \times 100000 / 1024 / 1024 \text{MB} = 6\text{MB}$ 。

DB2 通过两个数据库参数 locklist 和 maxlocks 触发锁升级。locklist 用来控制每个数据库可以使用的最大锁内存大小；maxlocks 用来控制每个应用可以占用的锁内存百分比，在多用户并发系统中，此参数的目的是限制某个应用占用过多的锁内存。锁升级发生的条件包括：

- 锁内存使用超出了 locklist 大小。
- 某个应用使用的锁内存空间达到了 locklist*maxlocks%。

当锁升级发生时，会在 db2diag.log 中记录锁升级的详细信息。在下例中，BASSBI.TP_USER_SVC_DAY 表升级为 X 锁：

```
2010-05-14-03.31.36.112432+480 E27792027A590 LEVEL: Warning
PID      : 27297          TID : 182431      PROC : db2sysc 8
INSTANCE: db2inst        NODE : 008        DB   : BASSDW15
APPHDL   : 0-55356       APPID: *N0.db2inst.100517004632
AUTHID   : BASSBI
EDUID    : 182431        EDUNAME: db2agntp (BASSDW15) 8
FUNCTION: DB2 UDB, data management, sqlcEscalateLocks, probe:2
MESSAGE : ADM5500W DB2 is performing lock escalation. The total number of locks
currently held is "1762251", and the target number of locks to hold is "881125".

2010-05-14-03.31.36.114655+480 E27792618A541 LEVEL: Warning
PID      : 27297          TID : 182431      PROC : db2sysc 8
INSTANCE: db2inst        NODE : 008        DB   : BASSDW15
APPHDL   : 0-55356       APPID: *N0.db2inst.100517004632
AUTHID   : BASSBI
EDUID    : 182431        EDUNAME: db2agntp (BASSDW15) 8
FUNCTION: DB2 UDB, data management, sqlcEscalateLocks, probe:3
MESSAGE : ADM5502W The escalation of "1762246" locks on table "BASSBI .TP_USER_SVC_DAY"
to lock intent "X" was successful.
```

因为表锁的粒度和范围比较大，锁的时间相对行锁时间更长，因此锁升级会对系统的并发性能造成影响，应该尽量避免。

11.3.5 锁转换 ■ ■ ■

锁转换（lock conversion）是锁的模式转换，即从一种模式转换为另一种模式。对于同一事务来说，在某一时刻，其获取的表锁和行锁只能处于一种模式，当需要更高级别的模式时，就会自动发生锁转换。举个简单的例子，某个事务读取某行数据，那么在该行上要加 S（或 NS）锁，这时该事务要更新此行，就需要获取 X 锁，S 锁会被转换为 X 锁。在锁转换的过程中，如果有不兼容的锁仍然会发生锁等待。

注意：锁转换与锁升级存在根本差别，锁转换是从弱类型的锁向强类型锁转换的过程，而锁升级是从行锁升级为表锁。

11.4 锁监控和诊断

理解了锁的机制和各种可能出现的锁现象还不够，掌握锁监控和分析诊断方法，挖掘出问题的根源才是根本。当遇到锁问题时，脑海中首先想到的是谁引起的锁？怎么解决？

接下来为大家介绍一些诊断的方法和工具。随着版本的演变，DB2 的锁监控功能也在不断完善和发展。以下我们在讨论各种锁现象的捕获与分析时，会介绍各种版本引进的工具，相信读者读了一定会有所收获。

11.4.1 锁的分析思路和方法 ■ ■ ■

首先要确认是锁的问题，这是非常重要的一步，否则就无法对症下药，做无用功。一般情况下，当出现以下症状时，我们可以怀疑锁的问题：

- 系统响应时间比较慢、吞吐量低、CPU 利用率较低、磁盘空闲等。
- 应用程序返回错误，如 SQL0911 错误可判断是锁超时或死锁引起的。
- 通过一些日常数据库监控工具，如快照监控（snapshot monitor）和事件监控（event monitor）等工具检测到锁比较多。

数据库快照是分析锁问题的最佳起点，因为数据库快照的很多元素都是一段时间的统计值，与锁相关的片段信息如下：

```
db2inst1@dpf1:~> db2 get snapshot for database on sample | more
...
Locks held currently          = 1
Lock waits                    = 206
```

Time database waited on locks (ms)	= 14587
Lock list memory in use (Bytes)	= 8832
Deadlocks detected	= 10
Lock escalations	= 3
Exclusive lock escalations	= 0
Agents currently waiting on locks	= 1
Lock Timeouts	= 4
Number of indoubt transactions	= 0

- Lock waits 表示在快照瞬间发生锁等的次数。
- Time database waited on locks(ms)表示花费在锁等待的时间总和，除以 Lock waits 即可计算出每次锁等平均花费的时间。当这个时间比较大时，可通过 11.4.4.3 节的方法进行诊断分析。
- Deadlocks detected 表示死锁的次数，当这个值有异常时，可通过 11.4.4.5 节的方法分析。
- Lock escalations 表示锁升级的次数，当出现锁升级时，可通过 11.4.4.2 节的方法分析。
Lock timeouts 表示锁超时发生的次数，当锁超时比较多时，可通过 11.4.4.4 节的方法分析。
- Lock held currently 表示监控时间点锁的数量；Lock list memory in use(Bytes) 表示当前使用的锁内存大小。
- Agents currently waiting on locks 表示当前正在等待锁的应用数。

提醒：监控和性能调优将在第 14 章和 15 章介绍。

注意：对于锁的快照监控结果，一定要在某个时间段内分析才有意义。

11.4.2 锁升级(lock escalation)的诊断分析 ■ ■ ■

当通过数据库快照发现锁升级比较多时，可查看 db2diag.log 分析哪些表发生的锁升级，然后进行相应的调优。避免或减少锁升级的方法包括：

- 减少锁，我们将在后续 11.5 节重点介绍。
- 增加锁内存资源。可通过增加 locklist 和 maxlocks 增加锁内存资源大小。在 DB2 9 以后，这两个参数都可以设成 automatic，即自动调整。

11.4.3 锁等 (lock wait) 的捕获与诊断分析 ■ ■ ■

在多用户并发系统中，出现锁等不可避免。当锁等数量较多并比较频繁的时候，会对系统的并发性和响应时间产生很大影响，这时需要重点关注和监控。锁是瞬时的，某一时刻可能监控到很多锁等，而另一时刻锁可能已经释放。因此，为了抓到锁等的原因，有时需要通过脚本执行多次。

find_lockwait.sh 脚本用于捕获锁等时间超过 2min 的应用，大家可以根据自己的应用情况修改。

```

#-----
# Program      : find_lockwait.sh
# Description  : The script will take a snapshot of the lockwait situation.
# Usage        : ./find_lockwait.sh <dbname>
# Date         : 2009-10-22
#-----
Server=$(hostname)

if [ "$1" = "" ]
then    echo ""
        echo "Parm 1 Database name (required)"
        return
else    db_name=$1
fi

export PATH=$PATH:${HOME}/sqllib/bin/
. ${HOME}/sqllib/db2profile
Datetime=`date +%Y-%m-%d-%H:%M:%S`
echo "select 'Application - '||APPL_NAME||' is on lock wait for last 2 mins' from
SYSIBMADM.LOCKWAITS          where          minute(current_timestamp          -
timestamp(LOCK_WAIT_START_TIME)) > 2;" > check_lockwait.sql
db2 connect to $db_name
if [ "$?" = "0" ]
then
    db2 -txf check_lockwait.sql > find_lockwait.log
else
    echo "Error connecting to database -> $db_name at $Datetime" >>
find_lockwait.log
fi
# rm check_lockwait.sql

```

很多情况下，仅仅知道发生了锁等待，并找到引起锁等待的应用还远远不够，作为 DBA，我们还需要找出引起锁等待的语句，这时 db2pd 可以大显身手了。

db2pd 常用的命令组合是：

```
db2inst1@dpf1> db2pd -d sample -locks showlocks wait -tra -app -dyn > db2pd.out
```

-locks 显示锁相关信息；showlocks 显示锁的详细信息，比如行锁所在的表 ID、表空间 ID、数据页和 slot（数据页和 slot 用来定位行数据）；wait 只显示处于锁等状态的锁信息，而忽略其他，大大简化了锁的分析；-tra=transactions，显示事务信息；-app=applications，显示应用相关信息；-dyn=dynamic sql，显示动态 SQL 相关信息。

以下是以上 db2pd 命令执行后的结果输出片段，我们将一步一步分析：

```

db2inst1@dpf1:~> db2pd -d sample -locks showlocks wait -tra -app -dyn |more
Database Partition 0 -- Database SAMPLE -- Active -- Up 4 days 06:03:44
Locks:

```

```

Address  TranHdl  Lockname Type  Mode Sts Owner Dur HoldCount  Att  ReleaseFlg rrIID
0x9A50CA00 9  0300050009000000000000000052 Row .NS W 11 1 0 0x00000000
0x40000000 0  TbspaceID 3  TableID 5  PartitionID 0 Page 0 Slot 9
0x9A50FF80 11 0300050009000000000000000052 Row .X G 11 1 0 0x00000008
0x40000000 0  TbspaceID 3  TableID 5  PartitionID 0 Page 0 Slot 9

```

-- 以上是锁的信息，事务 TranHdl 9 处于锁等状态，正在等待的行被 TranHdl 11 占据（通 lockname 可知）。通过 TbspaceID 和 TableID 可确定是哪张表。

Lockname，锁名称，对于一个行锁或者表锁来说，前两个字节为表空间 ID；紧接着的两个字节为对象 ID；然后紧接的 8 个字节为 RID（其中前两个为分区 ID，后 6 个字节的前四个代表数据页 ID，后两个代表 slot ID）；最后的一个字节则为锁类型。

本例中，0300050009000000000000000052 的含义，在 x86 平台，需要首先做个转化 0003 0005 0000 00000000 0009 52

表空间 3，对象是 5，表分区是 0，数据页 0，slot 0x09，52 代表行锁。

其后的 Sts 列为状态列，G 的意思为 Granted，也就是锁被成功得到；W 就是 Wait 的意思，代表锁等待；而 C 则是 Convert，代表正在从一个低级别的锁升级到高级别的锁（比如从 S 升为 X）。在分析锁数据时，应当从 Sts 为 W 或 C 的锁看起，然后根据对应的 Lockname 找到同样名字的 Sts 为 G 的那一行，然后根据第二列 TranHdl 得到事务句柄。

Transactions:

```

Address  AppHndl  TranHdl Locks  State Firstlsn  Lastlsn  LogSpace
0x9A3DBA80 4625 2 0 READ 0x0000000000000000 0x0000000000000000 0
0x9A3DC780 4626 3 0 READ 0x0000000000000000 0x0000000000000000 0
0x9A3DD480 4627 4 0 READ 0x0000000000000000 0x0000000000000000 0
0x9A3DE180 4628 5 0 READ 0x0000000000000000 0x0000000000000000 0
0x9A3DEE80 4629 6 0 READ 0x0000000000000000 0x0000000000000000 0
0x9A3DFB80 4630 7 0 READ 0x0000000000000000 0x0000000000000000 0
0x9A3E0880 4631 8 0 READ 0x0000000000000000 0x0000000000000000 0
0x9A3E1580 5921 9 4 READ 0x0000000000000000 0x0000000000000000 0
0x9A3E2F80 5899 11 3 WRITE 0x0000000006237049 0x0000000006237049 124
0x9A3E3C80 5964 12 0 READ 0x0000000000000000 0x0000000000000000 0

```

-- 以上是事务相关信息，事务 9 对应的 AppHndl 是 5921，上面有 4 个锁，事务状态是 READ；而事务 11 对应的 AppHndl 是 5899，上面有 3 个锁，事务状态是 WRITE

Applications:

```

Address  AppHndl  Status  C-AnchID  C-StmtUID L-AnchID L-StmtUID Appid
0x20AA7690 5921 Lock-wait 944 5 0 0 *LOCAL.db2inst1.110126212508
0x20AA0060 4631 ConnectCompleted 0 0 0 0 *LOCAL.DB2.110122162330
0x20A97690 4630 ConnectCompleted 0 0 0 0 *LOCAL.DB2.110122162329
0x20990060 5899 UOW-Waiting 0 0 85 1 *LOCAL.db2inst1.110126211438
0x20A90060 4629 UOW-Executing 0 0 0 0 *LOCAL.DB2.110122162328
0x20AD8430 5964 UOW-Waiting 0 0 207 1 *LOCAL.db2inst1.110126214851
0x20967690 4628 ConnectCompleted 0 0 0 0 *LOCAL.DB2.110122162327
0x20960060 4627 ConnectCompleted 0 0 0 0 *LOCAL.DB2.110122162326
0x209D4BB0 4626 ConnectCompleted 0 0 0 0 *LOCAL.DB2.110122162325
0x20A30060 4625 UOW-Waiting 0 0 970 1 *LOCAL.db2inst1.110122162324

```

-- 以上是应用相关信息，AppHndl 表示应用句柄，Status 表示应用状态，C-AnchID/C-StmtUID 和

L-AnchID/L-StmtUID 对分别用来唯一标识一条 SQL 语句，C 表示 current，当前执行的语句；L 表示 last，上一句执行的语句。在本例中，应用句柄 5921 处于 Lock-wait 状态，而应用句柄 5899 处于 UOW-Waiting 状态。C-AnchID/C-StmtUID 等于(944/5)的语句正在锁等，而(85,1)表示的 SQL 语句持有锁

Dynamic SQL Statements:

Address	AnchID	StmtUID	NumEnv	NumVar	NumRef	NumExe	Text
0x9CA64160	85	1	0	0	1	1	insert into t1 values('dddd', 'dddd')
0x9CB2FC60	207	1	0	0	1	1	select * from dba.snap_lockwait
0x9CB28EC0	686	1	0	0	1	1	update t1 set col1='aaxxx' where col2='xxxx'
0x9C9C0440	829	1	0	0	2	2	select * from t1 with rr
0x9CA094C0	850	2	1	1	1	1	select 'Application - ' APPL_NAME ' is on lock wait for last 5 mins' from SYSIBMADM.LOCKWAITS where minute(current_timestamp - timestamp(LOCK_WAIT_START_TIME)) > 5
0x9CA79ED0	944	5	1	1	1	1	select * from t1
0x9CB29650	970	1	0	0	1	1	select * from t1 for update with rr
0x9C9C00F0	977	3	1	1	1	1	insert into dba.snap_lockwait select * from SYSIBMADM.LOCKWAITS

-- 以上是动态 SQL 语句相关信息，(AnchID, StmtUID)用来唯一识别一条 SQL 语句，NumExe 表示 SQL 语句执行的次数，Text 表示 SQL 语句。在本例中，句柄为(944,5)代表的语句“select * from t1”正在等待句柄(85,1)代表的语句“insert into t1 values(111)”

通过以上分析可知，我们可以根据 db2pd -locks 选项定位锁信息，找到持有锁和锁等的事务句柄，然后根据 db2pd -tra 选项找到这几个事务句柄所属的应用句柄，接着根据 db2pd -app 找到这些应用句柄刚执行的或正在执行的 SQL 句柄（通过 AnchID 和 StmtID 识别），最后根据 db2pd -dyn 找到这些 SQL 句柄代表的 SQL 语句。在实际环境中，可根据需要写一个脚本，定时调度和捕获锁等待信息。

对于锁的监控，尽管锁快照也可以获取锁的信息，但我们建议使用 db2pd。在一个繁忙的系统中，锁列表可以达到几百兆大小，锁快照捕获时需要 DB2 内部数据结构加上内部锁，以保证自己独享内存，这种做法可能对性能造成负面影响。而 db2pd 不需要占用内部锁，因此推荐使用。

那么，通过以上 db2pd 命令选项是否一定能找到引起锁等待的 SQL 语句呢？答案是否定的，db2pd 具有一定的局限性。例如，如果一个事务执行完增、删、改操作后，又进行了多次读操作，而 db2pd 只能抓出当前正在执行（C-AnchID/C-StmtID）或上一条执行的语句（L-AnchID/L-StmtID），这时 db2pd 抓取的结果就不是真正占有锁的 SQL 语句了。

举例来说：

在 A 窗口中执行如下两条 SQL：

```
db2inst1@dpf1:~> db2 +c "insert into t1 values('eeee','eeee')"
```

```
DB20000I The SQL command completed successfully.
```

```
db2inst1@dpf1:~> db2 +c "select * from t1 where col1='eeee'"
```

```
COL1          COL2
-----
eeee          eeee

1 record(s) selected.
```

在 B 窗口中执行查询，出现锁等：

```
db2inst1@dpf1:~> db2 +c "select * from t1 where col1='aaaa' "
```

在 C 窗口中通过 db2pd 命令进行监控，并对结果进行分析，定位到引起锁等的语句是“select * from t1 where col1='eeee'”，而非“insert into ...”：

```
db2inst1@dpf1:~> db2pd -d sample -locks wait -tra -app -dyn
```

Locks:

Address	TranHdl	Lockname	Type	Mode	Sts	Owner
0x9A512080	11	0300050009000000000000000052	Row	.NS	W	12
0x9A516400	12	0300050009000000000000000052	Row	..X	G	12

Transactions:

Address	AppHndl	[nod-index]	TranHdl	Locks	State
0x9A3E2F80	5899	[000-05899]	11	4	READ
0x9A3E3C80	5964	[000-05964]	12	3	WRITE

Applications:

Address	AppHndl	Status	C-AnchID	C-StmtUID	L-AnchID	L-StmtUID	Appid
0x20990060	5899	Lock-wait	321	1	85	1	*LOCAL.db2inst1.110126211438
0x20AD8430	5964	UOW-Waiting	0	0	734	8	*LOCAL.db2inst1.110126214851

Dynamic SQL Statements:

Address	AnchID	StmtUID	Text
0x9CB697D0	85	3	CALL REORGCHK_IX_STATS ('T', 'DBA' ".SNAP_LOCKWAIT")
0x9C9C9E20	321	1	select * from t1 where col1='aaaa'
0x9CA09990	499	1	insert into t1 values('eeee','eeee')
0x7CDC0340	734	8	select * from t1 where col1='eeee'

在 9.5 版本，当使用 db2pd 无法捕获正确的锁等待 SQL 语句时，可考虑设置较小的 locktimeout 数据库参数，这样当出现锁超时，就可以利用 9.5 版本新增的 db2_capture_locktimeout 注册变量来捕获了。

11.4.4 锁超时 (lock timeout) 的捕获与诊断分析 ■ ■ ■

如果发现系统中锁超时的数量比较多，就要进一步分析，找到引起锁超时的 SQL 语句，进行优化。在默认情况下，在发生锁超时，并不会记录哪些语句发生了锁超时，也不会记录哪些语句引起的锁超时，这需要额外的工具和命令来捕获。DB2 在不同的版本中有不同的工具和

方法，如 9.1 的锁超时捕获方法主要是 db2pd 结合 db2cos 回调脚本来实现，而 9.5 引入了 locktimeout 注册变量参数。下面为大家详细介绍这两类机制。

1. 9.1 锁超时捕获

对于锁超时的捕获，在 DB2 9 中可以通过 db2pd 结合 db2cos 回调脚本来捕获，当发生锁超时时，会自动调用回调脚本，记录相关信息。具体操作可分为以下步骤。

(1) 改写 db2cos 回调脚本。

db2cos 脚本位于<inst_home>/sqllib/bin/db2cos (Unix/Linux) 或 <inst_home>\sqllib\bin\db2cos.bat (Windows)。在使用前，可将该脚本拷贝到 <inst_home>/sqllib/adm 目录。对于 locktimeout 事件，可改写 db2cos 脚本，调用 db2pd 命令将相关信息输出到文件中：

```
"LOCKTIMEOUT")
  echo "Lock Timeout Caught"                >> $logfile
  if [ ! -n "$database" ]
  then
    db2pd -inst                             >> $logfile
  else
    db2pd -db $database -locks -tra -app -dyn >> $logfile
  fi
;;
```

(2) 通过 db2pdcfg 配置锁超时事件。

配置 db2pdcfg 锁超时事件，一旦发生锁超时，就会调用回调脚本：

```
db2inst1@dpf1:~> db2pdcfg -catch locktimeout count=1
Error Catch #3
Sqlcode:          0
ReasonCode:       0
ADMCcode:         0
DiagText:
ZRC:              -2146435004
ECF:              0
Component ID:     0
LockName:         Not Set
LockType:         Not Set
Current Count:    0
Max Count:        1
Bitmap:           0x4A1
Action:           Error code catch flag enabled
Action:           Execute /home/db2inst1/sqllib/bin/db2cos callout script
Action:           Produce stack trace in db2diag.log
```

(3) 模拟锁超时，分析 db2cos 输出文件。

模拟一个锁超时，然后检查 db2diag.log，发现 db2cos 脚本已被调用：

```
2010-11-30-09.53.01.032917-480 I10290332G480      LEVEL: Event
PID      : 14568                      TID   : 2932861856  PROC  : db2sysc 0
INSTANCE: db2inst1                   NODE  : 000          DB    : DB1
APPHDL   : 0-1410                     APPID: *LOCAL.db2inst1.101130175237
AUTHID   : DB2INST1
EDUID    : 62                         EDUNAME: db2agent (DB1) 0
FUNCTION: DB2 UDB, trace services, pdInvokeCalloutScript, probe:10
START    : Invoking /home/db2inst1/sqllib/bin/db2cos from lock manager sqlplnfd

2010-11-30-09.53.07.092005-480 I10290813G463      LEVEL: Event
PID      : 14568                      TID   : 2932861856  PROC  : db2sysc 0
INSTANCE: db2inst1                   NODE  : 000          DB    : DB1
APPHDL   : 0-1410                     APPID: *LOCAL.db2inst1.101130175237
AUTHID   : DB2INST1
EDUID    : 62                         EDUNAME: db2agent (DB1) 0
FUNCTION: DB2 UDB, trace services, pdInvokeCalloutScript, probe:20
TOP      : Completed invoking /home/db2inst1/sqllib/bin/db2cos
```

在<inst_home>/sqllib/db2dump 下生成一个名为<pid>.<eduid>.<node>.cos.txt 的文件。通过分析该文件，可找到锁超时相关的语句。因为 db2cos 脚本调用的也是 db2pd -d sample -locks -tra -app -dyn，因此输出结果的分析同 11.4.2 锁等的分析是完全相同的。

2.9.5 锁超时捕获

通过前面的分析，我们看到 db2pd 捕获 SQL 语句的能力有限，为了更加有效、方便地捕获锁超时，9.5 版本引入了 db2_capture_locktimeout 注册变量。从最佳实践的角度分析，建议在配置这个参数的同时，创建 deadlock event monitor，有效捕获锁超时和死锁语句。以下举例说明。

(1) 首先设置参数，并创建事件监控器：

```
db2inst1@dpf1:~> db2set DB2_CAPTURE_LOCKTIMEOUT=ON
db2inst1@dpf1:~> db2 update db cfg for sample using locktimeout 15
db2inst1@dpf1:~> db2stop force
db2inst1@dpf1:~> db2start
db2inst1@dpf1:~> db2 connect to sample
Database Connection Information

Database server          = DB2/LINUX 9.7.1
SQL authorization ID    = DB2INST1
Local database alias    = SAMPLE

db2inst1@dpf1:~> mkdir locks
db2inst1@dpf1:~> db2 drop event monitor dlockevm
db2inst1@dpf1:~> db2 "create event monitor dlockevm for deadlocks with details history
write to file '/home/db2inst1/locks' "
```

```
DB20000I The SQL command completed successfully.
db2inst1@dpf1:~> db2 set event monitor dlockvm state=1
```

(2) 然后通过两个窗口模拟一个锁超时。

在第一个窗口中执行如下语句：

```
db2inst1@dpf1:~> db2 +c "insert into t1 values('ffff','ffff') "
db2inst1@dpf1:~> db2 +c "select * from t1 where col1='ffff' "
COL1          COL2
-----
ffff          ffff
1 record(s) selected.
```

在第二个窗口中执行查询，发生超时：

```
db2inst1@dpf1:~> db2 +c "select * from t1 where col1='xxxx' "
SQL0911N The current transaction has been rolled back because of a deadlock
or timeout. Reason code "68". SQLSTATE=40001
```

这时检查 <instance_home>/sqlib/db2dump 目录，发现生成了 db2locktimeout.0.72.2011-01-27-21-49-19 文件，这个文件包含两部分内容，一部分为锁请求相关信息，包含了锁超时的语句，申请的锁类型等，另一部分为锁拥有者相关信息，包含了当前锁的模式，还包含当前事务所执行的所有 SQL 语句历史信息。在本例中，“select * from t1 where col1='xxxx'”发生了锁等，而罪魁祸首是“insert into t1 values('ffff','ffff)”，这些都详细地记录在本文件中：

LOCK TIMEOUT REPORT

```
Date:                27/01/2011
Time:                21:49:19
Instance:            db2inst1
Database:            SAMPLE
Database Partition:  0
```

Lock Information:

```
Lock Name:           030005000B00000000000000052
Lock Type:           Row
Lock Specifics:      Tablespace ID=3, Table ID=5, Row ID=x0B000000000000000
```

```
Lock Requestor:      -- 锁请求者相关信息
System Auth ID:      DB2INST1
Application Handle:   [0-54]
Application ID:       *LOCAL.db2inst1.110128054807
Application Name:     db2bp
Requesting Agent ID:  72
Coordinator Agent ID: 72
Coordinator Partition: 0
Lock timeout Value:   15000 milliseconds
```

```

Lock mode requested:      .NS                      -- 请求锁的模式
Application Status:      (SQLM_UOWEXEC)
Current Operation:      (SQLM_FETCH)
Lock Escalation:        No

```

Context of Lock Request:

```

Identification:          UOW ID (1); Activity ID (2)

```

Activity Information:

```

Package Schema:          (NULLID )
Package Name:            (SQLC2H20NULLID )
Package Version:         ()
Section Entry Number:    201
SQL Type:                Dynamic
Statement Type:          DML, Select (blockable)
Effective Isolation:     Cursor Stability
Statement Unicode Flag:   No
Statement:                select * from t1 where col1='xxxx' -- 锁等语句

```

Lock Owner (Representative): -- 锁拥有者相关信息

```

System Auth ID:          DB2INST1
Application Handle:       [0-46]
Application ID:           *LOCAL.db2inst1.110128054759
Application Name:         db2bp
Requesting Agent ID:      19
Coordinator Agent ID:     19
Coordinator Partition:    0
Lock mode held:           ..X                      -- 锁持有模式

```

List of Active SQL Statements: Not available

List of Inactive SQL Statements from current UOW:

```

Entry:                    #1
Identification:           UOW ID (3); Activity ID (2)
Package Schema:           (NULLID )
Package Name:             (SQLC2H20)
Package Version:          ()
Section Entry Number:     201
SQL Type:                 Dynamic
Statement Type:           DML, Select (blockable)
Effective Isolation:      Cursor Stability
Statement Unicode Flag:    No
Statement:                select * from t1 where col1='ffff' -- SQL 语句

```

```

Entry:                    #2
Identification:           UOW ID (3); Activity ID (1)
Package Schema:           (NULLID )
Package Name:             (SQLC2H20)
Package Version:          ()
Section Entry Number:     203

```

SQL Type:	Dynamic
Statement Type:	DML, Insert/Update/Delete
Effective Isolation:	Cursor Stability
Statement Unicode Flag:	No
Statement:	insert into t1 values('ffff','ffff') --SQL 语句

注意：仅仅设置 db2_capture_locktimeout 注册变量是不够的，还必须创建一个死锁事件监控器才能正确完整地捕获锁超时相关的语句信息。

11.4.5 死锁 (deadlock) 的捕获与诊断分析 ■ ■ ■

死锁对性能有严重影响，应尽量避免。当监控到有死锁发生时，应该想办法记录引起死锁的 SQL 语句。目前最常用的方法是通过死锁事件监控器 (deadlock event monitor) 实现。以下通过一个实例演示死锁的诊断和分析过程。

(1) 创建 deadlock 事件监控器，将结果输出到事先创建的目录/home/db2inst1/deadlock 下：

```
db2inst1@dpf1:~> cd /home/db2inst1
db2inst1@dpf1:~> mkdir deadlock
db2inst1@dpf1:~> db2 connect to db1
Database Connection Information

Database server          = DB2/LINUX 9.7.1
SQL authorization ID     = DB2INST1
Local database alias     = DB1

db2inst1@dpf1:~> db2 " create event monitor dlockevm for deadlocks with details
history write to file '/home/db2inst1/deadlock' "
DB20000I The SQL command completed successfully.
db2inst1@dpf1:~> db2 set event monitor dlockevm state=1 --让事件监控器生效
DB20000I The SQL command completed successfully.
```

(2) 模拟一个死锁发生。

打开两个窗口，在第一个窗口中创建两张表：t1 和 t2，然后在 t1 表中插入一行数据：

```
-- 创建 t1 和 t2
db2inst1@dpf1:~> db2 drop table t1
db2inst1@dpf1:~> db2 drop table t2
db2inst1@dpf1:~> db2 "create table t1 (col1 char(10) )"
db2inst1@dpf1:~> db2 "create table t2 (col1 char(10) )"

-- 在 t1 表中插入数据
db2inst1@dpf1:~> db2 +c "insert into t1 values('aaa') "
DB20000I The SQL command completed successfully.
```

这时在第二个窗口中，在 t2 表中插入一行数据：

```
-- 在 t2 表中插入数据
db2inst1@dpf1:~> db2 +c "insert into t2 values('bbb') "
DB20000I The SQL command completed successfully.
```

然后，在两个窗口中分别执行对 t2 和 t1 表的查询。注意：在第一个窗口中输入完命令后，先不要执行，当第二个窗口也输入完命令后再同时执行，否则会出现锁超时而不是死锁。

在第一个窗口中访问 T2 表：

```
-- 访问 t2 表， 注意先别执行
db2inst1@dpf1:~> db2 +c "select * from t2"
```

在第二个窗口中访问 t1 表：

```
-- 访问 T1 表， 注意先别回车
db2inst1@dpf1:~> db2 +c "select * from t1"
```

当两个窗口命令都输入后，同时执行（或很短的切换时间），会发现其中一个事务由于死锁而回滚。在本例中，第二个窗口的事务回滚了：

```
db2inst1@dpf1:~> db2 +c "select * from t1"
SQL0911N The current transaction has been rolled back because of a deadlock
or timeout. Reason code "2". SQLSTATE=40001
```

（3）死锁结果分析。

在对死锁事件分析之前，先关闭事件监控器，并将结果输出到一个文本文件 deadlock.txt 中：

```
db2inst1@dpf1:~/deadlock> db2 flush event monitor dlockevm
DB20000I The SQL command completed successfully.
db2inst1@dpf1:~/deadlock> db2 set event monitor dlockevm state=0
DB20000I The SQL command completed successfully.
db2inst1@dpf1:~/deadlock> ls
00000000.evt db2event.ct1

db2inst1@dpf1:~/deadlock> db2evmon -path /home/db2inst1/deadlock > deadlock.txt
Reading /home/db2inst1/deadlock/00000000.evt ...
```

打开 deadlock.txt，找出引起死锁的 SQL 语句：

```
-----
                        EVENT LOG HEADER
Event Monitor name: DLOCKEVM
Server Product ID: SQL09071
Version of event monitor data: 11
Byte order: LITTLE ENDIAN
Number of nodes in db2 instance: 1
Codepage of database: 1208
Territory code of database: 1
Server instance name: db2inst1
-----
```

```

-----
Database Name: SAMPLE
Database Path: /home/db2inst1/db2inst1/NODE0000/SQL00001/
First connection timestamp: 01/27/2011 21:47:59.211609
Event Monitor Start time: 01/27/2011 22:34:04.842280
-----

3) Deadlock Event ...
Deadlock ID: 1 --Deadlock ID 是 1
Number of applications deadlocked: 2 --参与死锁的应用有两个
Deadlock detection time: 01/27/2011 22:37:09.787693
Rolled back Appl participant no: 2 --回滚的应用参与者编号是 2
Rolled back Appl Id: *LOCAL.db2inst1.110128063201 --回滚的应用 ID 是*LOCAL.db2inst1.
110128063201
Rolled back Appl seq number: : 0006

4) Connection Header Event ...
Appl Handle: 136
Appl Id: *LOCAL.db2inst1.110128063201 --回滚的应用连接头信息
Appl Seq number: 00006
DRDA AS Correlation Token: *LOCAL.db2inst1.110128063201
Program Name : db2bp
... --省略

5) Deadlocked Connection ...
Deadlock ID: 1
Participant no.: 2
Participant no. holding the lock: 1
Appl Id: *LOCAL.db2inst1.110128063201 --回滚的应用连接信息
Appl Seq number: 00006
Appl Id of connection holding the lock: *LOCAL.db2inst1.110128054759 --拥有锁的
应用 Appl ID
Seq. no. of connection holding the lock: 00001
Lock wait start time: 01/27/2011 22:37:04.601677
Lock Name : 0x030005000400000000000000000052
Current Mode : none
Deadlock detection time: 01/27/2011 22:37:09.788844
Table of lock waited on : T1
Schema of lock waited on : DB2INST1
Data partition id for table : 0
Type of lock: Row
Mode of lock: X - Exclusive
Mode application requested on lock: NS - Share (CS/RS)
Deadlocked Statement:
Type : Dynamic
Operation: Fetch
Text : select * from t1 --发生死锁回滚的 SQL 语句
... -- 省略持有的锁的信息

```

```

6) Deadlock statement history ...      --历史 SQL 语句信息
Deadlock ID           : 1
Participant No        : 2
Stmt history ID       : 2              --history ID=2, 表示第二个执行的语句
Type                  : Dynamic
... -- 省略
Statement text         : select * from t1  --SQL 语句

7) Deadlock statement history ...
Deadlock ID           : 1
Participant No        : 2
Stmt history ID       : 1              --history ID=1, 表示第一个执行的语句
Type                  : Dynamic
... -- 省略
Statement text         : insert into t2 values('bbb')  --SQL 语句

8) Connection Header Event ...
Appl Handle: 46
Appl Id: *LOCAL.db2inst1.110128054759
Appl Seq number: 00018

9) Deadlocked Connection ...
Deadlock ID: 1
Participant no.: 1
Participant no. holding the lock: 2
Appl Id: *LOCAL.db2inst1.110128054759
Appl Seq number: 00018
Appl Id of connection holding the lock: *LOCAL.db2inst1.110128063201
Seq. no. of connection holding the lock: 00001
Lock wait start time: 01/27/2011 22:37:03.515204
Lock Name             : 0x030006000500000000000000000052
Current Mode          : none
Deadlock detection time: 01/27/2011 22:37:09.795276
Table of lock waited on      : T2
Schema of lock waited on     : DB2INST1
Data partition id for table   : 0
Type of lock: Row
Mode of lock: X - Exclusive
Mode application requested on lock: NS - Share (CS/RS)
Deadlocked Statement:
  Type      : Dynamic
  Operation: Fetch
  Text      : select * from t2

... --省略持有的锁信息

10) Deadlock statement history ...
Deadlock ID           : 1
Participant No        : 1

```

```

Stmt history ID      : 2
Type                 : Dynamic
... -- 省略
Statement text       : select * from t2

11) Deadlock statement history ...
Deadlock ID          : 1
Participant No       : 1
Stmt history ID      : 1
Type                 : Dynamic
... -- 省略
Statement text       : insert into t1 values('aaa')

```

通过“3) Deadlock Event”可知，Deadlock ID 是 1，由于死锁发生回滚的应用 Appl ID 是 110128063201。

通过“5) Deadlocked Connection”可知，App Id 110128063201 在表 t1 上行上申请 NS 锁，但该表行被 110128054759 应用加上了 X 锁，同时，110128063201 在 t2 表的行上拥有 X 锁。

通过“6) Deadlock statement history”和“7) Deadlock statement history”可知，App Id 110128063201 执行的两条语句分别为 insert into t2 values('bbb') 和 select * from t1。

通过“9) Deadlocked Connection”可知，App Id 110128054759 在表 t2 上行上申请 NS 锁，但该表行被 110128063201 应用加上了 X 锁，同时，110128054759 在 t1 表的行上拥有 X 锁。

通过“10) Deadlock statement history”和“11) Deadlock statement history”可知，App 110128063201 执行的两条语句分别为 insert into t1 values('aaa')和 select * from t2。

11.4.6 9.7 锁事件监控器 ■ ■ ■

随着 DB2 版本的演变，锁监控工具变得越来越强大，但不可否认的是，针对锁等、锁超时和死锁都需要单独的工具来监控，这无形中增加了监控的复杂度和难度。为彻底改变这种情况，减少 DBA 的负担，DB2 9.7 版本引入了全新的锁事件监控器模型，使用统一的方法来捕获锁超时、锁等待和死锁，这就是 CREATE EVENT MONITOR FOR LOCKING 语句。

要使用 CREATE EVENT MONITOR FOR LOCKING，需通过几个数据库参数来设置，以下是默认的值。通过参数的解释，我们很容易就知道每一个参数分别用来捕获不同的事件。

Lock timeout events	(MON_LOCKTIMEOUT) = NONE
Deadlock events	(MON_DEADLOCK) = WITHOUT_HIST
Lock wait events	(MON_LOCKWAIT) = NONE
Lock wait event threshold	(MON_LW_THRESH) = 50000000

这几个参数可以配置成不同的值，有关它们的详细介绍请参看 DB2 信息中心。为了简化，我们建议将前 3 个参数都设成 HIST_VALUES，将 MON_LW_THRESH 设成 10000（10 秒），即锁等待超过 10 秒时即开始捕获相关信息：

```
db2inst1@dpf1:~ > db2 update db cfg for sample using MON_LOCKWAIT hist_and_values
MON_DEADLOCK hist_and_values MON_LOCKTIMEOUT hist_and_values MON_LW_THRESH 10000
```

参数设置完成后，即可开始创建锁监控器进行锁事件的捕获：

```
db2inst1@dpf1:~ > db2 "create event monitor lockevmon for locking write to unformatted
event table (table locks) "
DB20000I The SQL command completed successfully.

db2inst1@dpf1:~ > db2 set event monitor lockevmon state=1
DB20000I The SQL command completed successfully.
```

这个事件监控器与我们熟悉的死锁监控器是不是有些不同？没错，for locking 是全新引入的事件类型，write to unformatted event table 选项会将锁事件写到未格式化的表里（unformatted event table, UE）。之所以叫未格式化，是指里面的信息需要解析才可理解，解析的方法包括 Java 程序和存储过程，接下来我们会给大家介绍。

首先介绍 Java 程序，在 DB2 实例的 sample 目录下自带了一个 Java 解析程序，这个 Java 程序需要编译成 class 才能执行，编译方法如下：

```
--将 db2evmonfmt.java 和 DB2EvmonLocking.xsl 拷贝到另外一个目录下。
# cp /home/db2inst1/sqllib/samples/java/jdbc/db2evmonfmt.java
/home/db2inst1/sqllib/samples/java/jdbc/DB2EvmonLocking.xsl /home/db2inst1

# su - db2inst1
$ db2inst1@dpf1:~> ~/sqllib/java/jdk32/bin/javac db2evmonfmt.java （32 位平台，如果是 64 位平台，将 jdk32 改成 jdk64）

$ db2inst1@dpf1:~> ls -alt db2evmonfmt*
-rw-r--r-- 1 db2inst1 db2iadm1 16502 2011-01-28 00:04 db2evmonfmt.class
-r-xr-xr-x 1 db2inst1 db2iadm1 38697 2011-01-28 00:02 db2evmonfmt.java
```

然后模拟一个锁等、锁超时和死锁，具体步骤省略。

当出现锁等、锁超时或死锁后，就可以用我们刚编译过的 Java 程序对未格式化的表进行解析了：

```
db2inst1@dpf1:~> ~/sqllib/java/jdk32/jre/bin/java db2evmonfmt -d sample -ue locks
-ftext -u db2inst1 -p password > db2locks.out
```

-ue <table>指定未格式化的表名，-ftext 是将输出格式化为文本文件，-u、-p 用来指定用户名和密码。

以下是解析结果的一个片段，包含了一个死锁事件和一个锁超时事件：

```
SELECT evmon.xmlreport FROM TABLE ( EVMON_FORMAT_UE_TO_XML( 'LOG_TO_FILE',FOR EACH
ROW OF ( SELECT * FROM locks ORDER BY EVENT_ID, EVENT_TIMESTAMP, EVENT_TYPE, MEMBER )))
AS evmon

-----
Event ID          : 8
```

```

Event Type           : DEADLOCK           --事件类型死锁
Event Timestamp      : 2011-01-28-00.24.16.770628
Partition of detection : 0
-----

Deadlock Graph
-----
Total number of deadlock participants : 2           --有两个应用参与了死锁
Participant that was rolled back      : 2           --第二个参与者应用发生了回滚
Type of deadlock                     : local

Participant   Participant   Deadlock Member Application Handle
Requesting Lock Holding Lock
-----
1             2             0             0324
2             1             0             0136

Participant No 2 requesting lock
-----
Lock Name      : 0x03000600050000000000000000052
... -- 省略

Participant No 1 requesting lock
-----
Lock Name      : 0x03000500070000000000000000052
... -- 省略

Attributes      Requester      Requester
-----
Participant No   2             1
Application Handle 0324         0136
Application ID    *LOCAL.db2inst1.110128081329 *LOCAL.db2inst1.110128063201
Application Name  db2bp         db2bp

Current Activities of Participant No 2           --参与者 2 应用当前活动
-----
Activity ID      : 2
Uow ID          : 5
Stmt text       : select * from t2 where col1='yyyy'

Past Activities of Participant No 2           --参与者应用 2 过去的活动
-----
Past Activities wrapped: no
Activity ID      : 1
Uow ID          : 5
... -- 省略
Stmt text       : insert into t1 values('abcd')

Current Activities of Participant No 1           --参与者应用 1 当前活动

```

```

-----
Activity ID          : 2
Uow ID              : 17
... -- 省略
Stmt text           : select * from t1 where col1='xxx'

Past Activities of Participant No 1                --参与者应用 1 过去的活动
-----
Past Activities wrapped: no
Activity ID         : 1
Uow ID              : 17
Stmt text           : insert into t2 values('ddddaa')

-----
Event ID            : 9
Event Type           : LOCKTIMEOUT                --锁超时事件的捕获
Event Timestamp      : 2011-01-28-00.32.21.002275
Partition of detection : 0
-----

Participant No 1 requesting lock
-----
Lock Name            : 0x03000500080000000000000000052
... -- 省略

Attributes           Requester                     Owner
-----
Participant No       1                             2
Application Handle    0136                         0324
Application ID        *LOCAL.db2inst1.110128063201      *LOCAL.db2inst1.110128081329
Application Name      db2bp                          db2bp
... -- 省略

Current Activities of Participant No 1
-----
Activity ID          : 1
Uow ID              : 18
Stmt text           : select * from t1 where col1='2222'

Past Activities of Participant No 1
-----
Activities not available

Current Activities of Participant No 2
-----
Activities not available

Past Activities of Participant No 2
-----

```

```
Past Activities wrapped: no
Activity ID      : 1
Uow ID          : 7
Stmt text       : insert into t1 values('11111')
```

除了通过 db2evmonfmt 解析未格式表，还可以通过 SYSPROC.EVMON_FORMAT_UE_TO_TABLES 存储过程将未格式表转换为一组关系表，语法如下：

```
>>-EVMON_FORMAT_UE_TO_TABLES--(--evmon_type--,--xsrschema--,---->
>--xsrobjectname--,--xmlschemafile--,--tabschema--,----->
>--tbsp_name--,--options--,--commit_count--,--fullselect--)----><
```

在这个存储过程中，evmon_type 指定为 LOCKING，options 指定为 RECREATE_FORCE，表示重建关系表，tabschema 指定创建的关系表的模式，tbsp_name 指定表空间，fullselect 是指定查询和过滤条件进行格式化：

```
db2inst1@dpf1:~> db2 "call
sysproc.EVMON_FORMAT_UE_TO_TABLES('LOCKING',NULL,NULL,NULL,NULL,NULL,'RECREATE_F
ORCE',-1,'SELECT * FROM locks ORDER BY event_timestamp')"
Return Status = 0
```

这个存储过程的结果是一组关系表，以下是这几张表的含义。

- LOCK_EVENT 表对应发生的锁事件，每个事件对应一条记录。
- LOCK_PARTICIPANTS 表标识锁事件的参与者，每个参与的应用程序对应一行记录。
- LOCK_PARTICIPANT_ACTIVITIES 表包含了参与事件的应用程序曾经和当前正在执行的语句。

这些表中都有一个 XMLID 列来唯一标识每个事件，这个列的内容格式如下：

```
<event_header>_<event_id>_<event_type>_<event_timestamp>_<partition>
```

例如，对锁事件，event_header 统一为 db2LockEvent，如果是锁等待，则 event_type 为 LOCKWAIT，如果是死锁，则 event_type 为 DEADLOCK，如果是锁超时，则 event_type 为 LOCKTIMEOUT。

可通过 SQL 语句对锁相关事件进行查询：

```
db2inst1@dpf1:~> db2 "select substr(lp.xmlid,1,64), lp.participant_no,
lp.participant_type, lp.APPLICATION_HANDLE, lp.participant_no_holding_lk,
lpa.ACTIVITY_ID, lpa.ACTIVITY_TYPE, varchar(lpa.stmt_text,50) as STATEMENT from
lock_participants lp, lock_participant_activities lpa where lp.xmlid=lpa.xmlid and
lp.participant_no=lpa.participant_no order by lp.xmlid desc, lp.PARTICIPANT_NO,
lpa.ACTIVITY_ID "
```

-- 以下示例结果包含一个死锁事件和一个锁超时事件

```
db2LockEvent_9_LOCKTIMEOUT_2011-01-28-00.32.21.002275_0      1 Requester
136      2      1 current      select * from t1 where col1='2222'
db2LockEvent_9_LOCKTIMEOUT_2011-01-28-00.32.21.002275_0      2 Owner
```

```

324      -      1 past      insert into t1 values('11111')
db2LockEvent_8_DEADLOCK_2011-01-28-00.24.16.770628_0      1 Requester
136      2      1 past      insert into t2 values('dddaa')
db2LockEvent_8_DEADLOCK_2011-01-28-00.24.16.770628_0      1 Requester
136      2      2 current    select * from t1 where col1='xxx'
db2LockEvent_8_DEADLOCK_2011-01-28-00.24.16.770628_0      2 Requester
324      1      1 past      insert into t1 values('abcd')
db2LockEvent_8_DEADLOCK_2011-01-28-00.24.16.770628_0      2 Requester
324      1      2 current    select * from t2 where col1='yyyy'

```

以下语句用来统计各类锁事件发生的次数：

```
db2inst1@dpf1:~> db2 "select substr(event_type,1,20) AS EVENT_TYPE, count(*) AS
COUNT from LOCK_EVENT group by EVENT_TYPE"
```

EVENT_TYPE	COUNT
DEADLOCK	3
LOCKTIMEOUT	15

2 record(s) selected.

以下语句用来统计当天各类锁事件发生的次数：

```
db2inst1@dpf1:~> db2 "select substr(event_type,1,20) AS EVENT_TYPE, count(*) AS
COUNT from LOCK_EVENT where date(event_timestamp)=(current date) group by
event_type"
```

EVENT_TYPE	COUNT
DEADLOCK	2
LOCKTIMEOUT	12

2 record(s) selected.;

对未格式化表（UE 表）的维护，有如下建议：

- 创建独立的表空间，考虑到内联 LOB 的高效率，建议创建独立的 `pagesize` 为 32 页的表空间。
- 当锁事件发生频繁的时候，会导致 UE 表增长很快，建议定期删除不需要的数据。注意：在删除以前需要首先通过 `set ... state=0` 将事件监控器先关闭，否则会出现锁等导致数据无法删除。
- 当不需要锁事件监控器时，可以通过 `drop` 命令删除。但删除事件监控器并不会删除 UE 表，UE 表必须通过手工删除。如果没有删除 UE 表，以后再创建事件监控器时，不能再使用同样的 UE 表。

11.5 锁和并发调优



锁往往是一种症状的体现，并非问题的根源。当出现锁问题的时候，我们的基本思路是尽可能减少锁，这样就可以减少锁冲突、锁等待和锁超时的几率。那么，从应用层面上和数据库层面有哪些建议呢？

- 在应用程序设计时尽可能采用最弱的隔离级别，隔离级别从弱到强依次为：UR→CS→RS→RR。隔离级别越低，锁的范围和持久性就越低，并发性就越高。如果不能从整个应用改变隔离级别，可考虑在 SQL 语句级别通过 WITH 设置隔离级别，如 `select ... from ... with UR` 表示通过 UR 隔离级别查询数据。
- 在应用程序中尽量避免长事务，尽快提交事务，释放锁资源。
- 数据库物理设计优化，比如针对某些 SQL 语句在表上创建合适的索引。需要搜索的行数越少，需要的行锁就越少。
- SQL 语句调优，SQL 语句执行得越快，潜在的占有锁的时间就越短。
- 3 个锁注册变量：DB2_EVALUNCOMMITTED、DB2_SKIPINSERTED 和 DB2_SKIPDELETED。在应用允许的范围内，这 3 个参数可以减少锁的行数，大大提高并发性能。
 - DB2_EVALUNCOMMITTED: 验证未提交的数据。在 CS 或 RS 隔离级别，默认情况下，在决定某一行数据是否满足查询条件前，DB2 会锁住扫描的行。如果这一行上面已经有锁，当扫描时就要发生锁等待。DB2_EVALUNCOMMITTED 的目的就是提前验证未提交的数据，而尽量避免锁等。举个简单的例子，A 事务将 t1 表的 col1 列由 20 改为 5，未提交，这时 B 应用根据 col1 过滤条件查询 `select * from t1 where col1=10`，缺省情况下，B 应用将等待 A 事务提交，并验证提交后的值；而当这个注册变量设置后，DB2 会先验证未提交的值是否满足查询条件，如果不满足，则不加锁。如果要查询的值等于未提交的值，则仍需要等待。
 - DB2_SKIPINSERTED: 跳过插入行。在 CS 或 RS 隔离级别，在默认情况下，如果新插入的行没有提交，那么另外一个应用程序扫描表时会处于锁等状态，等待新行插入或回滚。很多环境下，并不需要等待新插入的数据提交，这时可通过 DB2_SKIPINSERTED 设置，忽略未提交的新插入数据。
 - DB2_SKIPDELETED: 跳过删除行。在 CS 或 RS 隔离级别，缺省情况下，如果新删除的行没有提交，那么另外一个应用程序扫描表时会处于锁等状态，等待删除行提交或回滚。使用该选项，应用程序不会发生锁等，而是假定数据没有删除。

需要注意的是，DB2_SKIPINSERTED 和 DB2_SKIPDELETED 这两个变量实际上某种形式的脏读，系统上线后，DBA 更改时一定要慎重。建议在开发阶段就与业务部门确认是否可用，否则上线后再更改，需要对应用进行全面测试。

这几个参数的设置方法如下，需重启实例才生效：

```
db2inst1@dpf1:~> db2set DB2_EVALUNCOMMITTED=ON
```

```
db2inst1@dpf1:~> db2set DB2_SKIPINSERTED=ON
db2inst1@dpf1:~> db2set DB2_SKIPDELETED =ON
```

11.6 Currently Committed 机制

前面我们提到了 4 种隔离级别，分别是 UR、CS、RS 和 RR，其中 RR 和 RS 由于锁的粒度较大，对并发的影响比较大，在实际生产中使用并不多，UR 会造成脏读，最多用于 SQL 语句级别，而 CS 作为默认的隔离级别使用比较多。CS，游标稳定性隔离级别，顾名思义，就是在游标所在的行加锁，读完该行即释放锁，如果某个事务在查询数据行时，有另外一个事务正在进行增、删、改操作，那么读取操作就需等待，直到增、删、改操作提交并释放锁。这种锁机制会对系统的并发性造成影响，也是被 Oracle 爱好者诟病的地方之一。

DB2 9.7 引入的当前已落实（Currently Committed, CC）机制从根本上解决了该问题。CC 是 CS 隔离级别的一种新实现，目的是防止写操作阻碍读操作（write blocks read），减少锁等而提高并发性。在 CC 机制下，一个事务做 update 和 delete 操作，但未提交，另一个事务读取该行时会从日志中获取已落实版本的数据，即 update 和 delete 之前的数据，而无须等待；一个事务做 insert 操作，但未提交，另一个事务读取时会忽略新插入数据，无须等待。

那么，这种机制在内部是如何实现的呢？答案是锁管理器和日志。我们知道，DB2 通过日志和锁来保证事务一致性，比如一个 update 操作，DB2 会将更改前和更改后的数据写入日志，同时在对应行上加。当启用了 CC 机制，DB2 会在每个行锁上增加一个标识，这个标识是以下几个值的其中之一。

- No information: 表示记录已经加锁。
- Uncommitted insert identifier: 表示这行是新插入的行，还没有提交。
- Log information: 表示这行没有提交，包含了当前已落实数据的日志 LSN。

当另外一个事务读取数据时会根据这个标识选择相应的处理方法，如果是 No information，则按传统的 CS 隔离级别机制，在当前行加读锁；如果是 Uncommitted insert identifier，则表明该行是新插入的行，还没有提交，读取时会忽略该行；如果 Log information，则 DB2 会根据日志信息中的 LSN（日志序列号）从日志文件或日志缓冲池中获取已经提交的数据，即更改之前的数据（或叫 before image）。

“当前已落实”从日志中获取已落实版本的数据，DB2 首先从日志缓冲区中查找数据，当更新事务仍处于活动状态时，已落实版本数据会处于日志缓冲区或磁盘上的活动日志文件中。由于 locklist 包含日志信息，DB2 不需要对所有日志文件进行搜索来查找已落实版本数据，而是直接访问合适的日志文件。当 DB2 无法在活动日志文件中找到相关记录时，DB2 将切换到“当前已落实”不启用的状态，即读操作会等待写操作落实。需要注意的是，启用“当前已落实”特性需要更多的日志表空间。

当前已落实（CC）机制是通过 CUR_COMMIT 数据库参数来配置的，在默认情况下，这个参数是开启的。如果从低版本升级到 9.7，该参数是关闭的，更改该参数值需要断开所有连接才会生效：

```
db2inst1@dpf1:~> db2 get db cfg for sample
...
Currently Committed                (CUR_COMMIT) = ON
```

以下我们通过一个例子来演示未启用和启用了当前已落实机制下数据访问情况。

1. 未启用“当前已落实”

打开两个窗口，模拟两个事务。在第一个窗口中先将当前已落实参数值关闭，然后创建一张表 t1，插入几行数据：

```
db2inst1@dpf1:~> db2 update db cfg for sample using CUR_COMMIT OFF
DB20000I The UPDATE DATABASE CONFIGURATION command completed successfully.
SQL1363W One or more of the parameters submitted for immediate modification
were not changed dynamically. For these configuration parameters, all applications
must disconnect from this database before the changes become effective.

db2inst1@dpf1:~> db2 force applications all
DB20000I The FORCE APPLICATION command completed successfully.
DB21024I This command is asynchronous and may not be effective immediately.

db2inst1@dpf1:~> db2 connect to sample

Database Connection Information

Database server          = DB2/LINUX 9.7.1
SQL authorization ID     = DB2INST1
Local database alias     = SAMPLE

db2inst1@dpf1:~> db2 "create table t1 ( id int, name char(10) )"
DB20000I The SQL command completed successfully.
db2inst1@dpf1:~> db2 "insert into t1 values(1,'aaa'),(2,'bbb'),(3,'ccc') "
DB20000I The SQL command completed successfully.
db2inst1@dpf1:~> db2 commit
DB20000I The SQL command completed successfully.
```

然后继续在第一个窗口中将第 2 行数据更改为 new，但不要提交：

```
db2inst1@dpf1:~> db2 +c "update t1 set name='new' where id=2"
DB20000I The SQL command completed successfully.
```

这时在第二个窗口中检索 id=2 的数据，出现锁超时：

```
db2inst1@dpf1:~> db2 "select * from t1 where id=2"
SQL0911N The current transaction has been rolled back because of a deadlock
or timeout. Reason code "68". SQLSTATE=40001
```

2. 启用“当前已落实”

打开两个窗口，模拟两个事务。在第一个窗口中先将当前已落实参数值打开：

```
db2inst1@dpf1:~> db2 update db cfg for sample using CUR_COMMIT ON
DB20000I The UPDATE DATABASE CONFIGURATION command completed successfully.
SQL1363W One or more of the parameters submitted for immediate modification
were not changed dynamically. For these configuration parameters, all applications
must disconnect from this database before the changes become effective.

db2inst1@dpf1:~> db2 force applications all
DB20000I The FORCE APPLICATION command completed successfully.
DB21024I This command is asynchronous and may not be effective immediately.

db2inst1@dpf1:~> db2 connect to sample
Database Connection Information

Database server          = DB2/LINUX 9.7.1
SQL authorization ID    = DB2INST1
Local database alias    = SAMPLE

db2inst1@dpf1:~> db2 "select * from t1"

ID          NAME
-----
      1 aaa
      2 bbb
      3 ccc
3 record(s) selected.
```

继续在第一个窗口中将第 2 行数据由 bbb 更改为 new，但不要提交：

```
db2inst1@dpf1:~> db2 +c "update t1 set name='new' where id=2"
DB20000I The SQL command completed successfully.
```

然后在第二个窗口中进行查询，以下是查询结果，可以看到，id=2 返回的结果是 bbb，即已落实的数据，或叫更新前的数据：

```
db2inst1@dpf1:~> db2 "select * from t1 where id=2"

ID          NAME
-----
      2 bbb

1 record(s) selected.
```

11.7 小结



DB2 中的锁曾经困扰了无数 DBA 和开发人员,相信每个初学者都有着不堪回首的痛苦学习经历。特别是早期版本,锁等待、死锁、锁超时和锁升级等各种现象出现时,由于 DB2 提供的工具和方法有限,使得这些问题的诊断和分析变得非常困难,同时由于 DB2 的锁机制会影响并发性能,这些都为大家所诟病。随着 DB2 版本的演变,提供的锁监控和分析工具越来越多,在性能方面也有一些改进,特别是 9.7 版本引入的当前已落实特性可以大大提高并发性能。

11.8 判断题



(1) DB2 支持 4 种隔离级别,分别为 UR、CS、RS、RR。

T: 正确

F: 错误

(2) 不同应用程序可以同时为同一行加共享锁。

T: 正确

F: 错误

(3) 用户可以通过调整 LOCKTIMEOUT 参数,控制应用程序等待锁的最长时间。

T: 正确

F: 错误

(4) 用户无法使用任何方法捕获锁超时事件。

T: 正确

F: 错误

(5) 锁升级和锁转换是一回事。

T: 正确

F: 错误

第四部分 DB2 监控和调优



DB2 进程/线程模型

不论是 DB2，还是其他任何数据库产品，从操作系统的角度看无疑只是一组进程和线程的结合，与其他应用程序没有任何区别。因此，想要从底层理解数据库的工作原理，一定要在操作系统层面上理解进程和线程的作用，以及数据库如何使用这些进程和线程，达到高效数据访问的目的。

作为一个单纯维护系统的 DBA，可能并不需要深入了解这些知识，就可以完成简单的数据库维护工作。但是，当系统发生问题或者需要进行调优时，理解数据库在操作系统中的运行原理至关重要。

本章主要讨论进程与线程的原理，以及 DB2 主要使用到的一些进程和线程，内容安排如下：

- 概述。
- 从操作系统看进程和线程。
- DB2 8/9.1 进程模型。
- DB2 9.5/9.7 线程模型。

12.1 提要



从这一章开始，相比起之前所介绍的运维流程，我们准备探讨一些相对高级的内容。作为一名合格的 DBA，仅仅掌握常规的运维操作还远远不够，当数据库出现故障；或者当用户抱怨应用程序性能下降，而其他部门一股脑地将问题归结于数据库的时候，就需要我们勇敢地站出来，在最短的时间内找到问题，或者用强有力的证据证明问题并非出在数据库端。

怎样才能做到这点？首先需要深入理解数据库的工作机制，即数据库架构。数据库架构可以从不同的角度进行描述。一般来说，有两种描述架构的方法最为常见：一种叫做进程/线程模型；而另一种叫做内存模型。

顾名思义，进程/线程模型是从进程/线程的角度描述一个 DB2 系统是如何工作的；而内存模型则是从内存的角度，来分析数据库如何管理存储在内存中的数据。想要真正的理解数据库的架构，必须从进程/线程与内存两个角度去看待 DB2，仅仅知道其中任何一种都只是管中窥豹。

12.2 从操作系统看进程和线程

在学习 DB2 的进程/线程模型前，首先需要了解什么是进程和线程。对于那些有着深厚操作系统管理经验的用户来说，可以直接跳过这一小节进入下一部分。对于那些刚刚接触 DB2 与操作系统不久的用户来说，笔者希望大家花费几分钟，用本小节提供的信息巩固一下学校中所学到的基础知识。

说起进程和线程，就要从计算机的历史开始谈起。

在很久很久以前……当计算机还只能被穿着白大褂的科研人员在实验室中研究的时候，那个年代的系统都只能在同一时间处理一个任务。科学家们通过卡片机将一块块打了千疮百孔的卡片插入系统，然后计算机通过读取卡片上的指令进行计算。

随着计算机的体积越来越小、处理功能越来越强大，一个新的需求就出现了：为什么不能让计算机同时处理多个任务？有的时候，一个傻瓜提出的问题十个聪明人也解答不了。这个看似简单的问题，但是对于计算机来说却不是那么简单。

计算机的核心是中央处理器(即 CPU)，一般来说，一个 CPU 只能在一个给定的时间点运行一个操作(抛开 CPU 流水线的概念)，使用一个 CPU 同时运行超过一个任务是不可能的。

不过规则就是为了用来被打破的，这种简单的规则毕竟难不倒众多聪明的科学家。人们发现，在系统处理任务的时候，实际上真正 CPU 处理所占据的时间，与整个程序完成的时间相比并不是很多。CPU 只是一个运算单元，所有的信息都保存在寄存器中，而寄存器的大小是非常有限的，因此每当寄存器里的信息不包含所需要的数据时，CPU 就要从缓存(1 级缓存、2 级缓存等硬件)与内存中读取数据。

一般来说，从缓存与内存读取数据是非常快的。但是对于不管是早期的计算机，还是近期的系统，内存尽管越来越大，但是依然不能容纳下成百上千倍爆炸的信息量。因此，大量的信息需要存储在一种可以永久保存并且廉价的介质上，这种介质就是磁盘。

当 CPU 需要的数据不在内存中的时候，系统就要从磁盘上读取数据。但是磁盘读取的速度比起 CPU 的速度，差距可能在千万倍。而在等待磁盘数据的期间，CPU 是不能处理其它任务的。

因此，如果有办法能够在 CPU 等待的时候，让 CPU 进行一些其他的计算，岂不是能够更有效地利用宝贵的计算资源？而从用户的角度看，尽管一个磁盘读写对于 CPU 讲可以说是几个世纪的概念，但是在人类的眼中那还是相当地快的。如果要是能够让 CPU 抓紧一切空隙，来运行多个任务，从用户的角度看来这些任务岂不是在同时运行？

这就引出了进程(process)的概念。这个概念就是，系统使用一些特定的结构标明不同的任务，然后让有限个数的 CPU 在不同的时段运行不同的任务，让这些任务看起来在同时发生。而在 Unix/Linux/Windows 系统中，这种标明任务的结构就叫做进程。

简单地说，操作系统中没有什么真正的魔法能够让不同的任务在一个 CPU 上同时执行。它所做的只不过是将会一个个任务切分成很小的时间段，然后在不同的任务之间跳跃（一会执行这个，一会执行那个.....）。

随着技术的发展，人们逐渐发现，进程有它的优势，但是对于越来越复杂的系统，却也有一些显著的问题。

譬如说，每个进程能够看到的内存地址空间都是独立的（这个我们在下一章内存模型中再讨论），而每个进程想要和其他进程通信或者共享数据，就要通过一些特殊的方法来实现。如果能够让不同的执行类似操作的任务运行在一个进程中，并且它们之间也能够做到看起来同时运行，岂不是更好？

不幸的是，这个问题是在 UNIX 系统已经大量使用后才提出的。当时几个巨头分别开发了自己的 UNIX 产品，产品本身也不兼容。后来，几家巨头意识到统一 UNIX 标准化开发接口的重要性，决定制定一套统一的大家都必须遵守的标准接口。

“线程”这个概念就是在这个标准里提出的，也就是我们今天所熟悉的 POSIX THREAD，简称 pthread。

线程和进程很类似，在操作系统中都是代表着一个单独的任务。但与进程不同，线程从逻辑上讲是运行在一个进程之下的。同一个进程可以包含若干个线程，就好像一个工作狂可以同时接电话，写 email，外加开两个窗口写程序一样，拥有多个线程的进程就可以被看做是一个同时处理多个任务的单元。

从操作系统中，如何查找哪些进程在运行呢？在 UNIX/LINUX 系统，可以使用 ps -elf 命令，而在 Windows 里面，任务管理器就是您最好的朋友。

```
red 302 % ps -elf
F S UID      PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY      TIME CMD
4 S root      1    0  0  76   0 -   668 ?          2010 ?      00:00:02 init
1 S root      2    1  0 -40   - -     0 migrat 2010 ?      00:00:01 [migr
1 S root      3    1  0  94  19 -     0 ksofti 2010 ?      00:00:00 [ksof
1 S root      4    1  0 -40   - -     0 migrat 2010 ?      00:00:00 [migr
1 S root      5    1  0  94  19 -     0 ksofti 2010 ?      00:00:00 [ksof
1 S root      6    1  0 -40   - -     0 migrat 2010 ?      00:00:01 [migr
1 S root      7    1  0  94  19 -     0 ksofti 2010 ?      00:00:00 [ksof
```

```

1 S root      8      1 0 -40  - -      0 migrat 2010 ?      00:00:00 [migr
1 S root      9      1 0  94 19 -      0 ksofti 2010 ?      00:00:00 [ksof
5 S root     10      1 0  65 -10 -      0 worker 2010 ?      00:00:00 [even
5 S root     11      1 0  65 -10 -      0 worker 2010 ?      00:00:00 [even
5 S root     12      1 0  65 -10 -      0 worker 2010 ?      00:00:00 [even
5 S root     13      1 0  65 -10 -      0 worker 2010 ?      00:00:00 [even
1 S root     14      1 0  65 -10 -      0 worker 2010 ?      00:00:00 [khe1
1 S root     15      1 0  65 -10 -      0 worker 2010 ?      00:00:00 [kthr
1 S root     16     15 0  75 -10 -      0 worker 2010 ?      00:00:00 [kacp
1 S root     38     15 0  65 -10 -      0 worker 2010 ?      00:00:00 [kblo
1 S root     39     15 0  65 -10 -      0 worker 2010 ?      00:00:00 [kblo
1 S root     40     15 0  65 -10 -      0 worker 2010 ?      00:00:00 [kblo
1 S root     41     15 0  65 -10 -      0 worker 2010 ?      00:00:00 [kblo
1 S root     42      1 0  75  0 -      0 hub_th 2010 ?      00:00:00 [khub
1 S root     59     15 0  80  0 -      0 pdflus 2010 ?      00:00:00 [pdf1
1 S root     60     15 0  75  0 -      0 pdflus 2010 ?      00:00:03 [pdf1
1 S root     61      1 0  75  0 -      0 kswapd 2010 ?      00:00:03 [kswa
1 S root     62     15 0  71 -10 -      0 worker 2010 ?      00:00:00 [aio/
1 S root     63     15 0  71 -10 -      0 worker 2010 ?      00:00:00 [aio/
1 S root     64     15 0  71 -10 -      0 worker 2010 ?      00:00:00 [aio/

```

ps -elf 命令能够列出系统中全部的进程，而通过 grep 命令抓取用户名，则可以看到特定用户拥有的全部进程：

```

red 303 % ps -elf | grep -i "cse93000"
4 S root      14922 3711 0 96 19 - 2170 ?   08:47      00:00:00 sshd: cse93000 [priv]
5 S cse93000 14929 14922 0 95 19 - 2171  08:47 ?   00:00:00 sshd: cse93000@pts/0
0 S cse93000 14930 14929 0 95 19 - 1450 rt_sig 08:47 pts/0 00:00:00 -tcsh
0 R cse93000 15098 14930 0 96 19 - 580 -      08:49 pts/0 00:00:00 ps -elf
0 S cse93000 15099 14930 0 94 19 - 397 pipe_w 08:49 pts/0 00:00:00 grep -i cse93000

```

对于线程来说，没有什么非常有效的命令列出系统中存在的所有线程。如果一定要查的话，可尝试使用某些调试工具(如 gdb、dbx 等)加载某一进程，列出其中包含的所有线程。

譬如我们建立一个多线程的程序叫做 a.out，然后通过 gdb（在 AIX 中为 dbx）可以看到其正在运行的所有线程：

```

red 326 % gdb a.out 15640
GNU gdb (GDB) 7.1
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /cs/home/cse93000/temp/a.out...(no debugging symbols
found)...done.
Attaching to program: /cs/home/cse93000/temp/a.out, process 15640

```

```

Reading symbols from /lib/tls/libpthread.so.0...(no debugging symbols found)...done.
[Thread debugging using libthread_db enabled]
[New Thread 0xb7e9eba0 (LWP 15641)]
Loaded symbols for /lib/tls/libpthread.so.0
Reading symbols from /lib/tls/libc.so.6...(no debugging symbols found)...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...(no debugging symbols found)...done.
Loaded symbols for /lib/ld-linux.so.2
0xb7fe87a2 in _dl_sysinfo_int80 () from /lib/ld-linux.so.2
(gdb) where
#0 0xb7fe87a2 in _dl_sysinfo_int80 () from /lib/ld-linux.so.2
#1 0xb7f5a21b in read () from /lib/tls/libc.so.6
#2 0xb7efe518 in _IO_file_read_internal () from /lib/tls/libc.so.6
#3 0xb7efd29e in _IO_new_file_underflow () from /lib/tls/libc.so.6
#4 0xb7eff8bb in _IO_default_uflow_internal () from /lib/tls/libc.so.6
#5 0xb7eff6ad in __uflow () from /lib/tls/libc.so.6
#6 0xb7efa797 in getchar () from /lib/tls/libc.so.6
#7 0x08048f9a in main ()
(gdb) thread
[Current thread is 1 (Thread 0xb7e9f6c0 (LWP 15640))]
(gdb) info thread
  2 Thread 0xb7e9eba0 (LWP 15641) 0xb7f69198 in clone () from /lib/tls/libc.so.6
*  1 Thread 0xb7e9f6c0 (LWP 15640) 0xb7fe87a2 in _dl_sysinfo_int80 () from /lib/ld-linux.so.2
(gdb) detach
Detaching from program: /cs/home/cse93000/temp/a.out, process 15640

```

这个输出中显示了 3 个信息，其中 `where` 命令代表当前正在运行的线程的函数调用堆栈，而 `thread` 与 `info thread` 显示当前正在执行的线程和进程中所有的线程。

通过这些信息，用户可以捕捉到一个进程中所使用到的所有线程。不过，对于 DB2 来说，我们提供了一些更为简单的方法来监测 DB2 进程中的线程信息，将在下面的部分提到。

12.3 DB2 V8/V9.1 进程模型

在 V8 与 9.1 版本，DB2 使用的是进程模型，也就是 DB2 执行的每个任务(或作业)由操作系统的进程执行，同理，如果所有的任务都运行在同一个进程的不同线程里面，我们把这种架构叫做线程模型，这也是 V9.5 与 V9.7 中的改变。

DB2 是怎么用到这些进程的呢？如图 12.1 所示。

对于每个数据库连接，DB2 在逻辑上都分配一个标识，即应用程序 ID。对于这些连接到数据库的应用程序，DB2 会分配若干个代理进程(Agent Process)为他们服务。

针对每个应用程序的 SQL 请求，这些代理进程都会与 DB2 的包缓存和优化器进行通讯（两

者将在后文介绍)，然后把设定好的执行计划拿过来交给 codegen 和 runtime 去运行，然后把返回的结果集交给应用程序。

用客户与客服代表来比喻，如果把一个连接看做一个客户的话，那么这若干个代理进程就是公司的客服代表。当客户向客服发出一个调档请求，然后客服代表跑去找部门的经理签字，然后拿着经理的批示跑去找档案室提取档案，然后拿回来给客户。

好，从一英里外远观这个 DB2 进程模型，好像并没有什么复杂的。让我们来看一下 DB2 的简化版进程模型，然后讨论几个常见的进程。

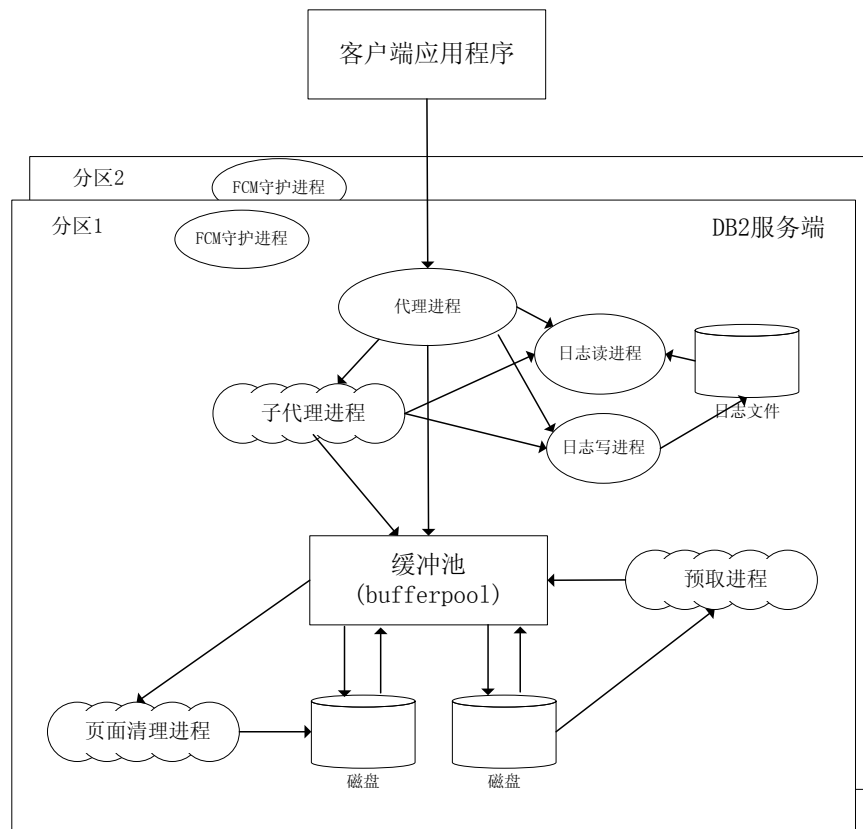


图 12.1 DB2 进程模型

12.3.1 代理进程 ■ ■ ■

代理进程的目的在于代表应用程序对数据进行处理。代理进程的数量就是该实例共可以分配多少个代理来服务用户连接。

在 V8/9.1 版本，与代理进程相关的参数主要有以下几个：（在 9.5 与 9.7 版本中有略微调整，不过总体思路不变）：

Max number of existing agents	(MAXAGENTS) = 400
-------------------------------	-------------------

Agent pool size	(NUM_POOLAGENTS) = 200(calculated)
Initial number of agents in pool	(NUM_INITAGENTS) = 0
Max number of coordinating agents	(MAX_COORDAGENTS) = MAXAGENTS
Max no. of concurrent coordinating agents	(MAXCAGENTS) = MAX_COORDAGENTS
Max number of client connections	(MAX_CONNECTIONS) = MAX_COORDAGENTS

MAXAGENTS 表示实例中该分区的最大代理数量。在 V8/V9.1 版本，可以根据系统中的并发数、业务类型、CPU 数量等手工调整 MAXAGENTS 数量。在 V9.5 和 V9.7 版本已经取消了该参数，改为完全自动。

一般来说，当不使用分区内并行的时候，MAX_COORDAGENTS = MAX_CONNECTIONS = MAXAGENTS。如果 MAX_CONNECTIONS > MAXAGENTS，就说明开启了 CONCENTRATOR 功能，表示一个代理可能需要为多个连接服务。除非特别需要，笔者不建议开启这个功能。

还有一个参数是代理池(agent pool)，用 NUM_POOLAGENTS 参数表示，就是最大可以被缓存的代理进程数量。不论是线程还是进程，每次创建时都有不小的开销。采用代理进程池可以保留一部分代理进程(线程)，这样当下一个应用程序需要处理时，就可以直接从代理池中获取，而无须新建，提升了系统的性能。

一般情况下，NUM_POOLAGENTS 参数不需修改，采用默认值即可。如果并发数很多，可以根据实例快照和数据库快照监控代理的使用情况，并进行逐步调整。

12.3.2 分区内并行 ■ ■ ■

对于比较复杂的查询，如果能够将其分解成很多块，让不同的代理进程并发执行，那么对查询的性能提升是有帮助的。为支持该特性，DB2 提供了分区内并行机制，通过 INTRA_PARALLEL 实例参数进行设置：

```
$ db2 get dbm cfg | grep -i "intra_parallel"
Enable intra-partition parallelism (INTRA_PARALLEL) = YES
```

缺省情况下，该参数是关闭的。可以通过以下命令将其打开。

```
$ db2 update dbm cfg using intra_parallel yes
```

继续用客服代表为例，如果这个部门经理觉得这个调档请求很复杂，比如涉及几百个人员的名单，那么可能他就会让一些其他的客服代表帮忙，而原先的客服代表则是坐在客户面前归纳其他客服代表返回的资料。这个就是分区内并行的原理。在 DB2 中，这个直接面对应用程序的代理叫做协调代理(coordinator agent)，而其他为其服务的代理则叫做子代理(subagent)。通过 ps 命令，如果当前系统有类似的业务用到了子代理，我们可以在 ps -elf 结果中看到类似下面的内容：

```
40001 A db2inst1 1941594 1814638 0 60 20 199db1510 4060 f100010041648298
17:05:29 - 0:00 db2agent (SAMPLE) 0
40001 A db2inst1 1994898 1896654 0 60 20 3b7cf2510 3200 17:10:26 -
1:09 db2agntp (SAMPLE) 0
```

其中显示了两个代理，一个叫做 `db2agent`，这是协调代理的名字；而另一个叫做 `db2agntp`，即子代理。

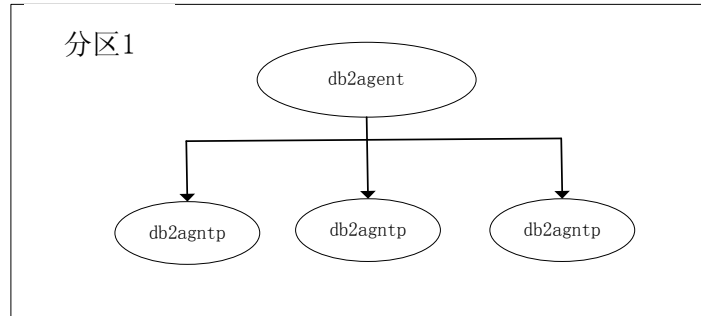


图 12.2 协调代理和子代理

那么分区内并行在什么情况下使用比较有效呢？以下是可以考虑的一些场景：

- 应用中的重要查询比较复杂，且需要读取大表数据；
- 每个分区有多颗 CPU；
- 系统中空闲 CPU 较多。

当满足以上条件时，可以考虑使用分区内并行，将一个大的任务分解成几个进程/线程同时执行，这样可以大幅度增加单位时间内的数据吞吐量。

但是如果主要查询的时间都很短，由于分区间的通讯需要耗费相当可观的时间，也增加了访问计划的复杂度，可能会造成优化器选择次优的计划。因此，在某些情况下，打开分区内并行并不一定会提升效率，甚至造成负面影响。特别是系统的 CPU 已经处于非常繁忙的状态时，增加并行处理会增加系统的负荷，造成 CPU 资源竞争。

一般来说，在多分区数据库（DPF）环境下使用分区内并行要特别小心。因每个分区对应的 CPU 数量都不会很多（一般 2~4 颗），打开分区内并行可能会加重每个分区的 CPU 负担，对系统产生负面影响。

根据笔者的经验，不建议在生产环境中使用分区内并行。如果一定要使用，请务必做好充分的性能测试。

12.3.3 分区间并行（DPF） ■ ■ ■

当系统更加庞大，包含更多数据的时候，如果我们能够把很多的数据平均分布到不同的机器中，让所有的机器一起执行，这样会对性能有很大的提高。这就是分区间并行的由来。

在 DB2 中，一个分区并不是单只一台物理的机器，一台机器可以包含多个分区。每一个分区，从某种意义上说可以看作是一个单独的数据库，他们有自己的数据和日志，有独立的内存和 CPU 资源，有自己的代理进程。但系统表是共享的。

当某个客户端程序连接数据库的时候，需要连接到其中一个分区，该分区叫做协调分区（Coordinator Partition），相对应的代理则叫做协调代理（Coordinator Agent）进程。协调代理将任务分发给其它分区，各分区的子代理进程并行处理，彼此通信，处理结束后将结果返回到协调代理，然后由协调代理汇总结果并返回给客户端。图 12.3 描述了这个结构。

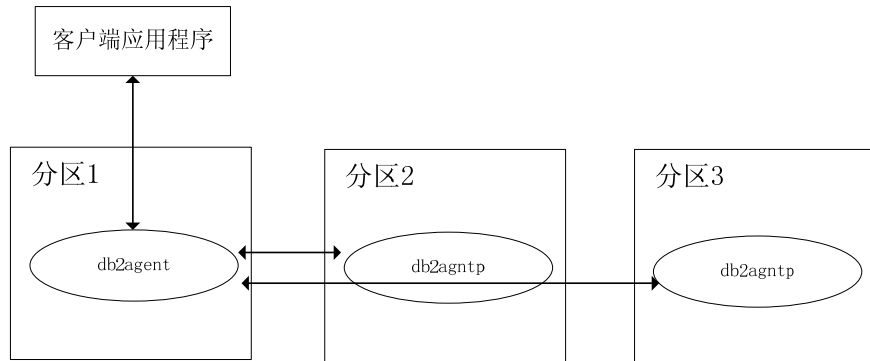


图 12.3 分区间并行

目前在关系数据库领域，在处理海量数据方面，DB2 多分区数据库拥有绝对优势。

12.3.4 预取进程（prefetcher） ■ ■ ■

前面我们提到过，当代理进程得到了访问计划以后，就会去 runtime 执行该计划得到结果。那么“执行一个查询”，这句话应该怎么理解？

我们知道，一般的查询不外乎就是做表之间的关联、数据的读取、结果的排序。从本质上说，不管多么复杂的查询，也不会脱离这几种操作。这些操作都有一个最基本的要求，就是必须要有数据可以操作。

那么问题就来了，我们怎样从磁盘上读到数据？每次需要的时候跑去读，还是用什么其他的方法？

答案就是都有。DB2 在访问数据的时候，既有同步访问也有异步访问。所谓同步访问比较容易理解，当客户说我要某某人的档案，客服代表立刻跑去档案室调出该人的档案，然后接着处理下一个客户档案。不过这种操作有一个问题，就是效率太低。

如果能有什么方法，把这个客户可能需要的所有人的档案都搬过来，然后根据客户的名单一个个检索，就不用一趟一趟地跑了。这就是预取进程（prefetch）的作用。它就相当于一个文档运送员，负责把档案室中的文档一车一车地运到办公室，所有的客服代表都可以在里面翻阅，比每次去档案室快多了。

预取进程采用异步 I/O，将磁盘上大段连续的数据读取至内存。当代理进程处理一条 SQL 时，认为预取效率更高时，就会通知预取进程：“去帮我把 xxx 表中从 A 页到 B 页之间的数据拿过来”。然后，预取进程在后台乖乖地将数据从磁盘搬到内存里，代理就可以轻松地从内存里

面拿到它想要的数据了。

图 12.4 描述了预取过程。

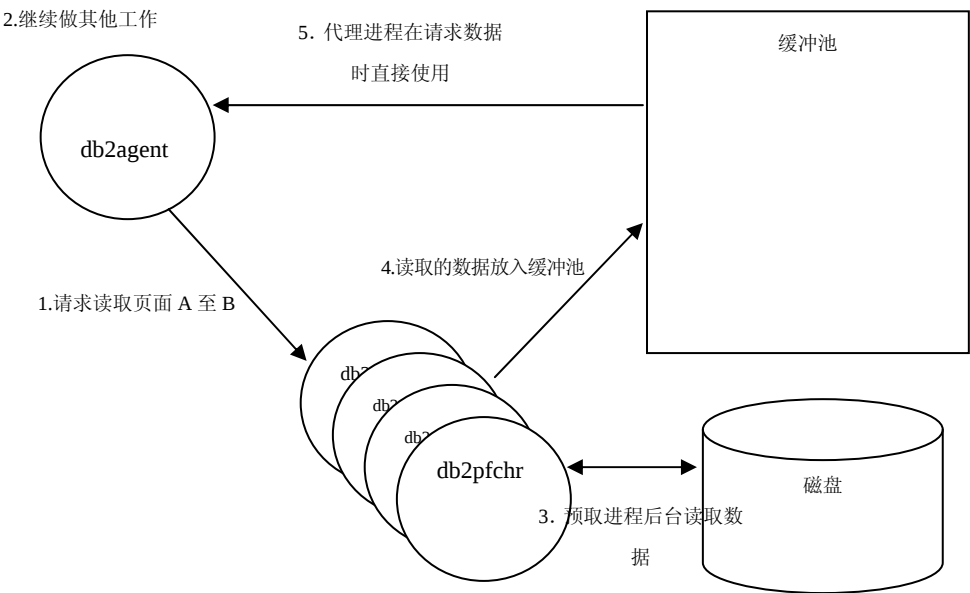


图 12.4 预取进程原理

在 DB2 进程中，预取进程的名字是 db2pfchr:

```
40001 A db2inst1 717036 1896654      0  60  20 1ba232510  1464 f1000002e0108a00
17:07:56      - 0:16 db2pfchr 0
40001 A db2inst1 737494 1896654      0  60  20 fea9b510   1464 f1000002e0108c00
17:07:56      - 0:26 db2pfchr 0
40001 A db2inst1 770160 1896654      0  60  20 32c4e0510  1464 f1000002f010a400
17:07:56      - 0:43 db2pfchr 0
```

预取进程的数量是通过 NUM_IOSERVERS 数据库参数设置的:

```
db2 get db cfg for sample | grep -i "ioservers"
Number of I/O servers              (NUM_IOSERVERS) = AUTOMATIC
```

一般情况下,将该参数设置为 Automatic 可以满足大部分应用的需求。预取一般发生在 OLAP 仓库类系统,因为该类系统通常需要处理的数据量较大,采用预取会提升系统的性能。对于 OLTP 系统,由于读取的数据相对较小,这时 DB2 一般不会进行预取。如果发现有大量预取发生的情况,根据经验,很大的概率是发生了全表扫描,这时需要对数据库进行性能调优。

12.3.5 页面清理进程 (Page Cleaner)

与预取进程相反, 页面清理进程负责将修改过的记录从内存写入磁盘。按照常规, 每次更改数据时直接写入磁盘是最简单的想法。但这需要每一次操作时都与 I/O 交互, 会造成极大的性能瓶颈。因此, 大部分关系型数据库, 都采用异步数据写入的机制。

DB2 异步数据写入的原理是, 当代理进程更改数据后, 所更改的数据依然保留在内存中, 而代理进程返回应用程序进行下一个操作。我们将包含了被修改的, 还没有写入磁盘的数据页称之为“脏页 (dirty page)”。

页面清理进程采用一定的机制将这些脏页写回磁盘上, 这个操作叫做页面清理 (page cleaning)。

以下两种情况会触发页面清理:

- 缓冲区中的脏页比例超过缓冲区总大小的 `CHNGPGS_THRESH` 比例
- 最早的还没有被写入磁盘的脏页与当前的所执行的 LSN 间隔超过了 `SOFTMAX` 所指定的大小。关于 LSN 请参见日志部分的相关章节。

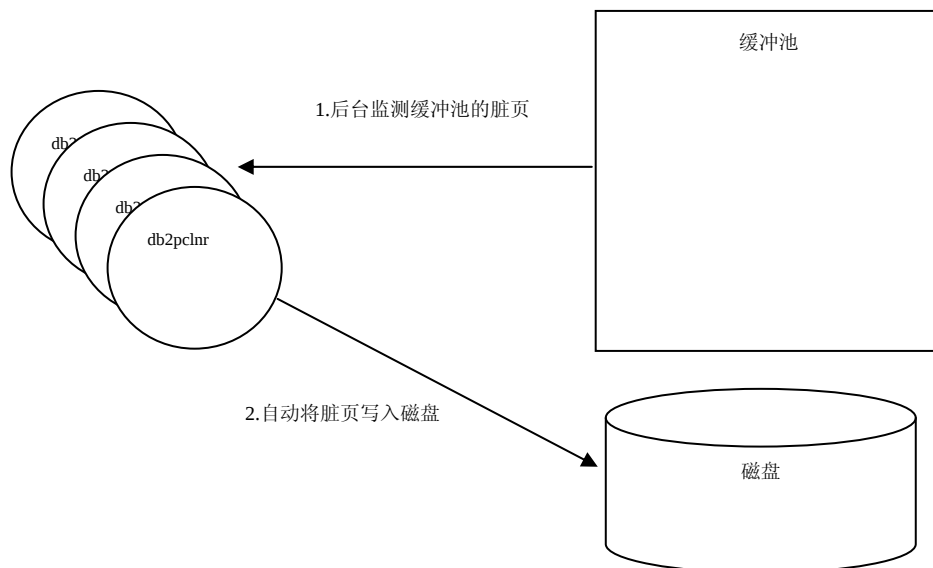


图 12.5 页面清理过程

继续客服代表的举例。当客服代表屋里的大桌子被堆得半满的时候, 另一个清理人员就跑过来, 将桌子上的一些被修改后的记录搬回档案室并更新原档案记录。当然, 那些没有被修改过的大桌子上的档案, 因为档案室无论如何还有一份拷贝, 所以当桌子放不下的时候直接销毁掉也不可惜。这样我们就可以确保文档运送员一车一车的新文件, 不至于很快将这个桌子占满了。

页面清理进程的名字是 db2pclnr:

40001 A db2inst1	929982	1896654	0	60	20	321ce1510	1300	17:07:55	-
0:00 db2pclnr 0									
40001 A db2inst1	962786	1896654	0	60	20	3236e1510	1300	17:07:55	-
0:00 db2pclnr 0									
40001 A db2inst1	1065030	1896654	0	60	20	3c4c7c510	1300	17:07:55	-
0:00 db2pclnr 0									
40001 A db2inst1	1544322	1896654	0	60	20	348eed510	1940	17:19:42	-
0:00 db2pclnr 0									
40001 A db2inst1	1597462	1896654	0	60	20	36a668510	1940	17:19:42	-
0:00 db2pclnr 0									
40001 A db2inst1	2138300	1896654	0	60	20	3f94fd510	1940	17:19:42	-
0:00 db2pclnr 0									

DB2 使用 NUM_IOSERVERS 数据库参数控制页面清理的个数。

Number of I/O servers	(NUM_IOSERVERS) = AUTOMATIC
-----------------------	-----------------------------

一般情况下, 将该参数设置为自动值(Automatic)可以满足大部分应用的需求。如果手工设置, 建议不要超过系统 CPU 的数量。这样在最坏情况下, 即使系统中某一时刻所有的页清除进程都开始运行, 也不会造成过大的 CPU 负担。

12.3.6 其他进程 ■ ■ ■

除了上面提过的几个进程(代理进程、预取进程与页面清除进程)外, DB2 还有很多其他的辅助进程。这些进程对于数据库的正常运行起到很重要的作用, 但是在日常的运维中并不用特别关心, 这里我们介绍几种主要常见的进程(总共有超过 70 种进程)。

1. db2sysc: 实例主进程

这是 DB2 中最重要的进程, 基本任何书籍都少不了介绍这个进程。这个进程是实例启动过程中的第二个进程(第一个是 db2wdog), 也是实例中所有进程的父进程与祖父进程。实例中的所有代理进程、预取进程, 还有以后提到的一些怪异的进程, 都是由这个进程派生出来的。

这个进程是整个实例的主体, 但是它本身并不真正做任何工作, 只是高高在上地观察着所有进程的运行。

2. db2wdog: 监视狗进程

该进程是 DB2 实例所启动的第一个进程。

正像它的名字一样, 这个进程就好像一个监视着手下所有绵羊的牧羊犬, 如果任何一个进程出于未知原因死掉了, 为了避免整个实例发生数据损坏, 该进程就会对所有的子进程发出信号 9, 杀掉所有进程。

在 UNIX 系统中, 任何一个进程结束都会发送一个消息给其父进程。既然 db2wdog 是第一

个启动的进程，而这个进程启动 db2sysc，那么所有 db2sysc 下的数据库进程都属于 db2wdog 的子进程。因此，db2wdog 可以监视所有进程退出的信号，只要发现任何一个进程是在未期待的情况下非自然退出的，则杀死整个实例避免数据损坏。

这样，诸位读者也可以理解为，手工杀死任何实例中的进程都会导致实例崩溃。这里的实例进程是除了 db2fmp 与 db2vend 外所有 db2wdog 的子进程。

3. db2tcpcm / db2ipccm: TCP 通信管理器与 IPC 通信管理器

对于不同的连接方式，应用程序可以选择通过 TCP 连接或者 IPC 连接。

IPC 是进程间通信，一般用于本地应用程序，比如说用户手工执行“db2 connect to sample”，这时这个“db2”进程就叫做前端进程，然后它用来启动一个 db2bp 的后端进程，这个后端进程在 db2 terminate 之前一直保持在那里，用于维持一个和 DB2 引擎之间的连接。

假如数据库是本地，并且没有用 loopback connection（就是编目自己机器为远程节点，然后编目数据库），一般这种连接使用 IPC 连接。而对于远程应用程序，比如 Websphere，一般的连接都是 TCP/IP。

这两个管理器当接收到连接信息后，就会创建出相对应的 DB2 代理进程，用于服务这个连接。

4. db2loggw/db2loggr: 写入/读取日志的进程

当 DB2 需要读/写日志的时候，该请求被发送至一个单独的进程中，即 db2loggr 和 db2loggw。其中 loggw 是写日志的进程，而 loggr 是读日志的进程。

5. db2fmp/db2vend: fenced mode process

这类进程则不属于 DB2 内核进程。手工杀掉这类进程不会造成实例崩溃。FMP 与 VEND 的作用就是运行一些并非使用 DB2 内核库的用户程序，比如存储过程。

当一个程序需要调用一个库(library)的时候，首先这个库必须加载入进程，然后进程从库中得到所需要的函数所在的指针，然后进行调用。在调用的过程中，如果该库函数发生非法内存访问，则会给该进程带来信号 11 的错误，也就是 sigsegv。当发生这种错误的时候，该进程崩溃。

所以，如果用户的存储过程存在一些非法内存调用，当数据库加载该存储过程到引擎后，就会立刻杀死实例，造成破坏，这是我们坚决不能允许的。

因此，一般情况下，我们都会把实例级参数 KEEPFCENCED 设置为 YES:

Keep fenced process	(KEEPFCENCED) = YES
---------------------	---------------------

该参数就是让数据库对每一个外来的存储过程都创建一个 db2fmp 在引擎之外执行。就算 db2fmp 中的用户代码造成了该进程的死亡，DB2 的监视狗也会忽略这些来自 FMP 的错误，而不会轻易地杀死实例。

db2vend 实际上就是一种特殊的 db2fmp，它不是用来运行存储过程，而是一些类似于 TSM 系统交互的程序，保证第三方的代码不会造成 DB2 引擎的崩溃。

6. db2acd/db2hmon: health monitor 进程

该进程的主要功能是提供自动维护服务，例如自动备份，自动 reorg/runstats、health check 等功能。一般来说，db2acd/db2hmon 可以被认为是一种 fmp 进程，杀死该进程不会造成实例崩溃。

12.3.7 实例 / 数据库启动步骤 ■ ■ ■

熟悉 Oracle 的人都很了解实例/数据库启动时内部的处理流程，但 DB2 的公开资料几乎没有提及该流程，本书给大家做个简单介绍。

首先用户用 telnet（或者 ssh）连入系统，这时用户会在屏幕上看到 shell 的界面。当用户输入 db2start，然后按下回车键的时候，都发生了什么呢？

（1）首先 shell 接收到回车的 ASD II 码，然后确认了这个是命令输入的键，然后将用户所敲入的所有字符解释一遍，看看是不是 shell 的保留命令（比如 export 之类的）。

（2）当 shell 确认不是自己的保留命令之后，就会创建一个新的进程。在这个新的进程中，程序到第一个空格或者 Tab 之前的第一个字符串，然后去 PATH 环境变量下所指向的所有路径中，从前往后找。当找到其中任何一个路径包含一个可执行文件匹配该字符串的时候，就会调用 execv 系统指令执行这个可执行程序。在 execv 的传入参数中，shell 会把自己找到的字符串按照双引号、单引号与空格这种顺序拆分开，得到若干个字符串，作为参数传入 execv。

（3）对于简单的 db2start 命令，shell 在 sqllib/adm 目录下找到该程序后，直接调用 execv 并且没有参数。

（4）当 db2start 的主程序代码被装载入新的进程以后就开始寻找到其中的 main 函数执行。在执行的时候，db2start 首先判断当前的 DB2INSTANCE 得到当前实例名，然后得到实例下 sqllib 路径，紧接着打开 sqllib/db2nodes.cfg 得到分区信息。

（5）对于每一个分区，db2start 都会发起一个 rsh 或者 ssh 的远程连接（如果是单分区，则直接调用），启动一个叫做 db2star2 的进程（这里的启动进程指的就是创建新的进程，然后 execv 新的可执行程序）。

（6）在 db2star2 中，它先检查当前 DB2NODE 变量来决定自己的分区，然后创建实例共享内存，最后调用一个叫做 db2sysc 的可执行程序。

（7）db2sysc 可执行程序被调用以后，它做的第一件事将就是创建一个自己的副本，然后把自己的名字改为 db2wdog。这就是 db2wdog 的由来，而 db2sysc 不过是这个旧的 db2sysc 的一个副本。

（8）这时，db2wdog 就开始等待了，然后新的 db2sysc 进程首先创建一个 FCM 内存段（如

果开启了 DB2_FORCE_FCM_BP)，然后创建其他的进程，例如 db2tcpcom（只有当 DB2COMM=TCP 被设置时，这个进程才会启动）、db2ipccm、db2gds、db2pdbc 等进程（实际上这些进程都是来自 db2sysc 可执行程序自身，只不过针对不同用途改成了不同的名字）。

（9）这个阶段，只要没有在中途发生错误，就意味着实例已经成功启动，然后消息就会返回给 db2star2 使其结束，所有分区的 db2star2 结束后，db2start 也会返回成功的消息，这样用户就会在屏幕上看到实例被成功启动了。

（10）这一步就轮到数据库的启动了。在没有连接或者 activate 命令的时候，数据库是不会自己启动的，因此第 9 步和第 10 步之间要等待多长时间，就取决于什么时候第一个连接的请求到达 db2tcpcom 或 db2ipccm 进程。

（11）当一个 connect 命令到达通信管理器的时候，DB2 会检测这个请求的数据库是否已经存在（sqllib、sqlbdir、sqldbdir），如果这个数据库根本不存在，则直接报数据库无法找到的错误。

（12）当数据库找到后，DB2 会再次判断数据库是否已经启动。如果数据库没有启动，根据 DATABASE_MEMORY 的设置创建一个数据库共享内存段，然后从 db2gds 进程中创建相应数量的 db2pfchr、db2loggr、db2loggw 与 db2pclnr 等进程并 attach 到数据库内存段，并创建一个代理进程服务于新的连接。

（13）检查数据库的状态，如果数据库状态不一致，则需要进行崩溃恢复。如果有索引在不可用状态并且数据库参数标明启动时创建索引，则继续创建索引。

（14）到此为止，如果中途没有错误发生，则数据库启动完毕，代理进程返回成功的消息给应用程序，然后应用程序向用户汇报数据库连接完毕。

从这里，大家可以看到，所谓的数据库和实例的启动不过就是一堆内存和进程的启动。这些进程每一个都有特定的作用，有一些用来监听端口，有一些用来得到用户的 SQL 执行，有一些用来做日志，还有一些用来进行 I/O 读/写。而内存呢，有一些负责存放缓冲池、有一些用来存放日志等。

所谓一个实例的启动，就是这一系列进程和内存的成功分配。

希望通过这几步演示大家能够解释在 DB2 8 和 9.1 中数据库的进程关系是什么样的。

注意：在 Solaris 操作系统中，由于其系统的特殊性，ps 命令无法显示更改之后的进程名，因此所有的 DB2 相关进程全部显示 db2sysc，需要用 db2ptree 命令显示其进程 ID 与进程名：

```
us2003:db2y9c 2% db2ptree
3176 db2wdog
  3177 db2sysc
    3178 db2gds
      3181 db2resyn
      3182 db2srvlst
```

```

3186 db2fmtlg
3187 db2loggr
3188 db2dlock
3189 db2pfchr
3190 db2pfchr
3191 db2pclnr
3192 db2pclnr
3179 db2ipccm
3185 db2agent (idle)
3180 db2tcpcm
10975 -csh
19872 -csh
19943 db2ptree
25494 -csh

```

12.4 DB2 9.5/9.7 线程模型

在 9.5/9.7 版本，DB2 的模型由进程模型(Process Model)变为线程模型(Thread Model)。在代码层面上，这样的架构改动需要无数的修改和测试，但是用户的角度，线程模型与进程模型的区别，在任务的相互依存关系看来着实有限。

在进程模型中，我们对每一种 DB2 进程都用“进程”二字称呼，但是它们实际上都有自己的名字，这个名字叫做引擎处理单元 (Engine Dispatched Unit, EDU)。每一个从 db2sysc 中派生出来的线程或进程（包括 db2agent、db2loggr、db2pclnr 等）都可以被称之为 EDU。

在 9.5/9.7 的线程模型中，每个 EDU 的作用基本和 8/9.1 时一样。而 db2start 时的区别，主要是从 db2wdog 创建 db2sysc 之后开始的。当 db2sysc 进程被 db2wdog 创建后，它自己则不再创建新的进程，而是所有的 EDU 都以线程的方式创建。

因此，在 9.5/9.7 中，所有的引擎相关任务完全封装在 db2sysc 进程中，在 ps 命令中不会看到其他引擎相关的任务（类似 db2fmp、db2vend 的进程不属于引擎进程）。

在进程模型中，如果想要监测某一个 EDU 的运行，只需要 ps -elf 就可以。在线程模型中，所有线程都封装在 db2sysc 进程中，那么如何监测每一个线程的运行状况呢？

为解决该问题，db2pd 工具中引入了一个新的参数：-edus，该参数可以列出当前 db2sysc 进程中有都运行了什么线程，而且每一个 EDU 自从创建到现在所消耗的 CPU 时间。根据这些 CPU 时间，我们可以找到哪个线程(或任务)消费了更多的 CPU 资源，从而作进一步的性能分析。

```

$ db2pd -edus

Database Partition 0 -- Active -- Up 80 days 02:02:27

List of all EDUs for database partition 0

```

db2sysc PID: 31523014

db2wdog PID: 31916042

db2acd PID: 22741144

EDU ID	TID	Kernel TID	EDU Name	USR
SYS				
18420	18420	13959349	db2agntdp (SAMPLE) 0	47.819685 0.930720
18163	18163	39125037	db2agntdp (SAMPLE) 0	45.746742 0.899355
17906	17906	58523857	db2agntdp (SAMPLE) 0	44.433269 0.862771
8419	8419	25886881	db2pfchr (SAMPLE) 0	6.404657 9.522108
8162	8162	40173633	db2pfchr (SAMPLE) 0	7.996857 12.630890
7905	7905	41549891	db2pfchr (SAMPLE) 0	11.740713 21.096566
7648	7648	77332569	db2pfchr (SAMPLE) 0	15.144031 22.875247
7391	7391	23330819	db2pclnr (SAMPLE) 0	2.816424 1.252599
7134	7134	76546165	db2pclnr (SAMPLE) 0	2.897723 1.391590
6877	6877	48889921	db2pclnr (SAMPLE) 0	2.907054 1.298173
5078	5078	25821207	db2pclnr (SAMPLE) 0	2.689734 1.156277
4821	4821	78839939	db2dlock (SAMPLE) 0	0.026056 0.010703
4564	4564	57213167	db2lfr (SAMPLE) 0	0.000070 0.000019
4307	4307	52363343	db2loggw (SAMPLE) 0	1.382791 2.136769
4050	4050	60752033	db2loggr (SAMPLE) 0	1.032843 0.971773
3793	3793	26083455	db2logmgr (SAMPLE) 0	0.185455 0.146322
3536	3536	24379629	db2logts (SAMPLE) 0	0.078673 0.010984
16591	16591	59048107	db2taskd (SAMPLE) 0	0.947635 0.178037
16334	16334	73203767	db2agntdp (SAMPLE) 0	0.453388 0.062238
16077	16077	52625459	db2evmgi (DB2DETAILDEADLOCK) 0	0.672271 0.340731
15820	15820	55640127	db2agntdp (SAMPLE) 0	0.472618 0.067260
15563	15563	49938505	db2agent (idle) 0	0.488789 0.075283
15306	15306	20578449	db2agntdp (SAMPLE) 0	0.469075 0.069074
15049	15049	26673315	db2wlmd (SAMPLE) 0	0.659992 0.334875
14535	14535	55967943	db2stmm (SAMPLE) 0	2.983204 0.103102
13729	13729	75497551	db2agent (idle) 0	0.000265 0.000080
13472	13472	32505927	db2agntdp (SAMPLE) 0	0.002232 0.001195
10067	10067	14352401	db2agent (SAMPLE) 0	964.647885 37.259222
3263	3263	63832205	db2agent (SAMPLE) 0	6.201576 1.147794
2828	2828	20250783	db2resync 0	0.495884 1.623417
2571	2571	78512271	db2tcpcm 0	0.142803 0.102343
1286	1286	68157657	db2ipccm 0	891.341070 507.970715
1029	1029	38076593	db2licc 0	0.024446 0.204877
772	772	44499051	db2thcln 0	0.154706 0.027115
515	515	31260741	db2aiiothr 0	969.326721 1151.582023
2	2	25297099	db2alarm 0	81.498307 87.873824
258	258	38535295	db2sysc 0	426.271595 448.221251

可以看到，在列表中有很多 EDU 的名字我们没有涉及，由于它们与 DBA 平日中的交集并不是很多，因此我们不需要在本书中详细讨论。如果读者想了解可以参考 IBM developerworks 上的相关文章。

12.5 小结



本章主要介绍了 DB2 8/9.1 版本使用的进程模型，以及 9.5/9.7 版本引入的线程模型。本章主要强调了几种常见的进程原理，如代理进程、预取进程和页面清理进程，对 db2sysc、db2wlog、db2acd 等进程也做了介绍。并详细介绍了 db2start 和数据库启动过程中进程和内存的分配情况。

DB2 中的进程通讯与依存关系极为复杂，对于 DBA 来说，了解常见的进程概念和操作即可。如果读者希望进一步了解相关信息，请参考 developerworks、信息中心等资料

12.6 判断题



(1) DB2 9.1 版本中首次引入线程模型。

T: 正确

F: 错误

(2) 预取进程(或线程)的设置越多越好。

T: 正确

F: 错误

(3) 在 OLTP 业务中，用户应当使用分区内并行机制。

T: 正确

F: 错误

(4) 页面清除机制会每隔一秒钟自动运行并检测脏页。

T: 正确

F: 错误

(5) 每种操作系统中实现线程的内部机制都是一样的。

T: 正确

F: 错误



DB2 内存模型

在任何一个应用程序中，进程与内存的关系都是相辅相成、不可分割的。脱离了进程则没有可执行资源，而脱离了内存则没有了相关数据，因此在理解一个产品原理的时候，一定要从进程与内存两个方面入手。

在前一个章节中，我们主要介绍了 DB2 的进程/线程模型。本章节将主要讨论 DB2 的内存模型，也就是 DB2 如何将大量不同的用户数据与控制资源存储在内存中，以及用户如何配置及监控 DB2 内存的使用情况。本章内容安排如下：

- 从操作系统看内存。
- DB2 8/9.1 内存模型。
- DB2 9.5/9.7 内存模型。
- 内存监控。

13.1 从操作系统看内存



在讲解 DB2 内存模型之前，我们首先要了解操作系统是如何管理进程的内存的。

对于 UNIX/Linux 和 Windows 操作系统，每一个进程都有一个自己的虚拟内存寻址空间（具体的解释可以参考后文中性能诊断调优章节），而它们可以看到的内存都只是针对该寻址空间有效的。

举例来说，一个隶属于某公司的客户销售代表，他所能看到的客户信息只是分配给自己的客户名单，而整个公司中所有的客户列表只有经理才能看到。

对于这种情况，如果经理手下有 10 名销售代表，每个销售代表管理着不同的客户账号，这就可以简单地将每个销售代表作为进程，客户经理就是操作系统，而每个客户代表所掌握的名单就是虚拟进程寻址空间。

操作系统作为一个掌控全局的检测者，能够看到系统中所有内存使用的情况。它根据一些算法，将有限的内存资源按照一个个标准单位（4K、64K 等页面大小）分配给不同的进程。而每个进程，都有一个 32 位或者 64 位的寻址空间，通过该寻址空间，操作系统可以将进程本地内存地址映射为全局虚拟内存地址空间，从物理内存或者交换分区中得到数据。

继续用客户代表名单举例，每一个客户代表的名单中，客户的 ID 都是互相独立的。比如客户代表 A 的名单中，1 号客户实际上与客户代表 B 名单中的 1 号客户截然不同。而经理则掌握着一个名单，将每一个客户代表手中的客户号码与真实的客户 ID 相互关联。这样，每个客户代表不需要知道该客户在公司中真实的 ID，只需要在请求数据的时候询问经理“请发给我 3 号客户的信息”，这是客户经理就可以通过名单将该代表手中的 3 号客户联系到公司系统中的真实客户 ID，从而得到用户信息。

既然每个进程中的内存都是互相独立的，那么如果想要在两个进程中共享信息该怎么办呢？举例来说，如何才能让两个客户代表共同服务于一个客户？这就涉及共享内存的概念了。刚才我们所提到的每个进程只有自己能见到的信息叫做私有内存，也就是其他的进程不可能通过操作系统直接访问到这些内存信息，因为该内存信息根本没有映射到其他进程所在的空间。而共享内存则完全相反，系统会单独创建出一块内存，将一个或者多个进程一起连接到该内存空间，使得一个进程所进行的修改在另一个进程中完全可见。

在 POSIX 标准中，共享内存使用函数 `shmget()` 创建。通过指定的共享内存 ID，多个进程可以使用函数 `shmat()` 将该段内存连接到进程，从而同时访问该段共享内存，从中查找或者更新数据。

在图 13.1 中，操作系统维持了内存块 A、B、C、D、E 和 F。其中进程 1 分配了私有内存 A 与 F，而进程 2 分配了内存块 B 和 C。但是内存块 E 与 F 是共享内存，进程 1 与进程 2 可以同时访问该内存中的资源。

对于内存块 A 和 F，由于该内存为私有，因此只有进程 1 才能够访问其中的资源。同理，对于 B 和 C，只有进程 2 才能够访问其中的数据。

那么 DB2 是使用私有内存还是共享内存呢？答案是都有。在 DB2 8/9.1 中，对于进程模型，自然共享内存起到了决定性的作用。当不同的 EDU 同时访问实例或者数据库相关信息的时候，这些信息必须存在于共享内存，才能够让所有的进程能够访问。而在 DB2 9.5/V9.7 中，由于使用了线程模型，几乎所有的工作线程都存在于同一个进程中。不过 DB2 并没有将共享内存完全转化成私有内存，因为一些 DB2 的特定工具依然需要从外部连接到实例与数据库内存中（例如 `db2pd`）。不过即使这样，在线程模型中 DB2 的内存结构也有了相当大的变化，因此在这一节中，我们同样将 DB2 8/9.1 作为一部分，而 9.5/9.7 版本作为另外一部分讨论。

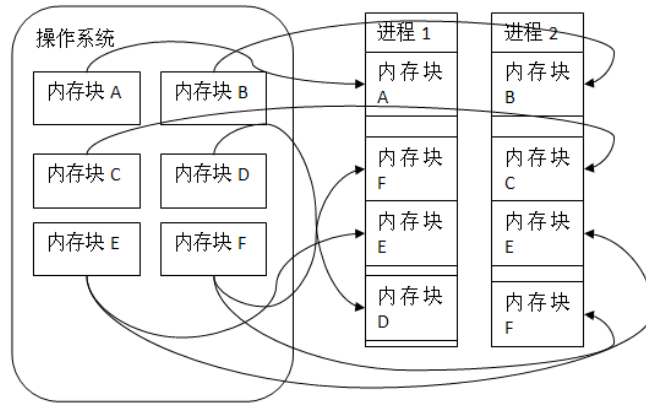


图 13.1 内存映射

13.2 DB2 8/9.1 内存模型

在 DB2 的环境中，一个实例可以包含多个数据库，一个数据库可以运行多个应用程序。因此，共享内存的类型也是依照这种类型区分，如图 13.2 所示。

- 实例共享内存。
- 数据库共享内存。
- 应用程序组共享内存。

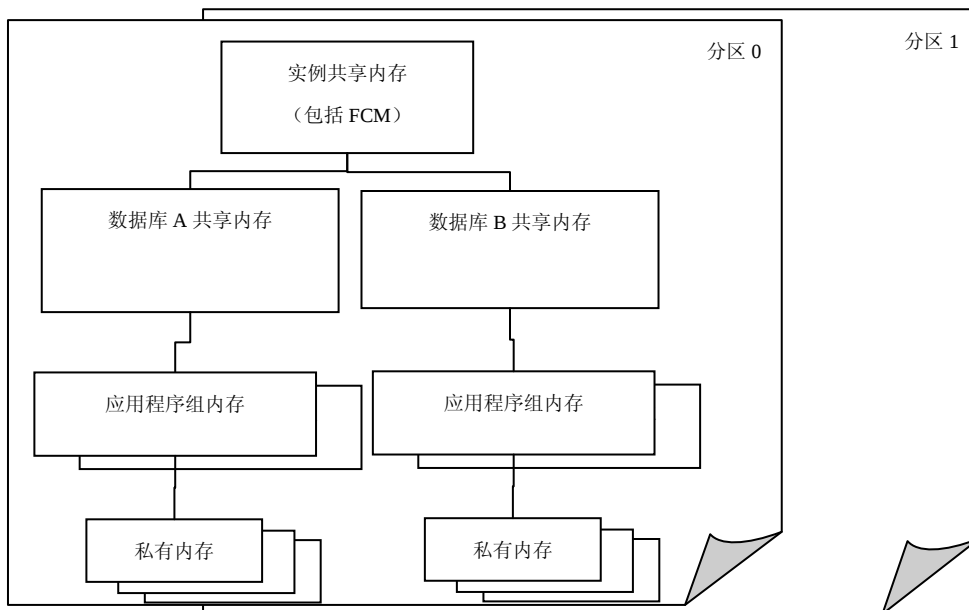


图 13.2 DB2 8/9.1 内存模型

这几部分内存，在 DB2 中是以不同的共享内存段（MemSet）分配的。在每个内存段中，又有不同类型的内存池（MemPool），用来装载不同用途的数据。同一个内存段中的不同内存池由 DB2 的内存管理模块进行控制，将不同的内存页分配给不同的内存池使用，然后每个内存池管理模块负责维护该内存池中每一个内存块（MemBlock）的使用。这就构成了 MemSet->MemPool->MemBlock 的内存分配关系。

13.2.1 实例共享内存段

实例共享内存段，顾名思义，就是实例级别的共享内存，包括一些内核控制块、FCM、监控和审计内存堆等，如图 13.3 所示。这部分内存是在 db2start 时就分配，在 db2stop 时回收。

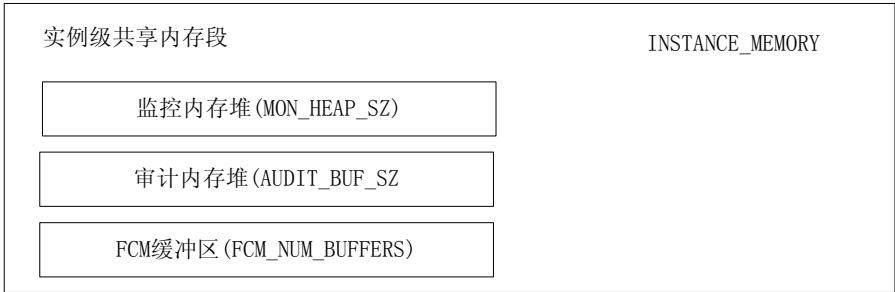


图 13.3 实例内存参数

一般来说，这块共享内存不需要很大的内存空间，由 INSTANCE_MEMORY 实例参数控制。下例中，10313 代表 10313*4KB=42242KB，大概 40MB。缺省情况下，该值是 AUTOMATIC，在实例启动时根据其他参数的设置自动计算大小。

```
db2 attach to <instance>
db2 get dbm cfg show detail | grep INSTANCE_MEMORY
Size of instance shared memory (4KB) (INSTANCE_MEMORY) = AUTOMATIC(10313)
AUTOMATIC(10313)
```

监控内存池可以由 MON_HEAP_SZ 调整：

```
Database monitor heap size (4KB) (MON_HEAP_SZ)=AUTOMATIC(90)
```

审计内存则可以由 AUDIT_BUF_SZ 调整：

```
Audit buffer size (4KB) (AUDIT_BUF_SZ)=0
```

FCM 主要用于分区间的数据通讯。通常情况下，与 FCM 相关的内存参数采用缺省值即可。

在 DPF 中，每一个逻辑分区维护一个自己的实例共享内存（不管是不是在同一个物理节点中）。

13.2.2 数据库共享内存 ■ ■ ■

通常情况下，数据库共享内存占据系统大部分的内存，包含缓冲区、锁列表、编目缓存、包缓存、database heap 与 utility heap 等内存池，如图 13.4 所示。

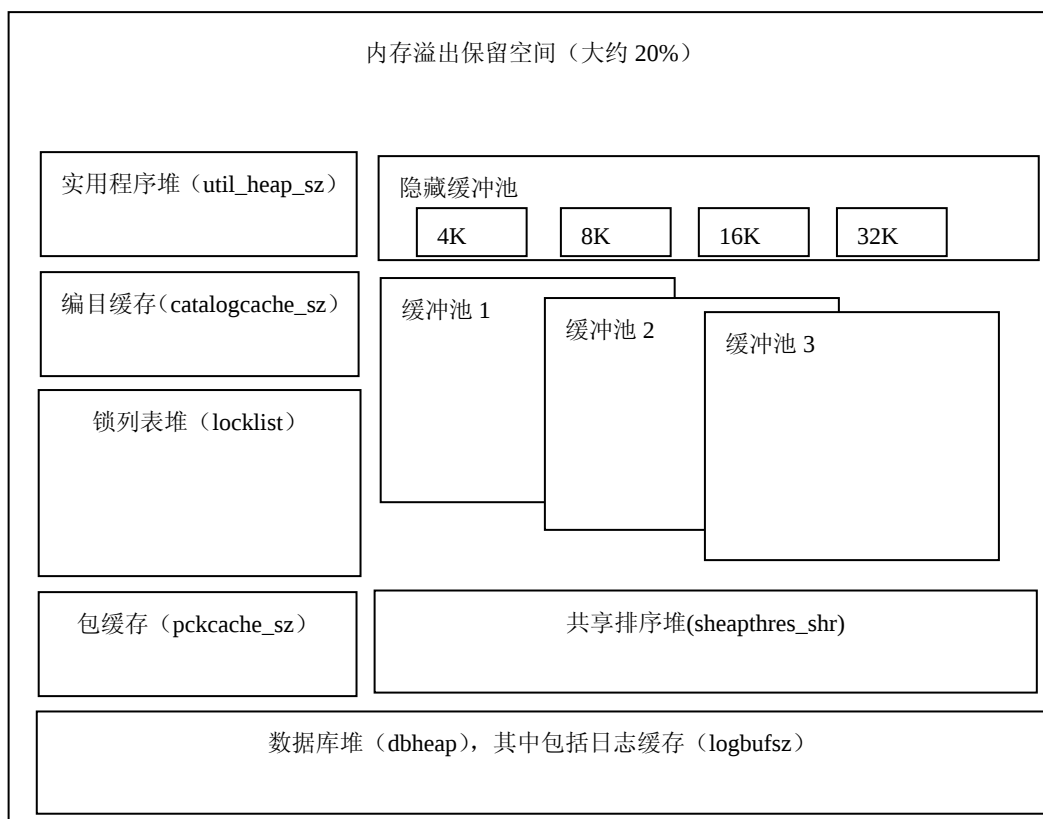


图 13.4 数据库共享内存

数据库共享内存大小的上限由数据库参数 `database_memory` 限制：

```
/home/db2inst1 $ db2 get db cfg for sample | grep -i "database_memory"
Size of database shared memory (4KB) (DATABASE_MEMORY) = AUTOMATIC(28112)
```

在 DB2 8 版本中，如果将 `database_memory` 设置为 `AUTOMATIC`，在数据库被激活的时候，系统会自动根据其他内存参数计算该数据库需要使用的内存上限，然后分配一个固定大小的内存。

以下介绍几个重要的数据库内存池。

1. 缓冲池

在数据库中，通常情况下，缓冲池是占用内存比例最大的。当 DB2 需要从磁盘上读取数据

时，该数据首先会读入缓冲池，然后当数据依然占据缓冲池的这段时间内，所有对该数据的访问可以直接通过内存，而不用请求额外的 I/O。这样做的好处是提升了性能。

除了用户所定义的缓冲区，系统还维护着 4 个隐藏缓冲池。这 4 个隐藏缓冲池分别对应着 4K、8K、16K 与 32K 的页面大小，当用户定义的缓冲池出于某些原因无法被分配时，系统会自动使用隐藏缓冲池。当然，系统性能在使用隐藏缓冲池时会受到极大的影响。

缓冲池的大小可以在创建缓冲池时指定，也可以由 `alter bufferpool` 命令修改：

```
/home/db2inst1 $ db2 "create bufferpool bp1 size 1024 pagesize 4096"
DB20000I The SQL command completed successfully.
/home/db2inst1 $ db2 "alter bufferpool bp1 size 2048"
DB20000I The SQL command completed successfully.
```

用户的缓冲池大小可以通过如下语句监测与更改（缓冲池的大小为 `NPAGES * PAGESIZE` 字节）：

```
$ db2 "select BPNAME, BUFFERPOOLID, NPAGES, PAGESIZE from syscat.bufferpools"
```

BPNAME	BUFFERPOOLID	NPAGES	PAGESIZE
IBMDEFAULTBP	1	1000	4096

```
1 record(s) selected.
```

很多用户也许有疑问，到底系统中应该使用一个巨大的缓冲池来存放所有表空间的数据好，还是用若干个相对较小的缓冲池，对应不同的表空间。

答案是“因地制宜”。对于一些应用程序需要经常访问，并且一定要保证性能的查询来说，可以使用单独的大缓冲池，确保该表中绝大部分数据都能被常驻内存；而对于其他一些相对并不是很关键的业务，则可以将它们放到一个较大的缓冲池中，让 DB2 自行调节哪些页面应该驻留内存。这个问题没有一个简单答案，需要用户在性能测试中找到适合自己系统的方案。

2. 锁列表

锁列表内存池（Lock List）是用来保存锁信息的内存，在锁与并发章节曾经提到过。锁列表中包含着该数据库中所使用到的所有的锁信息。一般来说这块内存段的大小完全取决于业务负载。有些系统中可能经常需要使用长事务进行数据的更改，这种情况下每一个事务可能会需要很多的锁资源，才能锁住所有被更改过的数据。在这种情况下，我们需要适当地增加锁列表的大小，否则系统中会经常出现锁升级现象。

```
$ db2 get db cfg for sample | grep -i "LOCKLIST"
Max storage for lock list (4KB) (LOCKLIST) = 100
```

这段内存是在数据库共享内存段中分配，在 9.1 版本之后支持自动调节功能，因此建议设置为 AUTOMATIC。

而在 DB2 8 中，当用户决定锁列表的大小时，应当考虑系统中平均并发业务的数量与吞吐量。一般来说，每一个锁占用 64 字节大小，如果用户系统中平均运行 100 个并发业务，每个业务平均使用 1000 个锁，那么锁列表的大小最好不要小于 $64 \times 100 \times 1000 = 6400000$ 字节，而为了打出一些余量，locklist 最好设置为不小于 2000 页（默认设置为 100 页，在大部分的生产系统中都需要进一步调整）。

3. 包缓存

包缓存（Package Cache）包含着被编译过的 SQL 信息。我们知道，当一条 SQL 准备执行的时候，如果该 SQL 不存在于包缓存中，则需要由编译器将其进行编译，编译后生成的包保存在该内存池中。当同样的语句再一次被调用，就可以略过编译部分，直接使用已经编译好的包。这一块内存存在繁忙的系统中可能会被分配得较大，譬如 500MB 甚至更多。但是过大的包缓存有时会使包缓存查找时间花费较多，所以需要在调优中寻找到一个最适合自己应用的值。一般来说，默认的值在大部分情况下都需要进行调整增大：

```
$ db2 get db cfg for sample show detail | grep -i "PCK"
Package cache size (4KB)          (PCKCACHESZ) = (MAXAPPLS*8)          (MAXAPPLS*8)
```

4. 编目缓存

由于 DB2 对系统表的高频率使用，让其常驻内存可以提高性能。编目缓存保存着一些重要编目信息的数据结构，当系统需要这些信息的时候，可以直接以二进制的方式在数据结构中提取出需要的信息，而不是依靠关系表查找出相应的行和列。一般来说，该参数不需要过于频繁地调整。当用户认为编目缓存过小时，可以逐步增加编目缓存的大小来适应系统的需求：

```
$ db2 get db cfg for sample show detail | grep -i "CATALOG"
Catalog cache size (4KB)         (CATALOGCACHE_SZ) = (MAXAPPLS*5)         (MAXAPPLS*5)
```

5. 数据库堆

数据库堆（Database Heap）是一块大部分人都不大注意的内存池。一般来说，在 9.1 之前，这部分内存主要保存系统的一些内部结构，而大小一般也不会引人注目。但是在表压缩推出之后，每个表的压缩字典都会保存在数据库堆中。尽管每个压缩字典的大小都不会太大（大概 100KB 左右），但是如果系统中使用了大量的压缩字典，可能会对数据库堆的大小产生严重的影响。尤其是当数据库运行多分区时，每一个分区中的每一个压缩表都要在该数据库的该分区中占用大概 100KB 左右的数据库堆。

当一张表没有被压缩时，占用 4K 的 db heap 大小。每个表空间占用 10K 的 db heap 大小。

```
$ db2 get db cfg for sample show detail | grep -i "dbheap"
Database heap (4KB)              (DBHEAP) = AUTOMATIC(1200)      AUTOMATIC(1200)
```

6. 实用程序堆

实用程序堆（Utility Heap）主要用于 Load 与 Backup/Restore 等操作。一般来说，这一部分内存不一定需要分配得过大。而在数据库激活时，该内存会被分配，但是不会被提交。也就是

说，这一部分内存池在数据库激活时并不占用物理内存空间。只有当 Load、Backup、Restore 运行时，才会真正占用相应大小的物理内存。

```
$ db2 get db cfg for sample show detail | grep -i "util"
Utilities heap size (4KB)          (UTIL_HEAP_SZ) = 5000          5000
```

13.2.3 应用程序组共享内存 ■ ■ ■

应用程序组内存仅在多分区，或者分区内并行，或者 concentrator 开启时有效。这是因为在这些情况下，每一个应用程序一般需要多个代理进程共同完成一个作业。这些代理进程之间可能需要交换数据，这些信息的交换就是由应用程序组共享内存完成的。

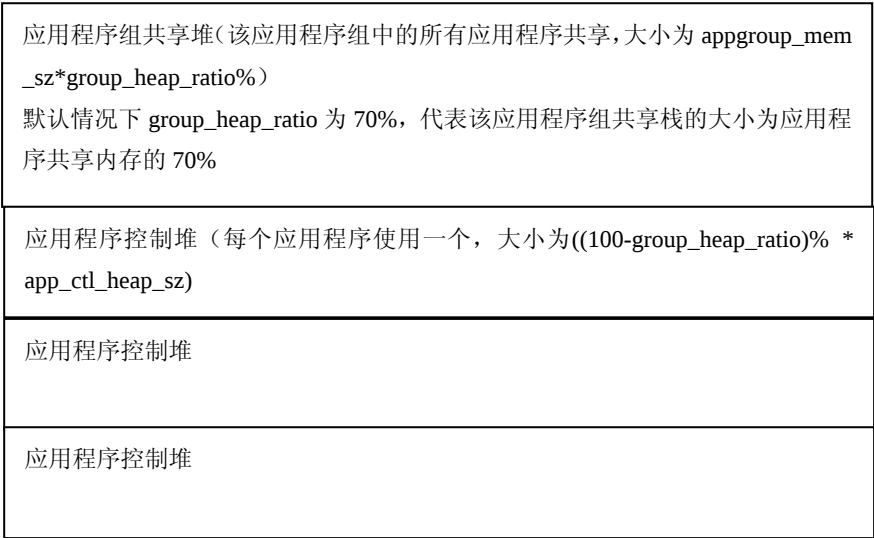


图 13.5 应用程序组共享内存

应用程序组共享内存的大小是由 appgroup_mem_sz 参数控制的。一个数据库可以有若干个应用程序组共享内存段，其中每一个应用程序组共享内存段中可以存在若干个应用程序控制堆，每一个堆得大小为 app_ctl_heap_sz。每一个应用程序控制堆可以对应一个应用程序。也就是说，每一个应用程序共享内存段中可以对应的应用程序数量由 appgroup_mem_sz / app_ctl_heap_sz 决定。对于应用程序内存，DB2 并没有提供一个参数控制应用内存的上限。

在一个应用程序组共享内存段中，其中 GROUP_HEAP_RATIO 所指定的比例用于应用程序共享堆，而剩下的 (100-GROUP_HEAP_RATIO)% 的部分则被划分为 (100-GROUP_HEAP_RATIO)% * app_ctl_heap_sz 大小的内存堆，数量则为 appgroup_mem_sz / app_ctl_heap_sz。

举例：

假设当前系统有如下配置：

```
Max size of appl. group mem set (4KB) (APPGROUP_MEM_SZ) = 400000
Max appl. control heap size (4KB) (APP_CTL_HEAP_SZ) = 512
```

```
Percent of mem for appl. group heap (GROUPHEAP_RATIO) = 70
```

那么我们可以计算得出下面的结论：

- 每个应用程序组共享内存段的大小为 $40000 \times 4096 = 163840000$ 字节。
- 每一个应用程序组共享内存段内，其中应用程序组共享堆大小为 $163840000 \times 70\% = 114688000$ 字节。
- 每一个应用程序组内存段内可以包含的应用程序数量为 $40000 / 512 = 78$ 。
- 每一个应用程序控制堆得大小为 $(100 - 70)\% \times 512 \times 4096 = 629146$ 字节。

13.2.4 私有内存 ■ ■ ■

私有内存是一类比较特殊的内存。在 DB2 V8 与 V9.1 的内存模型中，没有一个有效的方法能够监测系统中所有私有内存的使用量。

私有内存是进程使用 `malloc()` 等函数，在进程的私有内存空间内分配的内存。不同于使用 `shmget()` 分配的共享内存，私有内存无法被其他的进程所访问（也就是为什么叫它“私有”）。在 DB2 进程模型中，代理进程对于一些类型的内存，并不希望被其他的代理进程所访问，因此对于这些内存，将会由私有内存方式进行分配。

一般来说，下列内存池在私有内存中分配：

- 应用程序堆（`applheapsz` 控制）。
- 排序堆（私有排序的情况下，`sortheap` 控制）。
- 语句堆（`stmtheap` 控制）。
- 统计堆（`stat_heap_sz` 控制）。
- 查询堆（`query_heap_sz` 控制）。
- Java 解析器堆（`java_heap_sz` 控制）。
- 代理栈（`agent_stack_sz` 控制，仅适用于 Windows）。

这些内存池的作用各不相同，当应用程序运行某一条查询时，这些内存池会在需要的时候分配，当查询运行完成时被释放。不过要注意的是，这里的释放并不代表该物理内存真正回归操作系统。

就像内存的分配并不意味着内存的提交（从操作系统中拿走物理内存），内存的释放也并不真正意味着该物理内存页可以被操作系统重用。一般来说，释放一块内存后，这一段内存并不会直接被操作系统所拿到，而是回归进程所在的内存栈，当进程想要在结束前真正释放这块内存时，需要调用 `disclaim()` 函数将该物理内存释放回操作系统。

在 DB2 8/9.1 中，每一个代理进程池中的进程最大可以保留 8MB 的私有内存。假设一个代理进程在上午进行了很多复杂的查询，分配了大量的内存池。当下午的时候，该代理所对应的应用程序停止了，于是这个代理进程回到了进程池。即使该进程回到了进程池，为避免以后激活时再次从操作系统中提交内存，它也会在私有内存中保留 8MB 的内存。这样可能会产生一个

问题，当设置了大量的代理进程池时，需要保留的内存数量可能会很大（比如，1000 个代理进程池，需要保留最大 8GB 内存）。可以通过 `db2set DB2MEMMAXFREE=<字节大小>` 来设置代理进程最大保留的内存空间，比如可以设置为 2MB，或者 1MB，来避免消耗过多的内存。

13.3 DB2 9.5/9.7 内存模型

正如上一章进程模型中提到的，在 DB2 9.5/9.7 中 DB2 已经由进程模型转换为线程模型，内存模型得到了大大简化。因为所有的内存管理都被限定在同一进程中，自动内存调节功能也得到了增强。同时，除了 Linux 与 Windows 平台，其余平台已经不支持 32 位实例。

如图 13.6 所示，9.5/9.7 中的内存模型相对于以前版本，主要有几点变化：

- 应用程序内存发生了很大变化，由 `APPL_MEMORY` 数据库参数控制所有应用内存的大小。
- 共享应用程序组内存被应用程序内存所取代。
- 私有内存不再是每个进程所独有，而是被分配在一个单独的私有内存段空间中。
- `INSTANCE` 实例参数的意义发生了很大变化，它不再是实例共享内存段的上限，而是整个实例的内存上限。

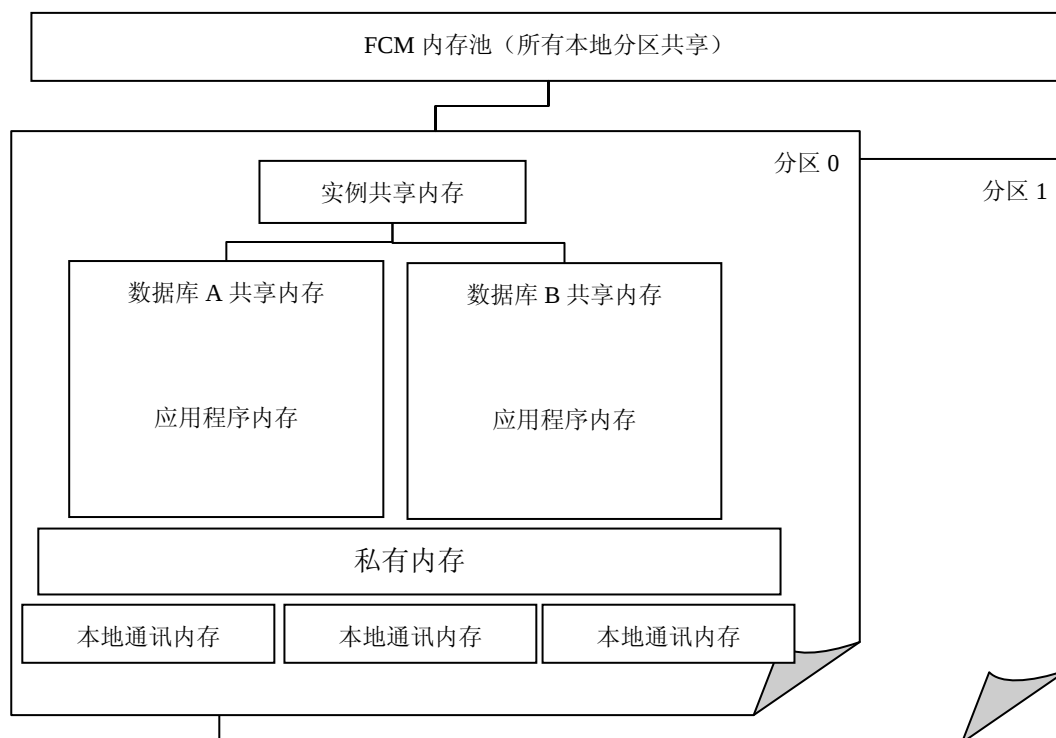


图 13.6 DB2 9.5/9.7 内存模型

13.3.1 实例内存 ■ ■ ■

在 DB2 8/9.1 时代，很多 DBA 经常问，却又得不到解答的一个问题就是：我的实例到底使用了多少内存？在进程模型中，当每个进程拥有自己的私有内存时，很难用一种有效的方法计算成百上千个进程所消耗的私有内存。

在 9.5/9.7 的线程模型中，由于所有的任务都运行在同一进程中，我们就有能力将所有的私有内存整合到一个单独的私有内存池中，然后计算整体实例的内存消耗。在 9.5/9.7 中，实例参数 `INSTANCE_MEMORY` 并不单指实例共享内存的大小，而是整个实例可以使用的最大内存大小，包括该实例下所有的实例共享内存、数据库共享内存、应用程序内存以及私有内存。换句话说，`INSTANCE_MEMORY` 参数可以控制 DB2 可以消耗的最大内存，而不是预先分配这么多内存。

通常情况下，不需要调整 `INSTANCE_MEMORY` 参数值，而使用缺省的 `Automatic` 值，在只有一个实例的情况下，`INSTANCE_MEMORY` 的上限为机器可用物理内存的 75-95%。

13.3.2 应用程序内存 ■ ■ ■

在 9.5/9.7 版本，应用程序内存的变化最大。由于引入了新的线程模型，应用程序共享内存组和私有内存的差别已经不复存在。相反，引入了一个新的 `APPL_MEMORY` 数据库参数控制应用程序内存的大小，如图 13.7 所示，该参数缺省值为 `AUTOMATIC`，在没有特殊原因的情况下，应当避免手工调节该参数。应用程序内存 (`APPL_MEMORY`) 和数据库共享内存 (`DATABASE_MEMORY`) 的和应该总是在 `INSTANCE_MEMORY` 参数的范围内。

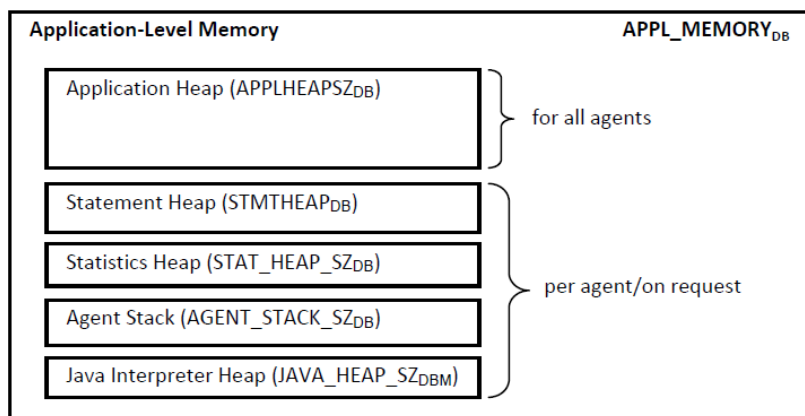


图 13.7 应用程序内存段

`APPLHEAPSZ` 参数的含义发生了变化。以前的版本代表着每个代理分配的应用程序堆。现在是每个应用一个应用程序堆，可能是多个代理共享，而不是为每个代理分配一个。建议设置 `APPLHEAPSZ` 的大小设置为 `Automatic`。

另外，不再需要多个应用程序组共享内存段，相对应的三个参数：APPGROUP_MEM_SZ、APP_CTL_HEAP_SZ 和 GROUPHEAP_RATIO 已经不再使用，DB2 将忽略这几个值。

可能很多 DBA 都有在 DB2 8/9.1 版本调整应用程序组内存的痛苦经历。9.5/9.7 内存模型的优点是，避免了繁琐的应用程序组内存设置，所有的应用程序组相关的设置都包含在 APPL_MEMORY 参数中，由数据库自动调节。

同时，语句堆、应用程序堆、统计堆等内存区也移至应用程序内存段。

Agent_Stack_SZ 参数，在 DB2 8/9.1 中只对 Windows 平台有效，但是在 9.5/9.7 中的线程模型中，则对所有系统都有效。该参数指的是每一个线程所使用的栈大小。在操作系统进程中，栈的作用可以用来储存函数调用，以及一些本地变量。因此，每一次的函数调用与本地变量的使用都需要在栈中分配一段空间。

在过去的进程模型中，每一个进程拥有自己的栈，在 AIX 平台中，32 位进程的第二内存段就被用来存放堆栈信息，也就是堆栈的整体可以达到 128MB。但是在线程模型中，所有的线程共享一个进程空间，既然每一个线程可以担负不同的工作，那就要求每个线程拥有独立的堆栈存放本地变量与函数调用指针。如果栈的大小不够的话，DB2 的线程就有可能使用超出栈大小的空间，从而导致一些查询报错，甚至实例崩溃。

一般来说，不需配置该参数，使用默认值即可。如果系统中经常运行复杂的查询（在编译器的编译阶段可能会使用到大量的堆栈资源），并且得到 SQL0973N 错误，可以考虑适当增加该参数。

13.3.3 自动内存调节 (Self Tuning Memory Management , STMM) ■ ■ ■

自动内存调节(STMM)，顾名思义，就是让 DB2 自己调节其内存的使用。实际上，STMM 在 9.1 版本就已经引入，但是由于 9.1 与 9.5/9.7 架构相差过大，我们主要在 DB2 9.5/9.7 内存架构中讨论。

STMM 的设计理念是，当系统中的业务类型不断变化的时候，DB2 通过一段时间内对各个内存池使用情况的统计，对各个内存池的大小逐渐地进行调节。

譬如说，上午的时候，系统中主要业务以 OLTP 为主，排序的操作很少。于是，我们会发现排序内存使用得很少，而缓冲池会被大量使用。于是，STMM 就会每几分钟统计一下数据，然后渐渐地增长缓冲区的大小，直到达到一个平衡状态。

而到了晚上，OLTP 业务停止，报表系统开始统计当天的销售数据，这时需要进行大量的排序，于是排序内存的使用上升，而包缓存的利用率逐渐下降。这样的话，DB2 的 STMM 就会慢慢减小包缓存的大小，将剩余出来的内存用来进行排序。

其中，“逐渐调节”这个词至关重要。有些用户的系统中业务类型的切换非常迅速，也许这一分钟还在跑 OLTP，下一分钟可能就会有大量的排序查询需要执行，然后几分钟以后再次切换

回 OLTP 业务。对于这种系统，STMM 不能及时地将内存池的大小进行如此迅速地切换（因为 STMM 的理念是基于一段时间的性能统计）。如果任何用户真的需要运行类似的业务，需要关闭 STMM。

在 9.5/9.7 版本的早期补丁中，IBM 建议对大型生产系统依然使用手工内存调节。不过随着补丁版本的更新，STMM 中各种各样的问题大部分已修复。从作者的角度讲，对于一个已经存在并且运行良好的系统，没有必要特意使用 STMM；而对于一个新建的系统，则可以考虑使用 STMM 以简化配置步骤。

具体地说，配置 STMM 时应当考虑以下几个方面：

- DPF 不建议使用 STMM。
- 对于相对来讲性能问题不重要的数据库，最好设置数据库或者实例级别的内存上限。
- 对于业务和数据量变化极快的数据库，建议手工设置内存。
- 当系统内存在多个数据库和实例时，建议使用 STMM 自动管理各个实例的内存。

当用户想要使用 STMM 时，可以在以下几个层面开启 STMM。

1. 实例级别

缺省情况下，INSTANCE_MEMORY 实例参数是 Automatic，如果该实例包含多个数据库，并且至少有一个数据库的 SELF_TUNING_MEM 参数为 ON，DATABASE_MEMORY 参数为 AUTOMATIC，那么该实例的内存上限就可以被 STMM 所动态修改。

因此，如果没有特别需求，一般建议实例的 INSTANCE_MEMORY 设置为 AUTOMATIC，这样当系统业务量增大时，实例可以自动扩展自己所需要的内存资源。

2. 数据库级别

数据库级别主要有两个参数：SELF_TUNING_MEM 与 DATABASE_MEMORY。其中 SELF_TUNING_MEM 是 STMM 的开关，用来开启和关闭数据库级别的 STMM；而 DATABASE_MEMORY 类似 INSTANCE_MEMORY，就是动态调节该数据库所占用的内存。当 SELF_TUNING_MEM 开关打开时，并且 DATABASE_MEMORY 设置为 AUTOMATIC，那么该数据库就可以在运行时由 STMM 监控，动态地增加或减少该数据库所需的内存。

3. 内存池级别

STMM 除了可以自动调节实例和数据库使用的内存大小外，更重要的是，STMM 可以在一个数据库的不同内存池之间调节内存的分配：如 BUFFERPOOL、SORTHEAP、LOCKLIST 和 PCKCACHESZ 等。如果用户期望这些内存池能够被 STMM 自动管理，可以将这些参数设置为自动(automatic)。STMM 需要至少 2 个或者以上的内存池设置为 AUTOMATIC。

简单地说，如果用户希望简化内存管理，可以采用 STMM。在 9.5 版本，64 位数据库环境，在创建实例和建库时，上述提到的 STMM 相关参数缺省均为 Automatic，用户不需要进行特殊配置。

13.4 内存监控

DB2 提供了多种命令监控实例和数据库的内存使用情况，其中有一些命令是 9.5/9.7 版本独有的。

13.4.1 db2mtrk ■ ■ ■

db2mtrk 是一个最古老的内存监视工具之一，这个命令可以显示出一个实例下所有的数据库与实例内存的使用情况：

```
/home/db2inst1 $ db2mtrk -i -p -v -d
Tracking Memory on: 2011/02/01 at 09:02:29

Memory for instance

    Other Memory is of size 103022592 bytes
    Database Monitor Heap is of size 327680 bytes
    FCMBP Heap is of size 66715648 bytes
    Total: 170065920 bytes

Memory for database: SAMPLE

    Backup/Restore/Util Heap is of size 65536 bytes
    Package Cache is of size 196608 bytes
    Other Memory is of size 196608 bytes
    Catalog Cache Heap is of size 196608 bytes
    Buffer Pool Heap (1) is of size 8650752 bytes
    Buffer Pool Heap (System 32k buffer pool) is of size 851968 bytes
    Buffer Pool Heap (System 16k buffer pool) is of size 589824 bytes
    Buffer Pool Heap (System 8k buffer pool) is of size 458752 bytes
    Buffer Pool Heap (System 4k buffer pool) is of size 393216 bytes
    Shared Sort Heap is of size 0 bytes
    Lock Manager Heap is of size 655360 bytes
    Database Heap is of size 35782656 bytes
    Application Heap (369) is of size 65536 bytes
    Application Heap (368) is of size 65536 bytes
    Application Heap (367) is of size 65536 bytes
    Application Heap (365) is of size 65536 bytes
    Applications Shared Heap is of size 196608 bytes
    Total: 48496640 bytes

Memory for agent 5943

    Other Memory is of size 196608 bytes
    Total: 196608 bytes
```

```

Memory for agent 7970

    Other Memory is of size 196608 bytes
    Total: 196608 bytes

Memory for agent 6685

    Other Memory is of size 589824 bytes
    Total: 589824 bytes

Memory for agent 6200

    Other Memory is of size 327680 bytes
    Total: 327680 bytes

```

db2mtrk 命令可以清楚地列出该数据库和实例中每一个内存池当前所使用的内存大小，一般来说，用户可以通过该命令简单地监视系统中各个内存池的使用情况。但是，由于 DB2 的内存管理模块会保留出一些缓冲内存，仅仅将该结果中所有的数值相加，并不能计算出真正实例的内存消耗（只能得到近似结果，可能会有一定的误差）。

13.4.2 db2pd -dbptnmem ■ ■ ■

db2pd -dbptnmem 这个命令是在 9.5/9.7 中独有的。该命令可以直接显示出本实例总共消耗的内存，而不需要用户将一个个内存池加到一起，这个数值可以真实地反映实例所消耗的内存总量：

```

/home/db2inst1 $ db2pd -dbptnmem

Database Partition 0 -- Active -- Up 0 days 15:02:10 -- Date 02/01/2011 09:08:43

Database Partition Memory Controller Statistics

Controller Automatic: Y
Memory Limit:          3556908 KB
Current usage:          392106 KB
HWM usage:              400298 KB
Cached memory:          180800 KB

Individual Memory Consumers:

Name                    Mem Used (KB) HWM Used (KB) Cached (KB)
=====
APPL-SAMPLE             160000      160000      159488
DBMS-db2inst1           31424       31424         64
FMP_RESOURCES           22528       22528          0
PRIVATE                 8320        8320        1920
FCM_RESOURCES           52074       60138          0
LCL-p26923634           128         128          0

```

LCL-p26923634	128	128	0
DB-SAMPLE	117504	117504	19328

其中 Memory Limit 就是当前 INSTNACE_MEMORY 所设置的上限。对于 AUTOMATIC 来说, 用户可以使用 db2 get dbm cfg show detail 看到当前的上限:

```
/home/db2inst1 $ db2 get dbm cfg show detail | grep -i "INSTANCE_MEMORY"
Size of instance shared memory (4KB) (INSTANCE_MEMORY) = AUTOMATIC(889227)
AUTOMATIC(889227)
```

Current usage 是当前所分配的内存大小 (并不是提交的)。HWM usage 是该实例自从最后一次启动以来, 曾经使用过最高的内存量。Cached memory 是已经被分配的内存, 但是当前并没有被使用到的部分 (操作系统已经无法将该内存分配给别的进程, 但是在 DB2 中这部分内存空闲可以用来分配给其他的内存池)。需要注意的是, Cached 部分是包含在 Mem Used 之内的。

紧接其下的列表则是对实例中各个内存段的详解。其中应用程序段, 实例段, FMP/FCM/私有内存段, 与数据库段的大小, 高水位, 与缓冲区大小都被详细地列在表中。

13.4.3 db2pd -memset / db2pd -mempool ■ ■ ■

如果用户想要详细地检查每一个内存段的信息, 可以使用 db2pd -memset 与 db2pd -mempool 命令。其中 db2pd -memset 可以直接使用, 也可以加上 -db 参数。对于没有 -db 参数的命令, db2pd -memset 给出实例共享内存段信息:

```
/home/db2inst1 $ db2pd -memsets

Database Partition 0 -- Active -- Up 0 days 15:47:00 -- Date 02/01/2011 09:53:33

Memory Sets:
Name          Address          Id          Size(Kb)  Key          DBP          Type          Unrsv(Kb)
Used(Kb)     HWM(Kb)         Cmt(Kb)     Uncmt(Kb)
DBMS          0x0780000000000000 75497519    31424      0xBFDF261    0            0            64
9984          10048            10048       21376
FMP           0x0780000001000000 109052135   22592      0x0          0            0            2
0             192              22592       0
Trace        0x0770000000000000 924844275   137547     0xBFDF274    0            -1           0
137547        0                137547      0
FCM           0x0780000002000000 174063652   271744     0xBFDF262    0            11           115520
156224        180416           156224      115520
```

而添加了 -db <数据库名> 的命令, db2pd -memset 则显示出该数据库的内存段信息:

```
/home/db2inst1 $ db2pd -memsets -db sample

Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:52:00 -- Date
02/01/2011 09:54:23

Memory Sets:
```

Name	Address	Id	Size(Kb)	Key	DBP	Type
Unrsv(Kb)	Used(Kb)	HWM(Kb)	Cmt(Kb)	Uncmt(Kb)		
SAMPLE	0x0700000040000000	228589595	117568	0x0	0	1
47040	47040	47040	70528			
AppCt1	0x0700000030000000	408944749	160064	0x0	0	12
576	960	960	159104			0
App368	0x0000000000000000	0	0	0x0	0	0
0	0	0	0			
App367	0x0000000000000000	0	0	0x0	0	0
0	0	0	0			
App365	0x0000000021B0000E	565182478	128	0x0	0	4
128	0	128	0			0
App369	0x0000000000000000	0	0	0x0	0	0
0	0	0	0			

仅仅观察内存段信息比较粗糙，如果用户发现某一个数据库内存段的大小非常可疑，那么可以使用 `db2pd -mempool` 得到更详细的内存池信息。类似 `memset`，`db2pd -mempool` 也为实例和数据库级：

```
/home/db2inst1 $ db2pd -mempool

Database Partition 0 -- Active -- Up 0 days 15:49:13 -- Date 02/01/2011 09:55:46

Memory Pools:
Address      MemSet      PoolName    Id      Overhead    LogSz      LogUpBnd    LogHWM
PhySz       PhyUpBnd    PhyHWM      Bnd BlkCnt CfgParm
0x07800000000000C40 DBMS      fcm          74      56960      3157880    1931054
3157880     3342336    1966080    3342336    Ovfl 1003    n/a
0x07800000000000AF8 DBMS      monh         11     122496    146815     368640     147175
327680     393216    327680     Ovfl 17     MON_HEAP_SZ
0x078000000000009B0 DBMS      resynch      62     23680     107320     1769472
107320     196608    1769472    196608     Ovfl 2      n/a
0x07800000000000868 DBMS      apmh         70     4512     734540     4980736     751676
786432     4980736    786432     Ovfl 24     n/a
0x07800000000000720 DBMS      kerh         52      0        610016     7733248     665808
720896     7733248    786432     Ovfl 72     n/a
0x078000000000005D8 DBMS      bsuh         71     65376     2225836     11403264
2259976     2359296    11403264    2359296    Ovfl 37     n/a
0x07800000000000490 DBMS      sqlch        50      0        2268773     2293760
2268773     2293760    2293760    2293760    Ovfl 203    n/a
0x07800000000000348 DBMS      krcbh        69      0        70496     65536     70672
131072     65536     131072     Ovfl 15     n/a
0x07800000200000868 FCM       fcmsess      77     65376     14517800     6692864
14517800    14811136    6750208    14811136    Ovfl 5      n/a
0x07800000200000720 FCM       fcmchan      79     65376     38981312     36675584
41666624    39190528    36700160    41943040    Ovfl 8      n/a
0x078000002000005D8 FCM       fcmbp        13     65376     66603968     53100544
88599488    66715648    53149696    88735744    Ovfl 11     n/a
0x07800000200000490 FCM       fcmctl       73     395424    12927551     47820816
12927551    13565952    47841280    13565952    Ovfl 3041    n/a
```

```

0x0780000020000348 FCM      eduah      72      24384      25600024      25600064
25600024      25624576      25624576 25624576      Ovfl 1      n/a
0x0780000010000348 FMP      undefh     59      8000      122900      22971520      122900
131072      23003136 131072      Phy 1      n/a

```

而对于某一个数据库，则可以增加-db 参数：

```

/home/db2inst1 $ db2pd -mempool -db sample

Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:53:56 -- Date
02/01/2011 09:56:19

Memory Pools:
Address      MemSet      PoolName      Id      Overhead      LogSz      LogUpBnd      LogHWM
PhySz      PhyUpBnd      PhyHWM      Bnd BlkCnt CfgParm
0x0700000040001910 SAMPLE      utilh      5      0      1888      20512768      2360
65536      20512768      65536      Ovfl 8      UTIL_HEAP_SZ
0x0700000040001680 SAMPLE      pckcacheh  7      40032      105267      Unlimited
106903      196608      Unlimited 196608      Ovfl 4      PCKCACHESZ
0x0700000040001538 SAMPLE      xmlcacheh  93      48832      147616      20971520
147616      196608      20971520 196608      Ovfl 1      n/a
0x07000000400013F0 SAMPLE      catcacheh  8      51840      83520      Unlimited 83520
196608      Unlimited 196608      Ovfl 9      CATALOGCACHE_SZ
0x0700000040001160 SAMPLE      bph      16      0      8600384      Unlimited
8600384      8650752      Unlimited 8650752      Ovfl 64      n/a
0x0700000040000ED0 SAMPLE      bph      16      0      783104      Unlimited 783104
851968      Unlimited 851968      Ovfl 5      n/a
0x0700000040000C40 SAMPLE      bph      16      0      520960      Unlimited 520960
589824      Unlimited 589824      Ovfl 3      n/a
0x07000000400009B0 SAMPLE      bph      16      0      389888      Unlimited 389888
458752      Unlimited 458752      Ovfl 2      n/a
0x0700000040000720 SAMPLE      bph      16      0      324352      Unlimited 324352
393216      Unlimited 393216      Ovfl 2      n/a
0x07000000400005D8 SAMPLE      shsorth  18      0      0      20512768      0
0      20512768      0      Ovfl 0      SHEAPTHRES_SHR
0x0700000040000490 SAMPLE      lockh      4      0      606848      720896      606848
655360      720896      655360      Ovfl 1      LOCKLIST
0x0700000040000348 SAMPLE      dbh      2      444064      35041958      43581440
35044718      35782656      43581440 35782656      Ovfl 598      DBHEAP
0x0700000030000720 AppCtl      apph      1      0      10843      524288      10843
65536      524288      65536      Phy 14      APPLHEAPSZ
0x07000000300009B0 AppCtl      apph      1      0      10888      524288      10888
65536      524288      65536      Phy 17      APPLHEAPSZ
0x0700000030000868 AppCtl      apph      1      0      10888      524288      12872
65536      524288      65536      Phy 17      APPLHEAPSZ
0x0700000030000490 AppCtl      apph      1      0      11214      524288      22228
65536      524288      65536      Phy 20      APPLHEAPSZ
0x0700000030000348 AppCtl      appshrh  20      1952      113656      81920000
139600      262144      81920000 262144      Phy 25      application shared

```

我们可以看到, `mempool` 列表中详细地列出了给定的内存段中内存池的信息, 其中的第三列就是内存池名, `bph` 代表缓冲池, `utilh` 则是 `utility heap`, `dbh` 自然就是数据库堆。而最后一列则是调整该内存池的设置参数, 譬如 `lockh` 对应的就是 `LOCKLIST`, 而 `shsorth` 对应的则是 `SHEAPTHRES_SHR`。

通过观察该列表, 我们可以观察第九列 `PhySz`, 代表该内存池所使用的物理内存大小。如果用户发现某一个内存池使用了过高的内存, 则可以通过调整数据库及实例参数改变其大小 (一般来说需要重启数据库或者实例)。

13.5 小结



在这一章中, 我们简要地描述了 DB2 的内存模型及几个常见的内存池的作用。

对于刚开始接触 DB2 的用户来说, DB2 的内存设置的确非常让人头疼, 不过在 9.1 之后的自动内存维护系统中, DB2 的内存设置得到了极大的简化, 很多在 DB2 8 中需要手工调节的参数可以被指定为自动维护, 这样一般的用户可以完全将这些参数交给 DB2 去维护, 自己只需要从大的层面监控内存的使用情况即可。

一般来说, 在 STMM 启动时, 系统预期会保留出大约 10% 的空闲内存。因此如果用户的系统有 64GB 内存, 用户应当期待系统空闲内存保持在 5GB 以上。当空闲内存持续减小并且低于该警戒线时, 用户可能需要检查 DB2 内存设置, 找到消耗内存最高的内存池, 然后研究如何降低内存的使用率。

在有关系统性能监视与调优的章节中, 我们会继续对内存监控进行深入阐述。

13.6 判断题



(1) DB2 的内存模型在 V8/V9.1/V9.5/V9.7 中没有任何更改。

T: 正确

F: 错误

(2) 在 DB2 V9.5 中, `INSTANCE_MEMORY` 的设置应当大于实例中所有数据库 `DATABASE_MEMORY` 大小的总和。

T: 正确

F: 错误

(3) 在 DB2 V8 中, INSTANCE_MEMORY 设置为 AUTOMATIC 意味着启动了自动内存调节功能。

T: 正确

F: 错误

(4) 在 DB2 V9.5 中可以使用 db2pd -dbptnmem 监测实例的内存消耗情况。

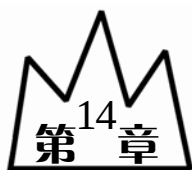
T: 正确

F: 错误

(5) db2mtrk 与 db2pd 均可以用来显示内存池的大小。

T: 正确

F: 错误



DB2 监控工具

监控数据库是 DBA 日常工作中的一个重要职责。为什么要进行数据库监控呢？通常有以下几个原因：

- 检查数据库是否正常稳定的运行
- 调优数据库或应用程序的性能。
- 调整实例和数据库参数。
- 更好的理解应用程序的负载和用户的活动。
- 进行问题诊断和分析。

DB2 提供了很多监控工具，主要分为两类：实时监控和跟踪监控。实时监控相当于照相机，记录数据库某一个时刻的快照信息，包括：SNAPSHOT 快照监控、db2pd、db2top 和 9.7 版本引入的 in-memory metrics 等工具，实时监控可以用于日常监控；而跟踪监控类似于摄像机，提供了更详细的数据库活动，包括：事件监控器和 9.7 引入的 activity monitor，事件监控器可能会产生较大的数据量，对系统造成比较大的影响，因此一般用于问题诊断。

本章主要侧重于几个常用工具的使用，对性能监控的分析部分在下一章详细介绍。本章内容安排如下：

- SNAPSHOT 命令行监控。
- SNAPSHOT 管理视图。
- db2pd。
- db2top。
- 事件监控器。

14.1 snapshot 命令行监控

SNAPSHOT 快照监控是最常用的性能监控工具，它相当于照相机，将数据库当前的活动情况记录下来，监控的对象包括实例、数据库和每个应用程序的活动，以及表空间、缓冲池、表、动态 SQL 语句、锁和排序等信息。

SNAPSHOT 监控的对象是通过开关控制的，这些开关缺省是关闭的，从最佳实践的角度，建议在初始配置的时候就在实例级别将开关打开。实例级别的开关设置需要重启实例才会生效，在生产环境要特别注意。

```
$ db2 update dbm cfg using DFT_MON_BUFPOOL ON DFT_MON_LOCK ON DFT_MON_SORT ON
DFT_MON_STMT ON DFT_MON_TABLE ON DFT_MON_TIMESTAMP ON DFT_MON_UOW ON
```

SNAPSHOT 监控命令本身比较简单，难的是对监控结果的分析。监控结果是由一些监控元素组成，监控元素分以下几类：

计数器 (counter)：用来存储累计值，比如自实例启动以来数据库发生的总排序次数 (total sorts)、死锁个数 (deadlocks)、读的行数 (rows read) 等。计数器类监控元素值为性能调优提供了很好的线索。

计量/瞬时值 (gauge)：记录某个监控元素的当前值，比如当前发生排序的次数 (active sorts)，当前锁的个数 (locks) 等。计量值反映的是当前系统的活动情况。

高水位值 (high water mark)：记录一个监控元素在打开监视器开关以来所达到的最大值或最小值，通过高水位值可获取峰值时的数据。

在实际分析和诊断问题时，通常需要进行多次抓取快照，分析一段时间内的数据库活动，以下脚本可作为参考。

```
#!/bin/ksh
maxCount=10
intervalCount=1
sleepTime=120
if [ $# -eq 0 ] ; then
echo "Usage: snaps.ksh <database>"
exit
fi
DB=$1
db2 terminate 2>&1
db2 connect to $DB
db2 reset monitor all
db2 update monitor switches using bufferpool on lock on sort on statement on table
on uow on timestamp on
while (( intervalCount <= maxCount )) ; do
dateStamp=`date +"%m%d"`.`date +"%H%M"`
db2 get snapshot for database manager > dbmsnap.$dateStamp 2>&1
```

```
db2 get snapshot for database on $DB > $DB.dbsnap.$datestamp 2>&1
db2 get snapshot for applications on $DB > $DB.appsnap.$datestamp 2>&1
db2 get snapshot for tables on $DB > $DB.tablesnap.$datestamp 2>&1
db2 get snapshot for tablespaces on $DB > $DB.tbpsnap.$datestamp 2>&1
db2 get snapshot for locks on $DB > $DB.locksnap.$datestamp 2>&1
db2 get snapshot for bufferpools on $DB > $DB.bpsnap.$datestamp 2>&1
db2 get snapshot for dynamic sql on $DB > $DB.dynsql.$datestamp 2>&1
intervalCount=intervalCount+1
sleep $sleeptime
done
db2 terminate
```

对 SNAPSHOT 监控结果的分析，请参看第 15.1.2 节“数据库级别性能监控”。

14.2 snapshot 管理视图

snapshot 命令行监控是很多 DBA 的监控首选,但最大的问题是监控结果不容易分析和统计。因此，从 DB2 9 开始增加了新的监控管理视图，这些视图与 snapshot 命令行是对应的，读者可以根据自己的习惯选择适合的方式。表 14.1 所示是常用的监控管理视图。

表 14.1

监 控 对 象	管 理 视 图	描 述
Database	SYSIBMADM.SNAPDB	数据库快照
	SYSIBMADM.SNAPDB_MEMORY_POOL	数据库内存快照
	SYSIBMADM.SNAPHADR	HADR 快照
Table	SYSIBMADM.SNAPTAB	表监控快照信息
	SYSIBMADM.SNAPTAB_REORG	表重组快照信息
Dynamic SQL	SYSIBMADM.SNAPDYN_SQL	动态 SQL 语句信息
Application	SYSIBMADM.SNAPAPPL	应用程序快照信息
	SYSIBMADM.SNAPAPPL_INFO	应用程序细节信息
Table spaces	SYSIBMADM.SNAPTbsp	表空间快照
	SYSIBMADM.SNAPCONTAINER	表空间容器快照
DBM	SYSIBMADM.SNAPDBM	实例快照

注意：snapshot 管理视图需要将实例监控开关打开，否则无法捕获所有快照，在会话级设置无效。db2 reset monitor all 命令只对 snapshot 命令生效，对管理视图不起作用。

SYSIBMADM.SNAPDYN_SQL 管理视图对于查找存在性能问题的动态 SQL 语句比较有用。

(1) 找到执行时间最长的语句 20 条语句（注：num_executions+1 是避免 num_executions 为 0 而导致溢出）：

```
SELECT    rows_read / (num_executions + 1) as avg_rows_read,
          rows_written / (num_executions + 1) as avg_rows_written,
          stmt_sorts / (num_executions + 1) as avg_sorts,
          total_exec_time / (num_executions + 1) as avg_exec_time,
          substr(stmt_text,1,1000) as SQL_Stmt
FROM SYSIBMADM.SNAPDYN_SQL ORDER BY avg_exec_time desc fetch first 20 rows only
```

(2) 找到排序最多的语句 20 条语句：

```
SELECT    stmt_sorts / (num_executions + 1) as avg_sorts,
          total_exec_time / (num_executions + 1) as avg_exec_time,
          substr(stmt_text,1,1000) as SQL_Stmt
FROM SYSIBMADM.SNAPDYN_SQL ORDER BY avg_sorts desc fetch first 20 rows only
```

14.3 db2pd

□ -----

对于大多数的数据库监控脚本来说，快照已经可以提供足够多的信息用来了解系统当前的状态。但是，在一些业务量非常繁忙的系统中，频繁地快照对系统性能造成的负面影响可能会比较显著。

因此，我们需要一种更加“轻量”级的工具，减小快照对系统性能造成的影响，即 db2pd。db2pd 不需要获取 DB2 内部锁（latches）和引擎资源返回监控数据，因而速度更快、性能更好。db2pd 直接调用 shmat 连接到数据库与实例的共享内存，不需要建立到数据库的连接。db2pd 基本不会对系统性能造成明显的影响。

db2pd 工具中包含了大量的公开与未公开的参数，可以指定抓取不同的信息。如果用户希望获取尽可能多的信息，可以使用“db2pd -everything”来得到大部分公开的信息。

当系统为多分区环境时，用户可以使用 -alldbp 获取本机所有分区的信息，或者使用 -dbp 参数指定分区的信息。当分区中存在多个物理节点时，db2pd 必须运行在逻辑分区所在的物理节点，才能够有效地连接到该分区的数据库与实例共享内存（不过在 V9.5 FP7 与 V9.7 FP3 之后，如果指定分区在远程物理节点，可以使用 -global 参数运行，例如 db2pd -db sample -dbp 3 -global。在 V9.7 FP4 之后不需要指定 -global 也可以获得远程物理节点的逻辑分区信息）。

本节我们将介绍一些最常用的 db2pd 参数。

1. -appl（应用程序）

```
/home/db2inst1 $ db2pd -db sample -appl
```

```

Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:00:18 -- Date
02/09/2011 13:19:47

Applications:
Address                               AppHandl [nod-index] NumAgents  CoorEDUID  Status
C-AnchID      C-StmtUID              L-AnchID      L-StmtUID      Appid
WorkloadID    WorkloadOccID  CollectActData              CollectActPartition
CollectSectionActuals
0x0780000000BB0080 1368      [000-01368] 1          6340      ConnectCompleted
0              0              0              0          *LOCAL.DB2.110209181941
0              0              N              C              N
0x0780000000D30080 1361      [000-01361] 1          9649      ConnectCompleted
0              0              0              0          *LOCAL.DB2.110209181934
0              0              N              C              N
0x0780000000B90900 1367      [000-01367] 1          6083      ConnectCompleted
0              0              0              0          *LOCAL.DB2.110209181940
0              0              N              C              N
...
External Connection Attributes
Address                               AppHandl [nod-index] ClientIPAddress
EncryptionLvl SystemAuthID
0x0780000000BB0080 1368      [000-01368] n/a          None        DB2INST1
0x0780000000D30080 1361      [000-01361] n/a          None        DB2INST1
0x0780000000B90900 1367      [000-01367] n/a          None        DB2INST1
...
Trusted Connection Attributes
Address                               AppHandl [nod-index] TrustedContext
ConnTrustType      RoleInherited
0x0780000000BB0080 1368      [000-01368] n/a
non trusted        n/a
0x0780000000D30080 1361      [000-01361] n/a
non trusted        n/a
0x0780000000B90900 1367      [000-01367] n/a
non trusted        n/a
...
Autonomous Routine Connections
Address      AppHandl [nod-index] Status      Autonomous Routine Handl
[nod-index] Status

```

该参数列出数据库中当前的连接。这部分信息等同于 db2 list applications show detail 的输出，其中包括用户连接和内部连接。

2. -edus （输出 EDU 列表）

```

/home/db2inst1 $ db2pd -edus

Database Partition 0 -- Active -- Up 1 days 02:47:05 -- Date 02/09/2011 13:22:17

List of all EDUs for database partition 0

```

```
db2sysc PID: 15495898
db2wdog PID: 15905328
db2acd PID: 16515828
```

EDU ID	TID	Kernel TID	EDU Name	USR (s)	SYS (s)
=====					
8140	8140	45470427	db2agntdp (SAMPLE)	0.011683	0.004746
7883	7883	45449935	db2agent (instance)	0.129638	0.035331
7625	7625	23568741	db2evmli (DB2DETAILDEADLOCK)	0.008639	
0.003344					
7368	7368	23687495	db2fw7 (SAMPLE)	0.007430	
0.002963					
7111	7111	23937459	db2fw6 (SAMPLE)	0.006850	
0.002905					
6854	6854	23544137	db2fw5 (SAMPLE)	0.007486	0.003022
6597	6597	23105987	db2fw4 (SAMPLE)	0.006313	0.002895
6340	6340	23896483	db2fw3 (SAMPLE)	0.006735	0.002818
6083	6083	37511591	db2fw2 (SAMPLE)	0.006950	0.003029
5826	5826	23638353	db2fw1 (SAMPLE)	0.006816	0.002934
5569	5569	36524385	db2fw0 (SAMPLE)	0.006884	0.002941
5312	5312	22663545	db2pfchr (SAMPLE)	0.002289	0.000062
5055	5055	18145713	db2pfchr (SAMPLE)	0.001918	0.000174
4798	4798	34701709	db2pfchr (SAMPLE)	0.001007	0.000489
4541	4541	34693475	db2pclnr (SAMPLE)	0.000888	0.000059
4284	4284	28410537	db2pclnr (SAMPLE)	0.000834	0.000063
4027	4027	33325553	db2pclnr (SAMPLE)	0.000672	0.000047
3770	3770	23232937	db2pclnr (SAMPLE)	0.001056	0.000106
3513	3513	23142907	db2pclnr (SAMPLE)	0.000784	0.000060
3256	3256	21086861	db2pclnr (SAMPLE)	0.000519	0.000066
2999	2999	21234059	db2pclnr (SAMPLE)	0.000844	0.000045
2742	2742	45376141	db2dlock (SAMPLE)	0.001700	0.000405
2485	2485	28393989	db2lfr (SAMPLE)	0.000916	0.000047
2228	2228	101548201	db2loggw (SAMPLE)	0.001391	0.001500
1971	1971	28226185	db2loggr (SAMPLE)	0.262575	0.136376
9649	9649	29348357	db2stmm (SAMPLE)	0.303967	0.155759
9311	9311	26772003	db2agent (SAMPLE)	5.478374	1.948298
8797	8797	24396297	db2lused (SAMPLE)	2.259700	0.939468
8540	8540	25105137	db2w1md (SAMPLE)	4.829751	1.396294
1544	1544	23695789	db2taskd (SAMPLE)	1.952098	0.660229
1286	1286	23179547	db2resync	0.003287	0.006637
1029	1029	23453957	db2ipccm	0.252390	0.209137
772	772	21987619	db2licc	0.000895	0.002205
515	515	23253487	db2thcln	0.001387	0.000674
2	2	23212527	db2alarm	0.578932	0.648263
258	258	23630207	db2sysc	66.938321	35.026101

-edus 在 9.5 之后引进。该参数输出系统中所有 EDU 的列表，其中最后两列代表着该 EDU 自从启动以来所执行的用户 CPU 与系统 CPU 时间，可以用来监测 EDU 的 CPU 消耗率。需要

注意的是，该输出中的用户与系统 CPU 代表着运行该线程系统以来总共的消耗量。如果用户希望使用该命令监测特定时间内的 CPU 消耗量，需要在给定时间的开始与终结各收集一次数据，然后对每一个线程 ID 计算两个时间点之间 CPU 使用的差量。

从 9.7 Fixpack 4 之后，IBM 增加了 interval 参数，可获取该段时间之内的 CPU 消耗量，输出结果中 USR DELTA 与 SYS DELTA 列是用户 CPU 与系统 CPU 对于该线程的消耗差。有了这个参数，就不用我们手工进行计算了：

```
/home/db2inst1->db2pd -edus interval=3
```

EDU ID	TID	Kernel TID	EDU Name	USR (s)
SYS (s)	USR DELTA	SYS DELTA		
7019	7019	6041785	db2agent (instance) 0	2546.724523
1209.814230	0.030072	0.014472		
3677	3677	1626271	db2loggr (MYDB) 0	1536.515249
758.441578	0.012801	0.005872		
515	515	1872075	db2aiothr 0	47.365481
0.000446	0.000440			48.620590
8539	8539	2682997	db2lused (MYDB) 0	30.989594
0.000331	0.000318			28.705540
8025	8025	4903007	db2taskd (MYDB) 0	28.602356
0.000250	0.000205			26.337251

3. -osinfo （操作系统信息）

```
/home/db2inst1 $ db2pd -osinfo
```

Operating System Information:

OSName: AIX
 NodeName: db2host1
 Version: 6
 Release: 1
 Machine: 00C78E5F4C00

CPU Information:

TotalCPU	OnlineCPU	ConfigCPU	Speed(MHz)	HMTDegree	Cores/Socket
16	8	16	1902	2	n/a

Physical Memory and Swap (Megabytes):

TotalMem	FreeMem	AvailMem	TotalSwap	FreeSwap
12288	2467	n/a	12288	2760

Virtual Memory (Megabytes):

Total	Reserved	Available	Free
24576	n/a	n/a	5227

Message Queue Information:

```

MsgSeg      MsgMax      MsgMap      MsgMni      MsgTql      MsgMnb      MsgSsz
n/a         4194304      n/a         n/a         n/a         4194304      n/a

Shared Memory Information:
ShmMax      ShmMin      ShmIds      ShmSeg
68719476736 1          131072      0

Semaphore Information:
SemMap      SemMni      SemMns      SemMnu      SemMsl      SemOpm      SemUme
SemUsz      SemVmx      SemAem
n/a         131072      n/a         n/a         65535      1024        n/a
32767      16384

CPU Load Information:
Short      Medium      Long
3.633591  3.440857  3.298279

CPU Usage Information (percent):
Total      Usr        Sys        Wait      Idle
27.000000  1.125000  15.500000  10.375000 73.000000

```

该参数输出本机操作系统信息，包括内存和 CPU 数量、IPC 信息与设置及 CPU 使用率等信息。此参数可以从高层面理解当前系统的设置。

4. -bufferpool （缓冲池信息）

```

/home/db2inst1 $ db2pd -db sample -bufferpool

Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:00:15 -- Date
02/09/2011 13:28:16

Bufferpools:
First Active Pool ID      1
Max Bufferpool ID          1
Max Bufferpool ID on Disk 1
Num Bufferpools            5

Address      Id  Name      PageSz      PA-NumPgs  BA-NumPgs  BlkSize
NumTbsp     PgsToRemov CurrentSz PostAlter  SuspndTSCt Automatic
0x0700000032E8A500 1  IBMDEFAULTBP 8192      1000      0          0          7
0          1000      1000      0          False
0x070000003164D820 4096 IBMSYSTEMBP4K 4096      16         0          0          0
0          16         16         0          False
0x070000003164EA20 4097 IBMSYSTEMBP8K 8192      16         0          0          0
0          16         16         0          False
0x0700000032E88100 4098 IBMSYSTEMBP16K 16384     16         0          0          0
0          16         16         0          False
0x0700000032E89300 4099 IBMSYSTEMBP32K 32768     16         0          0          0
0          16         16         0          False

```

Bufferpool Statistics for all bufferpools (when BUFFERPOOL monitor switch is ON):

BPID	DatLRds	DatPRds	HitRatio	TmpDatLRds	TmpDatPRds	HitRatio	IdxLRds	IdxPRds
HitRatio	TmpIdxLRds	TmpIdxPRds	HitRatio					
1	921	425	53.85%	0	0	00.00%	1156	
521		54.93%	0	0	00.00%			
4096	0	0	00.00%	0	0	00.00%	0	
0		00.00%	0	0	00.00%			
4097	0	0	00.00%	0	0	00.00%	0	
0		00.00%	0	0	00.00%			
4098	0	0	00.00%	0	0	00.00%	0	
0		00.00%	0	0	00.00%			
4099	0	0	00.00%	0	0	00.00%	0	
0		00.00%	0	0	00.00%			

BPID	DataWrts	IdxWrts	DirRds	DirRdReqs	DirRdTime	DirWrts	DirWrtReqs
DirWrtTime							
1	0	0	188	18	22	0	0
4096	0	0	0	0	0	0	0
4097	0	0	0	0	0	0	0
4098	0	0	0	0	0	0	0
4099	0	0	0	0	0	0	0

BPID	AsDatRds	AsDatRdReq	AsIdxRds	AsIdxRdReq	AsRdTime	AsDatWrts	AsIdxWrts	AsWrtTime
1	0	0	0	0	0	0	0	
4096	0	0	0	0	0	0	0	
4097	0	0	0	0	0	0	0	
4098	0	0	0	0	0	0	0	
4099	0	0	0	0	0	0	0	

BPID	TotRdTime	TotWrtTime	VectIORds	VectIOReq	BlockIORds	BlockIOReq	FilesClose
NoVectAvl	UnRdPFetch						
1	1133	0	0	0	0	0	0
4096	0	0	0	0	0	0	0
4097	0	0	0	0	0	0	0
4098	0	0	0	0	0	0	0
4099	0	0	0	0	0	0	0

该参数需要打开 bufferpool 监控器开关。使用这个参数，可以看到数据库启动以来每个缓冲池的使用情况，包括逻辑读、物理读等信息，同时会计算出命中率。

5. -logs （日志信息）

```
/home/db2inst1 $ db2pd -db sample -logs
```

```
Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:02:56 -- Date
02/09/2011 13:30:57
```

Logs:

```
Current Log Number 0
```

```

Pages Written                0
Cur Commit Disk Log Reads   0
Cur Commit Total Log Reads  0
Method 1 Archive Status      n/a
Method 1 Next Log to Archive n/a
Method 1 First Failure       n/a
Method 2 Archive Status      n/a
Method 2 Next Log to Archive n/a
Method 2 First Failure       n/a
Log Chain ID                 0
Current LSN                   0x00000000032C8010

```

Address	StartLSN	State	Size	Pages	Filename
0x070000003002F5F0	00000000032C8010	0x00000000	1000	1000	S0000000.LOG
0x0700000030018A70	00000000036B0010	0x00000000	1000	1000	S0000001.LOG
0x07000000300192D0	0000000003A98010	0x00000000	1000	1000	S0000002.LOG

该参数可以用来监测系统日志的使用情况。例如 Current LSN 可以表示当前的 LSN 位置，通过两次捕捉的 LSN，可以根据两者间的时间差计算出系统中日志使用的速率，从而了解系统中 IUD（Insert Update Delete）的吞吐量。

6. -tablespaces （表空间信息）

```
/home/db2inst1 $ db2pd -db sample -tablespaces
```

```
Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:05:51 -- Date
02/09/2011 13:33:52
```

Tablespace Configuration:

Address	Id	Type	Content	PageSz	ExtentSz	Auto	Prefetch	BufID	BufIDDisk
FSC NumCntrs MaxStripe			LastConsecPg Name						
0x0700000033987160	0	DMS	Regular	8192	4	Yes	4	1	1
1 0 3			SYSCATSPACE						
0x07000000339888C0	1	SMS	SysTmp	8192	32	Yes	32	1	1
1 0 31			TEMPSPACE1						
0x070000003398C000	2	DMS	Large	8192	32	Yes	32	1	1
1 0 31			USERSPACE1						
0x070000003398D760	3	DMS	Large	8192	32	Yes	32	1	1
1 0 31			IBMDB2SAMPLEREL						
0x0700000033990080	4	DMS	Large	8192	32	Yes	32	1	1
1 0 31			IBMDB2SAMPLEXML						
0x0700000033991540	5	DMS	Large	8192	4	Yes	4	1	1
1 0 3			SYSTOOLSPACE						
0x0700000033992A00	6	SMS	UsrTmp	8192	4	Yes	4	1	1
1 0 3			SYSTOOLSTMPSPACE						

Tablespace Statistics:

Address	Id	TotalPgs	UsablePgs	UsedPgs	PndFreePgs	FreePgs	HWM
Max HWM	State	MinRecTime	NQuiescers	PathsDropped			
0x0700000033987160	0	16384	16380	14008	0	2372	14008

14008	0x00000000	0	0	No				
0x07000000339888C0	1	1	1	1	0	0	0	
0	0x00000000	0	0	No				
0x070000003398C000	2	4096	4064	1824	0	2240	1824	
1824	0x00000000	0	0	No				
0x070000003398D760	3	4096	4064	608	0	3456	608	
608	0x00000000	0	0	No				
0x0700000033990080	4	4096	4064	1440	0	2624	1440	
1440	0x00000000	0	0	No				
0x0700000033991540	5	4096	4092	108	0	3984	108	
108	0x00000000	0	0	No				
0x0700000033992A00	6	1	1	1	0	0	0	
0	0x00000000	0	0	No				

Tablespace Autoresize Statistics:

Address	Id	AS	AR	InitSize	IncSize	IIP	MaxSize	LastResize
LRF								
0x0700000033987160	0	Yes	Yes	0	-1	No	None	None
No								
0x07000000339888C0	1	Yes	No	0	0	No	None	No
0x070000003398C000	2	Yes	Yes	0	-1	No	None	No
0x070000003398D760	3	Yes	Yes	0	-1	No	None	No
0x0700000033990080	4	Yes	Yes	0	-1	No	None	No
0x0700000033991540	5	Yes	Yes	0	-1	No	None	No
0x0700000033992A00	6	Yes	No	0	0	No	None	No

Containers:

Address	TspId	ContainNum	Type	TotalPgs	UseablePgs	PathID	StripeSet
Container							
0x0700000033988680	0	0	File	16384	16380	0	0
/home/db2inst1/db2inst1/NODE0000/SAMPLE/T0000000/C0000000.CAT							
0x0700000033989D80	1	0	Path	1	1	0	0
/home/db2inst1/db2inst1/NODE0000/SAMPLE/T0000001/C0000000.TMP							
0x070000003398D520	2	0	File	4096	4064	0	0
/home/db2inst1/db2inst1/NODE0000/SAMPLE/T0000002/C0000000.LRG							
0x070000003398EC80	3	0	File	4096	4064	0	0
/home/db2inst1/db2inst1/NODE0000/SAMPLE/T0000003/C0000000.LRG							
0x070000003398EF20	4	0	File	4096	4064	0	0
/home/db2inst1/db2inst1/NODE0000/SAMPLE/T0000004/C0000000.LRG							
0x070000003398F1C0	5	0	File	4096	4092	0	0
/home/db2inst1/db2inst1/NODE0000/SAMPLE/T0000005/C0000000.LRG							
0x070000003398F400	6	0	Path	1	1	0	0
/home/db2inst1/db2inst1/NODE0000/SAMPLE/T0000006/C0000000.UTM							

该参数可以显示出数据库中表空间的信息，包括当前的使用量、高水位、容器所在位置和大小，以及一些与配置相关的参数。

7. -locks （锁信息）

```
/home/db2inst1 $ db2pd -db sample -locks
```

```
Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:08:15 -- Date
02/09/2011 13:36:16
```

Locks:

Address	TranHdl	Lockname	Type	Mode	Sts	Owner	Dur	HoldCount	Att
ReleaseFlg rrIID									
0x07000000305D2B00	2		00030005000000000000000000552	Row			..X	G	2
1 0	0x000000008	0x40000000 0							
0x07000000305D1880	2		4141414141504161B4F02AB641	Internal	P		..S	G	2
1 0	0x000000000	0x40000000 0							
0x07000000305D1B80	2		00030005000000000000000000452	Row			..X	G	2
1 0	0x000000008	0x40000000 0							
0x07000000305D2800	2		0003000500000000000000000054	Table			.IX	G	2
1 0	0x00003000	0x40000000 0							

该参数可以列出数据库中所有的锁列表。这个参数对系统所造成的性能影响远小于繁忙系统中的锁快照,当使用该参数与-`applications`和-`agents`结合时可以用来诊断系统中锁相关的问题。

8. -agents (代理信息)

```
/home/db2inst1 $ db2pd -db sample -agents
```

Option -agents is an instance scope option. The database option has been ignored.

```
Database Partition 0 -- Active -- Up 0 days 00:10:40 -- Date 02/09/2011 13:38:34
```

Agents:

```
Current agents:      16
Idle agents:         0
Active coord agents: 14
Active agents total: 14
Pooled coord agents: 2
Pooled agents total: 2
```

Address	AppHandl	[nod-index]	AgentEUID	Priority	Type	State
ClientPid	Userid	ClientNm	Rowsread	Rowswrtn	LkTmOt	DBName
LastPooled						LastApplId
0x0780000000B10080	7	[000-00007]	1544	0	Coord	Inst-Active
36701276	db2inst1	db2bp	268	7		NotSet SAMPLE
*LOCAL.db2inst1.110209182801		n/a				
0x0780000000B15EE0	8	[000-00008]	5399	0	Coord	Inst-Active
36701276	db2inst1	db2stmm	0	0		NotSet SAMPLE
*LOCAL.DB2.110209182802					n/a	
0x0780000000B90080	9	[000-00009]	5656	0	Coord	Inst-Active
36701276	db2inst1	db2taskd	3	0		NotSet SAMPLE
*LOCAL.DB2.110209182803					n/a	
0x0780000000B95EE0	10	[000-00010]	5913	0	Coord	Inst-Active
36701276	db2inst1	db2wlmd	0	0		NotSet SAMPLE
*LOCAL.DB2.110209182804					n/a	

```

0x0780000000BB0080 11      [000-00011] 6170      0      Coord      Inst-Active
36701276      db2inst1 db2lused 0      0      3      SAMPLE
*LOCAL.DB2.110209182805      n/a
0x0780000000BB5EE0 12      [000-00012] 6427      0      Coord      Inst-Active
36701276      db2inst1 db2fw0 0      0      3      SAMPLE
*LOCAL.DB2.110209182806      n/a
0x0780000000BE0080 13      [000-00013] 6684      0      Coord      Inst-Active
36701276      db2inst1 db2fw1 0      0      3      SAMPLE
*LOCAL.DB2.110209182807      n/a
0x0780000000BE5EE0 14      [000-00014] 6941      0      Coord      Inst-Active
36701276      db2inst1 db2fw2 0      0      3      SAMPLE
*LOCAL.DB2.110209182808      n/a
0x0780000000C20080 15      [000-00015] 7198      0      Coord      Inst-Active
36701276      db2inst1 db2fw3 0      0      3      SAMPLE
*LOCAL.DB2.110209182809      n/a
0x0780000000C25EE0 16      [000-00016] 7455      0      Coord      Inst-Active
36701276      db2inst1 db2fw4 0      0      3      SAMPLE
*LOCAL.DB2.110209182810      n/a
0x0780000000C40080 17      [000-00017] 7712      0      Coord      Inst-Active
36701276      db2inst1 db2fw5 0      0      3      SAMPLE
*LOCAL.DB2.110209182811      n/a
0x0780000000C45EE0 18      [000-00018] 7969      0      Coord      Inst-Active
36701276      db2inst1 db2fw6 0      0      3      SAMPLE
*LOCAL.DB2.110209182812      n/a
0x0780000000C70080 19      [000-00019] 8226      0      Coord      Inst-Active
36701276      db2inst1 db2fw7 0      0      3      SAMPLE
*LOCAL.DB2.110209182813      n/a
0x0780000000C75EE0 20      [000-00020] 8483      0      Coord      Inst-Active
36701276      db2inst1 db2evml_ 0      0      3      SAMPLE
*LOCAL.DB2.110209182814      n/a
0x0780000000CF0080 0      [000-00000] 8741      0      Coord      Pooled      n/a
n/a      n/a      0      0      5      SAMPLE      *LOCAL.db2inst1.110209183809
Wed Feb 9 13:38:04
0x0780000000CF5EE0 0      [000-00000] 8998      0      Coord      Pooled      n/a
n/a      n/a      0      0      NotSet      SAMPLE      *LOCAL.db2inst1.110209183805
Wed Feb 9 13:38:03

```

该参数可以得到系统中的代理信息，当系统使用多分区或分区内并行时，每一个应用程序可能会对若干代理线程。使用该参数可以将所有的代理线程与应用程序互相对应。

9. -static （静态语句）

```

/home/db2inst1 $ db2pd -db sample -static | more

Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:13:14 -- Date
02/09/2011 13:41:15

Static Cache:
Current Memory Used      974918
Total Heap Size          1271398

```

```

Cache Overflow Flag          0
Number of References         41
Number of Package Inserts    4
Number of Section Inserts    0

Packages:
Address          Schema  PkgName
Version          UniqueID NumSec UseCount  NumRef  Iso QOpt Blk Lockname
0x0700000033F88D40 DB2INST1 SYSSH200
SYSLVL01 0      0      15      CS 5      B  5359534C564C303128EFECDC41
0x0700000033F883E0 NULLID  SQLDEFLT
Default Package 1  CONTOKN1 0      0      16      UR  5      U
434F4E544F4B4E310763DD2841
0x0700000033F80080 DB2INST1 SQLC2H21
AAAAAPAA 0      0      4      CS 5      B  4141414141504161B4F02AB641
0x0700000033FCCC40 DB2INST1 SYSLH202
SYSLVL01 0      0      6      CS 5      B  5359534C564C3031956EB7B441

Sections:
Address          Schema  PkgName
UniqueID SecNo NumRef  UseCount  StmtType Cursor  W-Hld Sect Size
0x0700000033FC0080 DB2INST1 SYSSH200
SYSLVL01 1      0      0      0      SQL_CURSH200C1 NO 0
0x0700000033FC02A8 DB2INST1 SYSSH200
SYSLVL01 2      0      0      0      SQL_CURSH200C2 NO 0
0x0700000033FC04D0 DB2INST1 SYSSH200
SYSLVL01 3      0      0      0      SQL_CURSH200C3 NO 0
0x0700000033FC06F8 DB2INST1 SYSSH200
SYSLVL01 4      0      0      0      SQL_CURSH200C4 NO 0
0x0700000033FC0920 DB2INST1 SYSSH200
SYSLVL01 5      0      0      0      SQL_CURSH200C5 NO 0
0x0700000033FC0B48 DB2INST1 SYSSH200
SYSLVL01 6      0      0      0      SQL_CURSH200C6 NO 0
0x0700000033FC0D70 DB2INST1 SYSSH200
SYSLVL01 7      0      0      0      SQL_CURSH200C7 NO 0
0x0700000033FC0F98 DB2INST1 SYSSH200
SYSLVL01 8      0      0      0      SQL_CURSH200C8 NO 0
0x0700000033FC11C0 DB2INST1 SYSSH200
SYSLVL01 9      0      0      0      SQL_CURSH200C9 NO 0
0x0700000033FC13E8 DB2INST1 SYSSH200
SYSLVL01 10     0      0      0      SQL_CURSH200C10 NO 0
0x0700000033FC1610 DB2INST1 SYSSH200
SYSLVL01 11     0      0      0      SQL_CURSH200C11 NO 0

```

该参数会列出系统中存在的静态语句包的信息，以及每个包所执行的次数。通过该信息，用户可以估算出哪些静态存储过程被调用的次数最多。

10. -dynamic （动态 SQL 信息）

```
/home/db2inst1 $ db2pd -db sample -dynamic
```

Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:14:56 -- Date 02/09/2011 13:42:57

Dynamic Cache:

Current Memory Used	974918
Total Heap Size	1271398
Cache Overflow Flag	0
Number of References	195
Number of Statement Inserts	37
Number of Statement Deletes	16
Number of Variation Inserts	22
Number of Statements	21

Dynamic SQL Statements:

Address	AnchID	StmtUID	NumEnv	NumVar	NumRef	NumExe	Text
0x0700000033F8ABA0	104	1	1	4	4		SELECT TRIGNAME FROM SYSCAT.TRIGGERS WHERE TABNAME='POLICY' AND TABSCHEMA='SYSTOOLS'
0x0700000033F84000	182	1	1	1	1		select * from employee
0x0700000033FCCFA0	199	1	1	4	4		SELECT COLNAME, TYPENAME FROM SYSCAT.COLUMNS WHERE TABNAME='POLICY' AND TABSCHEMA='SYSTOOLS'
0x070000003406FB80	230	1	1	1	1		SELECT IBM.TID, IBM.FID FROM SYSIBM.SYSTABLES AS IBM, SYSTOOLS.HMON_ATM_INFO AS ATM WHERE ATM.STATS_FLAG <> 'Y' AND IBM.TYPE IN ('S', 'T') AND ATM.CREATE_TIME = IBM.CTIME AND ATM.SCHEMA = IBM.CREATOR AND ATM.NAME = IBM.NAME ORDER BY IBM.FID, IBM.TID WITH UR
0x0700000034069140	249	1	1	1	1		DELETE FROM SYSTOOLS.HMON_ATM_INFO AS ATM WHERE NOT EXISTS (SELECT * FROM SYSIBM.SYSTABLES AS IBM WHERE ATM.NAME = IBM.NAME AND ATM.SCHEMA = IBM.CREATOR AND ATM.CREATE_TIME = IBM.CTIME) WITH UR
0x070000003407D820	314	1	1	2	2		SELECT POLICY FROM ...

Dynamic SQL Environments:

Address	AnchID	StmtUID	EnvID	Iso	QOpt	Blk
0x0700000033F8AD00	104	1	1	CS	5	B
0x0700000033F84120	182	1	1	CS	5	B
0x0700000033FCD100	199	1	1	CS	5	B
0x070000003406FD80	230	1	1	CS	5	B
0x0700000034069300	249	1	1	CS	5	B
0x070000003407D980	314	1	1	CS	5	B
0x0700000033F8FAA0	328	1	1	CS	5	B
0x070000003408C8A0	329	1	1	CS	5	B
0x070000003404FC40	390	1	1	CS	5	B
0x070000003406E6E0	398	1	1	CS	5	B
0x070000003406F640	455	1	1	CS	5	B
0x0700000033FC92E0	467	1	1	CS	5	B
0x070000003404F4C0	532	1	1	CS	5	B

```

0x070000003406EFA0 599 1 1 CS 5 B
...
Dynamic SQL Variations:
Address AnchID StmtUID EnvID VarID NumRef Typ Lockname
Val Insert Time Sect Size
0x0700000033F8B040 104 1 1 1 4 6 000000010000000100010D0056 Y
2011-02-09-13.33.04.724516 5896
0x0700000033F84460 182 1 1 1 1 6 0000000100000001000116C056 Y
2011-02-09-13.28.12.498387 6600
0x0700000033FCD440 199 1 1 1 4 6 0000000100000001000118E056 Y
2011-02-09-13.33.04.707592 6024
0x070000003407BA00 230 1 1 1 1 6 000000010000000100011CC056 Y
...

```

该信息包括了数据库的包缓存中所有的动态 SQL。每一个动态 SQL 有自己的 AnchID 与 StmtUID（两者的组合为唯一）。通过对应-applications 的输出，可以确定给定应用程序当前正在执行的语句和上一条已经执行过的语句（其中 C-AnchID 与 C-StmtUID 为当前执行的语句，而 L-AnchID 与 L-StmtUID 为上一条已经执行完成的语句）。

```

Address AppHandl [nod-index] NumAgents CoorEDUID Status
C-AnchID C-StmtUID L-AnchID L-StmtUID Appid
WorkloadID WorkloadOccID CollectActData CollectActPartition
CollectSectionActuals
0x0780000000B00080 7 [000-00007] 1 1544 UOW-Waiting 0
0 329 1 *LOCAL.db2inst1.110209182801 1 1
N C N

```

11. -tcbstats （表状态信息）

```

/home/db2inst1 $ db2pd -db sample -tcbstats

Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:18:46 -- Date
02/09/2011 13:46:47

TCB Table Information:
Address TbspaceID TableID PartID MasterTbs MasterTab TableName
SchemaNm ObjClass DataSize LfSize LobSize XMLSize
0x0700000033A1F6D8 0 1 n/a 0 1 SYSBOOT SYSIBM Perm 1
0 0 0
0x0700000033F6ECD8 2 -1 n/a 2 -1 INTERNAL SYSIBM Perm 1
0 0 0
0x0700000034024058 5 -1 n/a 5 -1 INTERNAL SYSIBM Perm 1
0 0 0
0x0700000033F4EBD8 0 5 n/a 0 5 SYSTABLES SYSIBM Perm 29
0 480 0
...

TCB Table Stats:
Address TableName SchemaNm Scans UDI RTSUDI

```

PgReorgs	NoChgUpdts	Reads	FscrUpdates	Inserts	Updates		
Deletes	OvFlReads	OvFlCrtes	RowsComp	RowsUncomp	CCLogReads	StoreBytes	
BytesSaved							
0x0700000033A1F6D8	SYSBOOT			SYSIBM	1	0	0
0	0	1	0	0	0	0	0
0	0	0	-	-			
0x0700000033F6ECD8	INTERNAL			SYSIBM	0	0	0
0	0	6	0	0	0	0	0
0	0	0	-	-			
0x0700000034024058	INTERNAL			SYSIBM	0	0	0
0	0	6	0	0	0	0	0
0	0	0	-	-			
...							

该信息包含着系统中在数据库开始后所被访问过的表的信息，其中包括该表的配置信息（例如所属的表空间），以及一些统计信息，包括 IUD（Insert Update Delete）的数量、所扫描的次数等。

如果增加 `index` 选项 `db2pd -db sample -tcstats index`，则可以同时得到索引信息。通过该信息用户可以了解每个表与索引被使用的次数。某些情况下用户可能想要查找所有不曾使用过的索引，就可以在运行应用程序一段时间后使用该命令得到相关信息。由于篇幅所限，这里不再赘述，读者可以自行尝试。

14.4 db2top

DB2 的快照与 `db2pd` 都是数据库监测的好帮手，但是如果用户没有一个趁手的脚本，可能很难在短时间内完整地阅读整个快照文件，并且有针对性地分析一组刚刚抓到的快照。很多人可能会问，有没有什么工具能够直观地显示出当前系统的状态，就好像操作系统的 `top` 或者 `topas` 工具那样？答案是有的，那就是 `db2top` 工具。`db2top` 最初是发布在 IBM 的 `alphawork` 上免费下载的。后来由于很多用户发现这个工具非常有用，IBM 决定将这个工具集成在 DB2 包中一并发行。

在早期的版本中，由于这个工具在设计之初并没有使用 DB2 内部函数调用，因此如果用户当前的 DB2 版本不包含该工具，可以在同系统下安装一个较新的 DB2 版本，然后将 `db2top` 可执行文件复制过来即可使用。

`db2top` 的原理就是在后台每隔一段时间收集一次快照，然后通过计算其与最近一次快照之间的数值差别与经过的时间，计算出一些列统计数据。

`db2top` 主要有以下两个作用：

- 实时监测系统。
- 捕捉历史快照信息。

提示：由于 db2top 所捕捉到的历史快照信息只能由 db2top 解析，不能直接转换成用户可以阅读的文本文件，因此在普通的性能数据收集，笔者依然建议使用普通的快照和 db2pd。

在多分区数据库系统，db2top 的使用需要特别谨慎。因为 db2top 的每一次刷新都是调用实例级的快照。如果没有指定针对某一个分区，对于几百个分区的数据库做全局快照会需要极大的内存空间，会对系统性能造成严重影响。

14.4.1 实时监测

简单地运行“db2top -d <数据库名>”就可以进入 db2top 的交互界面，如图 14.1 所示。

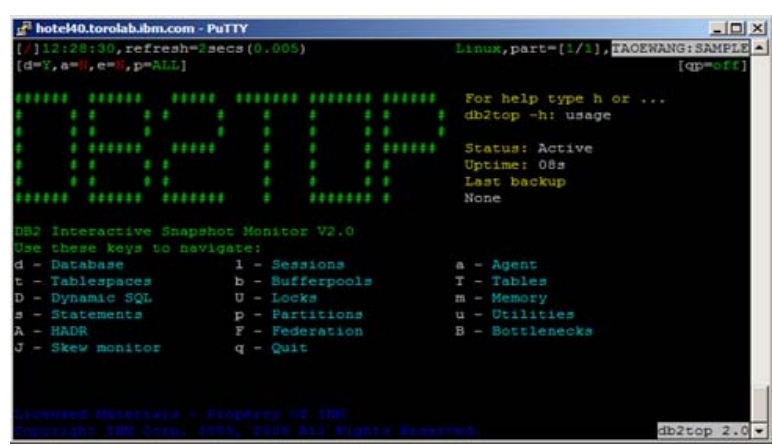


图 14.1 db2top 启动页面

左上角的时间戳显示的是最近一次的快照时间，旁边的 refresh 是每隔多少秒抓一次快照，在命令行中可以使用 -i 指定，在右边的括号中是真正使用了多长时间抓到该快照。如果我们发现括号中的时间大于指定的时间（比如系统很繁忙，可能用了几秒钟才抓到一组快照），我们就需要考虑增加 -i 的大小。上图中，最近一次的快照时间是 12:28:30，每隔 2 秒抓一次快照，最近一次快照用时 0.005 秒。

下面的一行显示的是当前的运算规则，其中 d 代表 Delta，表示是在两个快照之间做增量计算，还是简单地显示绝对大小（譬如 1 分钟前数据库有 1000 个逻辑读，现在有 1500 个，如果 d=Y，则显示 500，否则显示 1500），旁边的 a 代表 Active，即是否只显示活动变化的数值，e 代表扩展显示（Extended），在扩展显示打开的时候可以显示更多信息，p 代表分区，可能的数值有 All、CUR 和一般的数字，其中 ALL 代表所有分区，CUR 代表当前分区，而数字则代表某一个指定分区。右边的部分显示操作系统类型、当前监视的分区和实例与数据库名。

在主屏幕上，显示了一系列命令可以监测不同类型的信息。通过不同的命令可以进入子屏幕显示相关信息。

同时，用户可以使用 **h** 进入帮助菜单，在帮助菜单中显示了所有 db2top 支持的命令，每个命令区分大小写，譬如 **d** 代表数据库信息，而 **D** 则代表动态 SQL，如图 14.2 所示。



图 14.2 db2top 帮助页

关于如何阅读和分析每个子窗口的信息，我们将在后面详细讨论，在此之前，我们先来介绍一下捕捉历史快照信息模式。

14.4.2 历史信息收集 ■ ■ ■

使用 **-C** 参数，用户可以指定 db2top 将结果保存在文件中，而不是进入实时监测界面：

```
db2top -d sample -f collect.file -C -m 480 -i 15
[11:36:02] Starting DB2 snapshot data collector, collection every 15 second(s),
          max duration 480 minute(s), max file growth/hour 100.0M,
          hit [CTRL+C] to cancel...
[11:36:02] Writing to 'collect.file',
          should I create a named pipe instead of a file [N/y]? N
```

其中 **-m** 参数指定该收集模式运行多少分钟，**-i** 指定每隔多少秒收集一次快照。在上面的例子中，我们需要每隔 15 秒收集一次快照，总共运行 480 分钟。

在信息收集的过程中，该 db2top 命令不会返回命令行。当指定的收集时间过后，程序退出，用户可以使用如下命令重新播放所收集到的数据：

```
db2top -d sample -f collect.file -b 1 -A
```

可能有读者问，为什么要这样做呢？因为 db2top 图形监控方式只显示实时数据，很多用户没有条件一直盯着屏幕。这时，可以考虑在晚上收集信息，第二天进行分析，类似于看录像回

放。在重新播放的时候，如果用户希望直接跳转到某一个给定时间戳，可以使用类似下面的命令执行：

```
db2top -d sample -f collect.file /02:00:00
```

14.4.3 子窗口

当我们了解了交互模式与收集模式后，接下来着重介绍一下 db2top 中常用的子窗口，以及如何理解相关的输出内容。

1. 数据库 (d)

数据库窗口如图 14.3 所示。



图 14.3 db2top 数据库窗口

在数据库窗口中，主要显示了一些数据库整体相关的信息。其中最上面的方框内是当前一些系统资源的使用情况，譬如说 MaxActSess 就是当前活动的连接于总连接数的比例，当这个比例越高，说明数据库中活动连接所占的比重越大。在其下面的表格中，则显示了系统中一些常见的性能相关的参数，例如缓冲池大小、FCM 内存池大小、活动连接数量、死锁数量、逻辑读、物理读等。不过在图 14.3 中，我们看到所有的数值都是 0，为什么呢？仔细看看左上角，我们发现 d=Y，也就是说当前要显示的是差值。而我们的测试系统并没有任何活动连接，所以单位时间内（2 秒）所有的值的变化都为 0。如果想要显示绝对数值，我们可以按【k】键，显示结果如图 14.4 所示。

从图 14.4 中我们可以看到，一些譬如逻辑读物理读等信息不为 0，它们就是当前系统中快照所拿到的数值。



图 14.4 db2top 数据库绝对值信息

2. 表空间 (t)

当按下【t】键后，我们进入表空间子窗口如图 14.5 所示。

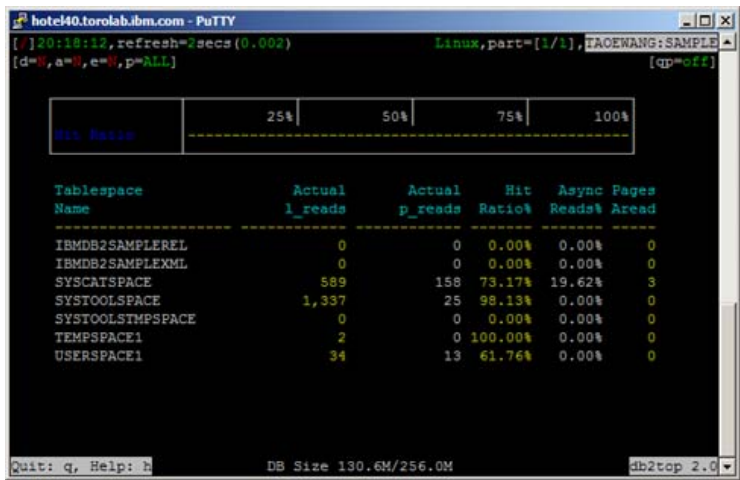


图 14.5 db2top 表空间窗口

表空间子窗口显示了表空间相关的信息。在这种有列表的表格中，用户可以按下键盘上的左、右方向键移动列表，显示不同的列，如图 14.6 所示。

在表空间子窗口中，比较重要的性能相关信息包括逻辑物理读的数量，而对于运维来说，重要的信息则是表空间使用情况等相关信息。

在屏幕的最下方则是数据库当前使用的大小与所配置的总大小。譬如我们有若干个 DMS 表空间，其中每一个的使用大约为 50%，那么最下方的信息就会显示当前使用与总大小的比例为一半左右。



图 14.6 使用箭头移动信息

3. 动态 SQL（D）

动态 SQL 列表包含了包缓存中的动态 SQL 信息，如图 14.7 所示。

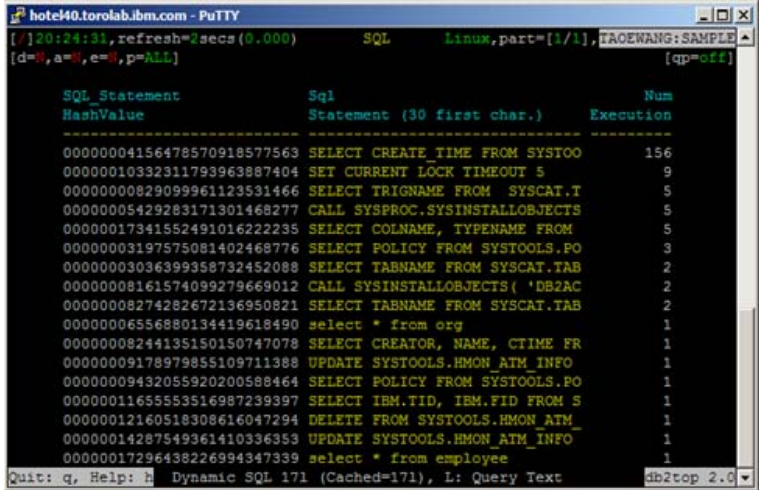


图 14.7 db2top 动态 SQL 窗口

需要注意的是，db2top 不支持上下滚屏，也就是说一个列表中无法上下滚屏。但是 db2top 提供了一种方法来选择用户需要看到的行。譬如，在 SQL 窗口，如果想要显示的 SQL 不包含“SELECT”的语句，那么可以按下【/】键进入搜索功能，然后键入“!SELECT”。当然，如果用户想要搜索包含 SELECT 的语句，则可以使用“SELECT”进行搜索。那么如果用户希望看到执行次数最多的 SQL 怎么办？

用户可以使用【z】键按照列排序。当按下【z】键后，会在屏幕的左上角出现提示，输入列号。列号码从 0 开始，例如 Num Execution 列就为 2，因此如果我们按下【z】键并输入 2，然

后按下回车键，就会出现类似图 14.7 中按照第三列排序的结果。

如果用户想要按照平均执行时间排序，则可以按【z】键并输入 4，然后按下回车键，得到类似图 14.8 所示的结果。

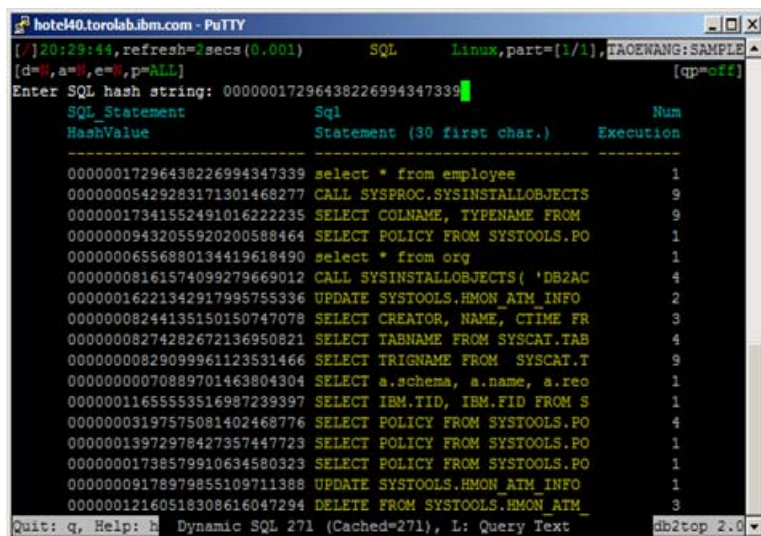


SQL_Statement HashValue	Exec Time	Avg ExecTime	Cpu Time	Avg CpuTime
00000017296438226994347339	0.168111	0.168111	0.010488	0.010488
00000005429283171301468277	0.667075	0.074119	0.002969	0.000329
00000017341552491016222235	0.546905	0.060767	0.017554	0.001950
00000009432055920200588464	0.053610	0.053610	0.003593	0.003593
00000006556880134419618490	0.029484	0.029484	0.001421	0.001421
00000008161574099279669012	0.050353	0.012588	0.001357	0.000339
00000016221342917995755336	0.022336	0.011168	0.009527	0.004763
00000008244135150150747078	0.033101	0.011033	0.004784	0.001594
00000008274282672136950821	0.036780	0.009195	0.006810	0.001702
00000000829099961123531466	0.075093	0.008343	0.006157	0.000684
0000000070889701463804304	0.007682	0.007682	0.005984	0.005984
00000011655553516987239397	0.006042	0.006042	0.006027	0.006027
00000003197575081402468776	0.019434	0.004858	0.002769	0.000692
00000013972978427357447723	0.004560	0.004560	0.002421	0.002421
00000001738579910634580323	0.003512	0.003512	0.002547	0.002547
00000009178979855109711388	0.003449	0.003449	0.003437	0.003437
00000012160518308616047294	0.009251	0.003083	0.008705	0.002901

图 14.8 db2top 按照 SQL 平均执行时间排序

其中 Avg ExecTime 为表中的第五列，因此按下【z】键然后选择 4，按下回车键后就可以得到上面的内容。

用户可以通过第一列语句的哈希值对应其真正的 SQL，具体方法是按下大写的 L，然后填入哈希值，如图 14.9 所示。



SQL_Statement HashValue	Sql Statement (30 first char.)	Num Execution
00000017296438226994347339	select * from employee	1
00000005429283171301468277	CALL SYSPROC.SYSINSTALLOBJECTS	9
00000017341552491016222235	SELECT COLNAME, TYPENAME FROM	9
00000009432055920200588464	SELECT POLICY FROM SYSTOOLS.PO	1
00000006556880134419618490	select * from org	1
00000008161574099279669012	CALL SYSPROC.SYSINSTALLOBJECTS('DB2AC	4
00000016221342917995755336	UPDATE SYSTOOLS.HMON_ATM_INFO	2
00000008244135150150747078	SELECT CREATOR, NAME, CTIME FR	3
00000008274282672136950821	SELECT TABNAME FROM SYSCAT.TAB	4
00000000829099961123531466	SELECT TRIGNAME FROM SYSCAT.T	9
0000000070889701463804304	SELECT a.schema, a.name, a.rec	1
00000011655553516987239397	SELECT IBM.TID, IBM.FID FROM S	1
00000003197575081402468776	SELECT POLICY FROM SYSTOOLS.PO	4
00000013972978427357447723	SELECT POLICY FROM SYSTOOLS.PO	1
00000001738579910634580323	SELECT POLICY FROM SYSTOOLS.PO	1
00000009178979855109711388	UPDATE SYSTOOLS.HMON_ATM_INFO	1
00000012160518308616047294	DELETE FROM SYSTOOLS.HMON_ATM	3

图 14.9 输入哈希值

按下回车键后就可以得到具体的语句，如图 14.10 所示。

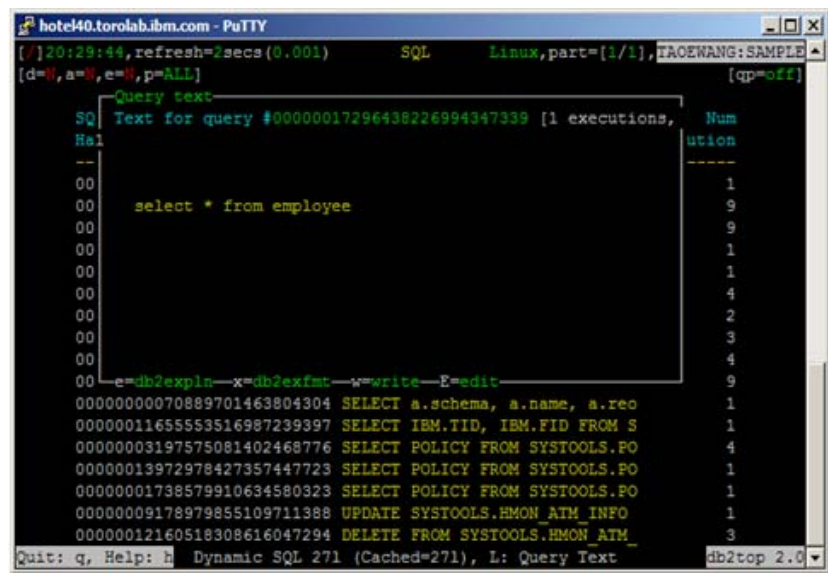


图 14.10 根据哈希值获得 SQL 语句

如果用户的数据库已经运行了 `sqllib/misc/EXPLAIN.DDL`，那么在这个窗口中按下【x】键就可以快速地得到当前语句的访问计划，如图 14.11 所示。

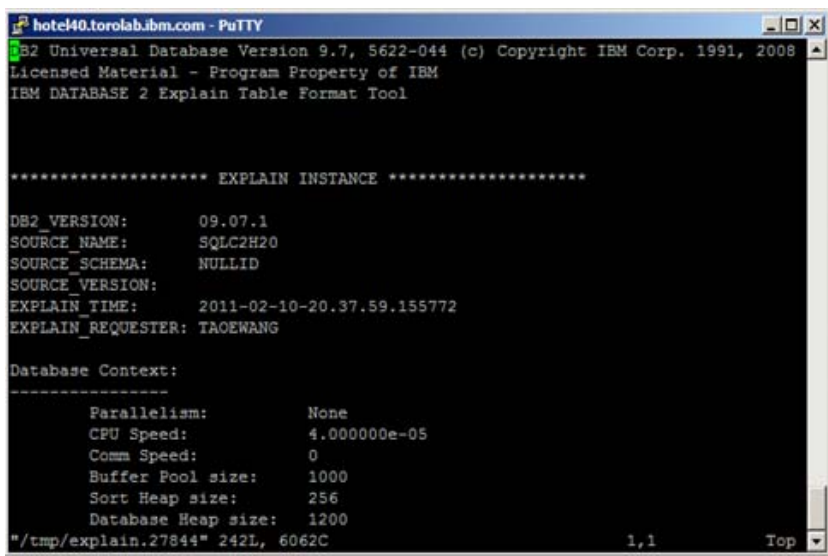


图 14.11 获得执行计划

4. 应用程序 (I)

启用程序子窗口显示系统中应用程序的列表，如图 14.12 所示。

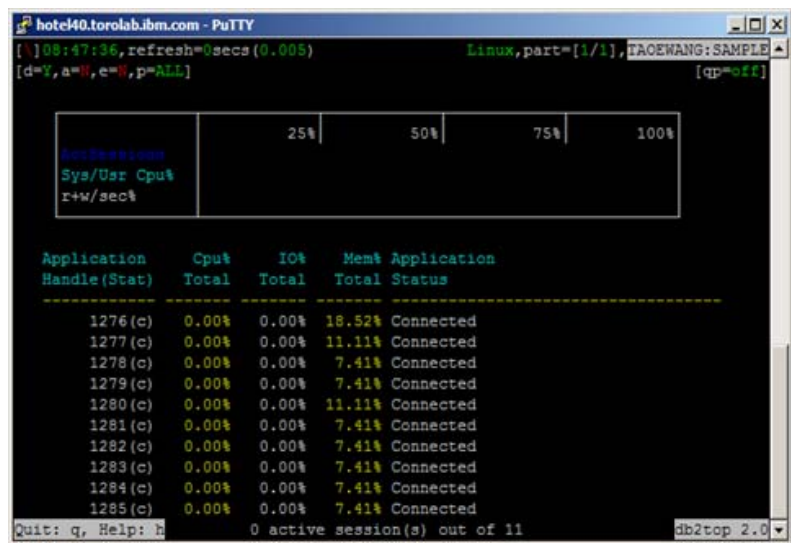


图 14.12 db2top 应用程序窗口

其中，第一列为应用程序句柄，第二列为 CPU，第三列为 I/O 和内存占用的百分比，第四列为应用程序状态，而应用程序名则在第五列，按键盘的左方向键移动列表即可。

在繁忙系统中，如果想要监测哪个应用程序占用大量的 CPU 或者 I/O 资源，可以通过应用程序子窗口迅速得到答案。通过结合该子窗口与动态 SQL，可以分析出当前使用高 CPU 的应用程序正在运行什么语句，然后可以进一步分析访问计划等。

5. 缓冲池（b）

命中率与逻辑物理读的数量是缓冲池子窗口所监控的重点，如图 14.13 所示。



图 14.13 db2top 缓存池窗口

对数据的具体分析请参阅有关性能监测与调优的章节,这里我们只是介绍 db2top 的主要功能。

6. 锁 (U)

按下【U】键就可以进入锁子窗口,如图 14.14 所示。

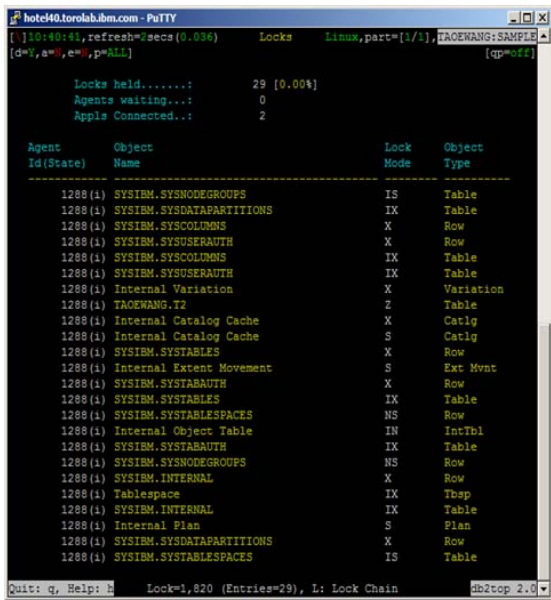


图 14.14 db2top 锁窗口

在一个繁忙的系统中,由于 db2top 不支持上下滚屏,所以可能无法看到所有的锁列表。因此,在处理锁的问题时,笔者建议通过 db2top 了解处于锁等待的应用程序,然后使用手工快照进行细致分析。

如果一定要使用 db2top,可以根据 Application Status 列排序,找到处于锁等待状态的应用,然后对应 Locked By 列找到其持有锁的应用程序。例如,当有两个应用程序:

```
App1:
db2 +c "create table t2 (c1 int)"
DB20000I The SQL command completed successfully.
db2 +c "insert into t2 values (1)"
DB20000I The SQL command completed successfully.

App2:
db2 +c "select * from t2 with RR"
```

这时我们看到应用程序 2 在锁等待。

然后通过 db2top 我们可以看到图 14.15 所示的界面。

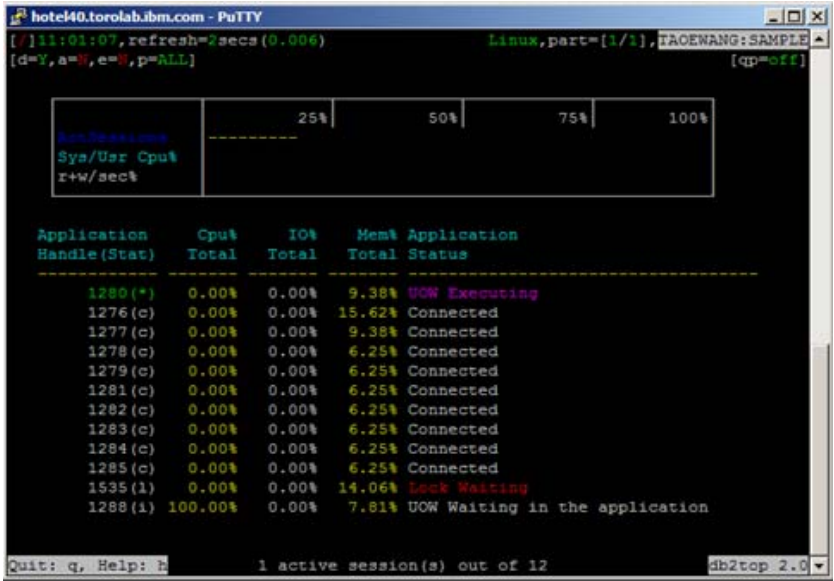


图 14.17 应用状态信息

其中 1288 为 UOW Waiting 状态，而 1535 为锁等待。也就是说，我们必须提交 1288 才能够释放锁，让 1535 获得。

7. 表 (T)

表子窗口显示了数据库自从启动以来所访问过的表，如图 14.18 所示。

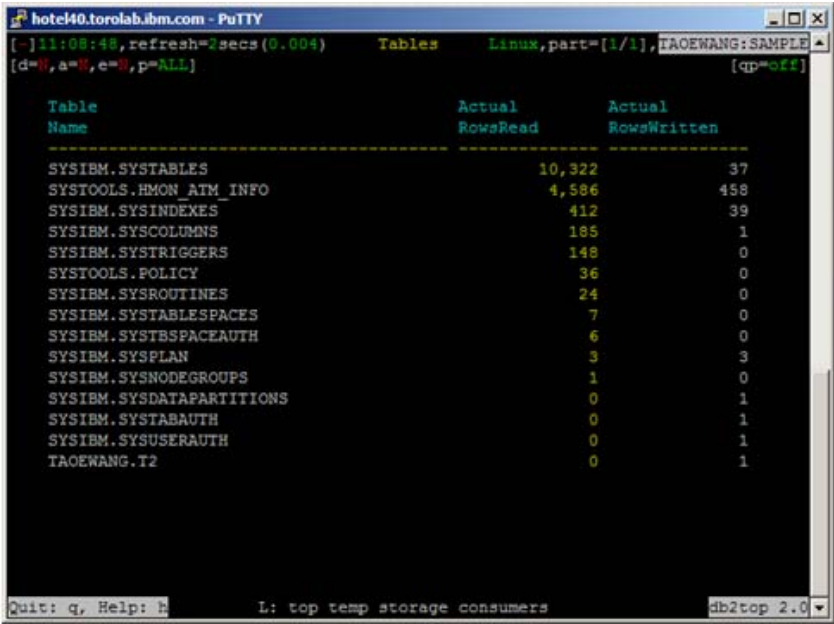


图 14.18 db2top 表窗口

按照第二列排序（z1）我们可以得到每个表所被读取的行数，而第三列则是每个表被写入的行数。

在运维中，用户可能想要统计每个表的大小，一般来说需要用表的 $NPAGE * PAGESIZE$ 计算。而 db2top 给出了更简单的监测方法，将每个表的大小直接显示在结果中，如图 14.19 所示。

Table Name	Table Size	Appl Handle	DB User
SYSIBM.SYSTABLES	600.0K	-	N/A
SYSIBM.SYSTOOLS.HMON_ATM_INFO	104.0K	-	N/A
SYSIBM.SYSINDEXES	296.0K	-	N/A
SYSIBM.SYSCOLUMNS	2.3M	-	N/A
SYSIBM.SYSTRIGGERS	72.0K	-	N/A
SYSIBM.SYSTOOLS.POLICY	40.0K	-	N/A
SYSIBM.SYSROUTINES	1.2M	-	N/A
SYSIBM.SYSTABLESPACES	88.0K	-	N/A
SYSIBM.SYSTBSPACEAUTH	104.0K	-	N/A
SYSIBM.SYSPLAN	304.0K	-	N/A
SYSIBM.SYSNODEGROUPS	56.0K	-	N/A
SYSIBM.SYSDATAPARTITIONS	144.0K	-	N/A
SYSIBM.SYSTABAUTH	216.0K	-	N/A
SYSIBM.SYSUSERAUTH	136.0K	-	N/A
TAOWANG.T2	8.0K	-	N/A

图 14.19 显示表的大小

14.5 DB2 事件监控器

事件监控器（Event Monitor）是当某个事件发生时记录信息的机制，比如死锁发生时，DB2 将死锁相关的信息写到表或文件中，便于事后的分析。与快照不同，事件监控器是基于事件的，通过 `CREATE EVENT MONITOR` 命令创建。事件监控器产生的数据量较大，会对原系统造成非常大的影响，因此在实际生产环境中的使用并不广泛，用得最多的就是用它来捕获死锁语句。事件监控器的创建语法如下：

```

CREATE EVENT MONITOR — event-monitor-name — FOR — event type | —
→ WRITE TO {
  TABLE — table options | — [MANUALSTART]
  PIPE — pipename | — [AUTOSTART]
  FILE — path-name | file options
}

```

可监控的事件类型包括 Database、Tables、Deadlocks、Statements、Tablespaces、Bufferpools、Connections 和 Transactions。监控结果可以输出到表、管道和文件，从性能的角度来看，建议写到文件。

事件监控器创建后，默认并不会自动启动，而需要通过 `SET event_monitor_name STATUS=1` 来激活。一旦激活，相关事件发生时就会记录。

如果将结果输出到文件系统，可以通过 `db2evmon` 解析数据结果：

```
db2evmon -path > event_monitor_target
```

我们曾经在 11.4.5 节中演示过死锁事件监控器的用法。

14.6 小结

本章主要介绍了 DB2 的监控工具，包括快照、`db2pd`、`db2top` 以及事件监视器。在日常监控时，快照和 `db2pd` 工具使用的最多，用户可编写脚本收集数据。`db2top` 的优点是图形界面，提供的统计数据比较直观，吸引了越来越多的人使用。事件监控器对性能的影响很大，一般只用于对死锁的监控和捕获。

本章是对 DB2 监控工具的介绍，至于在什么场景下使用，以及如何分析监控结果，我们将在下一章重点介绍。

14.7 判断题

(1) SNAPSHOT 可以用来获得某一时时间点的快照信息。

T: 正确

F: 错误

(2) `db2pd` 能够显示所有 SNAPSHOT 显示的内容。

T: 正确

F: 错误

(3) `db2pd` 对系统开销的影响远远小于 SNAPSHOT。

T: 正确

F: 错误

(4) `db2top` 可以用来实时监控系统。

T: 正确

F: 错误

(5) 在存在大量分区的系统中 (超过 100 个分区), 建议使用 `db2top` 的全局模式监控所有分区的状态。

T: 正确

F: 错误



性能监控和分析方法

性能监控在日常工作中必不可少，作为一名合格的 DBA，平日里对性能信息的收集与分析非常重要。能否在问题出现前发现苗头，或者性能下降时快速定位瓶颈，是衡量一个 DBA 工作能力的重要指标之一。

性能问题的症状一般体现在几个方面，响应时间慢、吞吐量低、对资源占用率高等。不同于传统意义上的问题诊断，当发生性能问题时，用户一般不会从数据库接收到任何错误信息，也就是说，没有任何错误信息可以帮助我们了解系统内部发生了什么。

因此，平日里收集大量的性能信息非常重要，当问题发生的时候，以平日收集的数据作为基准，将帮助 DBA 在最短的时间内，判断什么地方产生了瓶颈。

本章主要讲述性能监控和分析的方法，以及日常需要收集的重要性能数据。本章内容组织如下：

- 操作系统监控和数据收集。
- 数据库监控和数据收集。
- CPU/IO/内存瓶颈分析。
- 性能分析思路和方法。
- 5 个性能分析案例。

15.1 收集数据



从数据库的最终用户角度来看，CPU 使用率、内存使用率等都不重要，唯一重要的是查询

的响应时间。只要查询的响应时间在预期的范围之内，最终用户就不会认为系统存在性能问题。但是，从数据库管理员的角度看，查询的响应时间并不是唯一的指标。一般来说，性能分析应当从如下环节入手：

- 单位时间内的吞吐量（例如，每秒 1000 条查询交易，以及 300 条插入、更新、删除交易）。
- CPU 使用率。
- I/O 使用率。
- 内存占用率。

以上几点是在大层面上对一个系统的性能进行描述，但是仅仅通过这 4 点远不能有效地监控所有的性能指标，以及对未来可能发生的性能问题提供帮助。因此，我们要通过各种细节的指标来对各个系统子模块进行监控。

15.1.1 操作系统级别性能监控 ■ ■ ■

数据库作为一个软件运行在操作系统之上。在深入数据库模块之前，对整个操作系统的认识是至关重要的。

不管是 UNIX、Linux，还是 Windows，计算机系统的基础架构都是相通的。CPU 作为核心运算模块，从内存与 L2 缓存得到指令；内存作为主存储设备，负担着存储部分数据的任务；而磁盘作为永久存储设备，存放着大量的用户数据。因而，对于操作系统层面的性能监控应该着重于以下几个方面：

- CPU 使用率。
- I/O 使用率（包括网络 I/O）。
- 内存占用率。

在 UNIX/Linux 平台，提供了很多系统监控工具，如 `vmstat`、`iostat`、`top`、`ps` 等，而对于 Windows 来说，没有什么有效的命令行方法可以得到系统性能指标。作者建议使用微软的工具 `Process Explorer`（可以在微软的官方网站下载）。本文的示例中，Windows 部分的性能监测均使用 `PE` 工具。遗憾的是，`PE` 并不能保存完整的历史信息，因此，如果想要在 Windows 上监测并自动保存历史信息，可能需要借助一些第三方的工具。

注：本章中所有的命令行示例均在 AIX 系统下完成，对于其它操作系统请参见相关资料。

1. CPU 使用率

不同的操作系统之间监控 CPU 使用率的方法不尽相同。UNIX 与 Linux 的方法相近，而 Windows 则差别较大。

在 UNIX/Linux 系统中，`vmstat`、`lparstat` 与 `ps` 命令可以用来监视系统 CPU 使用情况。

```
/home/db2inst1 $ vmstat 1 5
```

System configuration: lcpu=8 mem=12288MB ent=0.40																			
kthr		memory				page				faults				cpu					
r	b	avm	fre	re	pi	po	fr	sr	cy	in	sy	cs	us	sy	id	wa	pc	ec	
3	0	2830247	282207	0	0	0	0	0	0	0	10	11735	6767	94	3	4	0	1.53	382.5
4	0	2830248	282205	0	0	0	0	0	0	0	11	15312	6836	90	3	7	0	1.45	362.6
0	0	2830192	282261	0	0	0	0	0	0	0	11	8981	6957	11	15	74	0	0.15	37.9
0	0	2830192	282260	0	0	0	0	0	0	0	7	6969	6224	8	14	78	0	0.13	32.6
11	0	2830192	282260	0	0	0	0	0	0	0	7	11474	5608	12	14	74	0	0.14	34.3

在上面的显示中，其中系统设置部分说明了该系统有 8 颗逻辑 CPU，内存为 12GB，而授权 CPU 为 0.4 颗，也就是说，LPAR 管理程序会保证该 LPAR 有最少 0.4 颗 CPU 的运算能力。

而在 vmstat 的主体，最后的 6 列分别代表的意义如下。

- us: 用户空间占用 CPU 百分比。
- sy: 内核空间占用 CPU 百分比。
- id: 空闲 CPU 百分比。
- wa: 等待输入/输出的 CPU 时间百分比。
- pc: 当前 LPAR 所用到的实际 CPU 运算量。
- ec: 对比 ent，当前 LPAR 的运算量所占百分比。

us 是指有多少 CPU 时间被消耗在用户程序中，这里的用户程序可以指数据库程序，也可以是其他任何非内核应用程序；

sy 是指有多少 CPU 时间被消耗在运行 UNIX/Linux 内核上，这里的内核可以是操作系统底层调用、进程优先队列计算、交换分区的读/写等；

id 是 CPU 既不在执行任何有效的指令，也不在等待 I/O 所消耗的时间；

wa 是指有多少 CPU 时间用来等待 I/O 指令的返回；

pc 是指当一个 LPAR 非常空闲的时候，根据管理程序的设置，管理程序有可能会将一部分 CPU 运算时间赋予另外的 LPAR 进行计算。但是根据 ent 的设置，该 LPAR 在最坏的情况下，系统也会保证有 0.4 的 CPU 运算资源“随用随到”。这样的话，us+sy+wa 所占的百分比是该 LPAR 真正使用到的 CPU 的百分比，也就是说，如果该 LPAR 非常空闲，而其中大部分 CPU 都被用在别的 LPAR 上，我们可能也会看到 us+sy+wa 占用的百分比很高。但是，这并不真正代表该 LPAR 占用过高的 CPU 资源。只有当 pc 列高于 ent 时，才说明该 LPAR 的 CPU 高于预期的最低值；

ec 列的计算很简单，就是用 pc 除以 ent 乘以 100 得到的结果。当 ec 列大于 100 的时候，需要注意的是 us、sy 与 wa 列。其中，us+sy 是该系统用户程序与操作系统消耗的总 CPU 量，而 wa 则是区分 I/O 瓶颈的一个重要指标。

一般来说，对于 UNIX/Linux 操作系统，尽量保持 CPU 使用率（us+sy+wa）在普通工作负载中维持在 50%以下，在高峰期保持在 90%以下。

vmstat 可以有效地监测系统的 CPU 使用率，但是如果用户希望获得进程的 CPU 使用情况，就要通过 top (topas) 或者 ps 命令。top 命令是实时监控程序，很难保存历史记录。因此，我们重点介绍 ps 命令，作为一个轻量级的命令，ps 可以提供非常多的信息：

```
/home/db2inst1 $ ps -elf | sort +5 -rn | more
  40411 A agrankin 11616526 12546314 120 60 20 1437f0c400 16892      * 18:29:17
- 3:21 db2sysc 2
  200001 A db2inst1 22073668 11706822 120 120 20 16af1aa400 106704      08:43:33
pts/43 0:00 ps -elf
  40401 A hirosojp 13754514 13811864 11 60 20 11af06b400 44732      * Jan 08
- 74:22 db2sysc 0
  240001 A hirosojp 12701870 12107938 8 60 20 17051c1400 14480      * Jan 08
- 35:47 ca-mgmt-lwd -i128 -p31138 -kb552547a -d/home/hirosojp/sqlllib/db2dump
-e/home/hirosojp/sqlllib/cf/ca-server -f
  240001 A hirosojp 12800140 12705868 7 60 20 14b252c400 14480      * Jan 08
- 35:39 ca-mgmt-lwd -i129 -p31139 -kb552557a -d/home/hirosojp/sqlllib/db2dump
-e/home/hirosojp/sqlllib/cf/ca-server -f
  40401 A walkerj2 20263266 20369668 3 60 20 13324cd400 109868      * 04:48:04
- 0:22 db2sysc
  200001 A db2inst1 22417772 11706822 3 61 20 15a4168400 248 f10001100cfdab20
08:43:33 pts/43 0:00 sort +5 -rn
  40401 A hirosojp 14278840 13742320 2 60 20 109f827400 24324      * Jan 08
- 30:23 db2sysc 1
  40401 A tonowak 1798396 1867790 2 60 20 13a8ceb400 20232      * Jan 07
- 6:36 db2sysc 0
  40401 A agrankin 12607886 12439970 2 60 20 106001a400 16692      * 18:29:17
- 3:28 db2sysc 3
  40401 A billpeck 59019366 59080720 2 60 20 155f957400 19400      * Jan 10
- 6:47 db2sysc
  40401 A mcorish 2674790 3112974 2 60 20 1229c8a400 23556      * Jan 07
- 29:52 db2sysc 0
  200001 A db2inst1 22266126 11706822 2 61 20 112fa4b400 244 f10001100ae6e388
08:43:33 pts/43 0:00 pg -n
  240001 A db2inst1 11706822 11567414 2 61 20 1331acd400 864      17:11:13
pts/43 0:00 -ksh
  40001 A jchwong 6148432 6103382 1 60 20 11b726d400 968      10:44:01
- 0:30 db2fcmdm 0
  40001 A jchwong 6173142 6078766 1 60 20 162918a400 956      10:44:00
- 0:30 db2fcmdm 1
  40401 A shenh 6320154 6406204 1 60 20 12dd2b7400 16332      * Jan 07
- 8:29 db2sysc 3
  40401 A shenh 6336554 6414400 1 60 20 10b782d400 16152      * Jan 07
- 8:23 db2sysc 2
  40401 A mariaj 19898710 14717236 1 60 20 10e4238400 17088      * 05:11:37
```

```
- 1:25 db2sysc
40401 A hirosojp 13856770 14004456 1 60 20 10c8c32400 24308 * Jan 08
- 29:59 db2sysc 2
```

以上显示了部分 ps 的输出，其中第 6 列为 C 列，用来表示当前进程所消耗的 CPU。如果 C 列很高，说明该进程消耗大量的 CPU 资源。这个数值在不同的操作系统中意义略有差别。在示例 AIX 环境中，该列代表 CPU 使用的计数器，每次系统时钟跳动的时候，如果发现该进程正在执行，则 C 列加一，然后该计数器每秒递减除二。

在 AIX 系统中，wait 进程是一类比较特殊的进程，在系统空闲的时候用来填充 CPU 时间。所以，如果用户看到 ps 列表中 wait 占用大量的 CPU 不必惊慌，这说明系统 CPU 使用率低，是正常现象。在 UNIX/Linux 操作系统中，vmstat 与 ps 是监测 CPU 使用的重要工具。

在 Windows 中，不存在 vmstat 与 ps 命令，PE 的系统信息部分就是最重要的参考数据，如图 15.1 和图 15.2 所示。

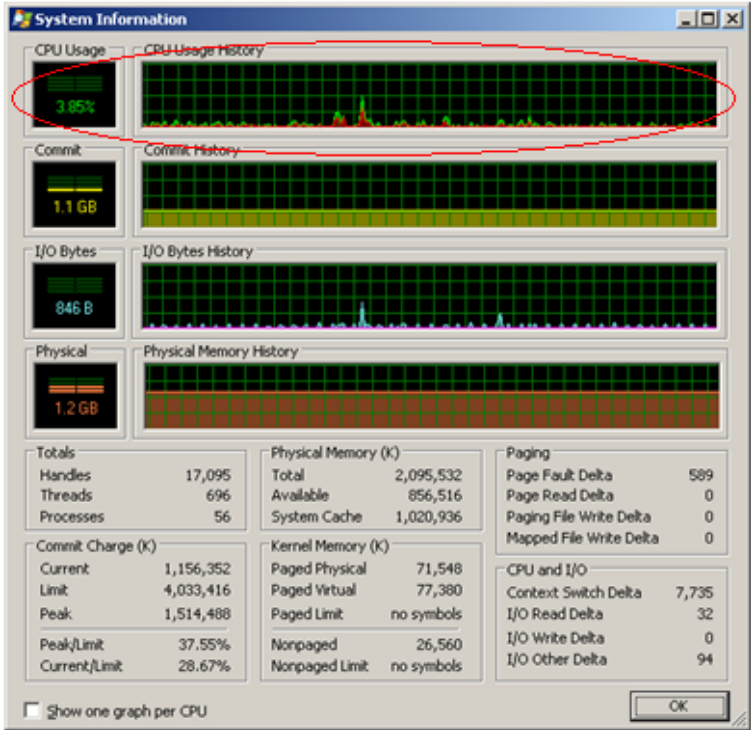


图 15.1 系统信息

在图 15.2 中，整体的系统 CPU 消耗大概在 5% 以下。其中 94.62% 被 System Idle Process 占用。该进程在系统空闲时被调用，代表空闲 CPU 使用率。其余的进程为用户进程，其中 procexp.exe 占用 1.54%，firefox.exe 浏览器占用 1.54%。

通过这些数据，用户可以迅速定位消耗 CPU 最多的进程。如果发现 DB2 进程消耗过多的 CPU，则可以通过下一章讲解的方法进行优化。

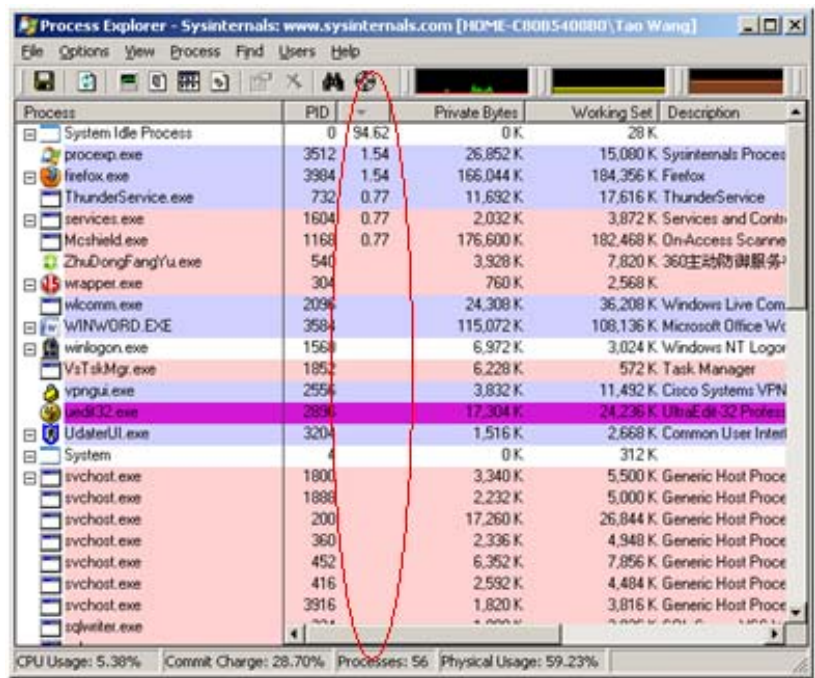


图 15.2 cpu 消耗

2. I/O 使用率

在 UNIX/Linux 平台，在 `vmstat` 输出结果中，`wa` 列表示有多少 CPU 时间用于 I/O 等待。一般来说，I/O 等待越少越好。可是，在某些数据仓库系统中，当大量的数据无法完全装载入内存的情况下，I/O 是不可避免的。这种情况下，跟踪所有磁盘的使用情况就至关重要了。

在 UNIX/Linux 中，`iostat -D` 选项可以用来收集磁盘 I/O 信息：

```
/home/db2inst1 $ iostat -D 1 2

System configuration: lcpu=8 drives=4 paths=4 vdisks=1

hdisk3      xfer: %tm_act    bps      tps      bread    bwrtn
            0.0      0.0      0.0      0.0      0.0
read:       rps  avgserv  minserv  maxserv  timeouts  fails
            0.0      0.0      0.0      0.0      0      0
write:      wps  avgserv  minserv  maxserv  timeouts  fails
            0.0      0.0      0.0      0.0      0      0
queue:     avgtime  mintime  maxtime  avgwqsz  avgsqsz  sqfull
            0.0      0.0      0.0      0.0      0.0      0.0
hdisk1      xfer: %tm_act    bps      tps      bread    bwrtn
            0.0      0.0      0.0      0.0      0.0
read:       rps  avgserv  minserv  maxserv  timeouts  fails
```

hdisk0		0.0	0.0	0.0	0.0	0	0
	write:	wps	avgserv	minserv	maxserv	timeouts	fails
		0.0	0.0	0.0	0.0	0	0
	queue:	avgtime	mintime	maxtime	avgwqsz	avgsqsz	sqfull
		0.0	0.0	0.0	0.0	0.0	0.0
	xfer:	%tm_act	bps	tps	bread	bwrtn	
		0.0	0.0	0.0	0.0	0.0	
	read:	rps	avgserv	minserv	maxserv	timeouts	fails
		0.0	0.0	0.0	0.0	0	0
	write:	wps	avgserv	minserv	maxserv	timeouts	fails
hdisk2		0.0	0.0	0.0	0.0	0	0
	queue:	avgtime	mintime	maxtime	avgwqsz	avgsqsz	sqfull
		0.0	0.0	0.0	0.0	0.0	0.0
	xfer:	%tm_act	bps	tps	bread	bwrtn	
		0.0	0.0	0.0	0.0	0.0	
	read:	rps	avgserv	minserv	maxserv	timeouts	fails
		0.0	0.0	0.0	0.0	0	0
	write:	wps	avgserv	minserv	maxserv	timeouts	fails
		0.0	0.0	0.0	0.0	0	0
	queue:	avgtime	mintime	maxtime	avgwqsz	avgsqsz	sqfull
hdisk3		0.0	0.0	0.0	0.0	0.0	0.0
	xfer:	%tm_act	bps	tps	bread	bwrtn	
		0.0	0.0	0.0	0.0	0.0	
	read:	rps	avgserv	minserv	maxserv	timeouts	fails
		0.0	0.0	0.0	0.0	0	0
	write:	wps	avgserv	minserv	maxserv	timeouts	fails
		0.0	0.0	0.0	0.0	0	0
	queue:	avgtime	mintime	maxtime	avgwqsz	avgsqsz	sqfull
		0.0	0.0	0.0	0.0	0.0	0.0
	xfer:	%tm_act	bps	tps	bread	bwrtn	
hdisk1		0.0	0.0	0.0	0.0	0.0	
	read:	rps	avgserv	minserv	maxserv	timeouts	fails
		0.0	0.0	0.0	0.0	0	0
	write:	wps	avgserv	minserv	maxserv	timeouts	fails
		0.0	0.0	0.0	0.0	0	0
	queue:	avgtime	mintime	maxtime	avgwqsz	avgsqsz	sqfull
		0.0	0.0	0.0	0.0	0.0	0.0
	xfer:	%tm_act	bps	tps	bread	bwrtn	
		0.0	0.0	0.0	0.0	0.0	
	read:	rps	avgserv	minserv	maxserv	timeouts	fails
hdisk0		0.0	0.0	0.0	0.0	0	0
	write:	wps	avgserv	minserv	maxserv	timeouts	fails
		0.0	0.0	0.0	0.0	0	0
	queue:	avgtime	mintime	maxtime	avgwqsz	avgsqsz	sqfull

hdisk2	queue:	avgtime	mintime	maxtime	avgwqsz	avgsqsz	sqfull
		0.0	0.0	0.0	0.0	0.0	0.0
	xfer:	%tm_act	bps	tps	bread	bwrtn	
		0.0	0.0	0.0	0.0	0.0	
	read:	rps	avgserv	minserv	maxserv	timeouts	fails
		0.0	0.0	0.0	0.0	0	0
	write:	wps	avgserv	minserv	maxserv	timeouts	fails
		0.0	0.0	0.0	0.0	0	0
	queue:	avgtime	mintime	maxtime	avgwqsz	avgsqsz	sqfull
		0.0	0.0	0.0	0.0	0.0	0.0

在以上输出中, `iostat` 把系统中的每一个磁盘都标识出来, 其中的 `tm_act` 指的是该驱动器占用率的百分比, 剩下的一些 `read/write` 相关信息就是读/写的速率。其中对于读写操作来说, `avgserv` 列代表着平均单个读写的时间。对于通常的存储设备来说, 如果单个 I/O 平均时间超过 10 毫秒就意味着 I/O 性能相对较低。

如果发现 I/O 异常, 可结合数据库监控找到引起 I/O 的“热表”, 并进行相应的优化。

在 Windows 中, PE 依然可以用来监测进程的 I/O。

首先, 在 `View→Select Columns` 中选择相应的 I/O 列, 如图 15.3 所示。

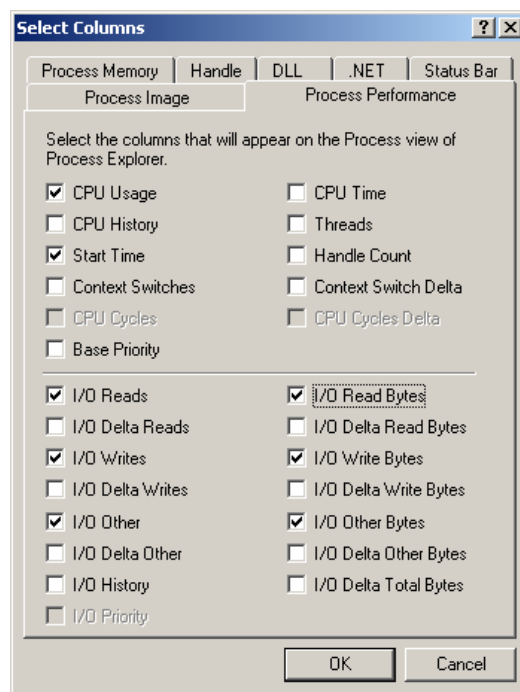


图 15.3 I/O 设置

单击 `OK` 按钮之后就会发现 I/O 相关的信息也显示出来了, 如图 15.4 所示。

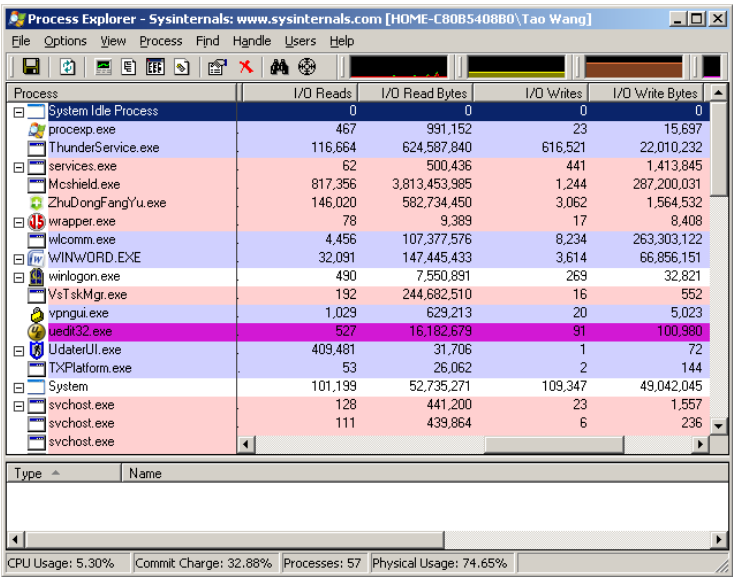


图 15.4 I/O 监控

3. 内存占用率

内存在计算机系统中至关重要，我们这里讨论的内存指虚拟内存(virtual memory)，包括物理内存(physical memory)与交换空间(swap space)。

从操作系统的角度来看，物理内存就相当于它的大脑中的记忆体，而交换空间则相当于记事本。当记忆体无法存放所有的内存请求时，它会根据某种算法将一部分内存放入交换空间，即 swapping。

相对于正常的物理内存访问，从磁盘的交换空间中读取数据会导致成百上千倍的性能下降。而当所有的物理内存与交换空间都被耗尽时，操作系统则会自动杀死一些进程以保证有足够的内存空间运行。如果操作系统根本来不及杀死足够多的进程，导致虚拟内存耗尽的时候，操作系统就会死机，需要重新启动系统。

由此可见，对于内存的监控不仅对性能，甚至对系统的可用性都至关重要。系统管理员与 DBA 一定要经常监视系统的内存使用情况，最好创建一些程序进行实时监控，以保证系统的 24x7 正常运行。

UNIX/Linux 平台，vmstat 仍然是一个非常好用的内存监控工具：

```
/home/db2inst1 $ vmstat 1 5

System configuration: lcpu=8 mem=12288MB ent=0.40

kthr    memory                page            faults          cpu
-----
r  b    avm  fre  re  pi  po  fr  sr  cy  in  sy  cs  us  sy  id  wa    pc    ec
```

1	0	2832106	288921	0	0	0	0	0	0	3	114384	208894	14	51	35	0	1.17	291.9
8	0	2832127	288900	0	0	0	0	0	0	3	114068	194184	12	59	28	0	1.81	452.0
1	0	2832106	288917	0	0	0	0	0	0	8	105946	207901	11	56	33	0	1.39	348.2
0	0	2832106	288917	0	0	0	0	0	0	1	110026	212747	14	49	37	0	1.08	270.1
1	0	2832106	288917	0	0	0	0	0	0	5	109269	211371	14	49	37	0	1.08	269.8

这里着重强调的部分是第三列和第四列，分别为 **avm** 和 **fre**。

- **avm** 是指当前系统中已经激活的虚拟内存页的数量。该数值并不包含文件系统缓存。
- **fre** 表示系统中平均空闲页的数量，每个页面的单位为 4096 字节。操作系统在内核中维护一个空闲页列表，在 AIX 系统，该列表中最少的页面数量可以由 **minfree** 参数指定。每当应用程序结束时，所有的工作页面都会被立刻返回给空闲列表。而文件系统缓存则驻留内存，并不会返还给空闲列表，除非它们被虚拟内存管理器盗取。因此，**fre** 并不能完全代表系统中可用的空闲内存。如果一个内存页被一个应用程序请求，那么最先分配的是其他已经结束的程序所占用的文件系统内存。

但是，如果用户想知道系统中到底有多少内存可以使用呢？可以使用 **svmon -G** 命令：

```
/home/db2inst1 $ svmon -G
```

	size	inuse	free	pin	virtual	mmode
memory	3145728	2976748	168980	929584	2846021	Ded
pg space	3145728	142373				
	work	pers	clnt	other		
pin	703601	0	0	225983		
in use	2760902	1398	214448			

在这个结果中，**clnt** 与 **in use** 交叉的那一项，也就是 214448，代表的就是有多少内存被文件系统使用。一般来说，这个值加上 **free** 列所对应的 168980，就可以初步认为是该系统中可以被应用程序所使用的内存。当然，具体深入的理解还要根据 **vmo** 的一些参数，在本文中我们就不一一阐述了，请参见相关资料。

刚才我们谈到了内存交换机制。如果操作系统的物理内存被耗尽，那么系统就会把一些内存放入交换空间，这种操作会大大降低系统的性能。通过 **vmstat** 的数据，我们同样也可以看到相应的信息。其中的第六列和第七列（**pi** 与 **po**）就是关于交换空间的 I/O 指标。在生产系统中，一般情况下，这两列应该永远为 0，表示没有任何内存交换。如果用户发现持续的交换发生，那么应该检查当前内存的使用量。

在 Windows 中，PE 依然可以用来监控系统的内存使用情况，如图 15.5 和图 15.6 所示。

其中的 **Working Set** 列代表该进程正在占用多少物理内存，而 **Private Bytes** 是私有内存的大小，可以由 **malloc** 等函数分配，而动态链接库所占用的内存空间不计入 **Private Bytes**，**Virtual Size** 是该进程所分配的虚拟内存大小，需要注意的是，进程虚拟内存占用的大小不一定和真正使用到的内存相等。一个应用程序可以分配一块很大的虚拟内存空间，但是真正使用的是其中有限的一部分，而不会造成内存的过高使用。

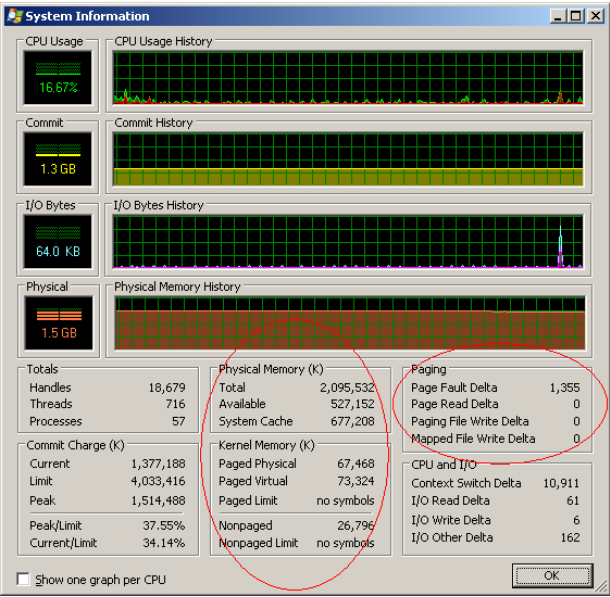


图 15.5 内存信息

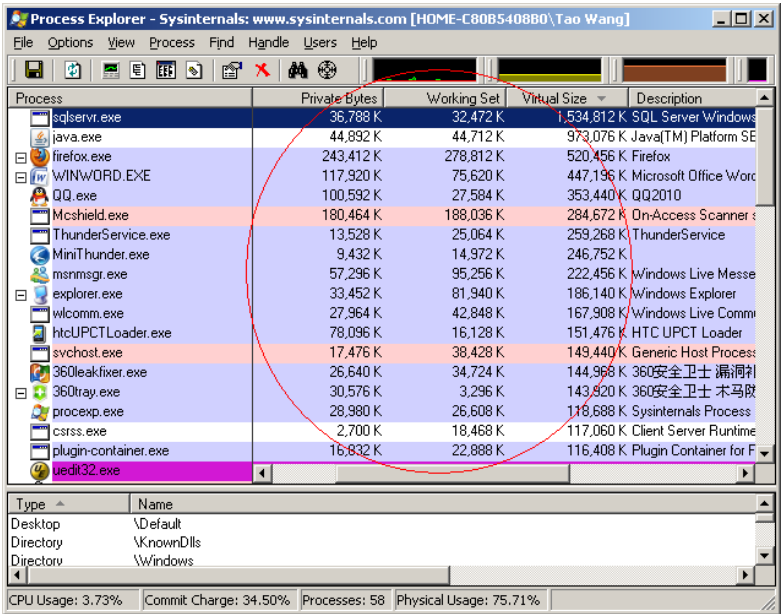


图 15.6 内存监控

4. 小结

在性能监控与性能分析中，操作系统级别的指标是首先应该了解的。只有当了解整体系统的状况（例如多少个 CPU、是否使用 LPAR、多少内存占用率、磁盘占用率等）后，才能更好地了解应该从什么方向入手分析特定的性能问题。

从操作系统层面，我们需要从 CPU、I/O 与内存 3 个方面入手。其中 `vmstat` 可以用来在高层面监测系统 CPU、I/O 与内存；`iostat` 可以用来专门监控 I/O。其它很多命令在是某些系统中独有的，例如 `svmon` 是 AIX 专有的；而在 linux 中，`free` 命令可以获得更加详细的内存使用情况。

15.1.2 数据库级别性能监控 ■ ■ ■

在了解了系统的运行情况之后，就要深入到数据库级别的监控。首先，从 DB2 的架构来看，一台主机上可以存在多个 DB2 版本，一台主机上可以运行多个实例，每一个实例可以跨越多个分区，每一个实例可以包含多个数据库。一个机器的全部资源都用于 1 个数据库，还是被 10 个数据库共享，分析步骤有很大的区别。因此，数据库级别分析的第一步就是了解整个系统的架构。

首先，通过 `db2greg -dump` 查看一台服务器安装了几个 DB2 版本：

```
/home/db2inst1 $ db2greg -dump | more
S,DB2,9.5.0.4,/opt/ibm/db2/V9.5,-,,4,0,a,1263345540,0
S,DB2,9.1.0.6,/opt/ibm/db2/V9.1,-,,6,0,a,1263349562,1
S,DB2,9.7.0.3,/opt/ibm/db2/V9.7,-,,0,0,a,1263394520,1
V,DB2GPRF,DB2INSTDEF,db2inst1,/opt/ibm/db2/V9.5,
I,DB2,9.7.0.3,db2inst1,/home/db2inst1/sqllib,,1,/opt/ibm/db2/V9.7,,
I,DB2,9.5.0.4,db2inst2,/home/db2inst2/sqllib,,1,,/opt/ibm/db2/V9.5,,
```

通过以上结果可知，本机安装了三个不同版本，分别是 9.5 Fixpack 4，9.1 Fixpak 6 与 9.7 Fixpack 3；有两个实例，第一个实例是在 9.7 Fix-pack 3 版本创建的 `db2inst1` 实例，实例目录位于 `/home/db2inst1/sqllib`；第二个实例是在 9.5 Fixpack 6 版本创建的 `db2inst1` 实例，实例目录位于 `/home/db2inst2/sqllib`。

其中，实例的信息最为重要，因为只有存在实例，才能够真正使用数据库。所以下一步要做的事，就是进入每个实例进行详细的数据收集。在此，以 `db2inst1` 为例进行说明。对于其它实例，请用同样的方法分析

当确定了需要分析的实例后，首先通过刚开始收集的 `ps` 输出大体了解该实例所消耗的资源。不同的 DB2 版本，分析方法略有不同。为了简单起见，本章节使用 9.7 单分区数据库作为示例，使用多分区的读者可以使用 `db2_all` 命令收集全部分区信息。

DB2 不同的版本间，分析方法略有不同。为了简单起见，本章节使用 `V9.7` 作为示例。如果其他版本需要的方法有显著不同将会着重说明，同时本章节示例使用单分区数据库，如果使用多分区，读者可以使用 `db2_all` 命令将所有收集信息的指令发送到全部分区。

由于实例运行在实例用户下，也就是说 DB2 进程的拥有者为 `db2inst1`。那么通过 `ps -elf | grep db2inst1 | grep db2` 就能得到实例用户名下所有 DB2 相关进程的列表：

```
(db2inst1@db2host1) /home/db2inst1 $ ps -elf | grep db2inst1
40401 A db2inst1 32518574 32522638 3 60 20 13598d7400 20956 * 20:55:42
- 0:43 db2sysc
```

```

240001 A db2inst1 32686422      1  0  60 20 1087c20400  4900      20:55:37
- 0:00 /home/db2inst1/sqllib/bin/db2bp 11706822A21406 5 A
240001 A db2inst1 32706866 32518574  0  60 20 17ed3fa400  2300 f10001100d6b2980
20:55:44 - 0:00 db2vend (PD Vendor Process - 258)
242001 A db2inst1 32784862 32522638  0  60 20 1742dd0400  9348      * 20:55:45
- 0:00 db2acd

```

在这个输出结果中，db2sysc 进程为实例的主进程，用来运行所有的 SQL 请求。

注意：在 DB2 8/9.1 非 Windows 平台中，DB2 使用进程模型而非线程模型。因此，每一个 DB2 代理都有其单独的进程，所以在非 Windows 平台的 DB2 8/9.1 中，上面的命令将会显示更多的 DB2 进程。

前文中已经提到，第 6 列 C 列作为 CPU 计数器，数字越高说明近期所占用的 CPU 就越高。在上面的输出中，db2sysc 的 C 列为 3，数字并不是很大，也就是说在最近的几秒钟，该进程并没有大量消耗 CPU 资源。

在了解了该实例所消耗的总 CPU 之后，就要使用一些 DB2 命令收集详细信息了。与很多人所认为的不同，收集信息并不是越多越好，因为可能对系统造成很大的负担，尤其是一些 trace。除非用户真正了解 trace 给系统带来的负担，或者在专家的指导下，否则在生产系统中应该尽量避免使用操作系统级，或者数据库级别的 trace。

对于一般的性能监控，我们需要收集如下类别的信息：

- 实例级快照。
- 对于每个数据库的数据库级快照。
- 对于每个数据库的内存使用信息。

在实例创建以后，作者建议在实例级尽可能打开所有的快照开关，否则收集的快照信息将不完整：

```

(db2inst1@db2host1) /home/db2inst1 $ db2 update dbm cfg using DFT_MON_BUFPOOL on
DFT_MON_LOCK on DFT_MON_SORT on DFT_MON_STMT on DFT_MON_TABLE on DFT_MON_UOW on
DB20000I The UPDATE DATABASE MANAGER CONFIGURATION command completed successfully.
(db2inst1@db2host1) /home/db2inst1 $ db2 get dbm cfg | grep -i dft_mon
Buffer pool                (DFT_MON_BUFPOOL) = ON
Lock                       (DFT_MON_LOCK) = ON
Sort                       (DFT_MON_SORT) = ON
Statement                  (DFT_MON_STMT) = ON
Table                      (DFT_MON_TABLE) = ON
Timestamp                  (DFT_MON_TIMESTAMP) = ON
Unit of work                (DFT_MON_UOW) = ON

```

下面我们将针对每种类别进行详细说明。

1. 实例级快照

实例级快照是针对该实例的一个总体说明，可以使用如下命令收集：

- db2 get snapshot for database manager。
- db2pd -edus。

```
(db2inst1@db2host1) /home/db2inst1 $ db2 get snapshot for database manager

Database Manager Snapshot

Node type                        = Database Server with local clients
Instance name                    = db2inst1
Number of database partitions in DB2 instance = 1
Database manager status          = Active

Product name                     = DB2 v9.7.0.3
Service level                    = s101006 (IP23212)

Private Sort heap allocated      = 0
Private Sort heap high water mark = 0
Post threshold sorts             = 0
Piped sorts requested            = 0
Piped sorts accepted             = 0

Start Database Manager timestamp = 01/12/2011 22:44:08.492955
Last reset timestamp              =
Snapshot timestamp                = 01/12/2011 22:44:16.331712

Remote connections to db manager = 0
Remote connections executing in db manager = 0
Local connections                 = 0
Local connections executing in db manager = 0
Active local databases            = 0

High water mark for agents registered = 1
Agents registered                  = 1
Idle agents                       = 0

Committed private Memory (Bytes) = 2228224

Switch list for db partition number 0
Buffer Pool Activity Information (BUFFERPOOL) = ON 01/12/2011 22:44:08.492955
Lock Information                  (LOCK) = ON 01/12/2011 22:44:08.492955
Sorting Information               (SORT) = ON 01/12/2011 22:44:08.492955
SQL Statement Information         (STATEMENT) = ON 01/12/2011 22:44:08.492955
Table Activity Information        (TABLE) = ON 01/12/2011 22:44:08.492955
Take Timestamp Information        (TIMESTAMP) = ON 01/12/2011 22:44:08.492955
Unit of Work Information          (UOW) = ON 01/12/2011 22:44:08.492955

Agents assigned from pool         = 0
Agents created from empty pool    = 2
Agents stolen from another application = 0
High water mark for coordinating agents = 1
```

```

Hash joins after heap threshold exceeded = 0
OLAP functions after heap threshold exceeded = 0

Total number of gateway connections      = 0
Current number of gateway connections    = 0
Gateway connections waiting for host reply = 0
Gateway connections waiting for client request = 0
Gateway connection pool agents stolen    = 0

Node FCM information corresponds to      = 0
Free FCM buffers                        = 128
Total FCM buffers                      = 128
Free FCM buffers low water mark         = 128
Maximum number of FCM buffers           = 8192
Free FCM channels                      = 128
Total FCM channels                     = 128
Free FCM channels low water mark        = 128
Maximum number of FCM channels          = 8192

Memory usage for database manager:

Node number                            = 0
  Memory Pool Type                      = Other Memory
    Current size (bytes)                 = 10813440
    High water mark (bytes)              = 10813440
    Configured size (bytes)              = 48365568

Node number                            = 0
  Memory Pool Type                      = FCMBP Heap
    Current size (bytes)                 = 851968
    High water mark (bytes)              = 851968
    Configured size (bytes)              = 851968

Node number                            = 0
  Memory Pool Type                      = Database Monitor Heap
    Current size (bytes)                 = 65536
    High water mark (bytes)              = 65536
    Configured size (bytes)              = 393216

```

以下是对实例快照文件各部分内容的解析。

首先注意该快照文件是针对哪个实例。

```

Instance name      = db2inst1
Product name       = DB2 v9.7.0.3
Service level      = s101006 (IP23212)

```

其次，快照时间也是很重要的一环。在分析性能问题的时候，对时间的概念是非常重要的。只有当明确了问题从几点几分开始发生，我们才能够通过对比好、坏情况的数据来判断可能的

瓶颈:

```
Start Database Manager timestamp      = 01/12/2011 22:44:08.492955
Last reset timestamp                  =
Snapshot timestamp                    = 01/12/2011 22:44:16.331712
```

接着是当前实例有多少应用连接。在这个例子里面, 该实例在 8 秒钟之前启动, 并且没有任何应用连接到数据库:

```
Remote connections to db manager      = 0
Remote connections executing in db manager = 0
Local connections                     = 0
Local connections executing in db manager = 0
Active local databases                = 0
```

接着是 FCM 相关信息, 使用多分区数据库的读者可以关注:

```
Node FCM information corresponds to    = 0
Free FCM buffers                       = 128
Total FCM buffers                      = 128
Free FCM buffers low water mark        = 128
Maximum number of FCM buffers          = 8192
Free FCM channels                     = 128
Total FCM channels                    = 128
Free FCM channels low water mark       = 128
Maximum number of FCM channels         = 8192
```

总的来说, 实例级快照主要是了解一下当前实例下有多少应用程序在执行, 实例启动了多长时间等。

在 DB2 9.5 之后, 数据库由进程模型演变为线程模型, 无法通过 `ps` 命令获得某一个代理使用的 CPU 时间, 可以使用 `db2pd -edus` 功能, 即列出当前实例下所有的 EDU:

```
(db2inst1@db2host1) /home/db2inst1 $ db2pd -edus

Database Partition 0 -- Active -- Up 1 days 00:01:32 -- Date 01/13/2011 22:45:40

List of all EDUs for database partition 0

db2sysc PID: 32780714
db2wdog PID: 32940506
db2acd PID: 32662006

EDU ID      TID          Kernel TID      EDU Name                      USR (s)
SYS (s)
=====
=====
9921        9921        125800525      db2agent (idle)               0.074223    0.009299
9664        9664        125882535      db2agntdp (SAMPLE )          0.217577    0.102190
9407        9407        120754391      db2throt                      0.006866    0.005147
```

9140	9140	124420269	db2agent (idle)	0.847119	0.391740
7859	7859	123863173	db2agent (SAMPLE)	0.373582	0.385199
7070	7070	107466997	db2agent (instance)	0.365549	0.199014
6813	6813	107098113	db2agent (SAMPLE)	5.074135	2.068768
6555	6555	105291911	db2evmli (DB2DETAILDEADLOCK)	0.018135	0.036966
4756	4756	107462771	db2fw1 (SAMPLE)	0.017193	0.037044
4499	4499	107294807	db2fw0 (SAMPLE)	0.016580	0.035801
4242	4242	105492541	db2pfchr (SAMPLE)	0.000117	0.000076
3985	3985	105529435	db2pfchr (SAMPLE)	0.000877	0.001700
3728	3728	105508987	db2pfchr (SAMPLE)	0.013284	0.049027
3471	3471	111489275	db2pclnr (SAMPLE)	0.016963	0.025738
3214	3214	114241569	db2dlock (SAMPLE)	0.337910	0.127040
2957	2957	113660029	db2lfr (SAMPLE)	0.000381	0.002645
2700	2700	114061411	db2loggw (SAMPLE)	0.986922	1.300911
2443	2443	106582097	db2loggr (SAMPLE)	555.510634	294.491338
2186	2186	107012181	db2logmgr (SAMPLE)	0.376233	0.440577
1673	1673	105422963	db2logts (SAMPLE)	0.031382	0.004762
8583	8583	111149181	db2lused (SAMPLE)	2.998484	3.351555
8288	8288	107999235	db2wlmd (SAMPLE)	4.568472	1.804187
8031	8031	81498253	db2taskd (SAMPLE)	3.903978	4.021501
1801	1801	100179991	db2stmm (SAMPLE)	6.745248	3.529151
1286	1286	105766985	db2resync	0.003240	0.009593
1029	1029	107491427	db2ipccm	0.433778	0.296067
772	772	106848351	db2licc	0.000542	0.001934
515	515	106516547	db2thcln	0.001530	0.000490
2	2	106459255	db2alarm	0.744841	0.675089
258	258	105545813	db2sysc	58.406570	29.840094

其中，EDU ID 就是每一个代理线程的 ID；EDU Name 是与 DB2 8/9.1 进程名一样的代理线程名称；最后两列是该线程所消耗 CPU 的多少，等同于 `ps -elf` 命令下的 TIME 列。

参考 `db2pd -edus` 的输出，用户可以了解实例中哪个 EDU 消耗最多的 CPU。

2. 数据库级快照

比起实例快照，数据库的快照包含更多有价值的信息。我们所说的数据库快照不仅仅包含 database snapshot，也包含其它应用快照、语句快照等信息，主要包括以下 6 种：

- 数据库快照：db2 get snapshot for database on <dbname>。
- 应用快照：db2 get snapshot for applications on <dbname>。
- 表空间快照：db2 get snapshot for tablespaces on <dbname>。
- 表快照：db2 get snapshot for tables on <dbname>。
- 动态 SQL 快照：db2 get snapshot for dynamic sql on <dbname>。
- db2pd：db2pd -db <dbname> -locks -trans -app。

这 6 种快照基本上可以涵盖大部分性能分析的需求以下将分别阐述每种快照包含的数据含义：

(1) 数据库快照

数据库快照，即 database Snapshot。数据库快照从宏观了对整个数据库的运行情况做了一个统计，包含很多重要的性能指标，应该作为性能分析的起点。

```
(db2inst1@db2host1) /home/db2inst1 $ db2 get snapshot for database on sample
```

Database Snapshot

```
Database name                = SAMPLE
Database path                =
/home/db2inst1/db2inst1/NODE0000/SQL00001/
Input database alias         = SAMPLE
Database status              = Active
Catalog database partition number = 0
Catalog network node name    =
Operating system running at database server= AIX 64BIT
Location of the database      = Local
First database connect timestamp = 01/13/2011 06:37:25.956206
Last reset timestamp         =
Last backup timestamp        = 01/12/2011 22:59:44.000000
Snapshot timestamp           = 01/13/2011 07:13:36.571054

Number of automatic storage paths = 0

High water mark for connections = 16
Application connects             = 55
Secondary connects total         = 14
Applications connected currently = 1
Appls. executing in db manager currently = 0
Agents associated with applications = 14
Maximum agents associated with applications= 16
Maximum coordinating agents      = 16

Number of Threshold Violations = 0
Locks held currently            = 0
Lock waits                      = 0
Time database waited on locks (ms) = 0
Lock list memory in use (Bytes) = 8832
Deadlocks detected              = 0
Lock escalations                = 0
Exclusive lock escalations      = 0
Agents currently waiting on locks = 0
Lock Timeouts                   = 0
Number of indoubt transactions = 0

Total Private Sort heap allocated = 0
Total Shared Sort heap allocated = 0
Shared Sort heap high water mark = 22
Post threshold sorts (shared memory) = 0
```

```

Total sorts                                = 1
Total sort time (ms)                      = 0
Sort overflows                            = 0
Active sorts                              = 0

Buffer pool data logical reads             = 6586
Buffer pool data physical reads            = 223
Buffer pool temporary data logical reads   = 0
Buffer pool temporary data physical reads  = 0
Asynchronous pool data page reads         = 93
Buffer pool data writes                    = 0
Asynchronous pool data page writes        = 0
Buffer pool index logical reads            = 3659
Buffer pool index physical reads           = 332
Buffer pool temporary index logical reads  = 0
Buffer pool temporary index physical reads = 0
Asynchronous pool index page reads         = 0
Buffer pool index writes                   = 0
Asynchronous pool index page writes        = 0
Buffer pool xda logical reads              = 0
Buffer pool xda physical reads             = 0
Buffer pool temporary xda logical reads    = 0
Buffer pool temporary xda physical reads   = 0
Buffer pool xda writes                    = 0
Asynchronous pool xda page reads           = 0
Asynchronous pool xda page writes          = 0
Total buffer pool read time (milliseconds) = 1048
Total buffer pool write time (milliseconds) = 0
Total elapsed asynchronous read time       = 90
Total elapsed asynchronous write time      = 0
Asynchronous data read requests            = 6
Asynchronous index read requests           = 0
Asynchronous xda read requests             = 0
No victim buffers available                = 0
LSN Gap cleaner triggers                   = 0
Dirty page steal cleaner triggers          = 0
Dirty page threshold cleaner triggers      = 0
Time waited for prefetch (ms)             = 63
Unread prefetch pages                     = 0
Direct reads                              = 2440
Direct writes                              = 0
Direct read requests                       = 206
Direct write requests                      = 0
Direct reads elapsed time (ms)             = 313
Direct write elapsed time (ms)             = 0
Database files closed                      = 0

Host execution elapsed time                = 1.252902

Commit statements attempted                = 234

```

```

Rollback statements attempted          = 0
Dynamic statements attempted           = 1715
Static statements attempted            = 288
Failed statement operations             = 0
Select SQL statements executed         = 519
Xquery statements executed              = 0
Update/Insert/Delete statements executed = 204
DDL statements executed                = 0
Inactive stmt history memory usage (bytes) = 0

Internal automatic rebinds             = 0
Internal rows deleted                  = 0
Internal rows inserted                  = 0
Internal rows updated                  = 0
Internal commits                       = 280
Internal rollbacks                     = 0
Internal rollbacks due to deadlock     = 0
Number of MDC table blocks pending cleanup = 0

Rows deleted                           = 0
Rows inserted                           = 0
Rows updated                           = 774
Rows selected                           = 1227
Rows read                              = 7720
Binds/precompiles attempted           = 0

Log space available to the database (Bytes)= 52921133
Log space used by the database (Bytes)  = 66867
Maximum secondary log space used (Bytes) = 0
Maximum total log space used (Bytes)    = 66963
Secondary logs allocated currently      = 0
Log pages read                          = 0
Log read time (sec.ns)                  = 0.000000004
Log pages written                       = 272
Log write time (sec.ns)                  = 0.000000004
Number write log IOs                    = 272
Number read log IOs                     = 0
Number partial page log IOs             = 251
Number log buffer full                  = 0
Log data found in buffer                 = 0
Log to be redone for recovery (Bytes)    = 67187
Log accounted for by dirty pages (Bytes) = 67187

Node number                            = 0
File number of first active log         = 14
File number of last active log          = 16
File number of current active log       = 14
File number of log being archived       = Not applicable

Package cache lookups                   = 804

```

```

Package cache inserts           = 36
Package cache overflows         = 0
Package cache high water mark (Bytes) = 1033454
Application section lookups     = 1715
Application section inserts     = 36

```

```

Catalog cache lookups          = 1025
Catalog cache inserts          = 29
Catalog cache overflows        = 0
Catalog cache high water mark  = 314255
Catalog cache statistics size  = 0

```

Workspace Information

```

Number of hash joins           = 2
Number of hash loops           = 0
Number of hash join overflows  = 0
Number of small hash join overflows = 0
Post threshold hash joins (shared memory) = 0
Active hash joins              = 0

```

```

Number of OLAP functions       = 0
Number of OLAP function overflows = 0
Active OLAP functions          = 0

```

```

Statistic fabrications         = 0
Synchronous runstats          = 0
Asynchronous runstats          = 0
Total statistic fabrication time (milliseconds) = 0
Total synchronous runstats time (milliseconds) = 0

```

Memory usage for database:

```

Node number                    = 0
  Memory Pool Type              = Backup/Restore/Util Heap
    Current size (bytes)        = 65536
    High water mark (bytes)     = 65536
    Configured size (bytes)     = 20512768

```

```

Node number                    = 0
  Memory Pool Type              = Package Cache Heap
    Current size (bytes)        = 1114112
    High water mark (bytes)     = 1114112
    Configured size (bytes)     = 1310720

```

针对以上数据库快照信息，我们详细分解每一段的含义。

首先是数据库名称与快照捕捉的时间：

```

Database name                = SAMPLE
First database connect timestamp = 01/13/2011 06:37:25.956206
Last reset timestamp         =
Last backup timestamp        = 01/12/2011 22:59:44.000000
Snapshot timestamp           = 01/13/2011 07:13:36.571054

```

上面的信息说明,快照抓取时间距离数据库第一个连接,已经过去了大约 36 分钟(01/13/2011 07:13 - 01/13/2011 06:37),换句话说,快照信息包含了 36 分钟的统计数据。

下面是数据库连接信息:

```

High water mark for connections = 16
Application connects            = 55
Secondary connects total       = 14
Applications connected currently = 1
Appls. executing in db manager currently = 0
Agents associated with applications = 14
Maximum agents associated with applications = 16
Maximum coordinating agents    = 16

```

通过以上片段,我们得知,连接数的高水位是 16,也就是在这半个小时之内,最高时总共有 16 个连接同时在线(包括一些非用户连接,比如数据库内部工具的连接);总共连接成功的数量为 55;当前用户连接数量为 1;当前有 0 个 DB2 代理线程正在执行事务;当前有 14 个 DB2 代理线程被使用(但是不在执行 SQL 任务)。通过这些信息,用户可以了解到数据库连接请求的频繁程度。譬如,正常情况下每小时平均有 100 个连接,可是当某些性能问题发生的时候,该数量也许会上升到 1000 甚至几千的级别。这种情况下就要分析,是否是过于频繁的连接导致的性能下降,还是性能下降导致的频繁连接请求。

接着是锁相关信息:

```

Locks held currently          = 0
Lock waits                    = 0
Time database waited on locks (ms) = 0
Lock list memory in use (Bytes) = 8832
Deadlocks detected            = 0
Lock escalations              = 0
Exclusive lock escalations    = 0
Agents currently waiting on locks = 0
Lock Timeouts                 = 0

```

包括当前数据库所持有的锁的总数、锁等待的数量、锁等待的总时间、当前数据库中锁列表用了多少内存、多少次死锁、多少次锁升级、多少次锁超时等。

这些信息对于锁问题的诊断和分析非常重要。通过研究给定时间单位内的锁等待数量与时间,可以估算出平均每个锁等待占系统运行总时间的百分比。

接着的片段是排序信息:

```

Total sorts                    = 1

```

Total sort time (ms)	= 0
Sort overflows	= 0

排序是 CPU 的杀手，过多的排序会造成 CPU 的极大消耗，应该尽量减少和避免。在一个给定的时间段内，如果发现总排序时间除以平均并发数量的结果很高，需要考虑对排序进行优化。

排序溢出是说，如果排序堆无法容纳排序数据，就会被溢出到临时空间。当需要排序的数据集很大，根本不可能将所有的数据都装入排序堆，对于仓库系统，溢出是不可避免的，但应该尽量减少。

排序是一种状态，根源在于 SQL 语句。在 16.2.2 章节，我们通过案例详细介绍了排序的诊断分析和调优过程。

接着的片段就是数据索引 I/O 了。由于篇幅所限，我们重点讨论几个实际中使用较多的指标：

Buffer pool data logical reads	= 6586
Buffer pool data physical reads	= 223
Buffer pool temporary data logical reads	= 0
Buffer pool temporary data physical reads	= 0
Asynchronous pool data page reads	= 93
Buffer pool data writes	= 0
Asynchronous pool data page writes	= 0
Buffer pool index logical reads	= 3659
Buffer pool index physical reads	= 332
Buffer pool temporary index logical reads	= 0
Buffer pool temporary index physical reads	= 0
Asynchronous pool index page reads	= 0
Buffer pool index writes	= 0
Asynchronous pool index page writes	= 0
Total buffer pool read time (milliseconds)	= 1048
Total buffer pool write time (milliseconds)	= 0
Total elapsed asynchronous read time	= 90
Total elapsed asynchronous write time	= 0
Asynchronous data read requests	= 6
Asynchronous index read requests	= 0
Time waited for prefetch (ms)	= 63
Direct reads	= 2440
Direct writes	= 0
Direct reads elapsed time (ms)	= 313
Direct write elapsed time (ms)	= 0

以上信息从数据，索引和临时数据三方面描述了一段时间来数据库所做的操作。其中，逻辑读可以理解为 DB2 向缓冲池请求的次数，在这一阶段 DB2 并不知道数据是在缓冲池还是磁盘。而物理读是说，如果请求的数据页不在缓冲池，需要从磁盘中读取数据页的次数。

逻辑读越多，需要的物理 I/O 就越少。以上信息包含了数据、索引和临时数据三个方面的逻辑读和物理读。通过研究这些参数之间的比例，DBA 可以对数据库的行为有一个大概的了解。例如：某一数据库 80% 的读取是索引，15% 是数据，5% 是临时数据；其中大约 90% 的读是从逻

辑读，即从内存中读取，另外 10% 为物理 I/O。通过这样的分析，DBA 会对什么样的操作占用主要资源有一个直观的认识。

缓存池读写时间也是一个非常重要的指标。不过需要注意，由于单纯的内存操作非常迅速，在计算内存池读写时间的时候，内存读并不会计算在内。因此，以下公式可用来计算平均每一物理读消耗的时间。

每一次物理读消耗的时间 = $\frac{\text{“Total buffer pool read time”}}{\text{“Buffer pool data physical reads”} + \text{“Buffer pool temporary data physical reads”} + \text{“Buffer pool index physical reads”} + \text{“Buffer pool temporary index physical reads”}}$

直接 I/O 操作一般用于大对象数据 (LOB)，如果用户发现系统大部分的时间都在等待直接读操作，那么需要对大对象的 I/O 进行优化。

以上是 I/O 片段信息，紧接着是吞吐量片段，或者事务情况：

```
Commit statements attempted      = 234
Rollback statements attempted    = 0
Dynamic statements attempted     = 1715
Static statements attempted      = 288
Failed statement operations      = 0
Select SQL statements executed   = 519
Xquery statements executed       = 0
Update/Insert/Delete statements = 204
DDL statements executed          = 0
Inactive stmt history memory usage (bytes) = 0
```

以上片段是说，在一段时间内，共提交、回滚了多少事务；执行了多少次动态和静态语句；有多少次增/删/改/查操作等。通过这部分数据，DBA 可以对当前应用程序的行为有一个直观的认识：是查询类型的事务多，还是主要在修改数据；平均每分钟执行了多少次事务；平均每个事务包含几条查询，几条增删改。对应用程序的行为了解越多，越能更好地进行性能分析和诊断。

接下来的片段是关于增、删、改、查的行数：

```
Rows deleted      = 0
Rows inserted     = 0
Rows updated      = 774
Rows selected     = 1227
Rows read         = 7720
Binds/precompiles attempted = 0
```

以上片段详细统计了，一段时间内删除、插入或修改了多少行数据；读取了多少行以及结果集数量。

Rows read/rows selected 是一个非常重要的性能指标，它表示为了检索一行数据需要读取多少行，该值越大，表示代价越高，需要的 I/O 越多，可调优的余地越大。

接下来的片段是关于事务日志：

```
Log space available to the database (Bytes)= 52921133
Log space used by the database (Bytes)      = 66867
Maximum secondary log space used (Bytes)    = 0
Maximum total log space used (Bytes)        = 66963
Secondary logs allocated currently          = 0
Log pages read                             = 0
Log read time (sec.ns)                     = 0.000000004
Log pages written                           = 272
Log write time (sec.ns)                     = 0.000000004
Number write log I/Os                      = 272
Number read log I/Os                       = 0
Number partial page log I/Os               = 251
Number log buffer full                      = 0
```

根据经验，发现很多系统的性能瓶颈并不在数据 I/O，而是在日志 I/O。我们知道，日志的目的是用来保证数据库的一致性，DB2 采用的是写日志优先算法，也就是在数据真正被写到磁盘之前先写日志。这样，日志 I/O 在很大程度上会影响数据库整体的性能。

通过以上数据，我们可以了解日志相关的统计信息，如日志空间的使用率、曾经使用的最大日志空间、共发生了多少次日志读写、读写日志花费的时间等。

以上是数据库快照中一些重要的片段信息。在案例分析部分，会针对一些重要的指标演示分析的过程和方法，加深理解。

(2) 应用程序快照

应用程序快照，即 Application Snapshot。应用程序快照对每一个数据库连接会产生一组快照，为了方便阐述，我们摘录其中一组数据：

```
(db2inst1@db2host1) /home/db2inst1 $ db2 get snapshot for applications on sample
...
      Application Snapshot

Application handle                = 2108
Application status                = UOW Executing
Status change time               = 01/13/2011 17:15:37.021724
Application code page            = 819
Application country/region code  = 1
DUOW correlation token           = *LOCAL.db2inst1.110113221440
Application name                 = db2bp
Application ID                   = *LOCAL.db2inst1.110113221440
Sequence number                  = 00001
TP Monitor client user ID       =
TP Monitor client workstation name =
TP Monitor client application name =
TP Monitor client accounting string =
```

```

Connection request start timestamp      = 01/13/2011 17:14:40.659640
Connect request completion timestamp    = 01/13/2011 17:14:40.668028
Application idle time                   =
CONNECT Authorization ID                = DB2INST1
Client login ID                        = db2inst1
Configuration NNAME of client           = db2host1
Client database manager product ID      = SQL09073
Process ID of client application        = 48324962
Platform of client application          = AIX 64BIT
Communication protocol of client        = Local Client

Inbound communication address           = *LOCAL.db2inst1

Database name                          = SAMPLE
Database path                          =
/home/db2inst1/db2inst1/NODE0000/SQL00001/
Client database alias                   = SAMPLE
Input database alias                   = SAMPLE
Last reset timestamp                   =
Snapshot timestamp                     = 01/13/2011 17:15:37.705285
Authorization level granted             =
  User authority:
    DBADM authority
    CREATETAB authority
    BINDADD authority
    CONNECT authority
    CREATE_NOT_FENC authority
    LOAD authority
    IMPLICIT_SCHEMA authority
    CREATE_EXT_RT authority
    QUIESCE_CONN authority
    SECADM authority
    DATAACCESS authority
    ACCESSCTRL authority
  Group authority:
    SYSADM authority
    DBADM authority
    CREATETAB authority
    BINDADD authority
    CONNECT authority
    IMPLICIT_SCHEMA authority
    DATAACCESS authority
    ACCESSCTRL authority
Coordinating database partition number  = 0
Current database partition number       = 0
Coordinator agent process or thread ID  = 6813
Current Workload ID                     = 1
Agents stolen                           = 0
Agents waiting on locks                 = 0
Maximum associated agents               = 1

```

```

Priority at which application agents work = 0
Priority type                             = Dynamic

Lock timeout (seconds)                   = -1
Locks held by application                 = 3
Lock waits since connect                  = 0
Time application waited on locks (ms)    = 0
Deadlocks detected                       = 0
Lock escalations                         = 0
Exclusive lock escalations                = 0
Number of Lock Timeouts since connected  = 0
Total time UOW waited on locks (ms)      = 0

Total sorts                             = 0
Total sort time (ms)                     = 0
Total sort overflows                      = 0

Buffer pool data logical reads            = 4666
Buffer pool data physical reads           = 63
Buffer pool temporary data logical reads  = 6396
Buffer pool temporary data physical reads = 0
Buffer pool data writes                   = 475
Buffer pool index logical reads            = 2
Buffer pool index physical reads           = 0
Buffer pool temporary index logical reads = 0
Buffer pool temporary index physical reads = 0
Buffer pool index writes                   = 9
Buffer pool xda logical reads              = 0
Buffer pool xda physical reads             = 0
Buffer pool temporary xda logical reads   = 0
Buffer pool temporary xda physical reads  = 0
Buffer pool xda writes                    = 0
Total buffer pool read time (milliseconds) = 17
Total buffer pool write time (milliseconds) = 158
Time waited for prefetch (ms)             = 84
Unread prefetch pages                     = 0
Direct reads                             = 4
Direct writes                             = 0
Direct read requests                       = 1
Direct write requests                     = 0
Direct reads elapsed time (ms)             = 0
Direct write elapsed time (ms)             = 0

Number of SQL requests since last commit  = 22
Commit statements                         = 0
Rollback statements                       = 0
Dynamic SQL statements attempted           = 22
Static SQL statements attempted            = 0
Failed statement operations                = 0
Select SQL statements executed              = 2

```

```

Xquery statements executed          = 0
Update/Insert/Delete statements executed = 0
DDL statements executed              = 0
Inactive stmt history memory usage (bytes) = 0
Internal automatic rebinds          = 0
Internal rows deleted                = 0
Internal rows inserted               = 0
Internal rows updated                = 0
Internal commits                     = 1
Internal rollbacks                   = 0
Internal rollbacks due to deadlock  = 0
Binds/precompiles attempted         = 0
Rows deleted                         = 0
Rows inserted                        = 0
Rows updated                         = 0
Rows selected                       = 149182
Rows read                           = 1117017
Rows written                         = 6396

UOW log space used (Bytes)          = 0
Previous UOW completion timestamp   = 01/13/2011 17:14:40.668028
Elapsed time of last completed uow (sec.ms)= 0.000000
UOW start timestamp                 = 01/13/2011 17:14:56.024485
UOW stop timestamp                  =
UOW completion status               =

Open remote cursors                 = 0
Open remote cursors with blocking   = 0
Rejected Block Remote Cursor requests = 0
Accepted Block Remote Cursor requests = 2
Open local cursors                   = 1
Open local cursors with blocking     = 1
Total User CPU Time used by agent (s) = 0.391806
Total System CPU Time used by agent (s) = 0.015351
Host execution elapsed time          = 3.402868

Package cache lookups                = 2
Package cache inserts                = 2
Application section lookups           = 22
Application section inserts           = 2
Catalog cache lookups                = 19
Catalog cache inserts                 = 1
Catalog cache overflows               = 0
Catalog cache high water mark        = 0

Workspace Information

Most recent operation                 = Fetch
Cursor name                          = SQLCUR201

```

```

Most recent operation start timestamp = 01/13/2011 17:15:37.021729
Most recent operation stop timestamp =
Agents associated with the application = 1
Number of hash joins = 1
Number of hash loops = 0
Number of hash join overflows = 1
Number of small hash join overflows = 0
Number of OLAP functions = 0
Number of OLAP function overflows = 0

Statement type = Dynamic SQL Statement
Statement = Fetch
Section number = 201
Application creator = NULLID
Package name = SQLC2H21
Consistency Token =
Package Version ID =
Cursor name = SQLCUR201
Statement database partition number = 0
Statement start timestamp = 01/13/2011 17:15:37.021729
Statement stop timestamp =
Elapsed time of last completed stmt(sec.ms)= 0.000050
Total Statement user CPU time = 0.335106
Total Statement system CPU time = 0.014749
SQL compiler cost estimate in timerons = 919550080
SQL compiler cardinality estimate = 2147483647
Degree of parallelism requested = 1
Number of agents working on statement = 1
Number of subagents created for statement = 1
Statement sorts = 0
Total sort time = 0
Sort overflows = 0
Rows read = 967833
Rows written = 6396
Rows deleted = 0
Rows updated = 0
Rows inserted = 0
Rows fetched = 0
Buffer pool data logical reads = 4052
Buffer pool data physical reads = 62
Buffer pool temporary data logical reads = 6396
Buffer pool temporary data physical reads = 0
Buffer pool index logical reads = 0
Buffer pool index physical reads = 0
Buffer pool temporary index logical reads = 0
Buffer pool temporary index physical reads = 0
Buffer pool xda logical reads = 0
Buffer pool xda physical reads = 0
Buffer pool temporary xda logical reads = 0

```

```

Buffer pool temporary xda physical reads    = 0
Blocking cursor                             = YES
Dynamic SQL statement text:
select * from t1 as A, t1 as B where A.c1=B.c1

    Agent process/thread ID                  = 6813
Memory usage for application:

    Memory Pool Type                        = Application Heap
    Current size (bytes)                    = 131072
    High water mark (bytes)                 = 131072
    Configured size (bytes)                 = 1048576

Agent process/thread ID                      = 6813
Agent Lock timeout (seconds)                 = -1
Memory usage for agent:

    Memory Pool Type                        = Other Memory
    Current size (bytes)                    = 262144
    High water mark (bytes)                 = 458752
    Configured size (bytes)                 = 12884901888
...

```

对于应用程序快照，首先需要注意时间与数据库名称。只有在正确的时间和正确的数据库抓到的快照才是有价值的数​​据，否则在用了半个小时分析文件后发现文件本身的时间就是错误的，就会浪费很多宝贵的资源。

```

Database name                               = SAMPLE
Database path                               =
/home/db2inst1/db2inst1/NODE0000/SQL00001/
Client database alias                       = SAMPLE
Input database alias                       = SAMPLE
Last reset timestamp                       =
Snapshot timestamp                         = 01/13/2011 17:15:37.705285

```

然后找到要分析的应用。

```

Application handle                          = 2108
Application status                          = UOW Executing
Status change time                          = 01/13/2011 17:15:37.021724
Application name                            = db2bp
Application ID                             = *LOCAL.db2inst1.110113221440
Connection request start timestamp          = 01/13/2011 17:14:40.659640
Connect request completion timestamp        = 01/13/2011 17:14:40.668028
Coordinator agent process or thread ID     = 6813

```

在应用程序快照中，每一个连接包含同样格式的一组数据，用户可以通过不同的组合找到自己想要的​​应用，比如通过应用程序名。上面的片段中，应用程序名是 db2bp，也就是 DB2 后台进程，用来与前台脚本进程交互的。

如果用户有许多命令行程序在运行，想要找到某个特定的连接，可以通过连接时间确定。假设用户知道连接发生在中午 12 点左右，则可以检查“Connection request start timestamp”时间戳定位连接。

如果连时间也不能确定呢？不要紧，既然 DBA 对该程序有兴趣，那么这个程序一定有什么特别的地方，比如正在执行什么东西，或者刚刚执行完什么东西。这样我们可以通过应用程序状态得到想要的连接。在上面这个例子中，笔者用一个窗口执行查询，因此只用搜索 UOW Executing 状态的应用程序就可以很轻易地定位。

“Status change time”描述的是最近一次状态变化的时间。根据这个状态变化的时间与快照捕获的时间，我们可以知道该应用在抓快照的一刹那，最后一个任务运行了多长时间。

“Coordinator agent process or thread ID”表示正在服务于该任务的进程或线程 ID。如果一个应用有多个线程在跑（比如分区内并行），可以在快照的最后看到所有的线程列表：

```
Agent process/thread ID          = 6813
Agent Lock timeout (seconds)     = -1
Memory usage for agent:
```

该代理线程 ID 可以与前文所述的 db2pd -edus 相对比，得到当前应用程序中谁消耗的 CPU 最多。

在应用程序快照中，绝大部分的参数都和数据库快照类似，只不过其数值针对某个应用程序，而数据库快照是对所有应用程序的统计。

接着是应用程序执行时间信息：

```
Total User CPU Time used by agent (s)    = 0.391806
Total System CPU Time used by agent (s)   = 0.015351
Host execution elapsed time               = 3.402868
Previous UOW completion timestamp         = 01/13/2011 17:14:40.668028
Elapsed time of last completed uow (sec.ms)= 0.000000
UOW start timestamp                      = 01/13/2011 17:14:56.024485
Most recent operation                    = Fetch
Cursor name                             = SQLCUR201
Most recent operation start timestamp     = 01/13/2011 17:15:37.021729
Most recent operation stop timestamp      =
```

以上内容非常重要。最上面的 3 条为 CPU 的使用情况与总运行时间。在一般情况下，如果分区内并行没有打开的情况下， $(\text{User CPU Time} + \text{System CPU Time}) / (\text{快照时间} - \text{应用程序连接时间})$ 可以得出该应用程序所占用单独 CPU 的百分比。假设我们有一个非常复杂的 SQL 需要运行 10 个小时，那么通过计算该百分比，我们可以了解该 SQL 是在主要进行 CPU 操作呢还是大部分时间花在等待锁和 I/O 上。

“Most recent operation”是指最近所进行的操作，在这个例子中是 Fetch，即数据抓取；该操作开始的时间是“01/13/2011 17:15:37.021729”。

如果想抓取一个 UOW Executing 的操作具体在执行什么语句，同样可以在应用程序快照中找到：

```
Dynamic SQL statement text:
select * from t1 as A, t1 as B where A.c1=B.c1
```

在应用快照的最后部分，就是当前正在执行的语句。

(3) 表空间快照

仅仅有数据库与应用程序快照并不能完整地表示系统的状态。比如在数据库快照中发现存在大量的逻辑读，通过应用程序快照可以细化到某两条特定的语句，但是该语句可能包含 10 个不同的表。如何知道是哪个表空间中的哪个表造成了如此多的逻辑读呢？这就需要表空间快照和表快照。

类似应用程序快照，表空间快照也是每个表空间一个段落：

```
(db2inst1@db2host1) /home/db2inst1/temp $ db2 get snapshot for tablespaces on sample
First database connect timestamp      = 01/13/2011 06:37:25.956206
Last reset timestamp                  =
Snapshot timestamp                    = 01/13/2011 17:46:16.668043
Database name                        = SAMPLE
Database path                        =
/home/db2inst1/db2inst1/NODE0000/SQL00001/
Input database alias                  = SAMPLE
Number of accessed tablespaces        = 5
.....
Tablespace name                      = SYSTOOLSPACE
Tablespace ID                        = 3
Tablespace Type                      = Database managed space
Tablespace Content Type              = All permanent data. Large table space.
Tablespace Page size (bytes)         = 4096
Tablespace Extent size (pages)       = 4
Automatic Prefetch size enabled      = No
Tablespace Prefetch size (pages)     = 32
Buffer pool ID currently in use      = 1
Buffer pool ID next startup          = 1
Using automatic storage               = No
Auto-resize enabled                  = Yes
File system caching                   = No
Tablespace State                     = 0x'00000000'
Detailed explanation:
Normal
Total number of pages                = 8192
Number of usable pages                = 8188
Number of used pages                  = 152
Number of pending free pages          = 0
Number of free pages                  = 8036
High water mark (pages)               = 152
```

```

Current tablespace size (bytes)      = 33554432
Maximum tablespace size (bytes)     = NONE
Increase size (bytes)               = AUTOMATIC
Time of last successful resize      =
Last resize attempt failed          = No
Rebalancer Mode                     = No Rebalancing
Minimum Recovery Time               =
Number of quiescers                 = 0
Number of containers                = 1

Container Name = /home/db2inst1/db2inst1/NODE0000/ SQL00001/SYSTOOLSPACE
  Container ID          = 0
  Container Type        = File (extent sized tag)
  Total Pages in Container = 8192
  Usable Pages in Container = 8188
  Stripe Set           = 0
  Container is accessible = Yes
  File system ID       = 9223372182883663873
  File system used space (bytes) = 80002359296
  File system total space (bytes) = 106568876032

```

Table space map:

Range Number	Stripe Set	Stripe Offset	Max Extent	Max Page	Start Stripe	End Stripe	Adj.	Containers
[0]	[0]	0	2046	8187	0	2046	0	1 (0)

```

Buffer pool data logical reads      = 31020
Buffer pool data physical reads     = 44
Buffer pool temporary data logical reads = 0
Buffer pool temporary data physical reads = 0
Asynchronous pool data page reads   = 0
Buffer pool data writes              = 15
Asynchronous pool data page writes   = 13
Buffer pool index logical reads      = 14611
Buffer pool index physical reads     = 16
Buffer pool temporary index logical reads = 0
Buffer pool temporary index physical reads = 0
Asynchronous pool index page reads   = 0
Buffer pool index writes             = 1
Asynchronous pool index page writes   = 1
Buffer pool xda logical reads        = 0
Buffer pool xda physical reads       = 0
Buffer pool temporary xda logical reads = 0
Buffer pool temporary xda physical reads = 0
Asynchronous pool xda page reads     = 0
Buffer pool xda writes               = 0
Asynchronous pool xda page writes     = 0
Total buffer pool read time (millisec) = 63
Total buffer pool write time (millisec) = 7

```

```

Total elapsed asynchronous read time      = 0
Total elapsed asynchronous write time     = 6
Asynchronous data read requests          = 0
Asynchronous index read requests          = 0
Asynchronous xda read requests            = 0
No victim buffers available                = 0
Direct reads                             = 342
Direct writes                             = 0
Direct read requests                      = 159
Direct write requests                     = 0
Direct reads elapsed time (ms)            = 42
Direct write elapsed time (ms)            = 0
Number of files closed                    = 0

```

...

在表空间快照开头部分是快照的时间和数据库信息。然后观察缓冲池读写，就可以轻松地定位哪个表空间被频繁使用：

```

Buffer pool data logical reads            = 31020
Buffer pool data physical reads           = 44
Buffer pool temporary data logical reads  = 0
Buffer pool temporary data physical reads = 0
Asynchronous pool data page reads         = 0
Buffer pool data writes                   = 15
Asynchronous pool data page writes        = 13
Buffer pool index logical reads            = 14611
Buffer pool index physical reads           = 16

```

另外一些关于表空间的详细信息也会被显示，由于本小节是关于性能监控，所以表空间的属性不再进行额外论述。

(4) 表快照

在绝大多数系统中，经常被访问的表可能就是几十个，我们把这些表叫做“Hot Table”（热表）。找到热表，并在物理和逻辑设计上进行优化、或者找到使用这些热表的 SQL 进行优化，是性能分析常用的手段。

```
(db2inst1@db2host1) /home/db2inst1/temp $ db2 get snapshot for tables on sample
```

Table Snapshot

```

First database connect timestamp      = 01/13/2011 06:37:25.956206
Last reset timestamp                  =
Snapshot timestamp                    = 01/13/2011 17:52:00.620074
Database name                         = SAMPLE
Database path                         = /home/db2inst1/db2inst1/NODE0000/SQL00001/
Input database alias                  = SAMPLE
Number of accessed tables              = 40
...
Table Schema                         = DB2INST1

```

```

Table Name           = T1
Table Type           = User
Data Object Pages    = 2804
Index Object Pages    = 32
Rows Read            = 1555617
Rows Written         = 0
Overflows            = 0
Page Reorgs         = 0
...

```

在表快照结果中，首先是快照捕捉时间，然后是每张表的快照序列，包括表模式、表名、类型、数据页数量、索引页数量、该表被读过的行数、写的行数、溢出数量和重组页数。

最有用的是数据页数量、索引页数量与读写的行数。这些属性标志着该表的大小，以及是否被频繁访问。如果我们发现一个表的页数很少，但是读的行数非常多（比如 1 个小时之内 1 亿行），那么我们可以合理地猜测该表在某些查询语句中可能处于 NLJOIN 的内部子节点。

（5）动态 SQL 语句快照

SQL 是很多性能问题的根源，因此需要找到哪些语句是“差”的 SQL。对于动态语句来说，采用 SQL 快照是最好的方法：

```
(db2inst1@db2host1) /home/db2inst1 $ db2 get snapshot for dynamic sql on sample
```

Dynamic SQL Snapshot Result

```

Database name           = SAMPLE
Database path           = /home/db2inst1/db2inst1/NODE0000/SQL00001/

Number of executions     = 1
Number of compilations   = 0
Worst preparation time (ms) = 3
Best preparation time (ms) = 3
Internal rows deleted    = 0
Internal rows inserted   = 0
Rows read                = 1406694
Internal rows updated    = 0
Rows written             = 7756
Statement sorts          = 0
Statement sort overflows = 0
Total sort time          = 0
Buffer pool data logical reads = 6121
Buffer pool data physical reads = 73
Buffer pool temporary data logical reads = 8016
Buffer pool temporary data physical reads = 3
Buffer pool index logical reads = 0
Buffer pool index physical reads = 0
Buffer pool temporary index logical reads = 0

```

```

Buffer pool temporary index physical reads = 0
Buffer pool xda logical reads      = 0
Buffer pool xda physical reads     = 0
Buffer pool temporary xda logical reads  = 0
Buffer pool temporary xda physical reads = 0
Total execution time (sec.microsec)= 1.928020
Total user cpu time (sec.microsec) = 0.535691
Total system cpu time (sec.microsec)= 0.015438
Total statistic fabrication time (milliseconds) = 0
Total synchronous runstats time (milliseconds) = 0
Statement text                      = select * from t1 as A, t1 as B where A.c1=B.c1
...

```

SQL 快照用来显示当前 Package Cache 中的动态 SQL。除了 SQL 语句本身，快照还会显示一些性能相关的数据，如该 SQL 执行的次数、总共读的行数、消耗的 CPU、逻辑物理读的数量、排序数量等信息。这些信息对于我们定位“差”的语句特别有用。用户可以通过对这些信息进行二次计算，得到自己真正想要的结果。例如，如果用户希望知道消耗 CPU 最大的 SQL 语句，可以对每一条 SQL 用 $(\text{Total user cpu time} + \text{Total system cpu time}) / \text{Number of executions}$ 公式计算，得到每次执行的平均 CPU 消耗时间，并按照从高到低排序。

3. 内存使用监控

我们把内存使用监控独立于数据库快照部分，主要是为了强调内存的重要性。

在性能问题分析的时候，内存使用率的监测是非常重要的的一环，很多内存相关的问题都会导致系统性能的严重下降。

一般来说，对于日常的内存监控，笔者建议收集如下信息：

- db2pd -osinfo。
- db2pd -dbptnmem。
- db2pd -memsets。
- db2pd -mempools。
- db2pd -db <dbname> -memsets。
- db2pd -db <dbname> -mempools。
- db2mtrk -i -d -k -v。

下面让我们分别来看看这些指令都收集了什么数据：

```
(db2inst1@db2host1) /home/db2inst1 $ db2pd -osinfo
```

```
Operating System Information:
```

```

OSName:   AIX
NodeName: db2host1
Version:  6
Release:  1

```

```

Machine: 00C78E5F4C00

CPU Information:
TotalCPU    OnlineCPU  ConfigCPU  Speed(MHz)  HMTDegree  Cores/Socket
16          8          16         1902        2           n/a

Physical Memory and Swap (Megabytes):
TotalMem    FreeMem    AvailMem    TotalSwap    FreeSwap
12288       285        n/a         12288        8592

Virtual Memory (Megabytes):
Total       Reserved   Available   Free
24576       n/a        n/a         8877

Message Queue Information:
MsgSeg      MsgMax     MsgMap      MsgMni       MsgTql      MsgMnb      MsgSsz
n/a         4194304   n/a         n/a          n/a         4194304    n/a

Shared Memory Information:
ShmMax      ShmMin     ShmIds      ShmSeg
68719476736 1          131072      0

Semaphore Information:
SemMap      SemMni     SemMns      SemMnu       SemMsl      SemOpm      SemUme
SemUsz      SemVmX     SemAem
n/a         131072    n/a         n/a          65535       1024       n/a       n/a
32767       16384

CPU Load Information:
Short      Medium     Long
1.045990   1.563797  1.625443

CPU Usage Information (percent):
Total      Usr        Sys        Wait       Idle
9.625000   1.250000  8.375000  0.000000  90.375000

```

osinfo 是 operating system information 的缩写，即操作系统信息。作者把它放在内存收集部分，因为最重要的就是其中内存的使用部分：

```

Physical Memory and Swap (Megabytes):
TotalMem    FreeMem    AvailMem    TotalSwap    FreeSwap
12288       285        n/a         12288        8592

Virtual Memory (Megabytes):
Total       Reserved   Available   Free
24576       n/a        n/a         8877

```

这个信息很好理解，系统的总物理内存为 12288MB，空闲内存 285MB，总交换分区 12288MB，空闲交换分区 8592MB。当然，这些数值并没有计入文件系统缓存，需要结合第一部分所讨论的 svmon 结果来理解。

	size	inuse	free	pin	virtual	mmode
memory	3145728	3069224	76504	943160	3883915	Ded
pg space	3145728	946403				
	work	pers	clnt	other		
pin	717177	0	0	225983		
in use	2988183	65	80976			

得到了系统级内存以后，`db2pd -dbptnmem` 是 9.5 版本引入的获得实例级内存分配的非常有用的命令。正像前文所描述的，在 9.5 中，DB2 从进程模型演变为线程模型，当 `db2pd attach` 到 `db2sysc` 进程的时候，它可以看到整个实例下所有的数据库共享内存。因此，在 9.5 以后，DB2 就可以通过查询单个进程下的内存映射来了解整个实例的内存使用情况：

```
(db2inst1@db2host1) /home/db2inst1 $ db2pd -dbptnmem

Database Partition 0 -- Active -- Up 1 days 07:46:43 -- Date 01/14/2011 06:30:51

Database Partition Memory Controller Statistics

Controller Automatic: Y
Memory Limit:          10670724 KB
Current usage:          366400 KB
HWM usage:              366656 KB
Cached memory:          191040 KB

Individual Memory Consumers:

Name                    Mem Used (KB) HWM Used (KB) Cached (KB)
=====
APPL-SAMPLE             160000      160000      155456
DBMS-db2inst1           49920       49920       10240
FMP_RESOURCES           22528       22528         0
PRIVATE                 20352       20352       7104
DB-SAMPLE               112832      112832      18240
LCL-p39874876           128         128         0
LCL-p48324962           128         128         0
LCL-p39874876           128         128         0
LCL-p48324962           128         128         0
LCL-p32662006           128         128         0
LCL-p32662006           128         128         0
```

这个信息是我们理解 DB2 内存使用的一个最重要的内容之一，它告诉我们整个实例的内存分配情况。在上面的例子中，实例的内存上限为 10670724KB，当前使用 366400KB，高水位 366656KB。其中，应用程序内存段对于 SAMPLE 数据库占用了 160000KB，实例共享内存占用 49920KB，FMP 资源占用 22528KB，私有内存为 20352KB，SAMPLE 数据库共享内存为 112832KB，剩下的几个 LCL 是本地连接时所创建的共享内存空间，用于本地进程信息交换。

而 **Cached** 这一项可以理解为在已经分配的内存中，有多少内存是还没有被真正使用。就拿 **APPL-SAMPLE** 内存段理解，当前分配的内存大小为 **160000KB**，但是由于系统没有什么应用连接，所以这部分内存中很大一部分还是空闲的，而 **DB2** 就会把这部分标识为 **Cached**，告诉用户这些内存尽管在逻辑上分配了，但是并没有被真正提交或者使用。

阅读了前面内存模型的读者可能还记得，在实例中会有多个不同的内存段，每一个内存段中可能会有一个或者多个内存池。而下面的 4 个命令就可以从实例和数据库级别取得内存段与内存池的具体信息：

```
(db2inst1@db2host1) /home/db2inst1 $ db2pd -memsets
```

```
Database Partition 0 -- Active -- Up 1 days 07:55:32 -- Date 01/14/2011 06:39:40
```

```
Memory Sets:
```

Name	Address	Id	Size(Kb)	Key	DBP	Type
Unrsv(Kb)	Used(Kb)	HWM(Kb)	Cmt(Kb)	Uncmt(Kb)		
DBMS	0x0780000000000000	578814120	49920	0x2EA98F61	0	10240
13824	13824	13824	36096			
FMP	0x0780000010000000	414187574	22592	0x0	0	2
0	704	22592	0			
Trace	0x0770000000000000	6291626	137576	0x2EA98F74	0	0
137576	0	137576	0		-1	

在 **db2pd -memsets** 中，**db2pd** 依然沿用 8/9.1 的设置，不显示数据库相关的内存段信息。在上面的输出中，我们可以看到实例共享内存段的大小为 **49920**，其中 **13824KB** 为已提交内存，**36096KB** 为未提交内存。通过对比 **ID** 列与 **ipcs -a** 的输出，我们可以将该内存段映射到操作系统级共享内存 **IPC** 段：

```
(db2inst1@db2host1) /home/db2inst1 $ ipcs -a | grep 578814120
```

```
m 578814120 0x2ea98f61 --rw----- db2inst1 build db2inst1 build 6
```

```
51118080 34111918 57487698 6:49:59 6:49:59 22:44:08
```

FMP 与 **trace** 内存段没有什么太多需要讨论的，一般这两个内存段很少造成性能问题。

如果需要深入到内存池，则需要通过 **db2pd -mempool** 实现：

```
(db2inst1@db2host1) /home/db2inst1 $ db2pd -mempool
```

```
Database Partition 0 -- Active -- Up 1 days 08:07:55 -- Date 01/14/2011 06:52:03
```

```
Memory Pools:
```

Address	MemSet	PoolName	Id	Overhead	LogSz	LogUpBnd	LogHWM
PhySz	PhyUpBnd	PhyHWM	Bnd	BlkCnt	CfgParm		
0x07800000000012A8	DBMS	fcm	74	0	0	1931054	0
0	1966080	0	0vf 0	n/a			
0x0780000000001160	DBMS	fcmssess	77	65376	1401568	1687552	
1401568	1572864	1703936	1572864	0vf 3	n/a		
0x0780000000001018	DBMS	fcmchan	79	65376	259584	507904	259584
393216	524288	393216	0vf 3	n/a			

0x07800000000000ED0	DBMS	fcmbp	13	65376	656896	925696	656896
851968	983040	851968	Ovf 3	n/a			
0x07800000000000D88	DBMS	fcmctl	73	111872		1594221	8675488
1594221	1769472	8716288	1769472	Ovf 11	n/a		
0x07800000000000C40	DBMS	monh	11	122496	150071	368640	153431
327680	393216	327680	Ovf 26	MON_HEAP_SZ			
0x07800000000000AF8	DBMS	resynch	62	41216		155320	2752512
155320	262144	2752512	262144	Ovf 2	n/a		
0x078000000000009B0	DBMS	apmh	70	4512		1262788	7929856
1263588	1441792	7929856	1441792	Ovf 73	n/a		
0x07800000000000868	DBMS	kerh	52	96	819552	4128768	874976
917504	4128768	983040	Ovf 96	n/a			
0x07800000000000720	DBMS	bsuh	71	0	766408	15138816	879788
917504	15138816	917504	Ovf 103	n/a			
0x078000000000005D8	DBMS	sqlch	50	0		2683025	2752512
2683025	2752512	2752512	2752512	Ovf 203	n/a		
0x07800000000000490	DBMS	krcbh	69	0	146824	131072	147088
196608	131072	196608	Ovf 18	n/a			
0x07800000000000348	DBMS	eduah	72	65440		2621384	2621408
2621384	2686976	2621440	2686976	Ovf 1	n/a		
0x07800000100000348	FMP	undefh	59	40000	614500	22971520	614500
655360	23003136	655360	Phy 5	n/a			

这个输出的第二例代表该内存池所属的内存段，譬如 DBMS 或者 FMP，第三列为该内存池的名称。我们可以看到前 5 个内存池为 FCM 相关内存，接着是 monitor heap，然后剩下的 resynch, apmh 等都为 DB2 内部所使用的内存池，不需要调整参数。第六列为逻辑大小；第九列为物理大小。一般来说，物理大小都要稍大于逻辑大小，而这两者有什么区别呢？DB2 拥有自己的内存管理模块，当一段程序请求内存管理模块分配一块内存的时候，管理模块会按照锁请求的内存大小分配一块稍大的以标准单位为倍数的内存。这个标准单位是多少呢？在不同的内存段中可能都不大相同，而且在版本之间可能会略有区别，所以没有深究的意义。用户只要知道，物理大小指的是 DB2 对于这个内存池真正分配了多少内存，而逻辑大小指的是 DB2 真正使用了所分配的内存的多少。

通过这些信息，用户可以了解实例下各个模块内存分配的大小，如果需要调整，可以根据最后一列调整相应的内存参数。其中 n/a 是说该内存是 DB2 内部维护的，没有相关的参数可以调节。

下一步就进入了数据库级别。对于实例下所有的数据库，都要收集 db2pd -db <dbname> -memsets 与 db2pd -db <dbname> -mempool:

```
(db2inst1@db2host1) /home/db2inst1 $ db2pd -db sample -memsets

Database Partition 0 -- Database SAMPLE -- Active -- Up 1 days 00:24:35 -- Date
01/14/2011 07:02:00

Memory Sets:
Name                Address                Id                Size(Kb)    Key                DBP                Type
```

Unrsv(Kb)	Used(Kb)	HWM(Kb)	Cmt(Kb)	Uncmt(Kb)				
SAMPLE	0x0700000030000000	954204320	112896	0x0	0	1	18240	
44672	54656	55296	57600					
AppCtl	0x0700000020000000	46137518	160064	0x0	0	12	0	
4608	7680	11456	148608					
App960	0x0000000000000000	0	0	0x0	0	0	0	
0	0	0	0					
App953	0x0000000000000000	0	0	0x0	0	0	0	
0	0	0	0					
App959	0x0000000000000000	0	0	0x0	0	0	0	
0	0	0	0					
App2485	0x0000000004D00073	80740467	128	0x0	0	4	0	
128	0	128	0					
App952	0x0000000000000000	0	0	0x0	0	0	0	
0	0	0	0					
App958	0x0000000000000000	0	0	0x0	0	0	0	
0	0	0	0					
App951	0x0000000000000000	0	0	0x0	0	0	0	
0	0	0	0					
App957	0x0000000000000000	0	0	0x0	0	0	0	
0	0	0	0					
App2108	0x000000001F300028	523239464	128	0x0	0	4	0	
128	0	128	0					
App950	0x0000000000000000	0	0	0x0	0	0	0	
0	0	0	0					
App956	0x0000000000000000	0	0	0x0	0	0	0	
0	0	0	0					
App949	0x0000000000000000	0	0	0x0	0	0	0	
0	0	0	0					
App2416	0x0000000006F000B5	116392117	128	0x0	0	4	0	
128	0	128	0					
App955	0x0000000000000000	0	0	0x0	0	0	0	
0	0	0	0					
App948	0x0000000000000000	0	0	0x0	0	0	0	
0	0	0	0					
App2612	0x000000000790004A	126877770	128	0x0	0	4	0	
128	0	128	0					
App954	0x0000000000000000	0	0	0x0	0	0	0	
0	0	0	0					

类似实例的 `memset`，对于数据库的 `memset` 显示的是与数据库相关的内存段，其中包括数据库内存段、应用程序控制段和应用程序段。每个段的大小也在第四列指定。

内存池的信息则需要 `mempool` 显示：

```
(db2inst1@db2host1) /home/db2inst1 $ db2pd -db sample -mempool

Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 06:34:16 -- Date
01/14/2011 16:46:26
```

Memory Pools:								
Address	MemSet	PoolName	Id	Overhead	LogSz	LogUpBnd	LogHWM	
PhySz	PhyUpBnd	PhyHWM	Bnd BlkCnt	CfgParm				
0x0700000040001910	SAMPLE	utilh	5	0	7200	20512768	11348	
65536	20512768	65536	Ovf 30	UTIL_HEAP_SZ				
0x0700000040001680	SAMPLE	pckcacheh	7	63584	1244897	Unlimited		
1318707	1507328	Unlimited	1507328	Ovf 408	PCKCACHESZ			
0x0700000040001538	SAMPLE	xmlcacheh	93	48832	147616	20971520		
147616	196608	20971520	196608	Ovf 1	n/a			
0x07000000400013F0	SAMPLE	catcacheh	8	48000	412733	Unlimited		
416573	524288	Unlimited	524288	Ovf 86	CATALOGCACHE_SZ			
0x0700000040001160	SAMPLE	bph	16	0	4504384	Unlimited		
4504384	4587520	Unlimited	4587520	Ovf 33	n/a			
0x0700000040000ED0	SAMPLE	bph	16	0	783104	Unlimited	783104	
851968	Unlimited	851968	Ovf 5	n/a				
0x0700000040000C40	SAMPLE	bph	16	0	520960	Unlimited	520960	
589824	Unlimited	589824	Ovf 3	n/a				
0x07000000400009B0	SAMPLE	bph	16	0	389888	Unlimited	389888	
458752	Unlimited	458752	Ovf 2	n/a				
0x0700000040000720	SAMPLE	bph	16	0	324352	Unlimited	324352	
393216	Unlimited	393216	Ovf 2	n/a				
0x07000000400005D8	SAMPLE	shsorth	18	0	0	20512768	122880	
131072	20512768	262144	Ovf 0	SHEAPTHRES_SHR				
0x0700000040000490	SAMPLE	lockh	4	0	606848	720896	606848	
655360	720896	655360	Ovf 1	LOCKLIST				
0x0700000040000348	SAMPLE	dbh	2	366592	34429632	43712512		
34472563	35192832	43712512	35192832	Ovf 2222	DBHEAP			
0x0700000030001910	AppCtl	apph	1	0	11482	1048576	82136	
131072	1048576	131072	Phy 25	APPLHEAPSZ				
0x0700000030001680	AppCtl	apph	1	0	10891	1048576	10891	
65536	1048576	65536	Phy 14	APPLHEAPSZ				
0x0700000030001538	AppCtl	apph	1	0	10936	1048576	10936	
65536	1048576	65536	Phy 17	APPLHEAPSZ				
0x07000000300013F0	AppCtl	apph	1	0	10936	1048576	10936	
65536	1048576	65536	Phy 17	APPLHEAPSZ				
0x07000000300012A8	AppCtl	apph	1	0	10936	1048576	10936	
65536	1048576	65536	Phy 17	APPLHEAPSZ				
0x0700000030001160	AppCtl	apph	1	0	10936	1048576	10936	
65536	1048576	65536	Phy 17	APPLHEAPSZ				
0x0700000030001018	AppCtl	apph	1	0	10936	1048576	10936	
65536	1048576	65536	Phy 17	APPLHEAPSZ				
0x0700000030000ED0	AppCtl	apph	1	0	10936	1048576	10936	
65536	1048576	65536	Phy 17	APPLHEAPSZ				
0x0700000030000D88	AppCtl	apph	1	0	10936	1048576	10936	
65536	1048576	65536	Phy 17	APPLHEAPSZ				
0x0700000030000C40	AppCtl	apph	1	0	10936	1048576	10936	
65536	1048576	65536	Phy 17	APPLHEAPSZ				
0x0700000030000AF8	AppCtl	apph	1	0	92856	1048576	93360	
196608	1048576	196608	Phy 1041	APPLHEAPSZ				
0x07000000300009B0	AppCtl	apph	1	0	10936	1048576	10936	

65536	1048576	65536	Phy 17	APPLHEAPSZ			
0x0700000030000868	AppCtl	apph	1	0	10936	1048576	12920
65536	1048576	65536	Phy 17	APPLHEAPSZ			
0x0700000030000720	AppCtl	apph	1	0	10936	1048576	10936
65536	1048576	65536	Phy 17	APPLHEAPSZ			
0x0700000030000490	AppCtl	apph	1	0	20979	1048576	82136
131072	1048576	131072	Phy 40	APPLHEAPSZ			
0x0700000030000348	AppCtl	appshrh	20	11648	745124	81920000	
773240	917504	81920000	917504	Phy 258	application shared		

其中第二列是每一个内存池所属的内存段,第三列是内存池名称。比如 utilh 代表 utility heap、bph 代表 bufferpool heap。其余列的信息与实例级内存池信息类似。

最后, 8.1 版本不支持 db2pd 工具, 必须使用 db2mtrk 工具。该工具在 8.2 后的版本继续提供, 但在内存分析时并不经常用到。由于 db2mtrk 对内存分配的跟踪方式与 db2pd 截然不同, 因此用户会在两者之间看到一些差异。一般来说, 在 db2pd 工具存在的前提下, 使用 db2pd 得到的结果更加实用。

```
(db2inst1@db2host1) /home/db2inst1 $ db2mtrk -i -d -p -v
Tracking Memory on: 2011/01/14 at 16:49:28

Memory for instance

Other Memory is of size 12582912 bytes
FCMBP Heap is of size 851968 bytes
Database Monitor Heap is of size 327680 bytes
Total: 13762560 bytes

Memory for database: SAMPLE

Backup/Restore/Util Heap is of size 65536 bytes
Package Cache is of size 1507328 bytes
Other Memory is of size 196608 bytes
Catalog Cache Heap is of size 524288 bytes
Buffer Pool Heap (1) is of size 4587520 bytes
Buffer Pool Heap (System 32k buffer pool) is of size 851968 bytes
Buffer Pool Heap (System 16k buffer pool) is of size 589824 bytes
Buffer Pool Heap (System 8k buffer pool) is of size 458752 bytes
Buffer Pool Heap (System 4k buffer pool) is of size 393216 bytes
Shared Sort Heap is of size 131072 bytes
Lock Manager Heap is of size 655360 bytes
Database Heap is of size 35192832 bytes
Application Heap (4667) is of size 131072 bytes
Application Heap (3960) is of size 65536 bytes
Application Heap (3959) is of size 65536 bytes
Application Heap (3958) is of size 65536 bytes
Application Heap (3957) is of size 65536 bytes
Application Heap (3956) is of size 65536 bytes
Application Heap (3955) is of size 65536 bytes
```

```
Application Heap (3954) is of size 65536 bytes
Application Heap (3953) is of size 65536 bytes
Application Heap (3952) is of size 65536 bytes
Application Heap (3951) is of size 196608 bytes
Application Heap (3950) is of size 65536 bytes
Application Heap (3949) is of size 65536 bytes
Application Heap (3948) is of size 65536 bytes
Application Heap (3947) is of size 131072 bytes
Applications Shared Heap is of size 917504 bytes
Total: 47316992 bytes
```

Memory for agent 8876

```
Other Memory is of size 458752 bytes
Total: 458752 bytes
```

Memory for agent 7402

```
Other Memory is of size 196608 bytes
Total: 196608 bytes
```

Memory for agent 5091

```
Other Memory is of size 196608 bytes
Total: 196608 bytes
```

Memory for agent 7145

```
Other Memory is of size 196608 bytes
Total: 196608 bytes
```

Memory for agent 4834

```
Other Memory is of size 196608 bytes
Total: 196608 bytes
```

Memory for agent 6888

```
Other Memory is of size 196608 bytes
Total: 196608 bytes
```

Memory for agent 4577

```
Other Memory is of size 196608 bytes
Total: 196608 bytes
```

Memory for agent 6631

```
Other Memory is of size 196608 bytes
Total: 196608 bytes
```

```
Memory for agent 4320

    Other Memory is of size 196608 bytes
    Total: 196608 bytes

Memory for agent 7660

    Other Memory is of size 196608 bytes
    Total: 196608 bytes

Memory for agent 6374

    Other Memory is of size 196608 bytes
    Total: 196608 bytes

Memory for agent 5843

    Other Memory is of size 196608 bytes
    Total: 196608 bytes

Memory for agent 5605

    Other Memory is of size 196608 bytes
    Total: 196608 bytes

Memory for agent 6100

    Other Memory is of size 327680 bytes
    Total: 327680 bytes

Memory for agent 5348

    Other Memory is of size 196608 bytes
    Total: 196608 bytes
```

15.1.3 数据收集的频度 ■ ■ ■

首先，用户需要了解什么是“性能”。在维基百科的定义中我们可以查到“**performance is characterized by the amount of useful work accomplished by a computer system compared to the time and resources used.**”，简单地说，性能就是在给定的时间和资源内，一个计算机系统所能处理的工作量。“给定的时间内”这个前提绝对不能被忽略。在作者的工作中，发现很多用户在处理性能问题时，经常收集一组快照之后就问：我们能不能从中看出什么问题？要知道，快照是一个给定的时间点下系统所处的状态。而根据“性能”的定义，我们至少要有一个给定的时间段，而不是时间点。

怎样确定一个时间段？两点确定直线，也就是在两个时间点中的过程才能称为一个时间段。

所有的性能分析都需要将数据标准化到一个给定的时间段中，可以是一个小时，也可以是一秒钟。

譬如，用户分别在 10:30、10:36、11:00 收集了 3 组数据。10:30 时数据库的逻辑读为 10000；10:36 时数据库的逻辑读为 13000；而 11:00 时数据库的逻辑读为 23000。通过这几组点，我们怎样了解系统在这 30 分钟内逻辑读的变化呢？我们可以用两个快照之间数值的差异除以时间，即可得到标准化后的指标。在这个例子中，10:30 至 10:36 的逻辑读为 $(13000-10000)/((10:36-10:30)*60)=8.3$ 每秒。而 10:36 至 11:00 的逻辑读为 $(23000-13000)/((11:00-10:36)*60)=6.9$ 每秒。这样我们可以清楚地看到，系统在 10:36 至 11:00 期间逻辑读的速率低于起始时间，大概降低了 16.8%。

因此，在分析性能数据时，一定要在不同时间点收集至少 3 次以上的数据才有分析的意义。针对仅仅一次的数据收集进行分析，在很大程度上是浪费时间。同时，数据只有在标准化以后才能够比较，单纯地用收集到的数值比较是没有意义的。

在明确了收集多次数据和量化数据这两个概念后，另外一个概念就是频率。

对于作者来说，在一个运行典型任务的系统中，每小时收集一次数据用来做普通的健康检查是一个比较合理的频率。这样，每天系统将会收集 24 次各种快照，通过这些数据，用户可以清楚地分析出什么时间段在运行什么业务，会对系统性能造成什么影响。

但是在一些特定的环境下，例如系统性能在每天的特定时刻突然下降，然后 10 分钟内恢复正常。针对这种问题，我们必须调整数据收集的频率，而其数值则取决于系统的业务量与所收集数据的多少。比如对于某个已经确诊为 CPU 使用率高的问题，我们可以不需要收集内存相关的信息，也不需要收集锁相关信息，这样能够减少很多数据收集产生的开销，也就能够让我们增加数据收集的密度，例如可以达到 1~2 分钟一次的频率。

说到数据收集产生的额外开销，顺便提一下事件监控器的使用。在一个繁忙的系统中，事件监控器会对性能造成非常重大的影响。举例来说，在一个平均每秒钟查询数量超过 10000 的 OLTP 环境下，如果打开 Statement 事件监控器，每一个查询都需要在执行前向事件监控器写入信息，会造成平均运行 1 毫秒的查询需要用额外的 5 毫秒写入监视信息，造成系统性能的急剧下降。

当然，死锁事件监视器对系统性能基本没有影响，而且几乎是唯一能够有针对性研究死锁问题的资源，强烈建议不要关闭（默认状态是开启的）。

以下是可能需要收集的信息，可以根据需要自行添加或者减少收集的数据项。建议将其可以写到脚本中，按照一定的频率调度：

```
#每台服务器
vmstat 1 5
iostat 1 5
svmon -G (AIX)或者 free (Linux)
ps -elf
```

```

ipcs -a
db2greg -dump
db2pd -osinfo
#系统中每个实例
    GET SNAPSHOT FOR DATABASE MANAGER
    db2pd -edus
    db2pd -dbptnmem
    db2pd -memsets
    db2pd -mempool
    db2mtrk -i -d -p -v
        #实例中每个数据库，每个分区
        GET SNAPSHOT FOR DATABASE ON <数据库名>
        GET SNAPSHOT FOR APPLICATIONS ON <数据库名>
        GET SNAPSHOT FOR TABLESPACES ON <数据库名>
        GET SNAPSHOT FOR TABLES ON <数据库名>
        GET SNAPSHOT FOR DYNAMIC SQL ON <数据库名>
        db2pd -db <数据库名> -locks -trans -app
        db2pd -db <数据库名> -memsets
        db2pd -db <数据库名> -mempool

```

15.1.4 小结 ■ ■ ■

本章我们着重讨论了数据收集和监控。数据的收集是性能分析的前提，在没有准确与详细的数据的前提下，即使再出色的性能分析专家也会一筹莫展。在下一小节中，我们将会详细讨论如何从如此繁多的数据中一步步地找到性能问题的瓶颈。

15.2 分析数据

正如前文中所讨论到的，分析数据必须建立在正确的数据和时间、足够且合理频率的采样的基础之上，通过标准化若干个时间点之间的性能统计数据，用户可以对一个系统在一定时间内的行为有一个全面而准确的了解。

当我们明白了系统的行为之前，绝对不要急于分析某一个特定模块（例如 CPU、I/O 等）。如果不能在第一时间找准大方向，只会在错误的方向上越走越远。

15.2.1 瓶颈分类与原理介绍 ■ ■ ■

一般来说，所有的性能问题都可以归结于以下 4 个大类中的一个或者若干个：

- CPU 瓶颈。
- I/O 瓶颈。
- 内存瓶颈。

- 系统懒惰。

下面，让我们来详细说明一下这 4 个大类。

1. CPU 瓶颈

顾名思义,CPU 瓶颈就是说系统需要处理的任务超出了 CPU 能够在给定时间内处理的极限。

在理解 CPU 瓶颈之前,我们首先要了解什么是 CPU 占用率。在计算机架构中,一个 CPU 内核在给定时刻只能处理单一的任务(超线程情况下可以认为是两个)。操作系统中的进程调度系统每隔一段时间被调用一次,用来重新选择该 CPU 需要执行的进程。当进程调度系统被触发的频率足够快时,从使用者的角度看,各个进程之间的执行为并行。

可是 CPU 调度程序是怎么选择应该执行什么进程呢?

每种操作系统的实现方法都略有区别,可是大体上都是通过优先队列进行选择的。每一个进程都有一个自己的优先级,当进程调度系统选择了一个进程的时候,就会略微降低其优先权,同时略微上升其他没有被选择到的进程的优先权。在选择的时候,调度程序简单地选择优先权最高的请求执行的进程。这样,如果一个进程被经常执行,其优先权就会相比其他进程低,这样当其他进程请求执行的时候,调度程序就会优先选择其他进程,从而达到并发的效果。

当整个系统中没有任何被请求执行的进程,那么调度程序就会选择一个默认永远执行,但优先权永远保持最低的进程不断循环执行一些没有意义的代码,比如在 AIX 中就是 wait 进程,在 Windows 中就是 System Idle Process 进程。而 CPU 使用率就是在单位时间内,CPU 选择到非等待进程的比例。当 CPU 执行一个进程的时候,每一个进程在操作系统内部都有一个控制块,定义着一些进程相关的信息。例如其中就包括当前执行到哪一条指令这种信息。

进程调度程序要做的就是把把这些信息装载到当前的 CPU Register 中,然后把执行权力交还给 CPU。

所谓用户 CPU 和系统 CPU 的区别就是当前执行的指令是在用户程序之中,还是在其所调用的操作系统内核程序之中。

所以,不管是常用的 vmstat,还是更加强大的 tprof,所做的就是高速采样(或者跟踪所有的调度)并且计算比率。如果在一万次采样中有五千次在运行用户程序,一千次在运行内核程序,两千次在等待 I/O(没有其他任何可以用来执行的进程,剩下的就是在 I/O 等待的进程),剩下的两千次在执行等待进程,那么我们就可以认为用户 CPU 占用率为 50%,系统 CPU 占用率 10%,I/O 等待 20%,空闲 20%。

那么我们就知道了,CPU 占用率也是在一个时间段的基础上判断的,而不是时间点。这样我们就可以得到更广义上的 CPU 占用。假设我们知道了系统有 X 颗 CPU,在给定的 Y 时间内,数据库所有代理线程使用的 CPU 为 Z(被调度程序选择、执行,并被调出之间的时间),那么该数据库对于整体系统 CPU 的占用率为 $100 * Z / (X * Y)$ 。这里, $X * Y$ 的积就是在 Y 时间内系统总共拥有的 CPU 时间,而 Z 就是数据库消耗的总时间。

CPU 瓶颈就是 CPU 在一段时间内保持非常繁忙的状态。我们知道进程调度系统的触发非常频繁，如果每一次被触发时都发现系统中有需要 CPU 执行的进程，那么就可以认为系统不能在请求执行的进程发出请求后立刻将其执行。这样从用户的角度来看，进程无法在给定的时间段内得到足够多的 CPU 执行时间，那么就是性能无法到达预期指标。

```
(db2inst1@db2host1) /home/db2inst1 $ vmstat 1 5

System configuration: lcpu=8 mem=12288MB ent=0.40

kthr      memory          page        faults          cpu
-----
r  b   avm   fre re  pi  po  fr   sr  cy  in   sy  cs  us  sy  id  wa   pc   ec
3  0 3575594 96839  0  0  0  0    0  0 959 165301 122143 53 30 12  4  2.93 732.6
4  0 3576397 96035  0  0  0  0    0  0  95 173056 142201 60 21 18  1  2.76 689.5
10 0 3577207 81159  0  0  0  0    0  0 784 182273 133386 53 29  7 11  2.87 717.5
6  0 3576774 74972  0  0  0  0    0  0 803 139161 128984 51 30  7 11  2.80 699.4
6  2 3575318 66185  0  0  0  0    0  0 329 173138 131693 56 26 14  4  2.92 730.3
```

在这个例子中，我们可以看到 us 与 sy 两列的总和大约占 80%，也就是说该系统的 CPU 占用率大概为 80%。

2. I/O 瓶颈

当一个进程发出一个 I/O 请求的时候，这个请求首先被操作系统内核得到，这时操作系统内核就会把这个进程做一个特殊的标识，表示该进程正在 I/O 等待。然后操作系统会把这个 I/O 请求发送给 I/O 调度程序，然后 I/O 调度程序就会根据这个请求，把它放入相应驱动器的 I/O 请求队列中。

当 I/O 驱动器从队列中拿到这个请求，物理移动硬盘磁头，等待盘片旋转到相应的扇区，通过电磁效应将保存在磁盘上的数据读入后，把相应的数据保存，返还给操作系统，并发出一个中断请求。当操作系统发现请求返回以后，就会把这些读取的数据放到进程指定的内存中，然后把进程重新放入等待执行的队列，这样下一次 CPU 调度程序就能够选择该进程继续执行。

从 CPU 调度程序的角度来看，每次选择需要执行的进程时，都从优先权最高的那一队列开始。如果最高的那一队列没有进程，就选择次高优先级队列，直到找到需要执行的进程。在寻找的过程中，调度程序并不会寻找等待队列中的进程(这是大概的理论，具体的实现方法不同操作系统之间，甚至同一系统的不同版本之间会略有区别)。

当所有的队列都寻找完毕，并不存在任何需要立刻执行的进程，假如等待队列中存在一个以上的 I/O 等待的进程，那么系统的状态就是 I/O 等待，否则就是空闲。这样，如果 vmstat 结果显示系统的等待时间过长，可以说明两件事：

- 没有大量的 CPU 执行请求。
- 经常存在处于 I/O 等待状态的进程。

这就是说，可能我们的进程需要经常从磁盘上读/写数据，而在每次读/写的时候，其等待时

间可能只有 5 毫秒，但是在 CPU 看来仿佛过了几年一样长，这样我们就可以认为，进程执行缓慢是由于需要经常等待 I/O，而不是无法得到 CPU 时间。因此，I/O 瓶颈是指，在一段时间内系统的采样中，请求 CPU 的进程占少数，并且存在很多等待 I/O 的进程。也就是说，系统的绝大部分的时间都花在 I/O 等待上。

```
(db2inst1@db2host1) /home/db2inst1 $ vmstat 1 5

System configuration: lcpu=8 mem=12288MB ent=0.40
```

kthr		memory				page				faults				cpu							
r	b	avm	fre	re	pi	po	fr	sr	cy	in	sy	cs	us	sy	id	wa	pc	ec			
9	2	3575474	115253	0	0	0	0	0	0	0	1688	149842	133991	6	1	12	81	0.25 62.5			
5	1	3575474	117884	0	0	0	0	0	0	0	1947	155883	134784	3	0	15	82	0.23 57.5			
3	0	3575521	108543	0	0	0	0	0	0	0	1835	157259	133759	3	0	16	81	0.23 57.5			
7	0	3574286	103290	0	0	0	0	0	0	0	998	183121	140385	2	0	16	81	0.22 55.0			
3	1	3575162	92172	0	0	0	0	0	0	0	297	172801	138019	2	1	15	82	0.24 60.0			

在这个例子中，我们可以看到 us 与 sy 两列的总和非常低，即该系统的 CPU 占用率很低，但是 wa 列数值很高，也就是说大部分系统运行时间都在等待 I/O。

3. 内存瓶颈

内存瓶颈可以分为两个情况来考虑。一个情况是操作系统中还有足够多的空闲内存，但是应用程序并没有请求足够多的内存；而另一个情况正好相反，应用程序希望得到足够多的内存，但是操作系统中没有足够的空闲内存，因而只能使用交换分区。

第一个情况可以认为是应用程序优化的问题，我们在下一章中性能调优会进行介绍，而第二个情况就是典型的内存瓶颈问题，可能是应用不够优化，也可能是硬件不够强大，甚至可能是应用程序的 bug。在本章我们着重讨论这种情况。

想要理解内存瓶颈，首先要理解操作系统怎样使用内存。

可能很多读者都会这样认为，内存不就是插在主板上的几根内存条吗，操作系统需要保存什么东西就放到里面，需要读东西就用某种方法拿出来，还有什么新鲜的呢？这样理解没错，可是距离我们想要了解的程度就相差太远了。

首先，正像大家了解的那样，计算机只能理解 1 和 0，在电容器和晶体管中，就会被代表为高电位和低电位，而内存可以看成是这种电容器与晶体管的大集合，当设备加电的时候，就能够保存每个电容器晶体管的状态，从而达到临时存储数据的目的。

对于操作系统来说，内存就是其除了一级和二级缓存以外的所有记忆体。由于一级和二级缓存在数据库级别来说基本属于不可控的（除非是对于数据库内核与操作系统内核的开发人员），我们着重讨论内存模块。

在操作系统中，内存就是相当于人类的大脑负责记忆的部分。一切 CPU（大脑思考的部分）

需要的数据都要能够从内存中找到。可是由于成本原因，内存的价格一直很昂贵，所以一台系统没可能把所有的数据全都放在内存中。

这样，随着越来越多的数据被读入内存使用，万一某一时刻内存全部耗尽了怎么办？

从理论上说，如果操作系统真的一点内存都没有了，就会造成内核程序的混乱，最终结果就是系统崩溃，或者其他任何诡异的事情都可能发生。因此，确保操作系统一直都有可用的剩余内存是非常关键的。但是人都比较贪心，每个花钱买了机器的人都想在上面运行尽可能多的业务（比如买了 1GB 内存电脑的人可能上面开 10 个窗口上网，买了 2GB 内存的就希望上面同时听着 MP3，后台开 5 个下载等）。所以为了那些有可能使用超出物理内存大小的业务，操作系统又引入了一个新的概念，即交换分区。

交换分区就是在硬盘上单独划分出一部分作为内存的延伸。当内存不够的时候就可以把不需要的数据从物理内存中转移到磁盘上。当然，如果这部分数据再次被需要的时候，系统还需要把这部分数据读进去。这个操作就叫做换页。把磁盘的数据读入物理内存叫做换入(page in)；而把物理内存写到磁盘上就叫做换出(page out)。

在 vmstat 中，单位时间内换入/换出的次数在 pi 和 po 两列显示：

```
(db2inst1@db2host1) /home/db2inst1/temp $ vmstat 1 5

System configuration: lcpu=8 mem=12288MB ent=0.40

kthr    memory                page                faults                cpu
-----
 r  b   avm   fre re  pi  po  fr   sr  cy  in   sy  cs us sy id wa   pc   ec
5  0 3581548 126333  0  0  0  0   0  0 2183 157678 137535 54 27 16  3  2.97 742.3
4  0 3581492 115553  0  0  0  0   0  0 2147 153813 136615 54 27 17  3  2.92 730.1
4  0 3580140 101085  0  0  0  0   0  0 1141 194272 122920 54 30 14  3  3.01 752.4
5  0 3581023 100202  0  0  0  0   0  0  148 166163 142600 58 23 19  0  2.82 703.8
6  0 3582140 82182  0  0  0  0   0  0 980 159421 136134 52 30  7 10  2.90 725.8
```

这样物理内存加上交换空间，其总大小就是操作系统能够直接操作的部分，也叫做虚拟内存。需要注意的是，前面内存模型部分中我们还提到了一个进程虚拟内存空间，与这个虚拟内存并不一样，我们在这一节稍后详细介绍。虚拟内存指的是操作系统可以访问的全部内存空间，在 db2pd -osinfo 输出中可以看到：

```
Physical Memory and Swap (Megabytes):
TotalMem    FreeMem    AvailMem    TotalSwap    FreeSwap
12288       611         n/a         12288        8973

Virtual Memory (Megabytes):
Total      Reserved    Available    Free
24576      n/a         n/a         9584
```

这里，物理内存（TotalMem）为 12288 兆，交换空间（TotalSwap）同样为 12288 兆，而虚拟内存总数为 24576 兆。

在操作系统中，每一个内存可操作单元都有一个固定大小的尺寸。在大部分操作系统的默认设置下都是 4096 字节，我们把一个 4096 字节的内存段叫做页。

在某些操作系统中，可以有超过一种大小的内存页。譬如在 AIX 中，除了 4096 字节的小内存页，同时也会维护 65536 字节的中内存页和 16 兆的大内存页。

类似数据库数据页大小的区别，数据页越大可存储的连续数据就越多，对大段的数据访问性能就会更好；但是如果用户数据在大内存页中只占据很小一部分，则大内存页可能会浪费空间。

对比起一个普通的内存读操作，如果操作系统发现需要的数据在磁盘上需要换入，则需要做什么操作呢？

- (1) 首先确定数据在磁盘上的位置。
- (2) 从物理内存中找到一块空闲的页面。
- (3) 把磁盘上的数据读入物理内存。
- (4) 更新操作系统内核中的页面表。
- (5) 把控制权返回给应用程序。

以上的这些操作对应用程序来说都是透明的，比如应用程序的一句 x86 汇编代码：

```
mov ebx, [eax]
```

这个指令指的是，把 `eax` 寄存器所指向的地址所在的内存中读入 32 位数放入 `ebx` 寄存器中。

这句代码在应用程序看来再正常不过，就是从内存中读入 32 位的一个数值而已。不过对于操作系统来说，假设应用程序需要读取的数值不在物理内存中，就需要调用换入操作的一系列动作，然后可能在 2 毫秒（算是比较快的 I/O 了）之后返回给应用程序继续执行。但是从 CPU 的角度来看，2 毫秒已经是长得不可思议的时间了。如果仅仅是普通的内存操作，也许几十纳秒就可以完成。也就是说，一个换页操作等同于上万个内存读。

这里我们想要强调的就是换页操作对性能产生的危害。一般来说当换页频繁发生时，系统大多不会仅仅产生几倍的性能下降，而是几百倍的性能下降，甚至完全瘫痪。

下一块需要讨论的内存主题是文件系统缓存。为什么操作系统中需要文件系统缓存呢？众所周知，从磁盘上读入数据的速度要比直接内存访问慢上万倍，如果有一种机制可以把经常访问的数据在内存中，岂不是能够很显著地提高性能？没错，这种机制就是文件系统缓存，并且广泛应用在大部分系统里。但是，如果应用程序不停地请求内存，而文件系统缓存也占用一部分内存，岂不是又要开始换页了？不会的，在文件系统的设计之初，设计者就考虑到如何处理文件系统数据页和应用程序数据页之间的关系了。这里我们把应用程序所请求使用的数据页叫做计算内存页，而文件系统所使用的数据页叫做非计算内存页。这样，如果应用程序不断地请求计算内存页，使得操作系统发现可用的物理内存过小，那么它就会在需要内存页的时候直接

使用非计算内存页（你的这块内存数据在硬盘上有副本，直接替换掉即可，不用引入任何 I/O）。

当然，真正的页面偷取算法比这个要复杂得多，感兴趣的读者请参见相关的操作系统文档。

根据已经讨论的话题，我们已经知道：

- 内存决不能耗尽，否则系统会崩溃。
- 虚拟内存分为物理内存部分和交换分区部分。
- 内存有不同的页面大小。
- 物理内存如果不够用，系统会写入磁盘（交换空间），性能急剧下降。
- 如果物理内存的消耗不多，那么操作系统会将物理内存的一部分用来存放数据页（非计算内存）。

至此，操作系统级的内存介绍部分先告一段落，下面我们来介绍进程级内存。刚才我们提到了进程虚拟内存空间，它是什么呢？

早在计算机的远古时代，操作系统并不支持多线程这个概念。整个操作系统就好像一个大线程，所有的执行都是先做一件事再做一件事，这个叫做串行系统。在这种串行系统中，所有的任务都可以访问全部内存。比如当一个用户执行了一个程序，那么在这个程序中，通过内存访问用户可以看到整个操作系统所有的内存。可是到 1969 年 Multics 操作系统诞生以后，并行计算就是人们追求的另一个方向了。因此问题产生了，怎样让不同的任务同时处理，并且互不干扰呢？答案就是进程虚拟内存空间。

所谓虚拟寻址空间，就是进程无法看到系统里面所有的内存了，每一个进程都有一块自己的虚拟内存空间，这段虚拟空间就是该进程能够访问的全部内容，并且被操作系统密切地监视着。操作系统内存管理模块有一个映射机制，把进程虚拟空间中的内存映射到真正的操作系统虚拟内存中（物理内存加上交换空间），如图 15.7 所示。

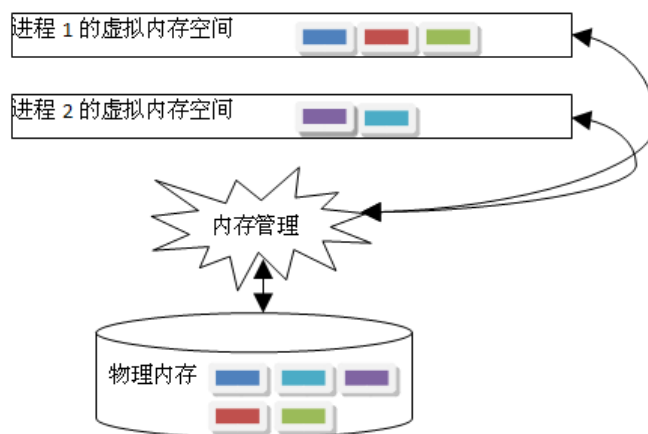


图 15.7 进程虚拟寻址空间

由于每一个进程都能够访问 32 位或者 64 位的寻址空间，对于进程来说，它们并不知道自己的这些内存地址都是虚假的。这样，操作系统允许这些进程分配一个很大的内存空间，但是在分配完成后的那一刻并不会立刻将物理内存中的内存页保留给相应的进程。相反，操作系统就像一个贪婪的地主，尽可能给自己保留内存。所以当进程分配了空间以后，操作系统只是说“哦，我知道了，我给你分配了 100 兆内存，放在你的虚拟地址 0x12345678 这个地方”。但是这个时候操作系统并不会真正地把 100 兆的物理内存（或者交换空间）指定给这个进程。

这时进程也不知道自己被开了空头支票，所以高高兴兴地认为自己已经拥有了 100 兆内存。这时进程开始使用这些内存。假设这个进程就是简单地向这 100 兆内存中的起始位置写入 100 个英文字符，这 100 个英文字符仅仅占 100 字节。这个时候操作系统看到进程真的开始用内存了，这才心不甘情不愿地找出一个空闲的物理内存页面（4096 字节），把进程的虚拟内存页面映射到物理内存，然后操作系统才会告诉应用程序内存写操作已经成功了。

这一步操作系统将虚拟内存页面映射到物理内存页面的过程叫做内存页提交，只有提交的内存页才能算是真正意义上的被使用了。而所谓“分配”的内存，只不过是进程自己一厢情愿的想法，以为自己得到了那么多的内存，实际上只是空头支票。

然后我们的进程继续写，直到写操作突破了 4096 字节，进入了一个新的内存页面，然后这个写请求发到操作系统的时候，操作系统才不得不又给进程分配一个物理页面，映射到进程虚拟空间。

总之，只有当一个物理页面真正被进程使用到了，这个物理页面才会被提交。

上面的例子是真正操作系统内存模型的简化版，不过已经完全能够解释内存提交的概念了。只有上面讨论的概念都清晰了以后，我们才能够真正了解和分析内存瓶颈。

一般说来，内存瓶颈就是发现开始换页，从 vmstat 来看就是 pi 和 po 列开始出现不为 0 的数值。

当然，其他所有的性能问题、瓶颈的发生只是表象，弄明白为什么系统正在换页才是我们真正需要掌握的内容。分析内存问题是一个博大精深的领域，本书将在稍后的部分尽量多介绍一些相关知识。

4. 系统懒惰

看到这个名字，很多读者一定会想“哪个棒槌给起的名字”。笔者认为，这个名字是最能直观反应该瓶颈的词语。实际上，这个瓶颈是一个现象，也就是当 CPU、内存换页、磁盘都不繁忙的时候，系统的性能依然不好，我们的应用程序“变懒了”。

与前面 3 个瓶颈不同，懒惰问题并非是由操作系统和硬件资源的不足引起的，只有找到问题根源才能解决。

最常见的系统懒惰是锁相关问题，由于应用程序的设计问题，或者 SQL 语句不够优化，导致多个事务等待锁资源而无法继续进行业务处理，造成 CPU、内存和 I/O 资源的浪费。

系统懒惰是一个充满变数的问题，没有一个神奇的方案能够一下定位和解决问题，只有从多个方面进行测试和排查

15.2.2 性能分析思路 ■ ■ ■

在讨论了这么多的原理以后，接下来开始我们的重头戏——性能分析。本小节我们着重探讨当发生性能问题时，DBA 应该以什么方式思考，才能够更加准确有效地解决问题。具体的例子将 15.2.3 性能分析案例部分重点介绍。

从大的方向来说，性能分析与其他所有的问题分析一样，根本的思路就是一步步地细化一个复杂问题，将一个复杂问题变为一个或者多个相对简单的问题，然后从中证明或者反证，重复这个步骤，直到细化到某一个基础问题为止。

但性能问题又和其余类型的问题有着本质的不同。在普通问题发生时，一般用户都可以看到一个错误代码，或者 `db2diag.log` 中的错误信息。但是性能问题发生的时候，通常情况下用户不会看到任何错误相关的记录。这样就给分析带来很多不便，很多人在处理这类问题的时候都无从下手。

在作者多年的性能分析调优的工作中，把该过程大体上分为以下 4 个部分：

- (1) 了解系统。
- (2) 明确问题。
- (3) 细化问题。
- (4) 定位问题。

1. 了解系统

读者可能会想，自己的系统每天都在使用，难道了解得还不够吗？比如这里有两颗 CPU，8GB 的内存，机房在三楼，还需要了解什么其他的呢？一台系统作为一个运行的整体，除了 CPU 和内存以外，还有很多其他的东西。譬如说：操作系统版本；系统中有几颗 CPU，具体的时钟速度是多少；使用的存储系统型号；存储中有多少块磁盘；数据库容器是如何在这些磁盘上分布的；系统中运行什么应用程序；应用程序逻辑是什么样的，是 OLTP 还是 OLAP；业务类型一天当中是否相同；业务高峰每天的时间；最近是否有做过什么软、硬件升级等。

看了这些问题，您觉得您还能在做进一步研究之前直接回答出来吗？如果可以，那么您对您的系统非常了解，这是一个对性能分析来说非常好的开端。如果不行，也不要灰心丧气，请在必要时召集公司的系统管理员，存储管理员，应用开发小组讨论，从而全面了解系统的行为。为什么了解系统这么关键？在进行性能分析的时候，分析人员经常需要回答一些自己提出的问题，譬如“我们一秒钟 300 条查询，20000 个逻辑读，200 个物理读，这样是否正常？”。如果您是公司中唯一的 DBA，除了您自己以外将不会有任何人能够回答这个问题。如果要让作者来

说，只能告诉您，这个数值听起来是有可能合理的（真实业务中可能出现的情况），但是具体是否正常，除了您自己以外没有人能够回答。而为了回答这个问题，数据库管理员需要对系统的正常行为有一个深入的了解，然后才能在问题发生的时候迅速回答“不对，每秒的物理读在我们系统中，正常的情形下应该不会超过 50”。

可是这些细节问题我们怎样了解呢，特别是有些公司里，DBA 可能负责管理 50 个数据库，不可能对每一台系统的所有性能指标都了如指掌啊。

这个就是日常数据收集的重要性了。DBA 不需要在每时每刻都能够张口就回答自己所管理的系统的性能指标（但是基础的配置一定要印在脑子里）。毕竟几十台系统，上百个应用程序，无数程序开发人员每天都在进行代码的修改，每天都有 4 个小时以上的会议讨论如何提升业务，这样根本不可能留给 DBA 很多的时间进行细节上的监控。所以，当一个细节的问题来临的时候，一定要有历史数据能够让 DBA 了解到，在正常的业务中，我想要了解的细节参数应该是什么样的。

所以，在了解您的系统部分，用户需要注意以下两点：

- 日常工作中对系统配置、修改已经更新进行随时跟踪，了解系统运行的状态。
- 保留至少一个月以上的历史数据，做到能够在需要时立刻检查系统正常状态下的一切指标。

在了解了系统之后，下一步就是明确发生的问题。

2. 明确问题

什么是明确问题？就好像大学中讲过的软件工程课程，所有的项目所经历的第一阶段就是明确用户需求。而在性能分析中也是一样的。当系统用户给 DBA 打来电话抱怨系统运行缓慢的时候，DBA 一定要明确到底最终用户在说什么。

很多时候，最终用户并不是计算机专家，他们可能并不知道自己在说什么。有的人可能认为自己非常聪明，总想着能够在自己的专业外对计算机的东西也可以指手画脚，因此作为 DBA 我们经常可以听到“我的系统变慢了，应该是我们程序连接的后台数据库慢了，请你们 DBA 帮忙看一下”。

先别忙着打开 Telnet 连接数据库服务器。让我们仔细地和用户聊一下，到底他们遇到了什么问题，来龙去脉是什么样的。否则被一个非科班出身，平时只用家里的 PC 连接过外接硬盘的人士牵着鼻子走，可就有点太说不过去了。要记住，我们 DBA 才是分析问题的专家。

根据作者多年的经验，大多时候我们都要假设最终用户是非专业的，这样我们就可以对他们一切自我感觉良好的分析完全忽略，只需要明白“到底发生了什么事情”。

当我们询问最终用户的时候，不要问一大堆计算机领域中高深的词语，甚至不要问太多计算机相关的东西。一般我们可以类似这样询问：

- (1) 发生了什么事情。
- (2) 从什么时候开始发生的。
- (3) 现在是否依然在发生。
- (4) 发生前是否做过什么更改或特别操作。
- (5) 以前是否发生过类似问题。
- (6) 如果应用效率低下，到底有多差，一倍的差距，还是 100 倍的差距。
- (7) 如果问题不再发生了，运行同样的业务是否可以重现问题。

相信这些问题，即使是非计算机专业的最终用户也是可以回答的。到现在，我们已经了解了问题，至少我们明白，问题是否可以重现，是否严重影响业务，是否和可能存在的软、硬件升级相关。

下一步，我们就要开始细化问题了，也就是性能分析的关键步骤。

3. 细化问题

在此之前，我们已经对系统有了一个整体的认识，并且保留了一部分历史数据可以用来对比，然后我们从最终用户那里得知了当前系统存在的问题现象。至此，前期的准备工作已经就绪，下面就让我们来真正分析问题。

数据库问题诊断说穿了并没有什么新奇的，与日常生活的问题处理一个道理。比如，某天晚上家里的台灯不亮了，我们通常采用如下的方法检查原因。首先检查台灯电源是否插牢，如果插销没问题，去开一下别的灯是否亮(这里假设其它灯都是关的)，确保保险丝没坏。如果别的灯能亮，只有台灯不行，可能是台灯灯泡问题，把它拧到一个可以亮的电灯上试一下，如果灯泡没问题，就可能是灯本身问题或者电源插座问题。这时，可以拿另外一个电器插到插座上试一下，如果没问题，就说明台灯本身坏了。

经过以上的分析，把一个复杂的问题分解为几小步，每一步都回答一个小小的问题，通过逻辑推导，把复杂的问题细化成相对简单的问题，从而得到解答。

是不是有点福尔摩斯的感觉？

从本质上来说，数据库性能的诊断分析和台灯诊断没区别，当然步骤会复杂一点，需要考虑的变数也更多。从电灯的例子来看，只有我们知道了台灯的工作原理（需要接电才能工作，家里的电由保险丝控制，家里有超过一个台灯）和周围的环境，才能够迅速地诊断问题。因此这也是我们在前文着重讨论系统原理的目的。

当一个最终用户反馈说应用性能下降时，我们也要像考虑台灯问题一样去考虑，应用程序连接的网络有没有问题（也就是插销牢不牢靠），其他的用户是否同样感受到性能下降（其他的台灯亮不亮），数据库和操作系统有没有明显的报错（钨丝有没有断）。

这些情况都清楚了以后，如果能够排除网络因素和用户的人为因素（比如用户可能在他的工作站开了 20 个网页窗口，1 个 MP3，3 个 QQ 账号，2 个 Word 再加上后台 10 个 BT 下载进程，估计不想慢都不行），我们就可以把问题限定在服务器系统了，如图 15.8 所示。

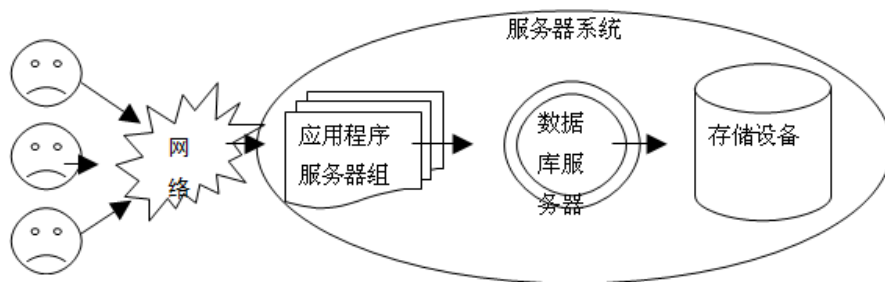


图 15.8 应用系统架构

现在很多应用采用三层架构：客户端、应用层和数据库，如图 15.8 所示。应用程序服务器处于中间层，负责从客户端接收信息，经过业务处理后将请求发送到数据库，数据库得到 SQL 请求后从存储设备读取数据，然后返还给应用程序。应用程序经过一系列数据整合后，把结果发送给最终用户。

从数据库来说，通常它不会直接和最终用户打交道。如果用户一直在抱怨自己的应用性能大幅度下降，而数据库服务器没有出现高 CPU、内存或者 I/O 的问题，同时大部分的连接都处于等待状态（UOW Waiting），这就是一个典型的数据库无高负载的情形。

不过我们说的“高”CPU，或者“大部分”UOW Waiting，怎样理解这个“高”和“大部分”呢？通过一个小时的数据收集，怎样判断当前 10% 的 CPU 使用率是高是低？怎样知道平均每秒 50 个查询是好是坏？怎样判断当前 300 个 UOW Waiting、10 个 UOW Executing、2 个 Lock Waiting 是繁忙还是空闲？

我们需要和历史正常性能数据对比。比如问题发生在周四下午三点半，该应用在每个周末都要将数据导入分析系统，并清理原系统数据。在接下来的一周内数据每天都会增加，直到周末再次清理。

很显然，对比周四与周一的数据是毫无意义的，真正有意义的是上周四下午 3 点半的性能，如果发现当时的 CPU 占用率为 60%，那么当前的问题绝对不是 CPU 瓶颈。

继续看下去的话，假设上个周四同样三点半到四点半的时间内，通过计算我们得知平均每秒 700 条查询、120 个 UOW Waiting、150 个 UOW Executing 和 30 个 Lock Waiting，并且用户不曾反映性能问题。那么对比问题发生时的数据，我们可以清晰地看到，问题发生时同时执行的连接减少，CPU 消耗率降低，查询数量降低，锁数量降低。此时直接打电话给网络管理员和应用程序服务器团队，让他们排查服务器与网络方面的问题即可。因为这个问题极有可能是应用程序服务器出于什么原因，无法将足够的工作交付给数据库系统执行。

刚才我们所做的就是将一个服务器系统性能问题细化为数据库服务器或者应用程序服务器的性能问题。如果能够在 3 个系统中（应用程序服务器、数据库服务器和存储服务器）中至少排除掉一个，都能够为我们性能分析带来很大方便。

但是有时候，很难排除以上任何一种。如果在上周四的数据中，发现上周四大概 6 个 UOW Executing，每秒 15 个查询，CPU 占用率 5%，那么我们不能够简单地说这个是应用程序服务器的问题。一般来说，正常的系统，性能下降意味着单位时间内的吞吐量减少。可是该例中，正常吞吐量为每秒 15 个查询，而性能不好的系统中为每秒 50 个查询，这一点明显有悖于正常行为。

让我们继续分析。通过对 CPU 使用率的理解，当前 CPU 使用率仅有 10%，由此可以推断 CPU 不应该成为瓶颈。那么问题可能出在哪呢？可能是应用层，比如修改了业务逻辑(需要更多的 SQL 语句)，也可能发生在数据库层，比如由于业务量的突然提高，导致更多锁等待，产生系统懒惰。

细化问题有一个前提，就是一定要小心不要走上错误的岔口。在每次回答一个小问题之前，都不要过度依赖某一类的数据。除了 CPU 使用率、并发数、吞吐量等指标，还包含更多的数据。我们一定要结合关键数据分析现象，千万不要被某几条现象所迷惑。

每一次回答小问题的时候，都要仔细思考一下，有没有其他任何数据能够证明该答案是错误的。注意，“错误”并不是笔误。每次回答问题之前，都要去假设心中的答案是一个错误的方案，然后尝试去找数据证明这一点。如果没有任何数据或者逻辑能够反证你的推论，这个推导才算是合格可用的。

在细化的过程中，有时候我们找不到一个完美的对比数据。比如有的时候当 DBA 被交予一个陌生的系统时，领导可能拍着桌子吼“这个系统跑了两年都是好好的，可是性能上周突然下降得很厉害，你给我把它搞定”。这个时候，用户可能并没有该系统的历史数据来对比，那么 KPI（Key Performance Indicator，关键性能指标）就是我们最好的朋友。

KPI 并不是凭空产生的，而是国内外无数专家在几十年的性能分析中总结出的一套相对适用于大部分系统的性能指标。

除非一个系统被设计得相当诡异，一般来说大部分系统都可以用 KPI 指标进行评估。当然，如果系统怪异到连最通常的 KPI 都不适用的情况下，那么这个系统的设计者估计也可以离职了。

最通常的 KPI 可以说是缓冲池命中率。相信大部分 DB2 DBA 都听说过，就是说有百分之多少的机会，一个逻辑读会发现数据已经存在缓冲池内部。这里我们先不给出具体数值，在下一章中，我们会列举很多常用的 KPI，以及它们的用法和意义。

数据收集频率很重要，每一次数据收集前，我们都要弄清楚“从这次的收集中我们需要研究什么方向”，以及“为什么在前一次的收集中我们没有收集到相关数据”。我们的目标是尽量减少无效的数据收集次数，但是由于性能问题的特殊性，人们不可能通过一次收集，就把所有可能发生性能问题的相关数据全部拿到。因此，如果 DBA 发现当前数据采样不能够很好地反映

问题，不要害怕进行多次数据收集。

这节我们着重强调了细化的作用，只有通过不断地细化一个复杂问题，才能将其定位到某一个或者几个特定的细节。在细化的过程中，我们需要不断地提出一个又一个假设，然后去证明或者反证这些假设。基本上我们要做的就是像侦探一样，在收集了大量的数据后，从其中找到真正能够证明我们猜测的东西。

4. 定位问题

类似细化问题，定位问题是性能分析的最后一步，也就是像柯南那样伸出一个手指头，斩钉截铁地指出，凶手就是你。

可是我们为什么要把它单拿出来，而不是同样归结于细化问题中呢？主要是两点原因：

- 很多时候，当细化到某一程度后就无法继续细化（比如说我很难定位到台灯里面哪一个电线断了）了。定位问题就是在若干个可能性中，通过各种技巧和经验将问题定位到其中一种或几种可能性，然后进行优化。
- 如果说细化问题是正向推导，那么定位问题很多时候需要反向推导，也就是说假设这个问题就是根源所在，我们能不能从原理上通过反向推导，推导出我们所看到的现象。

如果说我们所有的数据证据都指向一两种可能性，并且这一两种可能性都可以直接产生我们所看到的现象，那么我们可以将问题定位到这两个点上。

一般来说，定位最好不要超过 3 种可能性，否则在接下来的调优过程中会对执行者造成很大的困扰。

至此为止，性能分析的理论部分基本结束。不知道各位读者是否能够将“了解系统”、“明确问题”、“细化问题”与“定位问题”灵活运用于工作中。凭作者的感觉，不通过几年大量的案例练习与思考之前是很难实现的。

当然，经验是慢慢你积累的。很多人对性能监控调优是从一些最佳实践文档入手的，比如 IBM developerworks 上面的最佳实践文档就是很好的资料：

```
http://www.ibm.com/developerworks/data/bestpractices/db2luw/  
http://www.ibm.com/developerworks/data/library/techarticle/hayes/0102_hayes.html  
http://www.ibm.com/developerworks/data/library/techarticle/dm-0606schiefer/  
http://www.ibm.com/developerworks/data/bestpractices/systemperformance/
```

通过这些最佳实践文档，用户可以在对系统整体架构融会贯通之前，通过对比文档上推荐的一些配置来达到系统监控与优化的效果。

另一种手段则是根据自己或者他人的经验，在每次性能分析中按照特定的规则入手。比如依次检查 CPU、内存、实例内存、数据库内存、缓冲池命中率等，如果是一个有经验的 DBA 总结出来的步骤，可能会对很多新人有极大的帮助。这种方法不需要用户对系统与数据库原理有着深入的了解，也不需要系统的非正常行为有着狼一般的嗅觉，就可以处理一些常见的性

能问题。

不过在处理复杂问题时，这两种方法都有其局限性。如果一个性能问题是由非常罕见的原因引起的，按照最佳实践与常用规则的性能分析都很难定位到真正的问题所在。简单地说，这两种性能问题的分析方法取决于分析者的运气。

但是在使用细化问题的分析方法也有一定的问题，就是对于常见问题的分析时间与处理难度明显高于其他两个方法，同时与 DBA 的经验与能力有着直接的关系（无法遵循一个固定的分析路线，可能对很多人来说完全不符合其工作习惯）。

因此作者建议，DBA 在遵循最佳实践或规则步骤性能分析手段的同时，请尽量去尝试了解整体的系统架构，以及去尝试用简单问题的答案反推出用户所观察到的现象。当 DBA 对类似推理熟能生巧后，才能够结合两者之间的优势，摸索出最适合自己的性能分析的道路。当你拥有了自己的感悟后，也就能在这个领域称为专家了。

15.2.3 性能分析案例 ■ ■ ■

谈了那么多思路，可能很多读者并不能立刻完全理解。有了好的方法，还需要 DBA 的不断实践和积累。本章我们为每一类瓶颈问题准备了一个真实工作中的案例，希望对大家有所帮助。

1. 高 CPU 占用率

CPU 利用率居高不下是一个非常常见的问题。根据前面的介绍，相信各位心里已经有了思路。

第一步，就是判断 CPU 使用率高的进程是不是 DB2 进程。如果不是，自然可以排除 DB2 的问题。那么怎么知道哪个进程占用 CPU 呢？答案就是 ps 命令的 C 列。C 列越高，表示 CPU 利用率越高。

第二步，找到那些占用高 CPU 时间的 DB2 代理进程/线程。DB2 8/9.1 for UNIX/Linux 采用的是进程模型，ps 命令可以很轻易地输出哪些代理进程占用高 CPU。在 9.5 及以后的版本，ps 看到的就一个 db2sysc 进程，所以代理线程都封装在该进程中。可以多次执行 db2pd -edus 命令，对比 User/System CPU 列后找到占用高 CPU 的线程。

第三步，如果确实存在一个或多个代理进程/线程占用大量的 CPU，我们需要理解他们在做什么。这时，可以通过 db2pd 或者应用程序快照，找到当前这些代理正在执行的 SQL。

前面我们说过，不要仅仅通过一部分数据做出匆忙的判断，即使我们抓到了正在执行的 SQL，也不要急于下结论这几条 SQL 就是高 CPU 的罪魁祸首。

第四步，通过 SQL 快照，计算这些 SQL 语句的平均执行时间、读取的行数、CPU 的执行时间、排序行数等，然后分析 SQL 的访问计划，判断语句是否存在问题。

第五步，当所有的证据都证明该 SQL 确实占用了极高的 CPU，下一步就是对 SQL 做优化。很多情况下，DBA 可以在不修改语句本身的情况下，在数据库物理与逻辑设计上调优 SQL。

当然，也有一些情况，由于 SQL 本身的逻辑复杂，DBA 在数据库层面无法解决，这时就要联系应用开发人员，确认该语句的作用。根据我们的经验，很多开发人员在开发阶段更关注应用的功能，而非性能。而由于初期数据量较小，不会造成性能瓶颈，随着数据量的累积，性能问题才慢慢暴露出来。所以，遇到这类问题，通常的做法是协调开发人员，看是否可以调整 SQL 逻辑，或者增加一些过滤条件（比如只访问最近半年的数据）减少数据读取范围。

假设没有足够的证据证明某一个应用占据了绝对高的 CPU，则很有可能是系统的负载发生了变化。比如说，一个 2 年前设计得系统，2 CPUs / 8GB 内存，在正常业务下运行 20 个并发任务没有问题。可是上周开发主管打算在该系统中增加一个实时分析工具，要求同时运行 40 个并发查询。这种更改可能会严重地影响系统的 CPU 使用率，而作为 DBA 的一个重要职责，就在于当用户和开发者没有详细告诉你所有的情况时，需要自己通过数据作出合理的猜测和判断。

当然，也可能由于定期需要执行的系统维护脚本因故没有成功执行，导致 DB2 数据库的统计信息没来得及更新，造成了低效的访问计划，使得同样的查询需要更多的 CPU 资源。

让我们来看一个真实案例(该案例为笔者 2009 年 1 月曾经发布到国内知名论坛上的一篇文章)：

今天我们要讨论的是一个比较有趣（诡异）的性能问题，在每一步的推导之前请大家都停下来思考一下：如果自己的公司出现这个问题的话应当如何处理。

因为很多性能问题都类似于这个 case，问题的根源往往与现象看起来风马牛不相及，所以在看文章的时候千万不要只是记住了问题与结论，甚至记住每一步推论的步骤都不会为读者在工作时起到任何帮助，真正有用的内涵可以被概括成两个英文单词，在文章最后揭晓答案。

客户是一个世界上很有名的公司，他们最近新上了一套系统，名为 SUN Solaris 8 + DB2 V8 FP15，平时跑起来好好的一点问题都没有。可是，他们每周都会或多或少地遇上一两次 100%CPU 使用率，然后造成系统死机，不得不重新启动操作系统。

上面就是这次问题的现象，假设这个问题出在诸位读者的系统中，大家应当如何入手分析？为什么要那样入手分析？

首先是抓数据，没有数据的话所有的猜测也就只能称为“猜测”，而不能称为分析，在什么情况下抓什么数据，这是作为系统管理 DBA 的问题诊断的第一门课，也是必须要自学的一门课。每个人面对同样问题时需要的数据可能都是不同的，因为每个人分析问题的思路和切入点都不同。

对于这个问题，诸位都认为应该抓什么数据呢？

首先问题的现象就是操作系统 CPU 高，那么 nmon 是必需的，换句话说至少要有 vmstat, ps -elf。

另一方面是数据库，因为这个系统是完全的数据库系统，那么出问题基本可以认为是数据库问题了，从 DB2 角度来讲，我们需要检测整体情况，还要检测每个连接。所以也就是至少要有 snapshot for database/applications 与 db2pd -everything 的信息。

那么这些信息该怎么抓？什么时候抓？系统死机后肯定不能抓了，系统好好跑的时候抓了没用，只能在系统死机前一段时间的数据才有用。

而且就算只有了当时的一个 vmstat 又能看出什么？CPU 高.....高了.....高了.....然后呢？正常情况下 CPU 怎么样，CPU 增高的速率怎么样？抓信息不能只抓一个时间点，而是要从一段时间出发去理解问题。

也就是说，我们不能只抓一次这些数据，而是要不停地抓啊，那么问题就是，以什么样的速率去抓？

这个就是仁者见仁智者见智的问题了，没有标准答案，只要是适合自己的就好，根据这个问题的现象，平时一个小时抓一组数据，当 CPU 超过某一个正常标准时，应该有一个后台监测程序去一分钟抓一组数据，才能够比较准确地反映出问题。

因此我们可以写一个 `cron job` 文件平时一个小时抓一组数据，然后还要写一个后台守护进程几秒钟检测一下 CPU，如果超出了某一个使用率则调用另一个程序来一分钟抓一次数据，相信这些对诸位能够读到这里还没有睡着的读者来说都不困难。

也许你要问：那你怎么知道超过某个使用率的 CPU 一定会产生问题呢？说不定就是 `workload` 高了，过一会就好了呢？

没错，如果这样的话我们就把这些数据忽略掉不看，反正系统死机的时候我们肯定知道，而且我们肯定会在那之前就开始抓数据，所以我们肯定会抓到问题发生时的数据。

我们抓了这些信息都要看什么？很多情况下人们抓了无数的信息，然后就不知道该怎么办了。尽管大家都知道问题肯定是能够被这些信息反映出来的，但是在面对几百 MB 甚至几个 GB 的文本文件时有点狗咬刺猬无从下口的感觉。

在这个问题中，我们首先要知道，是不是 DB2 引起的问题。

当我们能够确认是 DB2 的问题，下一个要问的就是什么东西造成高 CPU，如果是 DB2 进程，那么是某一个进程吃掉了所有的 CPU，还是很多小的进程一起吃掉了所有的 CPU。

每次问的问题不要太多，让我们先来回答这两个。

从捕捉到的数据来看，问题发生的时候 CPU 的增长率非常快，基本上可以认为是几分钟之内就冲到了 100%，而不是像一些其他的 case 中可能用了 3~5 天的时间才渐渐增高到 100%。

这个现象可以说明什么？问题是短时间内产生的，而不是像内存泄露那种，需要几个小时到几天才能够有明显症状的问题。

那么下一个问题，我们有没有什么进程消耗了过高的 CPU？这个问题怎么看？很多人喜欢看 `ps aux`，实际上在对于 `warehouse` 这种系统来说，里面的那个百分比是完全没有用的。

想一想它的运算规则，是 CPU 运行总时间除以进程运行总时间.....

%CPU

(u and vflags) The percentage of time the process has used the CPU since the process started. The value is computed by dividing the time the process uses the CPU by the elapsed time

of the process. In a multi-processor environment, the value is further divided by the number of available CPUs because several threads in the same process can run on different CPUs at

the same time. (Because the time base over which this data is computed varies, the sum of all %CPU fields can exceed 100%.)

那么如果这个进程在刚开始的 10 个小时中完全不做工，最后的 1 秒钟内消耗所有的 CPU，`ps aux` 所得到的结果也是 0%。

所以 `ps` 中的 %CPU 基本没用。

那么怎么看一个进程的 CPU 使用率呢？`topas` 是一个不错的选择，但是只能在问题发生的时候看。此时就需要用 `ps -elf | sort +5 -rn | grep -i "db2" | grep -i "<instance owner>"` 了，C 列可不是玩具：

C

(-f, l, and -l flags) CPU utilization of process or thread, incremented each time the system clock ticks and the process or thread is found to be running. The value is decayed by the

scheduler by dividing it by 2 once per second. For the sched_other policy, CPU utilization is used in determining process scheduling priority. Large values indicate a CPU intensive

process and result in lower process priority whereas small values indicate an I/O intensive process and result in a more favorable priority.

按照 C 列降序排列，最高的就是 CPU 消耗最高的 DB2 进程。

在结果中，我们的确看到了几个进程有着高 C，那么下一步就是要验证这些进程到底在干什么。

怎么看？

对比 snapshot 看。既然我们知道了哪几个 db2agent 消耗了高 CPU，那么在 application snapshot 中查找对应的

PID，我们看到了大概 30 多个进程都在运行同样的一个查询语句。语句中包含 parameter marker，也就是“？”

每个语句消耗的 CPU 时间都不大相同，短的只有几秒钟，长的则有几十秒了。这点是可以理解的，因为不同的 parameter marker 所包含的参数可能不一样，那么造成消耗不同 CPU 的结果也可以理解了。

看到这里，我们可以大致认为这个查询语句是导致高 CPU 的直接原因（不一定是根本原因），那么下一个我们要问的就是，到底是先有鸡还是先有蛋？

是由于系统的某些问题导致这些查询都变慢了，然后一大堆一起执行造成系统更慢的恶性循环呢，还是说这些查询本来不应该这么多一起跑，但是现在由于什么原因它们一起执行造成了高 CPU 呢，或者说本来应该一大堆一起跑，但是每一个查询不应该消耗这么多 CPU 呢？

我们知道找到问题根源的思路肯定就在上面 3 种可能性的其中一种，那么让我们来一个个地排除和推导吧。

首先让我们来想想怎样证明或者排除第一种可能性。

从现有的数据我想不出好办法，你们呢？有什么建议？

既然想不出来就暂时不想，让我们看看第二种可能性。

第二种可能性就要看平时正常的数据库了。通过检查平时的数据库，我们发现该查询平时基本上不会同时出现 4 条以上。可是出现问题的时候却达到了 30 多条，这是为什么？

这个很有可能是切入的重点哦，因为这个间接地排除了第三种可能性，也就是平时不应该这么一大堆一起跑。

现在我们面前的是两种可能性，要不就是这些查询出于什么原因变慢了，原本单位时间内应该执行少量的查询，现在却需要执行一大堆，或者原本就不应该有那么多查询，但是某一段时间突然来了一堆。

下一步应该干什么？

不知道读者们都会怎么做，反正我是要客户就会重新抓数据，这一次加大正常时抓数据的密度，做到一分钟一次。

这次的数据发现了点有趣的东西。

在问题发生之前的几分钟内，从正常的数据库中我们发现了很多该查询出于 Lock Waiting 状态。

这个说明了什么？

估计有人看到这里就该明白了前因后果了.....

这是由于某些东西让这个查询造成了 Lock Waiting，这些查询都不能执行，然后越来越多的查询进来都被锁住。而后的某一时刻那个锁突然松了，这样所有的查询同时开始执行，造成了对系统十几倍的负担。

下一个问题就是，什么东西锁的呢？

在 application snapshot 中，我们发现所有的这些 lock 实际上都在等同样的一个 lock name，然后检查这个 lock name 所对应的应用程序，发现它在等另外的一个锁，然后再去看另外一个锁，却发现在 application snapshot 中显示这个锁的拥有者是它自己！

一个应用程序等它自己.....可能吗？

不可能.....难道是
bug？别忙着下结论.....

第三次抓数据，这次我们包含进 lock snapshot。

从抓到的 lock snapshot 中发现了什么呢？这个锁的拥有者的 agentid 实际上是 0！

怎么回事？

如果我们去 infocenter 查找 agent_id_holding_lock 就会发现：

If this element is 0 (zero) and the application is waiting for a lock, this indicates that the lock is held by an indoubt transaction. You can use either appl_id_holding_lk or the command line processor LIST INDOUBTTRANSACTIONS command (which displays the application ID of the CICS agent that was processing the transaction when it became indoubt) to determine the indoubt transaction, and then either commit it or roll it back.

明白发生了什么了吗？

问题的根源压根就不在 DB2 中，而是负责 2-phase commit 的应用程序服务器的错误（也不一定是它的错误，至少问题的根源不在 DB2 中了）。

这个问题发生的因果关系可以被表达为：

indoubt transaction 发生→锁不能被释放

→一个应用程序在等这个锁

→使用高 CPU 的查询在等待这个应用程序正在持有的锁

→应用程序不停地发同样的查询

→越来越多的查询都在锁等待

→应用程序服务器在若干长的时间之后自动解决 **indoubt transaction**（比如回滚）→锁释放

→很多查询同时得到共享锁，同时执行

→造成 CPU 使用率一下子冲高，系统性能严重下降

→新的查询不停地进入

→系统越来越慢，恶性循环

→系统死掉.....

那么怎么解决 **indoubt transaction** 呢？那就不是 DB2 的问题了，把这个问题交给应用程序小组解决即可。当然，如果业务允许，把那个查询改变成 UR 的 **isolation level** 也可以，不过很遗憾对于这个客户的业务不能够使用 UR，所以只能从应用程序层搞定了。

看了这么多的推导如果你还没有睡着，我想应该能发现点东西吧

首先，调优/调错并不是一种根据文档就可以简单完成的任务，而是需要很多的逻辑推导与证明/反证。

简单地说，就是如下的循环：

自己提出问题→想到应该如何去证明→抓数据→分析数据→提出下一步的问题.....

而这个循环也是适用于自己，也许其他人可能有不同的思路，不过所有的思路都可以被归纳为两个单词，即 **Narrow Down**

只有不停地对问题进行细分、证明或者排除其他的可能性，才能一步步到达问题的根源，而不是根据现象如同没头苍蝇一样到处乱撞，能不能解决问题全靠运气。

以上是一个典型的 CPU 高占用率的案例，分析思路也正像我们谈到的，关键在于理解是某几个任务造成高 CPU，还是整体负载造成 CPU 占用率居高不下。当这两个问题被回答以后，剩下的就是一些细节推导步骤，由于篇幅所限，就不在本书中一一列举了。

2. 高 I/O 占用

高 I/O 占用率指的是，应用程序的绝大部分时间花在了等待 I/O 上。

比较常见的表面现象就是 **vmstat** 中 **wait** 列所占比重较大，譬如超过 20%。当然，也会出现 **wait** 列低，但 I/O 仍有问题的现象。

根据我们的经验，很多情况下，I/O 问题是数据库性能最大的瓶颈。尽管 CPU 和 Memory 的速度一直在提高，但磁盘的 I/O 的读写速度一直没有太大突破。

一般来讲, I/O 问题可以从以下两个方面考虑:

- 数据 I/O。
- 日志 I/O。

这两个方面都很重要, 其中数据 I/O 主要是以读入为主。DB2 系统有自己的 `page cleaner` 用来将脏页写入磁盘, 因此这些写入以并行 I/O 为主。但是读取数据的时候, 如果内存无法将所有数据都装下, 那么进行物理读是不可避免的。而每一次的物理读都要经过我们之前所讨论的 I/O 步骤, 其开销将是很多应用程序性能的最大瓶颈。

但是对于日志来讲正好相反。除了回滚与前滚操作, 以及在 V9.7 中的 `Currently Committed` 功能外, DB2 在日常的操作中很少需要读取日志。每一次 DB2 进行数据的更新、插入和删除的时候, 都要在数据写入缓冲池之前被物理写入日志文件。因此, 日志 I/O 最常见的问题是写 I/O。

在系统发生性能下降时, 我们怎样能判断系统是否处于 I/O 瓶颈呢? 让我们用一件真实的案例来讲述怎样分析系统瓶颈在于 I/O:

这是一个使用 DB2 AS/400 系统的客户, 他们的一个新的项目准备建立在 DB2 LUW 上。新系统的设计是实时从主生产系统 (AS/400) 中提取被修改的数据, 然后将它们更新或者插入至 DB2 LUW 中以便分析。复制采用了 IBM CDC 软件。

这个系统最重要的需求是其实时性, 一定要确保 DB2 LUW 的更新速度足够快, 以便所有 AS/400 上更新的数据能够实时地更新至 LUW 系统。AS/400 生产系统更新的速度大约是一秒钟 3000 条记录以上, 也就是说, 为了能够与主系统同步, 新的系统必须能够达到至少每秒 4000 行左右的稳定更新速度。

这个新系统在交付最终用户之前的测试中, 数据更新速度能够达到 5000 行每秒, 但是当交付客户以后连入生产系统进行测试, 却发现更新速度降为不到 400 行每秒, 最好情况下也不过 1000 行每秒, 并且很难继续提高。

各位读者, 如果您遇到类似的问题, 应该从什么地方入手分析呢?

根据我们所讨论的思路, 首先是收集系统信息。该系统有 6 颗 CPU, POWER5 2102MHz, 64 位, 物理内存 20480MB。操作系统是 AIX 5.3, DB2 版本为 V9.1 Fixpack 5, 单分区, 只有一个实例, 不运行任何其它应用程序 (单纯的数据库服务器)。

对于每秒 5000 行的更新, 可以直接猜出来他们的 UPDATE 命令绝对不会复杂, 平均每条 UPDATE 最多几个毫秒就必须完成。在运行大量 UPDATE 的同时, 该系统也在处理少量的查询。从用户的问题描述来看, 这些查询并不是最主要的问题, 而且在性能最差的阶段停止运行这些查询也并不能改善性能。

通过分析数据发现, 频繁更新的表的表空间容器对应着 4 个 `hdisk`, 从 `vmstat` 和 `iostat` 看, `wait` 与 `tm_act` 并不很高。而表本身包含 15 亿行数据, 数据量大概在 100G 上下, 也就是每一行平均几十字节左右。

在问题汇报给我们之前, 用户曾经尝试着调整过 `DB2_PARALLEL_IO` 参数和 `INTRA_PARALLEL`, 但是效果并不明显。同时用户尝试重组表和进行统计数据收集, 同样没有任何显著效果。

IBM CDC 复制软件使用单一连接进行数据更新 (测试环境中就是单一连接), 当使用 4 个以上连接的时候可以达到 3000 行每秒的更新速度, 但是用户担心过多的连接数量对 DB2 AS/400 端造成性能影响。

因此, 在这里我们亟待回答的问题如下:

- (1) 为什么性能在生产环境下比在测试环境下要低。

(2) 如何提高性能。

这两个问题都不容易解决，而且我们遇到如下几点障碍：

第一，测试环境已经不复存在，而在生产环境中我们并没有一个基准性能指标代表着好的性能，所以我们没有任何可用的历史数据进行对比。

第二，根据客户所讲的，测试环境的硬件与设置与生产系统完全相同。不过，如果两者真的一模一样，能够出现如此明显的差别吗？因此，对这句话我们不予完全采信，只能够说明客户之前没有留意到两者之前有什么差别，需要我们自己找出来。

第三，我们并不能直接连接到系统，所有的分析，数据收集与建议都必须在远程操作，然后通过 E-mail 和电话让用户测试并得到反馈。

最后，该问题非常影响客户对本公司的信心，已经被客户的副总裁及本公司高层亲自过问，所有的数据收集与建议都要异常小心，并且需要尽快解决问题。

但我们也有一个优势，就是该问题可以无数次地重现。如果一个性能问题无法重现，将会对分析的困难造成几级数程度的增长。过去笔者曾经分析过的问题，是每个月末的月结程序性能时好时坏。好的时候几个小时完成，不好的时候要好几天。因为月结程序只能够在月末完成，而当把数据导出到测试环境下后完全无法重现问题，因此数据的收集只能最多一月一次，而且每次还不能保证问题的重现，为分析工作带来极大的困难。

扯远了，回到这个问题中，现在我们的优劣都很明显，那么如果读者您是处理这个问题的技术负责人，所有的眼睛都盯在您的身上，您将会怎样入手分析呢？

这个时候千万不要惊慌。慌乱并不能对解决问题带来哪怕一丁点的优势，只会持续地混乱我们的思路，造成负面影响。我们已经了解了用户的系统，并且理解了用户真正的问题，下一步就是要细化这个问题了。

首先，根据我们已经了解的问题：

(1) 同样的硬件和存储，在测试系统中可以达到至少 5 倍以上的吞吐量。

(2) 通过增加连接的数量同样在当前系统中可以增加吞吐量。

这个说明什么？说明不是硬件的资源限制造成的问题。别忙着下结论，用数据论证我们的猜测 (vmstat 结果)：

r	b	avm	fre	re	pi	po	fr	sr	cy	in	sy	cs	us	sy	i
d	wa	pc	ec	hr:mi:se											
2	1	4778836	28640	0	0	0	87	386	0	582	26388	6309	7	6	8
1	7	0.41	13.8	01:24:45											
3	1	4779889	27352	0	0	0	86	290	0	750	27283	7025	13	6	7
4	6	0.64	21.3	01:25:45											
2	1	4779798	27447	0	0	0	88	288	0	602	26666	6800	7	6	8
1	7	0.42	14.0	01:26:45											
2	1	4778357	29013	0	0	0	91	351	0	611	26074	6918	7	6	8
1	7	0.42	14.1	01:27:45											
3	0	4786967	21827	0	0	0	90	429	0	600	20939	5086	7	5	8
1	7	0.40	13.4	01:28:45											
2	1	4777482	29773	0	0	0	0	0	0	590	9594	2671	3	4	8
6	7	0.22	7.5	01:29:45											
3	1	4780683	23591	0	0	0	90	495	0	678	35375	7144	11	8	7
6	6	0.61	20.3	01:30:45											
2	1	4777576	76308	0	0	0	405	1704	0	787	46464	6451	8	7	7
9	7	0.49	16.2	01:31:45											
4	1	4777551	71157	0	0	0	0	0	0	580	26221	6210	7	6	8
1	7	0.42	13.9	01:32:45											
2	1	4777566	65947	0	0	0	0	0	0	675	25992	6873	8	6	8

```

0 6 0.46 15.3 01:33:45
2 1 4777567 60617 0 0 0 0 0 0 655 26187 6721 7 6 8
1 7 0.42 14.1 01:34:45

```

从数据看，系统的 CPU 占用率大概 15%，I/O 等待时间在 10% 之下，剩余的内存并不高，大概在几百兆左右徘徊。但是我们并没有任何 pi/po 的问题，因此很可能是文件系统缓存占用部分内存资源。

由此看来，硬件不存在瓶颈，那么现在把硬件放到一旁，我们接下来就深入 DB2 数据进行了分析。

由于是真实客户的案例，不能把大量涉及用户数据等敏感信息贴到本书中，我会尽量清晰地描述所收集到的数据。

首先我们来计算一下，串行操作每秒钟 5000 行的数据更新意味着什么。 $1/5000 \times 1000 = 0.2$ ，也就是说我们每条数据的更新不能高于 0.2 毫秒才能够达到这个吞吐量。0.2 毫秒是什么概念。平均每一个物理 I/O 大概需要 1~10 毫秒完成。如果取中间值 5 毫秒，也就是说假设内存访问不占用任何时间，需要至少确保每 25 次数据访问中最多一次进行物理 I/O。但是这个计算过于笼统，因为普通的 update 就算不使用任何物理 I/O，也没有可能一点花费都不占用。

那么我们怎样进一步估算出每一条 SQL 占用多少资源呢？

这里，我们需要看一下 SQL 快照：

```

Number of executions           = 1032098
Number of compilations         = 1
Worst preparation time (ms)    = 1865
Best preparation time (ms)     = 1865
Internal rows deleted          = 0
Internal rows inserted         = 0
Rows read                     = 1032076
Internal rows updated          = 0
Rows written                   = 1032076
Statement sorts                = 0
Statement sort overflows       = 0
Total sort time                = 0
Buffer pool data logical reads = 2066189
Buffer pool data physical reads = 423937
Buffer pool temporary data logical reads = 0
Buffer pool temporary data physical reads = 0
Buffer pool index logical reads = 10328952
Buffer pool index physical reads = 950895
Buffer pool temporary index logical reads = 0
Buffer pool temporary index physical reads = 0
Buffer pool xda logical reads  = 0
Buffer pool xda physical reads = 0
Buffer pool temporary xda logical reads = 0
Buffer pool temporary xda physical reads = 0
Total execution time (sec.ms)  = 9295.656929
Total user cpu time (sec.ms)   = 117.769741
Total system cpu time (sec.ms) = 65.283554
Statement text                  = UPDATE "USERSHEMA"."USERTABLE" SET "COLUMN1"
= ?, " COLUMN2" = ?, " COLUMN3" = ?, " COLUMN4" = ? WHERE " COLUMN5" = ? AND " COLUMN6"
= ? AND " COLUMN7" = ?

```

这里我们把表和列的名称都隐藏起来，只谈论数据本身。

我们知道，物理 I/O 的时候是不占用线程 CPU 的，因此，对于一百万的执行，通过计算用户 CPU 与系统 CPU 的总和，我们能够大约估算出每次执行使用的 CPU 时间： $(117+65)/1032098*1000=0.176$ 毫秒。也就是说，排除物理 I/O 的影响，平均每次执行大概需要 0.176 毫秒的 CPU 时间。

通过这个数据我们知道了什么？就算没有任何物理 I/O 的介入，单一连接能够达到的最快吞吐量是 $1000/0.176=5670$ 。这说明就算没有任何 I/O，单一连接在这个配置下最快达到 5670 行每秒，基本接近于我们测试时的峰值。

继续我们的计算，如果每次执行需要 0.176 毫秒 CPU 时间，为了能够确保每秒 5000 行的更新，我们的物理 I/O 几率需要降至 $1/((0.2-0.176)/5)=208$ ，即至少每 208 条 update 才能对应一个物理 I/O，也就是要保证 $100*(1-1/208)=99.5\%$ 以上的缓冲区命中率才行。那么我们当前的命中率是 $100*(1-((423937+950895)/(2066189+10328952)))=88.9\%$ ，现在我们看到缓冲区的命中率只有不到 90%，而我们的目标是提升到 99.5%。

不知道各位读者是否看明白了这段计算。也许大部分文档都在说，系统要最好保证 95% 以上的命中率才能有不错的性能，但是并没有讲这个数据是如何得到的（当然是 KPI 的指标，但是可能不能应用在所有的系统中）。但是在该问题中，我们已经量化了所有的指标，通过计算明确得知 99.5% 的命中率是保证每秒 5000 行更新的底线。因此，我们在分析问题中就会有一个非常明确的目标，并且 100% 确信，只要当我们达到了这个目标就可以完成任务（前提是假设 5 毫秒物理 I/O）。

这个就是细化问题的一个典型案例：怎样把一个复杂问题细化成一个量化后的简单问题。

提升命中率这个问题相对简单。我们来回顾一下什么是命中率。命中率就是说当我们 N 次尝试访问数据时，有多少次是能够直接在内存中找到数据的。假设其中有 M 次能够在内存中找到，这样命中率就是 $100*M/N$ 。那么我们怎样使得尽可能多的访问都直接在内存中找到呢？

对于 OLAP 来说，大部分的访问操作都需要访问大块的连续数据，这样数据库的预取功能可以把很多相关的数据一下读入内存，使得 DB2 在需要数据的时候可以经常发现它们已经被读入内存了。

不过对于 OLTP 来说就是另一个故事了。类似这个案例，OLTP 是一个典型的随机数据访问的系统。我们无法保证下一次的数据访问会在数据中的什么位置，因此预取在真正的随机数据访问中是完全没有作用的。

针对我们只有 20GB 的内存与 100GB 的数据，我们依然能够有大约 90% 的命中率，说明数据并不是真正随机的。其大部分依然是相互关联的关系，才能够让最多 20% 的数据存在内存中，使得超过 90% 的数据访问都不需要经过物理 I/O。

这样，我们也许就回答了客户的第一个问题：为什么该系统在测试中运行良好。

很显然有两种可能：

（1）测试时没有使用足够多的数据。可能使用了 30-40GB 的数据，造成绝大部分数据已经存在在内存中，从而避免了物理 I/O

（2）测试数据同样 15 亿行，但是需要 update 的数据随机性不够强，使得大部分需要 update 的数据已经存在内存中

从逻辑上推理，在给定硬件条件的情况下，上述两点是仅有的两种可能性造成测试与实际系统的差异。我们也可以看出，测试系统完全模拟真实环境是多么的重要。

接下来就是如何提升性能了。

在了解了 I/O 操作就是瓶颈的基础上（尽管 vmstat 没有显示高 wait，通过我们的分析得出的结论是在该系统中 I/O 是制约吞吐量的关键），我们能做的有以下两方面：

- （1）提升 I/O 的效率。
- （2）提升命中率。

通过刚开始的背景收集，该客户已经使用了最高端的存储之一，在提升 I/O 效率上做文章可能性不大。就算能做，可能产生的效果也不太高。想象一下，把 5 毫秒的 I/O 降低到 1 毫秒最多产生 5 倍的提升（400→2000）；可是如果把 90% 的命中率提高到 99%，那么性能的飞跃将是 10 倍（10% 的物理 I/O→1% 的物理 I/O，400→4000）。那么怎样提升命中率呢？我们知道该系统是伪随机数据访问，因此在预取上做文章也没什么可行性，也就是说数据重组肯定不能起到太大的帮助（通过客户已经做的实验证明，该推论完全正确）。那么最有可能的解决方案就是增大缓冲区，把尽可能多的数据都放入内存中。

根据我们的建议，客户将内存增至 70GB，然后发现单一的连接也可以轻松地达到 4000 行更新每秒，而最高峰值则超过 5000 行每秒。

I/O 瓶颈在日常的工作中屡屡皆是。大部分典型的 I/O 瓶颈都可以从 `vmstat` 和 `iostat` 上看出。但是即使两者没有显示高 `wait`，也不能够排除 I/O 瓶颈的可能性。

诸位读者一定要从快照的蛛丝马迹中尝试计算出单个物理 I/O 的速度，或者命中率的指标。一般来讲，当系统没有 CPU 瓶颈，没有换页，也没有锁等待的情况下，很大程度上 I/O 问题是最有可能发生的（当然不排除其他懒惰系统的情况，但是发生的比例相对较少）。

3. 内存瓶颈

内存瓶颈在某些意义上类似 I/O 瓶颈。譬如刚才的那个例子，虽然是 I/O 瓶颈导致的性能问题，但是从另一种角度也可以认为是内存的瓶颈（内存不够大，所以缓冲池不够大）。

这一部分我们要讨论两种不同意义上的内存瓶颈：

- 物理内存的限制，导致数据库内存池大小不足以负担给定的业务。
- 数据库参数的设置问题，导致使用的内存超出物理内存上限（开始换页）。

对于第一种原因，我们可以简单地用“系统容量”（`capacity`）来概括，也就是说业务量超出系统最大能够承受的能力。这种问题证明起来相当困难。一方面，如果想要正向的证明，需要进行如同上个案例那样一系列的计算，来从数字上证明当前的系统资源不足以负担给定的业务。另一方面，在大多数情况下，这种问题无法正向证明。能够说服领导添加硬件资源的，只有尝试证明无法通过其他手段进一步调优。这也是一个非常难以处理的论题。在证明中必须明确地阐述问题发生的根源，并且有能力证明没有其他方法可以有效地解决这个根源问题。

对于物理内存限制，导致内存池过小的这种问题，最明显的表象是其中一个或者多个内存池的使用率过高，甚至经常使用溢出内存（`Overflow Memory`）。

而当内存池为固定大小时，对于缓冲池人们会看到命中率低下；对于 `package cache` 则会看到大量的 `package cache` 插入和溢出；对于锁列表内存，则会在 `db2diag.log` 中看到大量的锁升级等，在快照和日志中，会有各种各样的信息从侧面告诉用户什么内存可能出现瓶颈，而 DBA 要做的就是从这些蛛丝马迹中找出真正问题的根源。

当 DBA 找到这些内存相关的现象时，就可以从这些方面入手，去理解系统中的工作负载与这些内存池的关系，以及是否有方法从其他方向减少其他内存池的使用，以便增加问题内存池的大小。

如果发现该系统的内存占用率居高不下，同时无法减少其他内存池的大小，则可以初步认为这个系统需要进一步扩容以满足业务。

另一种可能的瓶颈，就是数据库或操作系统参数的非正常设置，导致系统内存占用超出物理内存上限，最终导致换页。

从某种意义上说，这类问题是最容易解决的，但是另一方面却又是非常复杂的。因为数据库与操作系统的参数加起来成百上千，除非对整个数据库及操作系统的内核都有了一个完善理解，才能在第一时间反应过来，应该注意哪些参数。

首先，正如我们前面分析的那样，换页会对性能造成极大影响，在如今内存价格相对便宜的前提下，应该确保系统不发生换页。

然后，需要了解操作系统的计算内存和非计算内存分配原理，前面已经有详细介绍。这里以 AIX 操作系统为例，介绍 4 个虚拟内存控制参数：

- maxperm。
- maxclient。
- minperm。
- lru_file_repage。

这 4 个参数主要用于控制操作系统如何平衡计算内存与非计算内存之间的大小。

在推荐设置中，请使用下列参数设置当前系统（在 AIX5L 中，默认设置无法很好地运行 DB2，该设置已被 AIX6 采用为默认设置）：

```
maxperm%=90
maxclient%=90
minperm%=3
lru_file_repage=0
```

让我们来单独了解其中每一个参数的作用。

maxperm 的作用就是限制系统使用最大文件系统缓存。如果系统中文件系统的缓存所占用大小大于该系统物理内存大小给定的百分比，那么系统会倾向于牺牲文件系统缓存以满足计算内存。默认大小在 AIX5 中为 80%，而 AIX6 中为 90%。而 minperm 的作用正好相反，如果系统中文件系统的缓存所占的百分比低于该下限时，那么系统会倾向于牺牲计算内存以满足文件系统缓存（将会同时窃取文件系统缓存与计算内存页）。默认大小在 AIX5 中为 20%，而 AIX6 中为 3%。

maxclient 则类似 maxperm，不过其作用是特定为了 JFS2 文件系统缓存的。JFS2 文件系统为了增强其性能使用客户机文件，因此不受 maxperm 与 minperm 的影响。因此等同 maxperm，我们需要将设置调节为 90%。默认大小在 AIX5 中为 80%，而 AIX6 中为 90%。

也许有人会问，如果文件系统内存使用的大小介于 maxperm 与 minperm 之间怎么办？这个

问题将引出下一个极为重要的参数：`lru_file_repage`。对于数据库系统来说，我们已经有足够的缓冲池，通过 DB2 内部的页面控制机制，将最有可能经常使用的数据驻留内存，因此我们不需要操作系统另外为我们缓存更多的数据了。而这个参数的作用，就是如果文件系统内存介于 `maxperm` 与 `minperm` 之间，系统如何决定是从文件系统内存中偷取页面，还是从计算内存中偷取页面。在 AIX5 中，默认情况下为 1，也就是说页面替换只窃取文件页，除非文件页的重调入数量大于计算页数。但是我们不需要这个“除非”的部分，因此我们要将这个参数设置为 0，告诉 AIX 完全忽略重调入数量，只窃取文件页。在 AIX6 中该参数默认为 0。

如果用户的 DB2 运行于 AIX5 环境中，请一定确保这 4 个参数被正确设置。笔者曾经见过很多内存换页的问题，都是由于这 4 个参数的非正确设置直接导致的。

修改命令如下：

```
vmo -p -o maxperm%=90 -o maxclient%=90 -o minperm%=3 -o lru_file_repage=0
```

通过我们的分析，读者可以看到，想要进行参数调节需要对数据库及操作系统有一定程度的了解。一般来说，在完全理解数据库与操作系统每个参数的意义之前，请遵照系统的最佳实践指南进行初步调节，然后根据需要细微变动参数设置。

当内存换页发生时，最关键的部分就是分析出什么在占用着大量的内存资源。是 DB2，还是操作系统，还是什么其他进程？如果是 DB2，哪些内存池占用了大量的内存，还是私有内存占用了主要资源？

在 DB2 9.5 之后，`db2pd -dbptnmem` 可以用来监控每一个内存段的大小，实例的总内存消耗理论上不应该超过 `instance_memory` 的设置。而在 DB2 8/9.1 中，进程模型使得 DB2 无法中央集权控制总内存的消耗（主要是无法掌握每一个进程自己的私有内存空间），因此在 8/9.1 的系统中，内存监控相对比较复杂。

以下通过一个真实的案例，讨论一下如何分析系统设置导致的内存问题：

```
本案例发生于 2010 年 6 月，当用户执行脚本调整一些数据库参数设定后，系统持续宕机。
该系统为 AIX 5.3, DB2 V9.1 FP6, 64 位实例。
DB21085I Instance "db2inst1" uses "64" bits and DB2 code release "SQL09016"
with level identifier "01070107".
Informational tokens are "DB2 v9.1.0.6", "s081007", "U817474", and Fix Pack
"6".
Product is installed at "/opt/IBM/db2/V9.1".
```

```
AIX Info:
5300-08-03-0831
```

```
系统配置为 8 颗在线 CPU，物理内存为 8GB，交换分区为 6GB。
CPU: total:12 online:8 Threading degree per core:2
Physical Memory(MB): total:8192 free:9
Virtual Memory(MB): total:14336 free:4138
```

```
Swap Memory(MB): total:6144 free:4129
```

从上面抓到的数据我们已经可以看到，系统的交换分区被使用了 2GB，当前空闲内存为 9MB，说明系统内存使用过高。

系统中当前存在 5 个数据库，这里让我们用 DB1、DB2、DB3、DB4 和 DB5 代表。

用户具体的参数调整使用如下脚本，针对系统中所有的数据库运行了一遍：

```
db2 update dbm cfg using intra_parallel no health_mon off FEDERATED yes JAVA_HEAP_SZ
4096 KEEPFENCED yes DIAGPATH /${USER}/dbump_data DIAGLEVEL 3 util_impact_lim 10
db2 autoconfigure using mem_percent 70 apply DB and DBM
db2 connect to $DB
db2 select * from sysibm.sysbufferpools
db2 alter bufferpool IBMDEFAULTBP size automatic
db2 attach to ${USER}
db2 connect to $DB
db2 update db cfg for $DB using discover_db enable locktimeout 30 num_Iocleaners
AUTOMATIC num_IOservers AUTOMATIC logarchmeth1 TSM util_heap_sz 50000 logarchmeth2
DISK:/${USER}/log_dump immediate
```

在脚本的运行过程中，用户发现系统的内存使用量随着数据库的连接而显著上升，将缓冲池的大小调整为非 STMM 时系统则一切正常。

现在看起来好像一切证据都在指明 STMM 为缓冲池分配过多的内存。

以下是其中一个数据库的 DBM/DB CFG：

Current DBM CFG:

```
Size of instance shared memory (4KB) (INSTANCE_MEMORY) = AUTOMATIC
Sort heap threshold (4KB) (SHEAPTHRES) = 0
```

Current DB CFG:

```
Self tuning memory (SELF_TUNING_MEM) = ON
Size of database shared memory (4KB) (DATABASE_MEMORY) = COMPUTED
Database memory threshold (DB_MEM_THRES) = 10
Max storage for lock list (4KB) (LOCKLIST) = AUTOMATIC
Percent. of lock lists per application (MAXLOCKS) = AUTOMATIC
Package cache size (4KB) (PCKCACHESZ) = AUTOMATIC
Sort heap thres for shared sorts (4KB) (SHEAPTHRES_SHR) = AUTOMATIC
Sort list heap (4KB) (SORTHEAP) = AUTOMATIC
Number of asynchronous page cleaners (NUM_IOCLEANERS) = AUTOMATIC
Number of I/O servers (NUM_IOSERVERS) = AUTOMATIC
Default prefetch size (pages) (DFT_PREFETCH_SZ) = AUTOMATIC
Max number of active applications (MAXAPPLS) = AUTOMATIC
```

而我们当前的问题就是：为什么 alter bufferpool 命令会造成系统使用内存大小超过物理内存？

如果您作为一名 DBA，出现这种情况时将会如何入手分析问题？

首先，我们自然是要监测系统内存的使用情况：

```
Memory currently available: 717172736
Memory currently available: 7233536          // 7MB
```

```

Memory currently available: 4090081280
Memory currently available: 4217335808
Memory currently available: 11603968
Memory currently available: 4030660608
Memory currently available: 729288704
Memory currently available: 37617664
Memory currently available: 17100800
Memory currently available: 9191424
Memory currently available: 4776116224
Memory currently available: 6631424
Memory currently available: 1371914240
Memory currently available: 4210024448
Memory currently available: 4807700480
Memory currently available: 5275648
Memory currently available: 5033435136
Memory currently available: 4949757952
Memory currently available: 4950016000
Memory currently available: 5003309056
Memory currently available: 4995039232
Memory currently available: 4994469888
Memory currently available: 10358784          // 9MB

```

从这个输出中我们可以看到，系统的内存经常突然就完全被消耗掉了。

下一步，既然我们怀疑是 STMM 对缓冲池搞的鬼，那么监视 STMM 日志就变得非常重要了。

在 STMM 日志中，我们能够看到很多系统内存一下被消耗干净的信息：

```

2010-06-29-16.52.42.779932-240 A5344349A808      LEVEL: Event
PID      : 3129494          TID : 1          PROC : db2stmm (DB1) 0
INSTANCE: db2inst1         NODE : 000          DB   : DB1
APPHDL   : 0-440           APPID: *LOCAL.db2inst1.100629204342
AUTHID   : DB2INST1
FUNCTION: DB2 UDB, Self tuning memory manager, stmmGetDBMemDataAutomatic, probe:2460
Memory currently available: 4217335808          //4GB free
Set's configured size: 1766064128
...

2010-06-29-16.56.29.578532-240 A5356558A796      LEVEL: Event
PID      : 3657966          TID : 1          PROC : db2stmm (DB1) 0
INSTANCE: db2inst1         NODE : 000          DB   : DB1
APPHDL   : 0-644           APPID: *LOCAL.db2inst1.100629205329
AUTHID   : DB2INST1
FUNCTION: DB2 UDB, Self tuning memory manager, stmmGetDBMemDataAutomatic, probe:2460
Memory currently available: 11603968           //11MB free
Set's configured size: 1706229760

```

可以看到，在 4 分钟之内我们的内存从 4GB 直降为 11MB。

对比当前系统中 5 个数据库的参数：

DB1:

```

Size of database shared memory (4KB) (DATABASE_MEMORY) = AUTOMATIC(516928)
AUTOMATIC(516928)

```

```
Sort heap thres for shared sorts (4KB) (SHEAPTHRES_SHR) = AUTOMATIC(70383)
AUTOMATIC(70383)
```

```
Sort list heap (4KB) (SORTHEAP) = AUTOMATIC(3519) AUTOMATIC(3519)
```

```
2010-06-28-12.29.27.447597-240 I91499010A461 LEVEL: Info
```

```
PID : 2367616 TID : 1 PROC : db2agent (DB1) 0
```

```
INSTANCE: db2inst1 NODE : 000 DB : DB1
```

```
APPHDL : 0-996 APPID: *LOCAL.db2inst1.100628162926
```

```
AUTHID : DB2INST1
```

```
FUNCTION: DB2 UDB, buffer pool services, sqlbAlterBufferPoolAct, probe:90
```

```
MESSAGE : Altering bufferpool "IBMDEFAULTBP" From: "1000" To: "107195"
```

```
<automatic>
```

```
DB2
```

```
Size of database shared memory (4KB) (DATABASE_MEMORY) = AUTOMATIC(490400)
AUTOMATIC(490400)
```

```
Sort heap thres for shared sorts (4KB) (SHEAPTHRES_SHR) = AUTOMATIC(3208)
AUTOMATIC(3208)
```

```
Sort list heap (4KB) (SORTHEAP) = AUTOMATIC(165) AUTOMATIC(165)
```

```
2010-06-29-16.13.25.873096-240 I93362511A463 LEVEL: Info
```

```
PID : 245972 TID : 1 PROC : db2agent (DB2) 0
```

```
INSTANCE: db2inst1 NODE : 000 DB : DB2
```

```
APPHDL : 0-441 APPID: *LOCAL.db2inst1.100629201335
```

```
AUTHID : DB2INST1
```

```
FUNCTION: DB2 UDB, buffer pool services, sqlbAlterBufferPoolAct, probe:90
```

```
MESSAGE : Altering bufferpool "IBMDEFAULTBP" From: "100000" To: "974748"
```

```
<automatic>
```

```
DB3
```

```
Size of database shared memory (4KB) (DATABASE_MEMORY) = COMPUTED(532352)
COMPUTED(532352)
```

```
Sort heap thres for shared sorts (4KB) (SHEAPTHRES_SHR) = AUTOMATIC(72314)
AUTOMATIC(72314)
```

```
Sort list heap (4KB) (SORTHEAP) = AUTOMATIC(3615) AUTOMATIC(3615)
```

```
2010-06-29-16.13.37.372309-240 I93388870A462 LEVEL: Info
```

```
PID : 1761320 TID : 1 PROC : db2agent (DB3) 0
```

```
INSTANCE: db2inst1 NODE : 000 DB : DB3
```

```
APPHDL : 0-339 APPID: *LOCAL.db2inst1.100629201357
```

```
AUTHID : DB2INST1
```

```
FUNCTION: DB2 UDB, buffer pool services, sqlbAlterBufferPoolAct, probe:90
```

```
MESSAGE : Altering bufferpool "IBMDEFAULTBP" From: "1000" To: "1021990"
```

```
<automatic>
```

```
DB4
```

```
Size of database shared memory (4KB) (DATABASE_MEMORY) = COMPUTED(506576)
```

```

COMPUTED(506576)
Sort heap thres for shared sorts (4KB) (SHEAPTHRES_SHR) = AUTOMATIC(70393)
AUTOMATIC(70393)
Sort list heap (4KB) (SORTHEAP) = AUTOMATIC(3519) AUTOMATIC(3519)

2010-06-29-16.13.13.744925-240 I93339229A461 LEVEL: Info
PID : 245972 TID : 1 PROC : db2agent (DB4) 0
INSTANCE: db2inst1 NODE : 000 DB : DB4
APPHDL : 0-510 APPID: *LOCAL.db2inst1.100629201313
AUTHID : DB2INST1
FUNCTION: DB2 UDB, buffer pool services, sqlbAlterBufferPoolAct, probe:90
MESSAGE : Altering bufferpool "IBMDEFAULTBP" From: "4000" To: "662574"
<automatic>

```

```

DB5
Size of database shared memory (4KB) (DATABASE_MEMORY) = COMPUTED(562640)
COMPUTED(562640)
Sort heap thres for shared sorts (4KB) (SHEAPTHRES_SHR) = AUTOMATIC(70479)
AUTOMATIC(70479)
Sort list heap (4KB) (SORTHEAP) = AUTOMATIC(3523) AUTOMATIC(3523)

2010-06-29-16.13.32.202206-240 I93377866A461 LEVEL: Info
PID : 1089734 TID : 1 PROC : db2agent (DB5) 0
INSTANCE: db2inst1 NODE : 000 DB : DB5
APPHDL : 0-549 APPID: *LOCAL.db2inst1.100629201349
AUTHID : DB2INST1
FUNCTION: DB2 UDB, buffer pool services, sqlbAlterBufferPoolAct, probe:90
MESSAGE : Altering bufferpool "IBMDEFAULTBP" From: "4000" To: "994155"
<automatic>

```

通过上面的设置，不知道读者们看出了什么。

对于这 5 个数据库，DATABASE_MEMORY 的总和为 $516928+490400+532352+506576+562640=2608996$ (4KB)，将近 10GB。

而系统中我们刚才看到，总物理内存为 8G。

Physical Memory(MB): total:8192 free:9

这个说明了什么？应该是内存参数的设置问题，导致系统请求分配的内存高于物理内存。

再回头看一眼我们运行的脚本，把眼睛放在其中的两行上：

```

db2 autoconfigure using mem_percent 70 apply DB and DBM
db2 alter bufferpool IBMDEFAULTBP size automatic

```

这两个命令，结合 DB3、DB4 和 DB5 的 DATABASE_MEMORY，就是问题的所在：

```

Size of database shared memory (4KB) (DATABASE_MEMORY) = COMPUTED(532352)
COMPUTED(532352)
Size of database shared memory (4KB) (DATABASE_MEMORY) = COMPUTED(506576)
COMPUTED(506576)

```

```
Size of database shared memory (4KB) (DATABASE_MEMORY) = COMPUTED(562640)
COMPUTED(562640)
```

其中，`autoconfigure` 程序将每个数据库的内存使用上限设定为 70%，而对于 `DATABASE_MEMORY=COMPUTED` 来说，STMM 无法动态地调节该数据库的数据库共享内存大小。也就是说，单单这 3 个数据库共享内存就要占据超过 6GB 的内存空间，而且 STMM 无法对其进行动态调整。

因此，问题也比较清楚了。

首先，在用户过去从 V8 升级到 V9.1 的时候，没有将 `DATABASE_MEMORY` 从 `COMPUTED` 调整为 `AUTOMATIC`。其次，用户错误地在所有的数据库上使用 `autoconfigure` 程序，设置 `mem_percent` 为 70%。紧接着，用户又将缓冲池设置为 STMM 自动调节。可想而知，对于设置了巨大的数据库共享内存，STMM 自然是义不容辞地把缓冲池设得越大越好。而当 DB2 分配缓冲池中，自然要提交所使用到的内存，也就是操作系统不得不把相应大小的内存真正地分配给进程使用，这样就造成了系统内存存在所有数据库同时运行时迅速耗尽。

那么怎样修改呢？相信对各位来说已经不是个难题了。把那 3 个数据库的 `DATABASE_MEMORY` 设置为 `AUTOMATIC`，让 STMM 自动控制其大小，就可以立刻解决这个问题了。

在这个内存使用的案例中，我们可以发现，很多时候一个细微的设置不当就有可能导致严重的后果。因此作者在这里提醒大家，在做任何设置上的更改时，一定要在测试系统经过多次论证，并且还要保证测试系统完全能够在配置和业务量上模拟生产环境。

4. 系统懒惰

相比起 CPU、I/O 和内存瓶颈，系统懒惰是最不容易把握的。

在没有基准参照的情况下，很难一下子判断是否存在懒惰问题。譬如说，一台系统的吞吐量为一秒钟 500 行，从数据上看 CPU、I/O 与内存都没有大量的占用。可是你如何判断该一秒钟 500 行的性能是否还能继续提高呢？

在我们之前 I/O 部分的案例中，我们有一个很好的参照物，就是测试环境中的 4000 行每秒的吞吐量。但是如果测试环境中从来没有达到过这个数值，可能用户根本不会想到，系统的性能能够在增加内存后提高 10 倍。

于是，只有当存在一个合理的参照物，或者用户对系统的设计参数极端了解的前提下，我们才能够合理地怀疑一个系统是否懒惰，否则人们可以很轻易地将问题归结于应用程序设计。

而在分析懒惰系统的问题时，首先我们要问的一个问题就是：“你为什么认为该系统性能不佳。”

一般合理的回答包括类似“同样配置，运行类似业务的另一台系统的吞吐量比它高一倍”，或者“昨天它的性能还是好好的，可是今天在没有任何更改的情况下性能下降 2 倍”等。

而不合理的回答则包括“我猜你一定有什么魔法能让它跑得更快吧”，或者“我花了 10 万块买的硬件就应该比一万块的硬件快 10 倍”等。

当我们确定一个系统的确存在性能问题，并且从初步的分析看没有任何硬件资源瓶颈的时候，就该到懒惰系统分析的时候了。

要知道，懒惰系统的根源就是等待。我们先来定义这个“等待”。在操作系统里面，一个进程可以有多种状态，譬如说执行，或者说等待 I/O，或者等待消息队列，或者就是单纯地等待一段时间。从系统的内核来看，每次进程调度程序选择进程的时候，如果这个进程不是处于执行状态，则根本不会进入 ready queue 等待执行。因此，懒惰系统中所不执行的任务不会造成高 CPU（别忘了 CPU 高低的计算，是根据进程调度程序有多少几率会碰到请求执行的进程来计算的），因此我们把这些处于非执行状态的进程叫做等待状态。这个等待可以是等待 I/O，那么这个问题就变成了 I/O 瓶颈了，所以这一节我们不会关注这个状态。而其他所有的等待状态，比如等待一个消息队列，或者等待一个信号量(semaphore)，都可以算是通常意义上的等待状态。

譬如锁等待，实际上从线程的角度看，就是在等待一个消息队列，来通知正在等待的任务可以拿到锁继续执行。而其他依然被锁住的任务，都在等待某一个消息来通知它们。不过在本书的先前部分已经有锁等待相关的介绍了，这里我们同样对这种等待不予深究。这一节，我们所关心的懒惰系统就是非锁等待情况下，系统在不大量占用任何硬件资源的情况下，我们从什么地方入手分析。

首先，类似于我们曾经讨论过的，一定要对系统的整体有一个完整的认识，然后考虑到底什么地方有可能出现问题。其次，还要理解，这个性能问题是毫秒或秒级别的问题，还是分钟甚至小时级别的问题。譬如说，一个原本每秒 1000 条交易的系统如今每秒 500 条；与原本每秒 1000 条的交易如今每小时 1 条。在这两种情况下，所需要的分析思路天壤之别。

对于毫秒与秒级别的性能问题，我们应该着重关心的是 CPU、I/O 与执行时间之间的关系。一般在这种情况下，很难找到某一个模块大量地消耗时间（所谓大量，就是几百上千倍于其他部分）。在大多数情况下，系统依然运行良好，但是性能有些差强人意。而对于分钟甚至小时级的问题，相对起来就比较好解决了。一般来说，对于这类问题要从进程调用堆栈入手，找到当前进程正在等待什么，然后就可以直接入手解决。

本节将会讨论两个案例，一个案例是关于第一种秒级懒惰问题的，而第二个案例则是笔者曾经在论坛上发布过的案例，是关于分钟级懒惰系统的案例。

该案例发生在 2010 年 11 月，操作系统 AIX 5.3，DB2 版本 DB2 V9.5 Fixpack 4，系统为单分区。主要问题是系统中所运行的某一条插入语句有时性能极为不佳，但一般 20 分钟或半个小时以后系统就自己恢复正常了。

Number of executions	= 22985
Number of compilations	= 1
Worst preparation time (ms)	= 10
Best preparation time (ms)	= 4
Internal rows deleted	= 0
Internal rows inserted	= 0
Rows read	= 35774
Internal rows updated	= 0
Rows written	= 22350
Statement sorts	= 0
Statement sort overflows	= 0
Total sort time	= 0
Buffer pool data logical reads	= 62129

```

Buffer pool data physical reads      = 79
Buffer pool temporary data logical reads = 0
Buffer pool temporary data physical reads = 0
Buffer pool index logical reads      = 364270
Buffer pool index physical reads      = 958
Buffer pool temporary index logical reads = 0
Buffer pool temporary index physical reads = 0
Buffer pool xda logical reads         = 0
Buffer pool xda physical reads         = 0
Buffer pool temporary xda logical reads = 0
Buffer pool temporary xda physical reads = 0
Total execution time (sec.microsec)= 77412.138038
Total user cpu time (sec.microsec) = 3.695126
Total system cpu time (sec.microsec)= 0.419155
Total statistic fabrication time (milliseconds) = 0
Total synchronous runstats time (milliseconds) = 0
Statement text                        = insert into MYSCHEMA.MYTABLE (C1, C2, C3, C4,
C5) values (?, ?, ?, ?, default)

```

通过这部分的数值，我们可以计算出平均每条语句大约使用 3 秒以上的执行时间：

$$77412.138038/22985=3.37 \text{ 秒}$$

而 CPU 使用时间几乎可以忽略不计，而逻辑读数量为平均每条语句 18 个数据+索引逻辑读，物理读数量几乎忽略不计。

从哪个方面看，也没有道理使得每条插入使用 3 秒以上。难道是 CPU 使用率的问题，使得该插入不能够得到足够的 CPU 执行时间？

kthr	memory				page				faults				cpu				time				

r	b	avm	fre	re	pi	po	fr	sr	cy	in	sy	cs	us	sy	id	wa	pc	ec	hr	mi	se
221	42	1363475	12966		0	0	0	11685	63750		0	6771	155885	20355	36	11	51	2	1.91		
47.6 09:06:03																					
222	0	1415479	11871		0	0	0	19142	38818		0	3424	123710	13386	36	11	52	1	1.90		
47.6 09:06:20																					
202	2	1429497	12036		0	0	0	4305	7147		0	1938	38266	4336	18	6	74	2	0.98	24.5	
09:06:32																					
232	1	1447410	13191		0	0	0	11201	22036		0	1898	60155	5607	20	10	67	2	1.28	31.9	
09:06:41																					
115	0	1468633	13461		0	0	0	0	0	0	4306	68805	13294	33	5	54	8	1.54	38.5		
09:10:22																					
137	2	1472793	13084		0	0	0	0	0	0	3316	57720	10611	29	4	56	11	1.34	33.6		
09:13:11																					
176	0	1473322	13179		0	0	0	0	0	0	5223	82912	15431	38	6	50	6	1.77	44.3		
09:14:14																					
158	2	1476119	13375		0	0	0	0	0	0	5321	95523	18640	37	6	47	10	1.77	44.2		
09:16:07																					
198	2	1479280	13429		0	0	0	0	0	0	2578	44503	8458	26	3	65	5	1.25	31.2		
09:16:44																					
143	1	1479611	13284		0	0	0	0	0	0	1583	19488	4108	18	2	68	13	0.81	20.2		
09:17:07																					
280	1	1481335	13467		0	0	0	0	0	0	1051	20287	4812	24	1	63	11	1.06	26.4		

```

09:17:34
161 2 1484801 12554 0 0 0 2562 4432 0 5260 97553 16277 46 7 47 0 2.16 54.0
09:17:35
163 0 1487200 12428 0 0 0 347 763 0 5310 102879 16506 46 7 45 1 2.19 54.6
09:17:41

```

可以看到，**ec** 列依然不到 **100**，可是 **r** 列却出乎意料的高。

r 列就是平均等待执行的线程数量。每次进程调度程序启动的时候，系统都要统计内核中有多少进程或线程在请求执行。如果发现该列高（一般数值为 **10** 以下），则可以认为有很多的线程与进程在请求执行，但无法得到 CPU。

不过这个输出相当怪异，通过 **ec**、**us** 与 **sy** 列，我们根本看不出来系统的 CPU 占用率高。最高时不过 **70%**，但是 **ec** 列标明该 LPAR 远没有占用全部 **entitle** 的 CPU。

好像从中得不到什么真正的结论，让我们回过头来看看其他的信息。

通过 SQL 快照，我们明白了该插入语句并没有完全停顿，而是效率极为低下。同时系统中好像并没有什么明显的硬件资源瓶颈。当前令人奇怪的一点是超高的 **r** 列，不过暂时我们没有理论解释其原因。

既然如此，我们换一个角度，从应用程序快照入手：

```

Application handle           = 4921
Application status           = UOW Executing
Status change time          = 11/02/2010 09:19:45.044737
Application name             = db2jcc_application
Application ID               = 10.100.193.35.46497.10110201181

Snapshot timestamp          = 11/02/2010 09:20:04.089499

Previous UOW completion timestamp = 11/02/2010 09:19:45.044592
Elapsed time of last completed uow (sec.ms)= 0.000951
UOW start timestamp         = 11/02/2010 09:19:45.044669
UOW stop timestamp          =
UOW completion status       =

Total User CPU Time used by agent (s) = 0.042973
Total System CPU Time used by agent (s) = 0.002161

Most recent operation        = Execute
Most recent operation start timestamp = 11/02/2010 09:19:45.044738
Most recent operation stop timestamp =

Statement type               = Dynamic SQL Statement
Statement                    = Execute
Cursor name                  = SQL_CURSN300C11
Statement database partition number = 0
Statement start timestamp    = 11/02/2010 09:19:45.044738
Statement stop timestamp     =
Elapsed time of last completed stmt(sec.ms)= 0.000000
Total Statement user CPU time = 0.000060
Total Statement system CPU time = 0.000006
Dynamic SQL statement text:
insert into MYSCHEMA.MYTABLE (C1, C2, C3, C4, C5) values (?, ?, ?, ?, default)

```

上面显示出其中一条正在执行的应用程序。从快照中我们可以看到，当前系统中有非常多并发运行的同样的插入语句，而且它们的状态基本都是 **UOW Executing**。

这部分告诉我们什么？首先其状态为 **UOW Executing**，说明不是锁等待的问题。而当前快找时间为 **20:04**，对比状态改变时间 **9:19:45**，我们可以看到，该插入语句在 **19** 秒之内几乎不消耗任何 **CPU**，也没有任何明显的动作。

然后让我们看一下表的定义（这里作者把表名称和列名替换）：

```
CREATE TABLE "MYSCHEMA"."MYTABLE" (
  "C5" BIGINT NOT NULL GENERATED BY DEFAULT AS IDENTITY (
    START WITH +1
    INCREMENT BY +1
    MINVALUE +1
    MAXVALUE +9223372036854775807
    CYCLE
    NO CACHE
    ORDER),
  "C1" CHAR(11) NOT NULL WITH DEFAULT '',
  "C2" VARCHAR(11) NOT NULL WITH DEFAULT '',
  "C3" CHAR(3) NOT NULL WITH DEFAULT '',
  "C4" CHAR(3) NOT NULL WITH DEFAULT '');

ALTER TABLE "MYSCHEMA"."MYTABLE" ADD PRIMARY KEY ("C5");
ALTER TABLE "MYSCHEMA"."MYTABLE" ALTER COLUMN "C5" RESTART WITH 300214023;
```

该 **DDL** 与 **SQL** 是真实版本的无限简化版，不知道各位读者现在对问题的根源有了什么猜测没有。没有？没关系，我们继续分析。

难道和日志 **I/O** 有关？

9:00-9:20	
Log write time (sec.ns)	= 616.000000004
Number write log IOs	= 161984
9:20-9:40	
Log write time (sec.ns)	= 749.000000004
Number write log IOs	= 168059
9:40-10:00	
Log write time (sec.ns)	= 558.000000004
Number write log IOs	= 186572

从绝对数量上看，日志写的平均 **I/O** 并不慢，大概在 **3~4** 毫秒。而且根据用户，真正问题的发生只在 **9:00~9:20** 之间，**9:20** 以后系统已经恢复正常。

那么通过对比各个不同时段间 **I/O** 的速率，我们不能发现单位 **I/O** 开销的增长，也不能发现 **I/O** 数量的增长。从日志部分看，在问题发生与正常情况下，没有任何明显的区别。

到此为止，我们从快照中无法得到任何其他有用的信息来帮助我们深入理解，当前的 **UOW Executing** 到底在等什么？

通过 **CPU** 消耗率来看，我们可以 **100%** 断定，该线程并没有真正在内核中执行。但是 **UOW Executing** 却是说，该应用并没有锁等待，而是在 **DB2** 引擎中运行。

因此两者极为矛盾，明明是运行状态，却没有 CPU 或者 I/O 的占用，并且等待 20 秒。

下一步，我们就要去弄明白这个线程到底在做什么。

怎么查？这里我们就需要了解该线程的堆栈信息。我们想要看一看，能不能从函数调用堆栈中看出系统现在到底在做什么。

`db2pd -stack all` 是一个得到所有线程堆栈信息的命令，这个命令向 `db2sysc` 主进程通过消息队列发送一个消息。当 `db2sysc` 的主进程中的一个特殊的不停监听消息的线程得到该信息后，会向本进程中所有的线程发送一个信号（`signal`）。当线程收到信号后会直接跳转到 `signal handler` 模块，而在该模块中每个线程会产生一个独立的堆栈信息文件，放在 `db2dump` 目录下。

通过了解堆栈信息，我们可以知道，当线程收到信号的一刹那，该线程正在做什么。

简单起见，我们直接显示其中一个正在 `UOW Executing` 的任务所对应的堆栈：

```
<StackTrace>
-----Frame----- -----Function + Offset-----
0x090000003519EA40 sqloXlatchConflict + 0x260
0x090000003519E704 sqloXlatchConflict@glue15D + 0x74
0x090000003492D978 @101@sqldSeqGet__FP8sqeAgentiN22PP8SQLD_SEQ + 0x5C4
0x09000000348B1DD0 sqldSeqGenerate__FP8sqeAgentP8SQLD_SEQ + 0xC8
0x09000000363F026C sqlri_SeqGetNext__FP8sqlrr_cb + 0x2EC
0x09000000351DAEA0 sqlriSectInvoke__FP8sqlrr_cbP12sqlri_opparm - 0x814
0x0900000036450D00 sqlrr_execute_immediate__FP8sqlrr_cbi + 0x204
0x09000000364174BC sqlrr_execimmd__FP14db2UCinterface + 0x1D4
0x0900000036EC3124 sqljs_ddm_excsqlimm__FP14db2UCinterfaceP13sqljDDMObject +
0x1108
0x0900000034E266DC
sqljsParseRdbAccessed__FP13sqljsDrdaAsCbP13sqljDDMObjectP14db2UCinterface + 0x134
0x090000003521C700 sqljsParse__FP13sqljsDrdaAsCbP14db2UCinterface - 0x124
0x090000003521DB1C @63@sqljsSqlam__FP14db2UCinterfaceP8sqeAgentb + 0x160
0x09000000350A8754 @63@sqljsDriveRequests__FP8sqeAgentP14db2UCconHandle + 0x98
0x09000000350A89AC @63@sqljsDrdaAsInnerDriver__FP18SQLCC_INITSTRUCT_Tb + 0xDC
0x09000000350A84C0 sqljsDrdaAsDriver__FP18SQLCC_INITSTRUCT_T + 0xD8
0x090000003528CA94 RunEDU__8sqeAgentFv + 0x140
0x090000003528CF48 EDUDriver__9sqzEDUObjFv + 0x94
0x09000000352869F4 sqloEDUEntry + 0x57C
</StackTrace>
```

怎么读这个堆栈呢？从进程程序来说，堆栈的一个作用是保留函数调用信息。

譬如我们有一个程序：

```
void foo()
{
    printf("hello world\n");
}
void main()
{
    foo();
}
```

这个简单的 C 程序不过是从 `main` 入口函数调用了 `foo` 函数，在 `foo` 函数中调用了 C 库函数 `printf`。

那么在执行的时候，每当函数调用的时候，系统如何工作呢？

当用户从 Shell 中敲入 `./myprogram` 命令的时候，实际上 Shell 调用 `fork` 命令直接创建出一个 Shell 进程的镜像。在这个时候，新的进程与 Shell 完全一模一样。在新的进程创建完毕后，在新的进程中会调用 `execv` 函数，将指定的可执行程序装载入进程内存空间。在装载的时候，同时还要装载所有相关的动态链接库等。

当一个程序的所有可执行部分与动态链接库都装载完毕后，就可以真正开始执行了。

在执行的开始，内核首先寻找 `main` 函数所对应的地址。在每个进程中，所有对外函数都有一个列表，其中包括了该函数的入口地址。当内核找到了 `main` 的地址后，将指令执行寄存器跳转到 `main` 所对应的地址，也就是该进程所执行的第一个函数。

所谓“放入”堆栈，并不是真正分配一块内存然后把某一个东西放进去。当进程被 `execv` 的时候，进程的堆栈内存部分已经被分配，一般来说其大小为 `ulimit` 中 `stack` 的上限。

当堆栈被分配后，进程随时都会维护一个指向堆栈顶端的寄存器。当可执行程序在编译的时候，每次的函数调用都会将该堆栈寄存器向上移动一些。那么这个移动的部分就是该函数所使用的堆栈内存。一般来说，在堆栈中会保存函数调用的返回地址，以及一些本地内存变量（这也是早期的系统中缓冲区溢出攻击漏洞的原理，就是使用用户自己的函数返回地址替换堆栈中的地址，用来跳转到一些拥有 `root` 权限的系统调用）。

回到这个问题中，当我们执行这个用户程序的时候，首先堆栈中会被压入 `main` 函数的一些本地变量。然后当调用 `foo` 函数的时候，`main` 函数所执行的位置的下一条指令位置，也就是当 `foo` 结束时应该跳转回的地址，这个地址会被压入堆栈，然后当前执行指令寄存器会跳转到 `foo` 函数对应的地址。

同理，当 `foo` 函数调用 `printf` 的时候，`foo` 函数所对应的下一条指令也会压入堆栈，执行寄存器跳转至 `printf`。

假设用户通过 `procstack` 或者其他指令得到当前堆栈，那么系统会把堆栈中所有的函数调用过程显示出来。

譬如如果用户在执行 `printf` 的时候抓取堆栈信息，将会看到类似下面的输出：

```
printf + 0x123
```

```
foo + 0x10
```

```
main + 0x20
```

我们可以看到，堆栈中的函数调用顺序是从下往上的，也就是说，下面的函数将是上面函数的调用者。

在函数名后面的数字，代表着该函数的下一条指令的执行地址，也就是说，假如 `printf` 执行结束跳转回 `foo` 的时候，将会直接跳转到 `foo` 函数的 `0x10` 偏移继续执行。

操作系统常识普及完毕后，让我们回到 DB2。

通过下面的那个堆栈信息我们可以看到什么？

```
<StackTrace>
```

```
-----Frame----- -----Function + Offset-----
```

```
0x090000003519EA40 sqloXlatchConflict + 0x260
```

```
0x090000003519E704 sqloXlatchConflict@glue15D + 0x74
```

```
0x090000003492D978 @101@sqldSeqGet__FP8sqeAgentiN22PP8SQLD_SEQ + 0x5C4
```

```
0x09000000348B1DD0 sqldSeqGenerate__FP8sqeAgentP8SQLD_SEQ + 0xC8
```

```
0x09000000363F026C sqlri_SeqGetNext__FP8sqlrr_cb + 0x2EC
```

```
0x09000000351DAEA0 sqlriSectInvoke__FP8sqlrr_cbP12sqlri_opparm - 0x814
```

```
0x0900000036450D00 sqlrr_execute_immediate__FP8sqlrr_cbi + 0x204
```

```
0x09000000364174BC sqlrr_execimmd__FP14db2UCinterface + 0x1D4
```

```
0x0900000036EC3124 sqljs_ddm_excsqlimm__FP14db2UCinterfaceP13sqljDDMObject + 0x1108
```

```
0x0900000034E266DC
```

```
sqljsParseRdbAccessed__FP13sqljsDrdaAsCbP13sqljDDMObjectP14db2UCinterface + 0x134
```

```
0x090000003521C700 sqljsParse__FP13sqljsDrdaAsCbP14db2UCinterface - 0x124
```

```
0x090000003521DB1C @63@sqljsSqlam__FP14db2UCinterfaceP8sqeAgentb + 0x160
```

```
0x09000000350A8754 @63@sqljsDriveRequests__FP8sqeAgentP14db2UCconHandle + 0x98
```

```
0x09000000350A89AC @63@sqljsDrdaAsInnerDriver__FP18SQLCC_INITSTRUCT_Tb + 0xDC
```

```
0x09000000350A84C0 sqljsDrdaAsDriver__FP18SQLCC_INITSTRUCT_T + 0xD8
```

```
0x090000003528CA94 RunEDU__8sqeAgentFv + 0x140
0x090000003528CF48 EDUDriver__9sqzEDUObjFv + 0x94
0x09000000352869F4 sqloEDUEntry + 0x57C
</StackTrace>
```

首先，该进程的入口为 `sqloEDUEntry`，它调用了 `EDUDriver__9sqzEDUObjFv` 函数（实际上函数名为 `EDUDriver`，后面跟随的是一些编译器自己添加上去的东西）。按照同样的道理一直向上，我们可以看到函数的调用顺序：

```
sqlrr_execute_immediate → sqlriSectInvoke → sqlri_SeqGetNext → sqldSeqGenerate →
sqldSeqGet → sqloXlatchConflict
```

这是在说什么？根据函数名称我们可以猜出很多东西。

首先 `execute_immediate` 好像是说立刻执行某些东西，然后 `SectInvoke` 好像也是在说调用某些东西。

下一步是比较有意思的：`SeqGetNext`，好像在说得到一个 `sequence` 的下一个值。

继续往下看，`sqldSeqGenerate`。明显是在生成一个 `sequence` 数值。然后 `sqldSeqGet` 好像是真正去得到一个数值；最后是一个 `Latch` 冲突。

下一个常识普及讲座开始：什么是 `Latch`。

`Latch` 是一个 `DB2` 内部用来协调不同线程的东西。如果写过分布式系统程序的读者们可能会更清楚一些，分布式系统设计中复杂的部分之一就是系统间的协调。在多进程系统中，每个进程间如果想要互相协调，可以借助一些 `IPC`（`Inter Process Communication`）接口，比如说信号量（`semaphore`）、消息队列（`message queue`），或者套接字（`socket`）。这些不同 `IPC` 接口有着不同的功能和用法，不过最根本的目的还是一个，为了进程间互相了解，什么时候应该继续执行，什么时候应该等待别的事务。

而 `Latch` 也是一样，只不过 `Latch` 是 `DB2` 内部封装的一个线程协调工具。在不同的操作系统中底层所使用的系统调用都不大相同。而 `Latch` 的目的不过就是确保，有些时候线程应该等待其他任务的结束。

可以把 `Latch` 想象成 `DB2` 内部的锁，不过 `DB2` 的锁使用锁列表，但 `Latch` 是使用真正的系统调用来达到共享和互斥资源的目的。

那么 `Latch` 冲突就是说，还有其他的任务得到了这个 `Latch`，然后该线程需要等待。

这样的话，逻辑上我们就有了初步的解释：该任务在调用 `sequence` 相关的函数，用来生成下一个数值。但是由于有其他的任务锁定，该 `sqldSeqGet` 函数需要等待。

结合我们刚才所看到的现象，系统中有很多同时执行的 `insert`，也就是说，可能会有很多 `sqldSeqGet` 同时执行。继续从逻辑上推导，既然 `sequence` 是每一次生成一个数值，那么如果两个任务同时请求一个数值的时候怎么办呢？`sequence` 怎样确保分配的数值对第一个任务和第二个任务不同？因此 `Latch` 的作用在这里就明显了，就是为了确保没有两个任务能够得到同一个 `sequence` 数值。也就是说，把一组并行事务在这里转换成串行操作，每次只能对其中一个事务赋予唯一的 `sequence` 数值。

那么现在我们就明白了，问题应该出现在表的 `identity` 列。

我们回到表的 DDL：

```
"C5" BIGINT NOT NULL GENERATED BY DEFAULT AS IDENTITY (
  START WITH +1
  INCREMENT BY +1
  MINVALUE +1
  MAXVALUE +9223372036854775807
  CYCLE
  NO CACHE
  ORDER),
```

能够发现什么？“`NO CACHE`”不知道会不会一下子吸引大家的眼球。

我们刚才明白了，问题发生在该 `sequence` 不能足够快地为所有的任务分配下一个数值，也就是说，我们需要某一种机制来加快 `sequence` 的分配。

结合该 `identity` 列的定义我们发现，每次在分配数值的时候，该 `sequence` 都要去生成一个，而不是每次生

成若干个，然后从内存中直接抓取赋予那些请求的任务。

看到这里应该没有什么疑问了，凶手就是 **identity** 列。我们要做的就是不断调整 **CACHE** 的大小以满足业务需求。最后，当我们把 **CACHE** 调大为 **500** 的时候，性能问题得到全部解决。

不知道各位读者从这个案例中看到了什么？什么是 **Latch**？什么是堆栈？不错，这些信息是 **DBA** 需要了解的，但是并不是作者的重点。作者希望，读者应该能够充分理解性能诊断的其中最大的挑战性，就是没有任何指南或者步骤能够百分百确保你找到问题的根源。在大部分情况下，分析师都要通过无数的数据，在经历无数合理的猜测与推论后，一步步走向问题的答案。

作者建议，对于初次接触性能分析的用户，请参考 **IBM Developerworks** 的一些最佳实践文档。当对系统逐渐熟悉之后，可以尝试在性能分析中慢慢摆脱最佳实践文档带给我们思想上的局限性，拓展自己的思路，从不同的角度分析性能问题。

最后，作者用本人多年前在论坛上发表过的另外一篇懒惰系统的案例作为本小节的终结：

案例：db2 connect to <dbname> 无法连接数据库

描述：

db2 connect to <dbname> 无法连接数据库

背景介绍：

aix 平台。

系统原先运行良好，而后用户从 **V8 Fixpak 7b** 升级到 **Fixpak 15**。

在 **box** 中有 3 个实例，其中一个实例在 **db2iupdt** 后无法连接数据库。实例中仅有一个数据库。

问题诊断：

出现这种问题后第一个要做得是什么？

```
cd ~/sqllib/db2dump
mv db2diag.log db2diag.log.bak
<reproduce problem>
vi db2diag.log
```

得到下面的结果：

```
2007-12-12-20.42.45.029130-360 I1838C472          LEVEL: Warning
PID      : 1593388          TID : 1          PROC : db2agent (DB1) 0
INSTANCE: db2inst1        NODE : 000          DB   : DB1
APPHDL   : 0-7            APPID: *LOCAL.db2inst1.071213024245
FUNCTION: DB2 UDB, base sys utilities, sqleFirstConnect, probe:15
RETCODE  : ZRC=0x820F0004=-2112946172=SQL0_MEM_SIZE "Mem Mgt invalid size"
          DIA8563C An invalid memory size was requested.
```

```
2007-12-12-20.42.45.029814-360 I2311C755          LEVEL: Warning
PID      : 1593388          TID : 1          PROC : db2agent (DB1) 0
INSTANCE: db2inst1        NODE : 000          DB   : DB1
APPHDL   : 0-7            APPID: *LOCAL.db2inst1.071213024245
FUNCTION: DB2 UDB, base sys utilities, sqleFirstConnect, probe:16
DATA #1 : String, 297 bytes
Failed to allocate the desired database shared memory set.
```

```

Check to make sure the configured DATABASE_MEMORY + overflow
does not exceed the maximum shared memory on the system. Will
attempt to allocate the minimum possible db shared memory size.
Desired database shared memory set size is (bytes):
DATA #2 : Hexdump, 4 bytes
0x2FF13E20 : C601 4000                                ..@.

2007-12-12-20.42.45.030148-360 I3067C491             LEVEL: Severe
PID      : 1593388                TID : 1           PROC : db2agent (DB1) 0
INSTANCE: db2inst1              NODE : 000          DB  : DB1
APPHDL   : 0-7                   APPID: *LOCAL.db2inst1.071213024245
FUNCTION: DB2 UDB, base sys utilities, sqleFirstConnect, probe:17
RETCODE  : ZRC=0x850F0005=-2062614523=SQL0_NOSEG
          "No Storage Available for allocation"
          DIA8305C Memory allocation failure occurred.

2007-12-12-20.42.45.030341-360 I3559C646             LEVEL: Severe
PID      : 1593388                TID : 1           PROC : db2agent (DB1) 0
INSTANCE: db2inst1              NODE : 000          DB  : DB1
APPHDL   : 0-7                   APPID: *LOCAL.db2inst1.071213024245
FUNCTION: DB2 UDB, base sys utilities, sqleFirstConnect, probe:18
DATA #1 : String, 189 bytes
Failed to allocate the minimum possible database shared memory set.
Will now attempt to start up with a single small bufferpool.
Minimum possible database shared memory set size is (bytes):
DATA #2 : Hexdump, 4 bytes
0x2FF13E1C : ACC0 CCCC                                ....

2007-12-12-20.42.45.031566-360 I4206C472             LEVEL: Warning
PID      : 1593388                TID : 1           PROC : db2agent (DB1) 0
INSTANCE: db2inst1              NODE : 000          DB  : DB1
APPHDL   : 0-7                   APPID: *LOCAL.db2inst1.071213024245
FUNCTION: DB2 UDB, base sys utilities, sqleFirstConnect, probe:15
RETCODE  : ZRC=0x820F0004=-2112946172=SQL0_MEM_SIZE "Mem Mgt invalid size"
          DIA8563C An invalid memory size was requested.

2007-12-12-20.42.45.031762-360 I4679C755             LEVEL: Warning
PID      : 1593388                TID : 1           PROC : db2agent (DB1) 0
INSTANCE: db2inst1              NODE : 000          DB  : DB1
APPHDL   : 0-7                   APPID: *LOCAL.db2inst1.071213024245
FUNCTION: DB2 UDB, base sys utilities, sqleFirstConnect, probe:16
DATA #1 : String, 297 bytes
Failed to allocate the desired database shared memory set.
Check to make sure the configured DATABASE_MEMORY + overflow
does not exceed the maximum shared memory on the system. Will
attempt to allocate the minimum possible db shared memory size.
Desired database shared memory set size is (bytes):
DATA #2 : Hexdump, 4 bytes
0x2FF13E20 : BEBE 0000                                ....

```

```

2007-12-12-20.42.45.032023-360 I5435C491      LEVEL: Severe
PID      : 1593388          TID : 1          PROC : db2agent (DB1) 0
INSTANCE: db2inst1         NODE : 000         DB   : DB1
APPHDL   : 0-7             APPID: *LOCAL.db2inst1.071213024245
FUNCTION: DB2 UDB, base sys utilities, sqleFirstConnect, probe:17
RETCODE  : ZRC=0x850F0005=-2062614523=SQL0_NOSEG
           "No Storage Available for allocation"
           DIA8305C Memory allocation failure occurred.

```

```

2007-12-12-20.42.45.032214-360 I5927C646      LEVEL: Severe
PID      : 1593388          TID : 1          PROC : db2agent (DB1) 0
INSTANCE: db2inst1         NODE : 000         DB   : DB1
APPHDL   : 0-7             APPID: *LOCAL.db2inst1.071213024245
FUNCTION: DB2 UDB, base sys utilities, sqleFirstConnect, probe:18
DATA #1 : String, 189 bytes
Failed to allocate the minimum possible database shared memory set.
Will now attempt to start up with a single small bufferpool.
Minimum possible database shared memory set size is (bytes):
DATA #2 : Hexdump, 4 bytes
0x2FF13E1C : A665 D333                      .e.3

```

```

2007-12-12-20.42.45.032412-360 I6574C491      LEVEL: Severe
PID      : 1593388          TID : 1          PROC : db2agent (DB1) 0
INSTANCE: db2inst1         NODE : 000         DB   : DB1
APPHDL   : 0-7             APPID: *LOCAL.db2inst1.071213024245
FUNCTION: DB2 UDB, base sys utilities, sqleFirstConnect, probe:20
RETCODE  : ZRC=0x850F0005=-2062614523=SQL0_NOSEG
           "No Storage Available for allocation"
           DIA8305C Memory allocation failure occurred.

```

相信大家应该已经知道发生什么问题了吧。如果不知道？给个提示，下面的部分说明了什么？

```

2007-12-12-20.42.45.029814-360 I2311C755      LEVEL: Warning
PID      : 1593388          TID : 1          PROC : db2agent (DB1) 0
INSTANCE: db2inst1         NODE : 000         DB   : DB1
APPHDL   : 0-7             APPID: *LOCAL.db2inst1.071213024245
FUNCTION: DB2 UDB, base sys utilities, sqleFirstConnect, probe:16
DATA #1 : String, 297 bytes
Failed to allocate the desired database shared memory set.
Check to make sure the configured DATABASE_MEMORY + overflow
does not exceed the maximum shared memory on the system. Will
attempt to allocate the minimum possible db shared memory size.
Desired database shared memory set size is (bytes):
DATA #2 : Hexdump, 4 bytes
0x2FF13E20 : C601 4000                      ..@.

```

尝试分配 0xC6014000 的内存失败！为什么？看到了 SQL0_MEM_SIZE 和 SQL0_NOSEG 了，而且 size_t 是 32 位的，还用继续分析吗？

直接 db2level，一看果然是 32 位实例，让用户 db2iupdt -w 64 一遍这个问题不就搞定了。

可是敌人是顽固的，是无处不在的！用户 db2iupdt 以后还是会问：为什么我还是连不上呢？

把第一招再次使出来，抓新的 db2diag.log。

这一次错误信息果然换了一个：

```
2007-12-14-12.15.17.656977-360 I1A1093          LEVEL: Event
PID      : 1052786          TID : 1          PROC : db2
INSTANCE: iu88flaq         NODE : 000
FUNCTION: DB2 UDB, RAS/PD component, _pdlogInt, probe:120
START    : New db2diag.log file
DATA #1 : Build Level, 144 bytes
Instance "db2inst1" uses "64" bits and DB2 code release "SQL08028"
with level identifier "03090106".
Informational tokens are "DB2 v8.1.1.136", "s070720", "U810952", FixPak "15".
DATA #2 : System Info, 224 bytes
System: AIX host1 3 5 00027F15D600
CPU: total:8 online:8 Threading degree per core:2
Physical Memory(MB): total:7936 free:1527
Virtual Memory(MB): total:16128 free:7967
Swap Memory(MB): total:8192 free:6440
Kernel Params: msgMaxMessageSize:4194304 msgMaxQueueSize:4194304
                shmMax:68719476736 shmMin:1 shmIDs:131072
                shmSegments:68719476736 semIDs:131072 semNumPerID:65535
                semOps:1024 semMaxVal:32767 semAdjustOnExit:16384
Information in this record is only valid at the time when this file was
created (see this record's time stamp)
```

```
2007-12-14-12.15.17.655334-360 I1095A383          LEVEL: Error
PID      : 1052786          TID : 1          PROC : db2
INSTANCE: db2inst1         NODE : 000
FUNCTION: DB2 UDB, command line process, clp_start_bp, probe:3
MESSAGE  : CLP frontend unable to get REQUEST queue handle
DATA #1 : Hexdump, 4 bytes
0x0FFFFFFFFFFFFD6C8 : 870F 0042          ...B
```

```
2007-12-14-12.15.45.546601-360 I1479A383          LEVEL: Error
PID      : 2765134          TID : 1          PROC : db2
INSTANCE: db2inst1         NODE : 000
FUNCTION: DB2 UDB, command line process, clp_start_bp, probe:3
MESSAGE  : CLP frontend unable to get REQUEST queue handle
DATA #1 : Hexdump, 4 bytes
0x0FFFFFFFFFFFFD688 : 870F 0042          ...B
```

看看 870F0042 是什么：

Input ZRC string '0x870f0042' parsed as 0x870F0042 (-2029060030).

```
ZRC value to map: 0x870F0042 (-2029060030)
V7 Equivalent ZRC value: 0xFFFFF642 (-2494)
```

ZRC class :

```

Global Processing Error (Class Index: 7)
Component:
    SQL0 ; oper system services (Component Index: 15)
Reason Code:
    66 (0x0042)

Identifer:
    SQL0_QUE_NOT_EXIST
Identifer (without component):
    SQLZ_RC_QNOEXS

Description:
    Queue does not exist

Associated information:
    Sqlcode -902
SQL0902C A system error (reason code = "" occurred. Subsequent SQL
statements cannot be processed.

    Number of sqlca tokens : 1
    Diaglog message number: 8558

```

告诉我们 queue doesn't exist! 怎么办?

首先我们要理解 queue doesn't exist 是什么。

当我们用 `db2 xxxx` 命令做事情的时候, 这个 DB2 是一个可执行程序, 这个可执行程序被称为 **frontend process**。当 DB2 进程执行的时候, 会隐式地在后台生成一个 **db2bp** 的进程, 这个进程叫做 **db2 backend process**。所以当我们使用 `db2 connect to <db>` 命令连接数据库的时候, 真正与数据库相连接的是其后台的 **db2bp** 进程。而当连接成功以后, 后面所有的操作 (如 `select`), 都是通过 **queue** 这种 **IPC** 手段与 **db2bp** 进程进行通信的。

这里我们得到的错误信息就是说, 当 **db2 frontend process** 想找 **backend process** 的时候, 发现用来通信的 **queue** 不存在!

我们知道既然是 **IPC** 通信, 那么 **frontend process** 就不可能只是得到一个 **queue** 不存在就立刻报错。所以问一下用户: 您得到错误的时候是在 **issue** 了 **connect** 命令后立刻得到的还是等了一阵。用户回答: 当然是等了一阵咩。

现在该做什么了? 当然是要看看这个后台进程正在做什么, 竟然让前台进程等了半天也拿不到 **queue**, 人家不得不退出说 **queue** 不存在了……

怎么看? 还用问? **db2trc for the win!**

因此使用第二招法宝:

```

db2trc on -t -f db2trc.dmp
<reproduce problem>
db2trc off
db2trc flw -t db2trc.dmp db2trc.flw
db2trc fmt db2trc.dmp db2trc.fmt

```

出来的 FLW 文件不大，也就 40MB 左右，大概有 51 万行（比动辄上 G 的其他复杂问题小多了）。

别被这 51 万吓倒，让我全看？没那闲情逸趣。所以说看 **trace** 最关键的就是要知道，你现在正在干什么，想要看什么东西。

我们现在正在调查 backend process 为什么不创建 queue，首先让我们从得到的 db2diag.log 入手。

先到 **FMT** 文件中找关键字"**2007-**"（不包括引号）（问我为什么找这个关键字？那您觉得用什么关键字才能够最快地找到 **dump** 到 **db2diag.log** 文件中的内容呢？）。

只得到了一个 entry:

```
2170      data DB2 UDB command line process clp_start_bp fnc (3.3.41.7.0.3)
      pid 2765134 tid 1 cpid -1 node -1 sec 64 nsec 869465587 probe 3
      bytes 392
```

Data1 (PD_TYPE_DIAG_LOG_REC,384) Diagnostic log record:

```

2007-12-14-12.15.45.546601-360 I1479A383          LEVEL: Error
PID      : 2765134          TID   : 1          PROC  : db2
INSTANCE: db2inst1         NODE   : 000
FUNCTION: DB2 UDB, command line process, clp_start_bp, probe:3
MESSAGE  : CLP frontend unable to get REQUEST queue handle
DATA #1 : Hexdump, 4 bytes
0x0FFFFFFFFFFFFD688 : 870F 0042          ...B

```

可以看到 2170 对应 FLW 文件中的 2170，然后在 entry 中我们看到 clp_start_bp，顾名思义，就是 CLP 用来开启 backend process 的 function call，也就是说，这个 2170 是在 frontend process 中的。然后回到 FLW 文件，找 2170：

2166	64:869290682								sqlochgfileptr exit
2167	64:869405455								_pdlogInt data [probe 90]
2168	64:869450954								sqlowrite entry
2169	64:869464745								sqlowrite exit
2170	64:869465587								clp_start_bp data [probe 3]
2171	64:869466392								sqloclose entry
2172	64:869472755								sqloclose exit
2173	64:869479918								_pdlogInt exit
2174	64:869480276								pdLog exit

64 秒才跑到 2170，往上看，看看它一直在干什么，我们看到了好多 OpenMLNQue，而且是基本一秒钟一个：

```

1437      22:867813790 | | | | | sqloOpenMLNQue data [probe 1]
1438      22:867817987 | | | | | sqlogkey entry
1439      22:867818164 | | | | | sqlogkey data [probe 1]
1440      22:867819491 | | | | | sqlogkey exit [rc = 0x6F02C86F = 1862453359]
1441      22:867823317 | | | | | sqloOpenMLNQue exit [rc = 0x870F0042 =
-2029060030 = SQLO_QUE_NOT_EXIST]
1442      22:867823675 | | | | | sqlorest entry
1443      22:867823869 | | | | | sqlorest data [probe 10]
1444      23:867842081 | | | | | sqlorest exit
1445      23:867845317 | | | | | sqloOpenMLNQue entry
1446      23:867847142 | | | | | sqloOpenMLNQue data [probe 1]

```

```

1447      23:867850175 | | | | | sqllogkey entry
1448      23:867850525 | | | | | sqllogkey data [probe 1]
1449      23:867852012 | | | | | sqllogkey exit [rc = 0x6F02C86F = 1862453359]
1450      23:867855105 | | | | | sqloOpenMLNQue exit [rc = 0x870F0042 = -2029060030
= SQLO_QUE_NOT_EXIST]
1451      23:867855472 | | | | | sqlorest entry
1452      23:867855657 | | | | | sqlorest data [probe 10]
1453      24:867872812 | | | | | sqlorest exit
1454      24:867876195 | | | | | sqloOpenMLNQue entry
1455      24:867877582 | | | | | sqloOpenMLNQue data [probe 1]
1456      24:867880910 | | | | | sqllogkey entry
1457      24:867881256 | | | | | sqllogkey data [probe 1]
1458      24:867882545 | | | | | sqllogkey exit [rc = 0x6F02C86F = 1862453359]
1459      24:867885558 | | | | | sqloOpenMLNQue exit [rc = 0x870F0042 = -2029060030
= SQLO_QUE_NOT_EXIST]

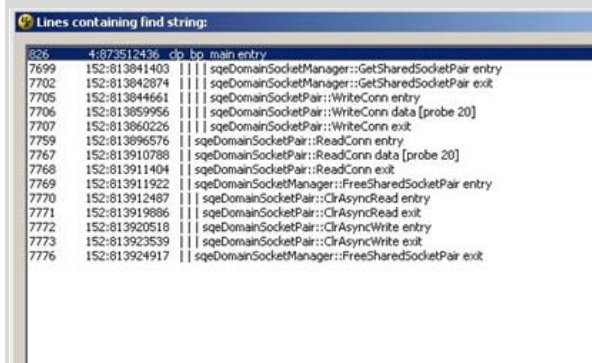
```

这说明了什么？frontend process 每秒钟检测一次 backend process queue 是不是 available，然后等了一分钟左右发现一直找不到，然后就 timeout 退出了。

不过现在还是没有说 backend process 在这段时间干了些什么呀，怎么办？

既然我们知道了 process 是 frontend process，那么 backend process 肯定在另外的地方咯。而且想要知道 backend process 在干什么，就要找这 64 秒之前的东西，像那些 100 多秒以后的事情就可以忽略不计了。

但是这里有 51 万行，怎么找？我们知道一个程序的入口都是 main，所以我们来搜索一下关键字 main，得到下图所示的内容。



除了第一个 clp_bp_main，其他的都是 100 秒之后的事情，不用管，去看第一个。

从名字上听起来，clp_bp_main 即 CommandLineProcess_BackendProcess_main，就是后台进程的入口。

```
pid = 3002686 tid = 1 node = 0
```

```

804      4:867219779 sqloDirectStandardFileDescriptorsToDevNull entry
810      4:867250656 | sqloAddOneReservedHandle entry
811      4:867265542 | sqloAddOneReservedHandle data [probe 10]
812      4:867266195 | sqloAddOneReservedHandle exit
813      4:867266457 | sqloAddOneReservedHandle entry
814      4:867266882 | sqloAddOneReservedHandle data [probe 10]
815      4:867267114 | sqloAddOneReservedHandle exit
817      4:867267312 | sqloAddOneReservedHandle entry
819      4:867267556 | sqloAddOneReservedHandle data [probe 10]

```

```

820      4:867268020 | sqloAddOneReservedHandle exit
821      4:867268268 sqloDirectStandardFileDescriptorsToDevNull exit
826      4:873512436 clp_bp_main entry
827      4:873535442 | sqloInitApplicationEnvironment entry
828      4:873548871 | | sqloGetEnvInternal entry
829      4:873557214 | | | EnvRegGetProfile entry
830      4:873558276 | | | EnvRegGetProfile exit
831      4:873558815 | | | EnvPrfGetValueByEnumIndex entry
832      4:873559169 | | | EnvPrfGetValueByEnumIndex exit [rc = 0x870F0104 =
-2029059836 = RC_ENV_NOT_FOUND]
833      4:873559409 | | | EnvRegGetProfile entry
834      4:873559809 | | | EnvRegGetProfile exit
835      4:873560012 | | | EnvPrfGetValueByEnumIndex entry
836      4:873560306 | | | EnvPrfGetValueByEnumIndex exit [rc = 0x870F0104 =
-2029059836 = RC_ENV_NOT_FOUND]
837      4:873560884 | | sqloGetEnvInternal exit [rc = 0x870F0104 = -2029059836
= RC_ENV_NOT_FOUND]
838      4:873569808 | | sqloxltc_app entry
839      4:873571249 | | sqloxltc_app exit
840      4:873571666 | | sqloBeginTypedCtxInternal entry

```

而那个对应的进程 id 3002686 就应该是其 backend process 的 PID 了。知道了这此以后，现在我们能看出，用户发出 db2 connect to <dbname>命令是在 trace 开启后 4.8 秒的时候，然后继续往下看，看看 bp 都在干什么。

往下找了大概不到 1000 行，我们突然发现了几个时间戳的 big jump:

```

1235      4:893112670 | | | | sqloPdbTcpipGetFullHostName entry
1236      4:893114440 | | | | | sqloPdbTcpIpResolveHostName entry
1237      4:893114912 | | | | | sqloPdbGetHostByName entry
1238      4:893115316 | | | | | sqloPdbGetHostByName data [probe 7]
1462      25:10592207 | | | | | sqloPdbGetHostByName data [probe 25]
1463      25:10594709 | | | | | sqloPdbGetHostByName exit
1464      25:10595152 | | | | | sqloPdbTcpIpResolveHostName exit
1465      25:10596028 | | | | | sqloPdbTcpipGetFullHostName exit
1466      25:10599298 | | | | | sqlogmbblkEx entry
1467      25:10628001 | | | | | sqlogmbblkEx mbt [Marker:PD_OSS_ALLOCATED_MEMORY ]
1468      25:10629198 | | | | | sqlogmbblkEx exit
1469      25:10630862 | | | | | sqloGetNextNodeList entry
1470      25:10631216 | | | | | sqloStreamFileGetString entry
1471      25:10631679 | | | | | sqlovsfh entry
1472      25:10631991 | | | | | sqlovsfh exit
1473      25:10634789 | | | | | sqloread entry
1474      25:10649979 | | | | | sqloread exit [rc = 0x870F0009 = -2029060087
= SQLO_EOF]
1475      25:10652511 | | | | | sqloStreamFileGetString exit
1476      25:10655747 | | | | | sqloGetNextNodeList data [probe 5]
1477      25:10698043 | | | | | sqloGetNextNodeList exit
1478      25:10699779 | | | | | sqloPdbTcpIpResolveHostName entry
1479      25:10700201 | | | | | sqloPdbGetHostByName entry
1480      25:10700605 | | | | | sqloPdbGetHostByName data [probe 7]

```

```

1810      45:128224793 | | | | | sqloPdbGetHostByName data [probe 25]
1811      45:128226786 | | | | | sqloPdbGetHostByName exit
1812      45:128227114 | | | | | sqloPdbTcpIpResolveHostName exit
1813      45:128228218 | | | | | sqloPdbTcpIpResolveHostName entry
1814      45:128229057 | | | | | sqloPdbGetHostByName entry
1815      45:128230102 | | | | | sqloPdbGetHostByName data [probe 7]
2259      65:245641277 | | | | | sqloPdbGetHostByName data [probe 25]
2260      65:245643055 | | | | | sqloPdbGetHostByName exit
2261      65:245643514 | | | | | sqloPdbTcpIpResolveHostName exit
2262      65:245663318 | | | | | sqloGetNextNodeList entry
2263      65:245663841 | | | | | sqloStreamFileGetString entry
2264      65:245664308 | | | | | sqlovsfh entry
2265      65:245664700 | | | | | sqlovsfh exit
2266      65:245666685 | | | | | sqloread entry
2267      65:245671716 | | | | | sqloread exit [rc = 0x870F0009 = -2029060087
= SQLQ_EOF]
2268      65:245672445 | | | | | sqloStreamFileGetString error [probe 15]
2269      65:245672908 | | | | | sqloStreamFileGetString exit [rc = 0x870F0009
= -2029060087 = SQLQ_EOF]
2270      65:245673296 | | | | | sqloGetNextNodeList exit [rc = 0x870F0009 =
-2029060087 = SQLQ_EOF]
2271      65:245674354 | | | | | sqlopartfmbblk entry
2272      65:245676469 | | | | | sqlopartfmbblk exit

```

从 4 秒跳到 25 秒，然后跳到 45 秒，最后跳到 65 秒。那 65 秒之后呢？由于在 64 秒 front end process 就出错退出了，所以不用管 65 秒之后的事情。

看看这几个大的跳跃：

```

1238      4:893115316 | | | | | sqloPdbGetHostByName data [probe 7]
1462      25:10592207 | | | | | sqloPdbGetHostByName data [probe 25]

1480      25:10700605 | | | | | sqloPdbGetHostByName data [probe 7]
1810      45:128224793 | | | | | sqloPdbGetHostByName data [probe 25]

1815      45:128230102 | | | | | sqloPdbGetHostByName data [probe 7]
2259      65:245641277 | | | | | sqloPdbGetHostByName data [probe 25]

```

都是同样的 function，同样的位置。这个 function 是干什么的？GetHostByName 应该不陌生，就是 DNS 地址解析，下面要做什么不用多说了，直接找到 db2nodes.cfg，看看 hostname，然后 ping 一下看看？用了 30 秒才返回第一个 package！

这里我们不研究 DNS 出了什么问题，反正与 DB2 无关。停止实例，把 db2nodes.cfg 中的 hostname 改成 ipaddress，然后 connect 一次就搞定了！

不同于上一个秒级的性能问题，该 connect 挂起实际上算是一种特殊的分钟级性能问题，直接导致的后果就是 connect 在规定的时间内无法连接数据库而报错。

这类问题可以通过 trace 入手。当然如果想要分析堆栈也不是不可以，只是没有 trace 这样直观地告诉用户数据库内部一步一步的动作和每个函数调用所消耗的时间。

15.2.4 小结 ■ ■ ■

作为数据收集与分析部分的小结，实际上我们并没有太多可以回顾的。

性能问题是一类不同于错误信息问题的特殊现象，而在分析时我们需要结合历史数据及对系统整体架构的认识，通过不同的手段一步步走向问题的根源。

性能问题的分析实际上与侦探电影相似（这也是为什么性能问题是作者最钟爱的问题类型之一）。就好像没有一个教程能够告诉人们，到底怎样成为一名杰出的侦探。同理，想要通过文字将所有类型的性能问题分析步骤一一写出是不大可能的。只有通过我们自己的努力，通过不断积累知识和经验，才能够从各种不同的角度去观察和分析一个现象，做到从不为人知的角落将隐藏的问题找出来。

15.3 判断题



（1）进行性能分析时，如果有正常情况下的性能数据作为对比，会起到事倍功半的效果。

T: 正确

F: 错误

（2）所有的性能问题都可以被归结于 4 个类型：CPU 瓶颈，I/O 瓶颈，内存瓶颈与系统懒惰。

T: 正确

F: 错误

（3）随机尝试修改不同的数据库参数，并不是进行性能调优最有效的方法。

T: 正确

F: 错误

（4）增加物理内存是解决内存瓶颈的唯一方法。

T: 正确

F: 错误

（5）高 I/O 开销一定是源于 I/O 子系统的配置问题。

T: 正确

F: 错误



优化器与性能调优

在上一章中，我们详细地介绍了性能数据收集与分析的思路，并且给出了 5 个不同的案例，来讲述如何在不同的情况下分析性能问题。

不过上一章我们主要强调的是历史性能数据收集的重要性，但是在用户的日常工作中，很有可能一个系统自从建立以来，其性能一直不尽如人意，那么我们如何帮助这种系统运行得更快呢？这就引出了我们这一章的论题：性能调优。在介绍性能调优之前，我们先要看一下 DB2 优化器的架构，理解 DB2 如何利用优化器来优化一个用户查询，使它能够以最高的效率执行。

本章内容安排如下：

- 优化器简介。
- 性能调优的思路和方法。
- 关键性能指标介绍（KPI）。

16.1 优化器简介

在介绍系统调优之前，我们先简要介绍一下 DB2 优化器的架构，只有这样，才能够更好的理解 DB2 内部执行过程，以及 DB2 如何优化一个用户查询，并使它能够以最高的效率执行。由于篇幅限制，我们尽量用简短的语言阐述优化器与系统调优细节原理，更深入和高级的内部优化器原理，将在《DB2 数据库高级管理》一书中详细讨论。

优化器是关系数据库的核心，相当于汽车的发动机，DB2 在业界最值得称道的功能之一就是其强大的优化器。在大多数情况下，用户不必手工干预优化器的执行，而且 DB2 优化器总是足够聪明地选择最佳访问计划，当然前提是数据库的统计信息被及时收集。

我们俯瞰一下优化器的构成。

首先，当一条查询被 DB2 接收以后，该查询会被查询重写器重新翻译，重写成一种更高效的能够被优化器直接理解的格式。一般来说，被重写的查询也是基于 SQL 的，但是重写器会利用一些固定的规则将某些指令翻译成不同的表达方式，或者重新安排子查询之间的顺序，然后将 DB2 的视图、触发器与约束等数据库对象带入，以便优化器能够更有效地理解一段复杂的查询。

紧接着，被重写后的查询会被送入 DB2 优化器模块。在优化器中，DB2 会以动态规划算法（如果在优化级别 `optlevel ≤ 2` 时，将会是用贪心算法），根据每个数据库对象所对应的统计数据，估算出按照不同访问数据库对象的顺序所得到的开销。

譬如说，我们有 3 个表 A、B、C。表 A 中有 10000 行相同的数据，而 B 与 C 中只有 1 行同样相同的数据（比如都是 0）。当需要 3 个表进行关联的时候，我们可以针对表 A 中的所有行与 B 关联，然后结果集与 C 关联。这样的关联需要扫描 A 中的所有行，然后对于每一行与 B 关联后，结果集中依然还有 10000 行（因为数据都相同），然后对这 10000 行再与 C 管理，还是得到 10000 行。

这样的计算也许效率不高，我们有没有更有效的访问数据的手段呢？

我们可以先用 B 与 C 关联，这样我们只需要扫描 1 行，然后将这 1 行与 10000 行的 A 进行关联。这种访问途径可以减少对中间结果集扫描 10000 次的开销。

从上面的例子我们可以看出，以不同的方式访问相同的数据，其开销可能截然不同。因此，DB2 优化器的作用就是根据 DB2 对每一个数据对象的了解，制定出一个最佳的访问途径。

当优化器计算出若干种访问数据的途径后，会选择预期最有效的一个途径，然后将这个计算出的规则发送给代码生成模块（CodeGen）生成可以运行的路径，然后该路径被发送至运行模块（Runtime）真正执行。

这是一个从一光年之外看到的优化器模块，具体的细节我们会留到下一本数据仓库的书中介绍。不过，对于数据运维的 DBA 来说，了解数据库对象的统计信息是至关重要的。

我们刚才谈到了，为了能够让 DB2 更好地生成一个最佳的访问计划，准确地理解每一个数据对象是非常重要的。对于刚才的例子，如果数据库不知道表 A 有 10000 行，优化器就不能有效地选择第二种访问方式的。

这些代表每个数据库对象状态的信息叫做统计信息，其相关信息可以使用 SYSSTAT 模式的视图访问：

```
/home/db2inst1 $ db2 "list tables for all" | grep -i "SYSSTAT"
```

COLDIST	SYSSTAT	V	2011-01-12-20.56.05.285121
COLGROUPDIST	SYSSTAT	V	2011-01-12-20.56.05.291256
COLGROUPDISTCOUNTS	SYSSTAT	V	2011-01-12-20.56.05.297720
COLGROUPS	SYSSTAT	V	2011-01-12-20.56.05.301516
COLUMNS	SYSSTAT	V	2011-01-12-20.56.05.304493
FUNCTIONS	SYSSTAT	V	2011-01-12-20.56.05.311247
INDEXES	SYSSTAT	V	2011-01-12-20.56.05.318353
ROUTINES	SYSSTAT	V	2011-01-12-20.56.05.328908
TABLES	SYSSTAT	V	2011-01-12-20.56.05.333256

从名字可以看出, 这些系统编目表有些是针对表对象的, 而有些则是针对索引或者列对象的。不同数据对象类型的统计信息使用不同的视图访问, 相关的列信息请参见 IBM DB2 信息中心。在第 10 章我们已经介绍了统计信息收集的命令: **Runstats**, 并介绍了 **Runstats** 执行的时机和最佳实践。

接下来我们介绍如何查看执行计划。可能很多读者已经了解了 **Visual Explain** 工具, 但是作者强烈推荐使用 **db2exfmt** 工具, 生成文本访问计划。

相比起图形化 **Visual Explain** 生成的访问计划, **db2exfmt** 工具的输出要完善许多, 并且包含很多重要的信息, 帮助用户理解“为什么 DB2 生成如此的访问计划”。

首先, 让我们来看一看如何收集 **db2exfmt** 信息:

```
/home/db2inst1 $ db2sampl

Creating database "SAMPLE"...
Connecting to database "SAMPLE"...
Creating tables and data in schema "DB2INST1"...
Creating tables with XML columns and XML data in schema "DB2INST1"...

'db2sampl' processing complete.

/home/db2inst1 $ db2 connect to sample

Database Connection Information

Database server          = DB2/AIX64 9.7.3
SQL authorization ID    = DB2INST1
Local database alias    = SAMPLE

(db2inst1@db2host1) /home/db2inst1 $ db2 -tvf ~/sqllib/misc/EXPLAIN.DDL > /dev/null
/home/db2inst1 $ db2 "select empno, firstnme, lastname from employee, department where
employee.workdept = department.deptno and mgrno='000060'"

EMPNO  FIRSTNME      LASTNAME
-----
000060  IRVING         STERN
000150  BRUCE          ADAMSON
000160  ELIZABETH     PIANKA
```

```

000170 MASATOSHI      YOSHIMURA
000180 MARILYN            SCOUTTEN
000190 JAMES              WALKER
000200 DAVID              BROWN
000210 WILLIAM            JONES
000220 JENNIFER           LUTZ
200170 KIYOSHI          YAMAMOTO
200220 REBA              JOHN

  11 record(s) selected.
/home/db2inst1 $ db2 set current explain mode explain
DB20000I The SQL command completed successfully.
/home/db2inst1 $ db2 "select empno, firstnme, lastname from employee, department where
employee.workdept = department.deptno and mgrno='000060'"
SQL0217W The statement was not executed as only Explain information requests
are being processed.  SQLSTATE=01604
/home/db2inst1 $ db2 set current explain mode no
DB20000I The SQL command completed successfully.
/home/db2inst1 $ db2exfmt -d SAMPLE -g TIC -w -1 -n % -s % -# 0 -o prod_sample_exfmt.txt
DB2 Universal Database Version 9.7, 5622-044 (c) Copyright IBM Corp. 1991, 2009
Licensed Material - Program Property of IBM
IBM DATABASE 2 Explain Table Format Tool

Connecting to the Database.
Connect to Database Successful.
Binding package - Bind was Successful
Output is in prod_sample_exfmt.txt.
Executing Connect Reset -- Connect Reset was Successful.

```

在创建了数据库之后，我们首先运行了“db2 -tvf ~/sqllib/misc/EXPLAIN.DDL”来创建访问计划表。EXPLAIN.DDL 文件存在于所有的实例目录 sqllib/misc 下，包含多个表的定义用来临时存储访问计划信息。

当运行了 DDL 后，用户可以使用命令“db2 set current explain mode explain”打开访问计划选项，按照普通方式执行 SQL，然后使用“db2 set current explain mode no”关闭访问计划选项。

当访问计划表被创建，并且访问计划选项被打开后，执行 SQL 的语句并不真正执行该 SQL，而是生成一个针对该 SQL 的访问计划。访问计划的信息存储于使用 EXPLAIN.DDL 生成的表中，使用 db2exfmt 工具将这些信息汇总到一个文件中。

db2exfmt 工具包含很多参数，这里我们不一一列举。一般来说，使用下列参数就可以很好地从一个数据库中得到最新收集的访问计划：

```
db2exfmt -d <数据库名称> -g TIC -w -1 -n % -s % -# 0 -o <输出文件>
```

让我们看一看 db2exfmt 生成的文件中都包含了什么信息。

(1) 系统信息:

```

DB2_VERSION:      09.07.3
SOURCE_NAME:      SQLC2H21
SOURCE_SCHEMA:    NULLID
SOURCE_VERSION:
EXPLAIN_TIME:     2011-01-18-14.02.17.151293
EXPLAIN_REQUESTER: DB2INST1

```

用户在分析的过程中首先要判断，一个文件是不是我们真正感兴趣的。假设如果问题发生在 3:00，但是文件的收集时间在 1:00，很有可能这个文件并不包含真正的问题语句。

(2) 系统设置:

```

Parallelism:      None
CPU Speed:        4.000000e-05
Comm Speed:       0
Buffer Pool size: 1000
Sort Heap size:   256
Database Heap size: 1200
Lock List size:   4096
Maximum Lock List: 10
Average Applications: 1
Locks Available:  13107

```

在数据库的几十个参数中，真正对优化器起作用的也只有这么几个（当然还包括一些 db2set 和多分区信息）。其中 Parallelism 可以是 None，代表单分区，或者 Interparallel 代表多分区，或者 Intraparallel 代表分区内并行，或者多分区与分区内并行的结合。

然后 CPU 速度则是当前 CPU 速度的一个估算值，注意该值并不代表任何有意义的单位，而是 DB2 在优化其中使用 CPU 速度作为一个参量。该数值在实例启动时自动计算。

Comm Speed 是通信速度，在多分区系统中默认为 100，可以根据需求调节。

Buffer Pool Size 是系统中缓冲池总页数的和。注意 DB2 优化器在计算访问计划时，并不知道都有哪些缓冲池将被使用（譬如说某些计划可能需要一些临时表，而临时表的缓冲池也许不同于数据表），因此缓冲池大小只是简单地将数据库中所有缓冲池的页面数量相加。

Sort Heap Size 是排序堆的大小，用来估算排序或者哈希关联是否会溢出。

Database Heap size 是数据库堆栈大小，在通常情况下对访问计划不会产生决定性的影响。

Lock List Size、Maximum Lock List、Average Applications 与 Locks Available 是锁列表信息，用来估算某些类型的数据访问是否有足够锁列表容纳相关数据。

(3) 用户查询:

```
Original Statement:
```

```
-----
```

```
select empno, firstnme, lastname
from employee, department
where employee.workdept = department.deptno and mgrno='000060'
```

这个查询就是用户所输入的查询，应当与用户的输入完全匹配。如果用户发现该查询不符合刚输入的命令，请详细检查先前的步骤。

(4) 查询重写器所重写的查询：

```
Optimized Statement:
-----
SELECT Q2.EMPNO AS "EMPNO", Q2.FIRSTNME AS "FIRSTNME", Q2.LASTNAME AS
"LASTNAME"
FROM DB2INST1.DEPARTMENT AS Q1, DB2INST1.EMPLOYEE AS Q2
WHERE (Q1.MGRNO = '000060') AND (Q2.WORKDEPT = Q1.DEPTNO)
```

可以看到，重写器在每个表后面加上了别名，然后重新组织了一下两个条件。

注意：在这里想要提一句，经常看到有用户问：到底哪一个条件应该放在语句的前面。在 DB2 中，答案是无所谓的。DB2 的优化器足够强大，可以通过不同的排列次序估算出执行时哪个条件在前会比较有利。因此用户自己的排列不会被优化器所认可，优化器完全基于开销对访问计划进行评估，例如图 16.1 所示的访问计划。

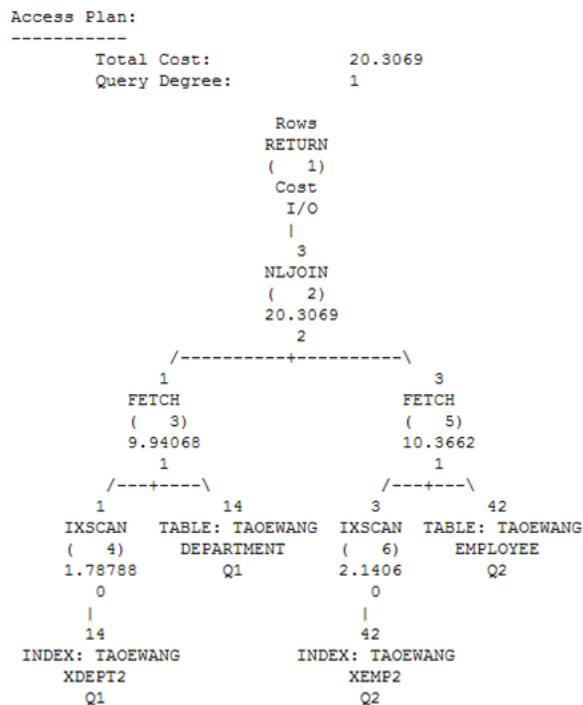


图 16.1 访问计划

这就是我们谈论了很久的访问计划。在这个访问计划中，我们首先看到，**Total Cost** 的估算为 20.3069。注意这个估算值并没有实际单位。其值并不是基于秒或者毫秒的，因此在比较两个访问计划开销时，一定要基于同一个系统。在不同系统间的开销不具有可比性。而下面的树状结构图就是访问计划本身。我们在读计划时从下往上，从左往右。

首先，左下方是 **IXSCAN**，也就是索引扫描。这个扫描使用了 **XDEPT2** 索引，统计信息显示该表中包含了 14 行，而 **IXSCAN** 预期返回一行，预期开销为 1.78788。然后返回的 **RID** 则通过 **FETCH** 操作从表中读出相应的行，通过嵌套循环关联(Nested Loop Join)与右子树关联。

在右子树中，我们有类似的操作。**IXSCAN** 使用 **XEMP2** 索引中读出，每次预期读取 3 个 **RID**，然后从 **EMPLOYEE** 表中读取相应的行。最后将结果集返回给 **RETURN**，代表查询的结束。

在嵌套循环关联中，其规则是对左子树的每一个元素都会执行一次右子树。譬如说，如果两个表 **A** 包含 (1,2,3)，而 **B** 包含 (3,4)，如果使用嵌套关联并且 **A** 在左子树中，则对于 **A** 中的每一条记录都要扫描一遍 **B** 表。也就是说，对于 1，会检查 **B** 表中是否包含这个数据，然后检查 2，以此类推。

但是如果 **B** 在左子树中，那么首先对于 3 会扫描整个 **A** 表，然后读取 4 再一次扫描 **A** 表中匹配的数据。

在这个访问计划中，我们暂时可以看出，DB2 估算的最佳访问计划是：首先用索引扫描从 **DEPARTMENT** 中得到预期一行，然后对于这些行使用索引扫描，从 **EMPLOYEE** 表中得到 3 行，最后返回结果。

在接下来的讲解中，我们会根据 **db2exfmt** 所提供的越来越多的信息不断完善这个访问计划的描述。

下面我们值得关注的信息是这部分：

```

1) RETURN: (Return Result)
  Cumulative Total Cost:          20.3069
  Cumulative CPU Cost:           129672
  Cumulative I/O Cost:            2
  Cumulative Re-Total Cost:       1.694
  Cumulative Re-CPU Cost:         42350
  Cumulative Re-I/O Cost:         0
  Cumulative First Row Cost:      19.8859
  Estimated Bufferpool Buffers:    3

Arguments:
-----
BLDLEVEL: (Build level)
          DB2 v9.7.0.3 : s101006
HEAPUSE : (Maximum Statement Heap Usage)
          112 Pages
PREPTIME: (Statement prepare time)

```

```

          34 milliseconds
    STMTHEAP: (Statement heap size)
          8192

    Input Streams:
    -----
          9) From Operator #2

              Estimated number of rows:      3
              Number of columns:             3
              Subquery predicate ID:         Not Applicable

              Column Names:
              -----
              +Q3.LASTNAME+Q3.FIRSTNME+Q3.EMPNO

```

该信息描述 RETURN(1)操作中的一些重要信息。

前面的一段是开销估算，例如总开销为 20.3069，其中 CPU 开销为 129672（注意，这些开销没有单位，在对比中只能与同数据库中的开销对比），I/O 开销为 2 等。

其中包括当前的数据库版本、使用了多少 statement heap、编译该语句的时间，以及一些可能的针对优化器的 db2set 信息（在本例中没有显示）。

Input Stream 说明该操作的输入是来自操作（2），预期返回 3 行，包括 3 列，这 3 列的名字分别是 LASTNAME、FIRSTNME 与 EMPNO。

现在，我们继续完善我们的访问计划描述。

先前的描述为：首先用索引扫描从 DEPARTMENT 中得到预期一行，然后对于这些行使用索引扫描，从 EMPLOYEE 表中得到 3 行，最后返回结果。

然后我们添加上一些信息：首先用索引扫描从 DEPARTMENT 中得到预期一行，然后对于这些行使用索引扫描，从 EMPLOYEE 表中得到 3 行，最后返回 3 行，包括 LASTNAME、FIRSTNME 与 EMPNO。

紧接着，看下面的数据：

```

    2) NLJOIN: (Nested Loop Join)
        Cumulative Total Cost:      20.3069
        Cumulative CPU Cost:       129672
        Cumulative I/O Cost:        2
        Cumulative Re-Total Cost:   1.694
        Cumulative Re-CPU Cost:    42350
        Cumulative Re-I/O Cost:     0
        Cumulative First Row Cost:  19.8859
        Estimated Bufferpool Buffers: 3

    Arguments:

```

```

-----
EARLYOUT: (Early Out flag)
      NONE
FETCHMAX: (Override for FETCH MAXPAGES)
      IGNORE
ISCANMAX: (Override for ISCAN MAXPAGES)
      IGNORE

Predicates:
-----
3) Predicate used in Join,
      Comparison Operator:          Equal (=)
      Subquery Input Required:      No
      Filter Factor:                0.0714286

      Predicate Text:
      -----
      (Q2.WORKDEPT = Q1.DEPTNO)

Input Streams:
-----
      4) From Operator #3

          Estimated number of rows:  1
          Number of columns:         1
          Subquery predicate ID:      Not Applicable

          Column Names:
          -----
          +Q1.DEPTNO

      8) From Operator #5

          Estimated number of rows:  3
          Number of columns:         4
          Subquery predicate ID:      Not Applicable

          Column Names:
          -----
          +Q2.WORKDEPT(A)+Q2.LASTNAME+Q2.FIRSTNME
          +Q2.EMPNO

Output Streams:
-----
      9) To Operator #1

          Estimated number of rows:  3
          Number of columns:         3

```

```

Subquery predicate ID:      Not Applicable

Column Names:
-----
+Q3.LASTNAME+Q3.FIRSTNME+Q3.EMPNO

```

这一段数据代表嵌套循环关联步骤，也就是 NLJOIN(2)。

首先出场的还是估算开销，紧接着是一些 NLJOIN 使用的特定参数，然后就是该 NLJOIN 的关联列，在这里就是：

```

3) Predicate used in Join,
   Comparison Operator:      Equal (=)
   Subquery Input Required:  No
   Filter Factor:            0.0714286

   Predicate Text:
   -----
   (Q2.WORKDEPT = Q1.DEPTNO)

```

这部分的意思是，经过根据表对象统计数据的估算，对于 DEPARTMENT 中的每一个返回行有大约 7.14% 的数据在 EMPLOYEE 中满足 Q2.WORKDEPT = Q1.DEPTNO 条件。

剩下的 Input Stream 和 Output Stream 就是输入和输出列，在此不用详细叙述。

现在，我们再来更新访问计划描述：首先用索引扫描从 DEPARTMENT 中得到预期一行，然后对于这些行使用 NLJOIN，与 EMPLOYEE 在(Q2.WORKDEPT = Q1.DEPTNO)列上进行关联，预期大约 7.14% 的数据满足关联条件，这样使用索引扫描从 EMPLOYEE 表中预期得到 3 行，最后返回 3 行，包括 LASTNAME、FIRSTNME 与 EMPNO。

接下来后面的信息是 FETCH(3)与 IXSCAN(4)。由于篇幅所限，这里就不贴出全部数据了，有兴趣的读者可以自己尝试。

这两个操作对应着 FETCH(3)与 IXSCAN(4)。其中 FETCH 是指对于索引扫描的结果中的每一个 RID 都要去表对象中将相应的行读出来，如图 16.2 所示。

```

      /-----+-----\
      1
      FETCH
      ( 3)
      9.94068
      1
      /-----\
      1      14
      IXSCAN  TABLE: TAOEWANG
      ( 4)    DEPARTMENT
      1.78788  Q1
      0
      |
      14
      INDEX: TAOEWANG      INDI
      XDEPT2
      Q1

```

图 16.2 Fetch 和 Ixscan 访问计划

而我们比较感兴趣的是 IXSCAN(4):

```

4) IXSCAN: (Index Scan)
    Cumulative Total Cost:          1.78788
    Cumulative CPU Cost:           44697
    Cumulative I/O Cost:           0
    Cumulative Re-Total Cost:       0.36944
    Cumulative Re-CPU Cost:         9236
    Cumulative Re-I/O Cost:         0
    Cumulative First Row Cost:      1.71904
    Estimated Bufferpool Buffers:    1
    Predicates:
    -----
    2) Start Key Predicate,
        Comparison Operator:         Equal (=)
        Subquery Input Required:     No
        Filter Factor:               0.0714286

        Predicate Text:
        -----
        (Q1.MGRNO = '000060')

    2) Stop Key Predicate,
        Comparison Operator:         Equal (=)
        Subquery Input Required:     No
        Filter Factor:               0.0714286

        Predicate Text:
        -----
        (Q1.MGRNO = '000060')

```

这里最关键的是它的两个 Predicate，乍一看都是(Q1.MGRNO = '000060')，不过仔细看，一个是 Start Key Predicate，一个是 Stop Key Predicate。Predicate 是说，在进行索引查询时，可以根据给定的键，通过索引 B+树的结构快速地定位数据节点，然后从节点中读取 RID，将其返回给 FETCH，从而可以在表对象中将对应的数据读取出来。

现在，我们再来重新组织一下访问计划描述：首先用 Start/Stop Key 索引扫描，根据 Q1.MGRNO = '000060' 的条件从 DEPARTMENT 中得到预期一行，然后对这些行使用 NLJOIN，与 EMPLOYEE 在(Q2.WORKDEPT = Q1.DEPTNO)列上进行关联，预期大约 7.14% 的数据满足关联条件，这样使用索引扫描从 EMPLOYEE 表中预期得到 3 行，最后返回 3 行，包括 LASTNAME、FIRSTNAME 与 EMPNO。

这样就变得很完善了，我们继续看最后的两个操作步骤，如图 16.3 所示。

```

      3
    FETCH
    (   5)
    10.3662
      1
    /---+---\
    3         42
  IXSCAN  TABLE: TAOEWANG
    (   6)      EMPLOYEE
    2.1406      Q2
      0
      |
      42
    DEX: TAOEWANG
      XEMP2
      Q2

```

图 16.3

这两个步骤是从 EMPLOYEE 表中读数据，同理前面所提到的 FETCH 和 IXSCAN，我们直接跳到 IXSCAN 步骤的详细信息：

```

6) IXSCAN: (Index Scan)
  Cumulative Total Cost:          2.1406
  Cumulative CPU Cost:           53515
  Cumulative I/O Cost:            0
  Cumulative Re-Total Cost:       0.72216
  Cumulative Re-CPU Cost:        18054
  Cumulative Re-I/O Cost:         0
  Cumulative First Row Cost:      1.93408
  Estimated Bufferpool Buffers:    1
  Predicates:
  -----
3) Start Key Predicate,
  Comparison Operator:            Equal (=)
  Subquery Input Required:        No
  Filter Factor:                  0.0714286

  Predicate Text:
  -----
  (Q2.WORKDEPT = Q1.DEPTNO)

3) Stop Key Predicate,
  Comparison Operator:            Equal (=)
  Subquery Input Required:        No
  Filter Factor:                  0.0714286

  Predicate Text:
  -----
  (Q2.WORKDEPT = Q1.DEPTNO)

```

这里的 Predicate 依然是 Start/Stop Key，而条件则是(Q2.WORKDEPT = Q1.DEPTNO)，所代表的含义是，对于从 DEPARTMENT 表中使用索引扫描和 FETCH 之后所拿到的行，得到每行的 DEPTNO 列，然后根据该列的数值，在 XEMP2 索引中使用(Q2.WORKDEPT = Q1.DEPTNO)条件，对其输入的每行得到对应的 RID。

因此，最终的访问计划描述为：首先用 Start/Stop Key 索引扫描，根据 Q1.MGRNO = '000060' 的条件从 DEPARTMENT 中得到预期一行，然后对于这些行使用 NLJOIN，与 EMPLOYEE 在 (Q2.WORKDEPT = Q1.DEPTNO)列上进行关联，预期大约 7.14%的数据满足关联条件，这样使用索引扫描，对于从 DEPARTMENT 表上返回的行，根据 Q2.WORKDEPT = Q1.DEPTNO 条件从 EMPLOYEE 表中预期总共得到 3 行，然后预期返回 3 行，包括 LASTNAME、FIRSTNAME 与 EMPNO 列。

由于分析 db2exfmt 并不是一件简单的工作，本文在这里也只是抛砖引玉，帮助读者初步了解如何阅读并理解 db2exfmt 的输出。如果用户想要进一步了解 DB2 优化器，我们将在今后出版的数据仓库部分中详细讨论优化器原理，以及一些基本的分析步骤。

16.2 性能调优简介

□ -----

类似性能诊断，性能调优在很多方面与问题诊断有相通之处，甚至从某种程度上说，性能调优是建立在成功的问题诊断的基础之上。

但是两者还是有一定的差异。

性能诊断强调的是“诊断”二字，也就是弄明白系统为什么慢，瓶颈出在什么地方。可是对于曾经真正做过系统调优的用户来说，也许深有体会的一件事就是，弄明白问题的出处和真正解决一个问题，有时候并不是完全相关的。

也许有人会问：既然一个问题的根源都已经被发现了，为什么不能解决呢？

有的时候，譬如说我们发现一个系统的瓶颈制约在 I/O 上，可是也许我们的系统已经使用最高端的存储设备，同时对于 24×7 的在线系统不允许我们花费一两天的时间将数据重新物理分布，那么在这种情况下，有没有什么其他的办法从侧面解决问题，这就是性能调优的最大的难处。

而另一方面，有的时候彻底地研究性能问题的根源相当困难与费时，那么如何能够在并不完全透彻地了解问题根源之前，就用某些方法绕过问题，也是性能调优专家需要精通的技能。

所以，调优的最终目的不是完全挖掘出问题的根源（与性能分析的终极目标不同），而是用不同手段提升系统的单位时间内吞吐量。有可能性能问题的根源根本不在数据库端（譬如应用程序的设计不好，造成很多锁等待，使得系统性能极不稳定），但是如果我们能够能够在数据库端找到某种方法绕过问题（比如使用 `currently committed` 功能）使性能提升，就是一次成功的性能调优。

正如读者现在可能在想的，从某种意义上说，性能调优比性能分析更加直观与灵活。有的时候只要了解了性能问题的现象，通过经验与一定程度上对应用与数据库的理解，就能够大体上给出一个调优的方向（譬如磁盘问题，还是 CPU 问题，还是内存问题）。

在这一节中，我们不想花费大量的篇章讨论性能调优的原理（关于性能分析的原理可以参照性能诊断章节），而是专注于如何解决和绕过一些常见的性能问题。

类似性能分析，在调优时，首先尝试理解系统的架构。至少要弄明白，该系统有多少 CPU，多少内存，多少连接之类的最基本的问题。

其次，是将问题细化到 CPU、内存、I/O 或者系统懒惰的大方向。在阅读了性能诊断篇章后，相信读者都会对这一步不再陌生。

再次，就是要给出一个期望值，尝试用不同的手段将某种指标提高到目标之上，然后再次重复细化问题的步骤，直到系统的整体性能达到预期。

因此，在系统调优时，我们面临以下两个难点：

- 如何找到一个合理的目标与期望值。
- 用什么手段达到目的。

要说这两个难点哪一个更重要，自然是第一点。只有寻找到一个正确的目标，才能保证我们其后的调优是有效的。如果目标都定位不准，那么也就无从谈起如何调优了。

在没有经历过完整的性能分析时，最常见的手段就是参照一些最佳实践文档，一步步采纳文档中的建议进行调优。

不要小看这种类似纯体力工作的步骤，对于大多数性能问题，这种方法可以简单有效地解决问题。不过由于市场上已经存在太多的关于最佳实践的文档，因此作者在本书中将不再讨论这一点。需要的读者可以在 IBM 网站上找到很多相关资料。

当最佳实践文档无法解决用户的难题，或者说文档中的很多建议无法在一个已经运行多年的系统上实现时，我们就需要一个更加具体的思路来找到合理的目标，这就是我们在前文中所提到的 KPI。

KPI 作为国内外无数性能分析专家多年来的总结，可以被大部分系统所采纳。当然，有些 KPI 是针对某一种特定业务（例如 OLTP），而另一种类型的业务（比如 OLAP）可能会在同样的指标中使用完全不同的建议值。

一般来说，KPI 可以被分为以下几个级别：

- 数据库级别。
- 实例级别。
- 操作系统级别。

对于数据库级别 KPI，其分析指标完全来自数据库本身。而实例级别则是针对实例中的一些指标进行监控。操作系统级别就是从高层面观察一个系统的行为，看看是否能够迅速定位出需要提高的部分。

这里我们先列出几种最常用的 KPI，让读者大致明白 KPI 的作用，然后我们会通过一些案例来说明如何进行系统性能调优。

后面我们将会列举更加详尽的 KPI 及其作用。本章调优中，只是抛砖引玉，简单介绍几个 KPI 的用法，如表 16.1 所示。

表 16.1

KPI 名称	来 源	计 算 公 式	建 议 数 值	详 细 说 明
数据索引缓冲池命中率	数据库、缓冲池、表空间快照	(逻辑读-物理读)/逻辑读	优秀: >95% 良好: >80%	代表着一个数据请求可以在不通过物理 I/O 的情况下，直接从内存池找到的几率。该项目的提高，代表使用 I/O 几率的降低
读行 v.s.选择行	数据库快照	Rows Read / Rows Select	OLTP 系统: <=5 SAP 系统: <=3	数据库读取的行数与真正在结果集中所选择的行数。该项目的数值越低，说明每次都读取真正有用的数据的比例越高
排序内存过量使用	实例快照		排序内存高水位 > SHEAPTHRES	当排序内存超出 SHEAPTHRES 时，实例将会分配额外的排序内存。可能导致内存持续的分配与回收，甚至使用临时表空间磁盘
物理磁盘数量	操作系统信息		每 CPU 内核对应 6~20 个物理磁盘数量	物理磁盘的数量决定着 I/O 的效率。当数据库数据被均匀分布到不同磁盘上时，从不同的页面中读取数据会使有更多的机会访问不同的物理磁盘，这样能够加快并行系统的 I/O 吞吐量

作者知道，上面的指标远远不能涵盖日常工作中所遇到的性能问题。不过通过表 16.1，至少读者有了一个关于什么是 KPI 的了解。在下面的 KPI 中我们将详细探讨几十个最重要与常用的性能关键指标。

接下来, 让我们来看几个真正的性能调优案例。正像前面所提到的, 针对系统优化问题的特殊性, 没有什么方法能够保证用户一定找到问题的答案。所以同样地, 我们将会给出相对完整的分析步骤和思路, 但是着重强调所表现出的现象, 而不是像上一章那样所强调后台的原理。

该案例发生在 2011 年 1 月初, 系统为 Linux, 64 位, 8 颗 CPU 内核, 32GB 内存, db2 版本为 DB2 9.5 Fixpack 5。

用户的问题是, 每年年底在运行结算程序时, 系统性能严重下降。该系统平时性能表现一般, 但是当每年年底的结算程序运行时, 一定要减少并发数量, 否则会轻易造成 100%CPU 使用率。

客户明白该应用很可能存在设计问题, 但是需要我们提供一些调优思路帮助解决问题。

系统中只有一个实例运行, 单分区。

既然是没有针对性模块的普通调优建议, 那么我们可以从快照入手分析。

用户重置快照后, 15 分钟收集了若干包括数据库、实例、表空间、SQL、应用程序等快照和一些系统信息, 希望我们能够从中给出建议。

从系统信息中, 我们可以看到系统的使用率维持在大概 50~70%左右。而在数据收集的最后几分钟冲到 90%以上。很显然, 我们的 CPU 使用显然是个大问题, 也从侧面验证了客户曾经的观测结果。

```
procs -----memory----- --swap-- -----io---- --system-- -----cpu-----
r b swpd free buff cache si so bi bo in cs us sy id wa st
3 0 305420 4602212 2072 23901232 0 0 22878 5994 4190 50413 21 16 58 5 0
15 1 305420 4434696 2088 24062144 0 0 4132 17042 4658 16947 11 8 80 1 0
2 1 305420 9139256 2096 19357272 0 0 20491 7516 3845 61736 18 21 59 2 0
2 1 305420 10577924 2112 17933692 0 0 46191 2136 4723 55323 19 14 59 8 0
19 0 305420 9975284 2128 18542952 0 0 81786 3156 5093 156035 56 19 20 5 0
2 1 305420 9579476 2144 18919188 0 0 61790 12208 5431 105615 38 13 38 11 0
1 1 305420 9247096 2152 19254984 0 0 61102 614 4628 35970 24 9 56 11 0
8 0 305420 8885696 2168 19630252 0 0 66905 2662 5085 44408 27 10 51 13 0
1 1 305420 8654744 2184 19857360 0 0 43444 3844 4095 20296 24 8 63 4 0
0 2 305420 8279600 2200 20219676 0 0 64951 10162 5624 12737 43 16 34 6 0
7 0 305420 10358480 2224 18125288 0 0 43894 663 3955 23715 36 13 47 3 0
...
procs -----memory----- --swap-- -----io---- --system-- -----cpu-----
r b swpd free buff cache si so bi bo in cs us sy id wa st
5 0 297096 5964328 4036 22506736 0 0 4334 93423 8621 57023 57 23 19 1 0
10 2 297096 5745696 4052 22740020 0 0 9579 42424 6255 55024 68 17 13 1 0
24 0 297096 5155376 4068 23321812 0 0 717 119550 8734 45119 63 26 11 0 0
24 1 297096 4769692 4084 23710536 0 0 2754 67116 5453 44193 71 23 5 0 0
19 0 297096 4267292 4100 24214268 0 0 1707 121084 7784 39624 67 28 5 0 0
36 0 297096 3824592 4120 24648828 0 0 546 89163 6764 64811 71 22 6 0 0
18 0 297096 3527636 4128 24962944 0 0 11949 55640 6592 63437 58 17 23 2 0
27 0 297096 3195580 4144 25290712 0 0 3430 64558 7308 32265 71 22 7 0 0
27 1 297096 2750700 4160 25733060 0 0 3695 92971 8972 41527 62 24 13 0 0
```

而从 ps -elf 输出中, db2sysc 进程的 C 列维持在 99:

```
4 S db2inst1 28735 28733 99 85 0 - 1983046 184466 08:22 ? 1-05:35:49 db2sysc
```

0

同时还有一个 db2fmp 也在使用不少的 CPU，不过比起 db2sysc 就少很多了：

```
4 S db2inst1 15638 28733 21 76 0 - 233743 semtim 21:27 ? 00:04:34 db2fmp
( ,1,1,0,0,0,0,0,1,0,8a1984,14,1e014,2,0,1,2331fc0,0x2100000000,0x2100000000,16000
00,50268004,2,47da800b
```

这点毫无疑问地说明了 db2sysc 中存在大量消耗 CPU 的任务。

然后在数据库快照中我们看到：

```
Buffer pool data logical reads = 3037962816
Buffer pool index logical reads = 854461868
```

这说明什么呢？数据逻辑读与索引逻辑读的比例相差将近 4 倍，也就是说非常多的读是走表扫描，而不是索引扫描。

于是，这是第一点可能有待提高的方面（当然，对于 OLAP 系统来说，有时候这种现象是正常的。分析人员一定要理解应用程序正在执行的业务）。

然后继续我们的观察：

```
Rows selected = 76395867
Rows read = 45143452324
```

Rows selected 与 Rows read 的比例极低。对比我们刚才的 KPI，在 OLTP 系统中预期数值是 5。而这个系统明显是 OLAP，但是其比例达到 500 以上也是明显不合理的（说明需要读平均 500 行才能真正选定一行用户需要的数据）。

然后我们进入 SQL 快照，通过计算 Total user cpu time / Number of executions 得到单一 SQL 的平均消耗 CPU 时间，可以看到：

```
Number of executions = 434
Total user cpu time (sec.microsec) = 22828.046168
Rows read = 11327044778
Rows written = 9804015280
Buffer pool data logical reads = 1280044999
Buffer pool data physical reads = 171604105
Buffer pool temporary data logical reads = 51594042
Buffer pool temporary data physical reads = 4860358
Buffer pool index logical reads = 22887712
Buffer pool index physical reads = 4403
Statement text = INSERT INTO SESSION.MYTEMPTABLE...
```

具体的 SQL 在这里不显示。通过观察 434 次执行的总 CPU 时间与逻辑、物理读的数量，可以发现该 INSERT 占用大量的资源（平均每个 INSERT 要读取 2600 万行）。

继续找还可以发现其他占用大量 CPU 的语句：

```
Number of executions = 402
Total user cpu time (sec.microsec) = 4708.016408
Buffer pool data logical reads = 542103826
Buffer pool data physical reads = 21217460
```

```

Buffer pool temporary data logical reads = 5240499
Buffer pool temporary data physical reads = 29094
Buffer pool index logical reads = 564654474
Buffer pool index physical reads = 3814
Rows read = 1007057175
Rows written = 147518950
Statement text = insert into SESSION.MYTEMPTABLE2...

Number of executions = 22461
Total user cpu time (sec.microsec) = 4635.475109
Rows read = 228327230
Buffer pool data logical reads = 33378048
Buffer pool data physical reads = 95206
Buffer pool temporary data logical reads = 16846
Buffer pool temporary data physical reads = 0
Buffer pool index logical reads = 101241520
Buffer pool index physical reads = 18
Statement text = SELECT COUNT(*) INTO :HV00011 :HI00011 ...

```

通过对比这些逻辑读与 CPU 消耗的时间，我们可以清楚地看到在重置快照到数据收集的这段时间内，到底哪些 SQL 占用了最多的资源。

然后继续应用程序快照，我们看到了当前有 10 个正在 UOW Executing 的连接，其中 8 个为可执行文件，2 个为 Java：

```

Application handle           = 48138
Application status           = UOW Executing
Status change time          = Not Collected
Application code page        = 819
Application country/region code = 0
DUOW correlation token       = A1C23E82.C281.012D592C1D49
Application name             = db2jccThread-7
Application ID               = A1C23E82.C281.012D592C1D49
...

Application handle           = 48326
Application status           = UOW Executing
Status change time          = Not Collected
Application code page        = 819
Application country/region code = 1
DUOW correlation token       = 161.194.65.138.1037.11010602420
Application name             = myprogram.exe
Application ID               = 161.194.65.138.1037.11010602420
Sequence number              = 00001

```

两个 Java 程序都是在执行同样的 INSERT，同时另外 8 个 .exe 程序都在执行前面所看到的那个消耗 CPU 最大的查询（查询本身大约为 350 行的 SQL）。

这样，我们就有足够的理由认为，这几个（尤其是那个消耗 CPU 最大的查询）查询需要被进一步优化，而入手点则应该从减少逻辑读开始。因为单纯的内存读并不占用 I/O，但是大量的内存访问绝对对 CPU 是一大负担。

当然，找到了这些 SQL 语句后，从 DBA 的角度，我们可以尝试加一些索引或其他一些物理设计影响优化器的执行计划，减少表扫描和排序等，降低 CPU 和其他资源的占用。如果在 DBA 层面无法解决的问题，比如 SQL 语句写的实在很差，那就要从查询的设计入手（开发人员的工作，而不是运维人员的任务），看是否可以从逻辑的角度减少数据访问的数量。

在上面的例子中，我们尽管没有明确提及，但是读者应该看到了，引导我们针对 SQL 进行调优分析的重要依据就是“数据读取/数据选择”的比例。尽管在 OLAP 系统中很难找到一个合适的数值来定义其好坏，但是每 500 个数据读对应一个选择的比例，在这种中等系统中过于夸张。假设如果有一种理想状态下的方法使得所读的每行都是要选择的数据，那么理论上系统性能将会提高 500 倍。因此，我们的调优方向就是尽可能减小这个比例。有些情况下，我们的调优并不能与问题的根源正面交锋，而是需要一些迂回的策略解决问题。

在接下来的案例中，由于作者并没有太多当时收集的性能统计数据，所以作者只是简要地阐述一下问题的现象和调优的思路。

该案例发生于 2010 年初国外的一家银行。该系统运行 DB2 V8，主要用于全国各地支行的柜台业务员在工作时间的一些业务处理（也就是 OLTP）。

系统的整体运行相对稳定，但是唯一有问题的就是其中的一个操作对应着数据库中的一个存储过程。该存储过程并不复杂，最主要的部分是一个 INSERT 操作，从 4 个表中做 SELECT 然后插入一个临时表。

问题就是，每周刚开始的时候该存储过程的性能还算说得过去，大概维持在 0.5 秒左右。可是随着数据量的增加该存储过程的性能不断下降。到了周五的时候就只有 3~4 秒了。然后第二个周一开始的时候，存储过程性能恢复正常，但是在一周中持续下降，到周五又变成 3~4 秒了。

通过描述，我们第一个问题就是：每天的数据量是否相同。

通过对系统的深入研究，发现用户每周的维护时间都使用脚本将一个大表清空一半左右的数据，然后进行 REORG、RUNSTATS 和 REBIND，开始新的一周的工作。

看到这里，相信很多读者都意识到发生了什么，就是随着数据量的上升，系统的统计信息无法反应最真实的当前状态，因此优化器一直认为表中的数据量相对较少，因而选择了一个对当前数据量并非最优的访问计划。

逻辑上的推理完全能够解释这个现象，根据当时所观察到的数据，我们同样发现，问题的大表所被读取的行数在一周内的每天都有增长，同时对应的语句在一周内不同时间使用动态 SQL 抓 exfmt，所显示的访问计划与维护完成之后的计划明显有所区别。

这些数据都可以从侧面证实我们的推论（与性能诊断相同，在调优中千万不要由于某几条快照信息就匆忙地下结论，一定要从多个方向和角度看待问题）。

但是用户的系统维护策略无法让他们每天运行 RUNSTATS 和 REBIND，因此我们必须使用其他的方法来保证存储过程使用最优的访问计划。

那么我们应该怎么办呢？

通过一般的逻辑判断，假设一个查询的访问计划对于大数据量时是最优的，那么当数据量减少后，尽管该计划可能并非最优，但是其执行时间应该不会比数据量占用更多时长。因为按照同样访问数据的方法，在数据量减小的情况下，性能不会比数据量多时更差。

在这个前提条件成立的情况下，我们测试了手工 **RUNSTATS** 和 **REBIND**，发现在数据量最多的情况下，使用最新的统计数据可以使存储过程维持在一秒之内。

那么我们下一步要做的就是看看如何让这个存储过程一直强制使用该访问计划。

可能有人说要用 **profile**，不过不需要那么复杂，我们有两个选择。

一种方法就是在删除数据之前做 **RUNSTATS** 与 **REBIND**，然后 **REORG** 表释放空间，另一种方法就是找一个周末删除数据之前做 **RUNSTATS+REBIND** 的存储过程，然后在今后的维护脚本中永远不进行 **REBIND**。

这两种方法的目的是是一样的，就是能够在数据量减小之前将该存储过程的访问计划设定好，这样在接下来的一周中就算数据量减小，也可以使用同样适用于大规模数据的访问计划。

这个案例告诉我们，性能调优并不一定是要解决系统性能问题的根源。在这个案例中，如果想要解决根源问题，需要每天定时进行 **RUNSTATS** 与 **REBIND**，但是在客户的系统中根本无法实现。这样就需要我们使用其他的方法“绕过”这个问题。所以，这里再次强调，性能调优不同于性能分析，我们的终极目标并不是找到根源并且硬碰硬地解决它，而是通过各种迂回的办法绕过这些问题。在很多时候，我们可能并不知道问题的根源，而是凭着处理过大量性能问题后得到的宝贵经验，对某一个或几个参数进行调整后提升性能，都是能够被认可的成功性能调优。

在书中，分析并且理解问题的根源是关键的，但是在用户的工作中，结果才是最重要的。因此读者千万不要因为阅读本书后而对自己的思想产生局限，而是要以此为基础拓展思路，争取早日获得属于自己的性能调优理念。

根据笔者的经验，很多数据库的性能问题都是 **SQL** 语句引起，对于 **DBA** 来说，即使找到了差的 **SQL** 语句，也很难直接修改 **SQL** 语句本身。这时，就可以考虑在数据库层面对数据库做一些优化。接下来，将重点介绍索引的使用和最佳实践，以及排序的根源及减少排序的思路和方法。

16.2.1 索引 ■ ■ ■

在 7.4 节，介绍了索引的算法、原理和命令使用，本节我们要谈的是索引的优缺点及如何有效地使用索引，即索引使用的最佳实践。

相信很多用户都有过类似经验，当发现一个 **SQL** 运行效率不佳的时候，在向别人询问时，他人的答案很多都是“加索引”。但是不知道大家思考过没有，什么样的索引才是有效的。

我们在前面提到了，索引实际上就是 **B+**树，树中的每一个节点包含键值与一系列 **RID**。索引之所以能够提升性能，就是在于当查找 **SQL** 的某个谓词时，可以使用 **B+**树迅速地找到对应的一系列键值，而不用去扫描整个表中所有的数据页。

因此，了解了 **B+**树的工作原理后，如何有效地使用索引也就呼之欲出了：对于给定谓词，

能够将谓词中的条件列与索引键值所代表的列相匹配，并且当优化器判定使用 B+树扫描的效率高于全表数据页扫描的时候，这个索引就是有效的。

在这里我们有两个条件。第一个条件就是，怎样能够让谓词条件与索引键值对应的数据列所匹配；而第二个条件就是，怎样让优化器认为使用索引的效率更高。

在平时的工作中，作者发现很多用户仅仅注意其中的一个条件。当发现建立的索引不能被使用时，很多人仅仅从其中一个方面入手但是忽略了另外一个因素。

首先，让我们看一看怎样将谓词条件与索引键值对应的列匹配。

在一个 B+树的键中，当包含多个元素的时候（多个数据列），其排列次序是顺序排列，也就是在一个给定的 B+树节点（键）中，按照键值 1、键值 2、键值 3 一个接一个地排列。

而当使用 B+数进行多个元素的查找时，首先匹配键值 1。如果键值 1 不符合查找的第一个元素所引用的范围，则查找其左子树或者右子树。当键值 1 匹配的时候，则进一步判定键值 2 是否满足引用的范围，以此类推。

这样我们能够看出，想要有效地利用 B+树，最重要的一点就是，对于键中的每一个元素（每一列），想要使用元素 2，必须要在元素 1 被判定后才有可能。也就是说，假设一个键中包含数据列 (C1,C2,C3)，那么想要判定 C2 的时候，给定的谓词条件中必须包含 C1。同理，想要判定 C3 的时候，谓词条件必须保证 C1、C2 列都被包含。只有满足这个条件的谓词，才能够在索引扫描时使用 Start/Stop Key 谓词检索。否则即使使用索引，也是 Sargable 谓词或者 Residual 谓词扫描整个 B+树，其开销在很多情况下甚至高于表扫描。

具体这些谓词含义的解释，我们将在《DB2 数据库高级管理》中的优化器部分详细讨论。现在我们给出 Start/Stop Key 与 Sargable 谓词显示的例子：

```
/home/db2inst1/temp $ db2 "create table t1 (c1 int, c2 int, c3 int, c4 int)"
/home/db2inst1/temp $ db2 "create index i1 on t1 (c1, c2, c3)"
/home/db2inst1/temp $ db2 "insert into t1 values (1,2,3,4)"
...-- 插入一些不同的数据
/home/db2inst1/temp $ db2 set current query optimization 0
/home/db2inst1/temp $ db2 set current explain mode explain
DB20000I The SQL command completed successfully.
(db2inst1@db2host1) /home/db2inst1/temp $ db2 "select * from t1 where c1=? and c2=?
and c3=? and c4=?"
SQL0217W The statement was not executed as only Explain information requests
are being processed. SQLSTATE=01604
/home/db2inst1/temp $ db2 set current explain mode no
DB20000I The SQL command completed successfully.
/home/db2inst1/temp $ db2exfmt -d SAMPLE -g TIC -w -1 -n % -s % -# 0 -o
prod_sample_exfmt.txt
DB2 Universal Database Version 9.7, 5622-044 (c) Copyright IBM Corp. 1991, 2009
Licensed Material - Program Property of IBM
IBM DATABASE 2 Explain Table Format Tool
```

```

Connecting to the Database.
Connect to Database Successful.
vim Output is in prod_sample_exfmt.txt.
Executing Connect Reset -- Connect Reset was Successful.

```

```

Original Statement:
-----
select *
from t1
where c1=? and c2=? and c3=? and c4=?

Optimized Statement:
-----
SELECT ? AS "C1", ? AS "C2", ? AS "C3", ? AS "C4"
FROM TAOEWANG.T1 AS Q1
WHERE (Q1.C4 = ?) AND (Q1.C3 = ?) AND (Q1.C2 = ?) AND (Q1.C1 = ?)

Access Plan:
-----
          Total Cost:          4.72413
          Query Degree:        1

          Rows
          RETURN
          ( 1)
          Cost
          I/O
          |
          0.0144
          FETCH
          ( 2)
          4.72413
          0.36
          /-----\
          0.36          9
          IXSCAN  TABLE: TAOEWANG
          ( 3)          T1
          1.75942      Q1
          0
          |
          9
          INDEX: TAOEWANG
          I1
          Q1

```

图 16.4

```

Predicates:
-----
3) Start Key Predicate,
   Comparison Operator:      Equal (=)
   Subquery Input Required:  No
   Filter Factor:            0.04

   Predicate Text:
   -----
   (Q1.C3 = ?)

3) Stop Key Predicate,
   Comparison Operator:      Equal (=)
   Subquery Input Required:  No

```

Filter Factor: 0.04

Predicate Text:

(Q1.C3 = ?)

4) Start Key Predicate,

Comparison Operator: Equal (=)

Subquery Input Required: No

Filter Factor: 0.04

Predicate Text:

(Q1.C2 = ?)

4) Stop Key Predicate,

Comparison Operator: Equal (=)

Subquery Input Required: No

Filter Factor: 0.04

Predicate Text:

(Q1.C2 = ?)

5) Start Key Predicate,

Comparison Operator: Equal (=)

Subquery Input Required: No

Filter Factor: 0.04

Predicate Text:

(Q1.C1 = ?)

5) Stop Key Predicate,

Comparison Operator: Equal (=)

Subquery Input Required: No

Filter Factor: 0.04

Predicate Text:

(Q1.C1 = ?)

下面的是使用 `sargable` 谓词的语句:

```

Original Statement:
-----
select *
from t1
where c4=? and c3=?

Optimized Statement:
-----
SELECT Q1.C1 AS "C1", Q1.C2 AS "C2", ? AS "C3", ? AS "C4"
FROM TAOEWANG.T1 AS Q1
WHERE (Q1.C3 = ?) AND (Q1.C4 = ?)

Access Plan:
-----
          Total Cost:          155.082
          Query Degree:        1

          Rows
          RETURN
          ( 1)
          Cost
          I/O
          |
          1.6
          FETCH
          ( 2)
          155.082
          7.99996
          /-----\
          40          1000
          IXSCAN    TABLE: TAOEWANG
          ( 3)      T1
          120.558    Q1
          4
          |
          1000
          INDEX: TAOEWANG
          I1
          Q1

```

图 16.6

```

Predicates:
-----
2) Sargable Predicate,
   Comparison Operator:      Equal (=)
   Subquery Input Required:  No
   Filter Factor:            0.04

   Predicate Text:
   -----
   (Q1.C3 = ?)

```

因此，简单地说，就是一个能够被使用 Start/Stop Key 谓词的索引，在查询中谓词所引用的列中一定要包含该索引的起始列。

譬如查询“SELECT * FROM MYTABLE WHERE C1=5 AND C2=3”，一个有效的索引起始列必须是 C1 或者 C2 列。如果一个索引即使包含了 C1 或者 C2，但是起始列是其他列，则依然不能被看成是合理有效的索引。

而第二个条件，如何影响优化器使用索引，就是一个更为复杂的问题。由于对优化器的详细介绍将在下一本数据仓库书中进行，因此在这里我们不讨论优化器的算法。

一般来说，想要影响优化器，需要从统计信息上做文章。如果用户发现即使建立的索引合法，但是有时候优化器依然使用全表扫描，这个时候用户可以将优化级别调为 0，或者在表上加 `volatile` 标识，使优化器更倾向于使用索引扫描而不是全表扫描。

`volatile` 的意思是可变的、易变的。我们知道，优化器选择是否执行索引依赖于统计信息，如果表数据变化很频繁，当统计信息收集的时候，有可能并不代表该表在其他时刻的统计信息。那么优化器如果完全根据统计信息制定访问计划的时候，有可能判定访问计划时所根据的统计信息并不能很好地代表当前的数据分布，这样这个访问计划很有可能并非最优。如果一个表被设定了 `volatile` 属性，优化器就会尽量忽略该表的统计信息，尽量使用索引扫描。

用户可以通过 `Alter table <表名> volatile` 来设定其属性。

在创建索引时，应该遵循以下最佳实践：

- 分析组合索引键的顺序，如：(a,b)和(b,a)是完全不一样的；
- 不要创建冗余索引，如：某张表有(a),(a,b)两个索引，则(a)索引是多余的；
- 验证索引是否被用到，如果没有用到，建议删掉；
- 尽可能通过 `include` 创建 `index-only` 索引，减少数据获取 I/O，提升效能。

索引是 DBA 的性能调优利器，它的主要优点可以概括为以下几点：

- 最主要的目的是提高查询速度；
- 避免不必要的表扫描，表扫描是 CPU 的第一杀手；
- 避免排序操作，排序是 CPU 的第二杀手；
- 减少死锁发生的概率。

但索引也有缺点，总结如下：

- 增加了 `insert/update/delete` 等操作的负担；
- 索引需要占用额外的磁盘空间；
- 增加了运维成本，如 `RUNSTATS`, `REORG`, `LOAD` 等操作都要维护索引。

因此，索引并不是越多越好，对于 DBA 来说，需要找出长期不需要的索引并将其删除，这也是一项比较重要的工作。那么怎样找出无效的索引呢？

以下几种方法可供选择：

(1) 通过 `db2pd-d sample-tcbstats-index`。有一个字段是 `scans`，找到一段时间内 `scans` 为 0 的值，表示这个索引没有被用到，即可删除：

TCB Index Stats:							
Address	TableName	IID	EmpPgDel	RootSplits	BndrySplits	PseuEmptPg	Scans
KeyUpdates	InclUpdats	NonBndSpts	PgAllocs	Merges	PseuDels	DelClean	
IntNodSpl							
0x9FF18424	T1	9	0	0	0	32	0
0	0	0	0	0	0		

(2) DB2 9.7 提供了一个新的监控表 MON_GET_INDEX 用来识别没有用到的索引:

```
db2inst1@dpf1:~> db2 "SELECT
substr(T.tabschema,1,18),substr(t.tabname,1,18),substr(S.INDSCHEMA,1,18),
substr(S.INDNAME,1,18) ,t.page_allocations,s.uniquerule,s.indextype FROM
TABLE(MON_GET_INDEX('',' ', -1)) as T, SYSCAT.INDEXES AS S WHERE T.TABSCHEMA =
S.TABSCHEMA AND T.TABNAME = S.TABNAME AND T.IID = S.IID and T.INDEX_SCANS = 0"
```

TABSCHEMA	TABNAME	IDXSCHEMA	IDXNAME	PAGE_ALLOCATIONS	UNIQUERULE	INDEXTYPE
DB2INST1	T1	DB2INST1	I1	0	D	REG

(3) DB2 9.7 还提供了一个方法用来查看索引最后使用的时间。9.7 在 syscat.tables, syscat.indexes 和 syscat.packages 视图中增加了一个字段 lastused, 用来表示此对象最后一次被访问的时间。01/01/0001 表示此对象从来没有被访问过, 因此可用其作为过滤条件:

```
db2inst1@dpf1:~> db2 "select indname, lastused from syscat.indexes where
lastused='01/01/0001' "
```

16.2.2 排序 ■ ■ ■

在 DB2 数据库系统监控中, 经常会遇到 CPU 资源使用过高问题, 造成此类问题的原因很多, 但最主要的原因有两个, 第一是过多的逻辑 I/O 读取, 第二就是过多的排序 (Sort), 这两个因素有时也称为 CPU 的两大杀手。所谓过多的逻辑 I/O 读取, 通常指发生在缓冲池中的表扫描。

排序是指对某些数据按照某个 (或某些) 字段从大到小或从小到大排列的过程。比如, 按照学生成绩从高到低排序, 根据姓氏字母顺序排序等。

在 DB2 数据库中, 哪些操作会引起表数据排序呢? 熟悉 SQL 的朋友立即会想到 Order By。没错, 如果排序字段上没有索引, 或者 DB2 认为索引的开销比表扫描更大时, DB2 就会对数据进行排序。

除 Order By 之外, 在以下操作中也可能引起排序:

- 对于包含 DISTINCT、GROUP BY、HAVING、INTERSECT、EXCEPT、UNION 等操作的 SQL 语句, 如果没有索引满足所取的行的顺序要求, 或者优化器认为排序的代价低于索引扫描, 就需要进行排序。
- 对于包含 Max/Min/Sum/Cube/Rollup/Rank 等聚集函数, DB2 会把语句重写为一个使用排序的嵌套子查询。
- 对于 Reorg, Create Index 等操作, 需要对表数据进行排序才能完成。
- 对于查询计划中的 Dynamic Bitmap Index ANDing (IXAND)、Hash Join (HSJOIN) 等操作的生成的 Hash 表会放在 SORTHEAP 中。

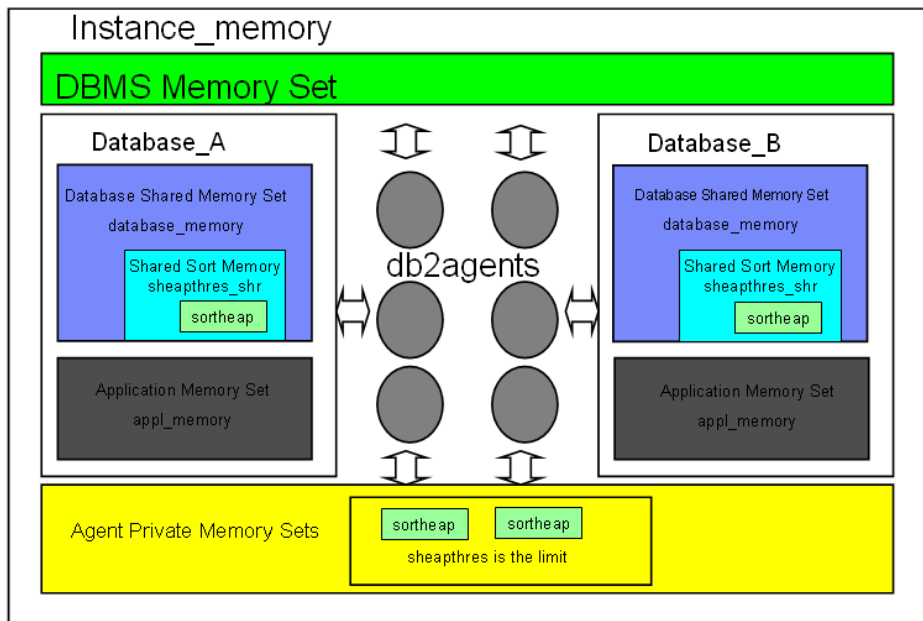
通常情况下, 大量的排序会对性能造成极大影响。比如, 排序会导致很高的 CPU 使用时

间；增加 SQL 执行时间；增加锁超时和死锁发生的几率；排序会严重消耗有限的内存空间；排序溢出会引起临时表空间的频繁 I/O 等。在实际运维过程中，经常发现很多人对排序的监控和调优缺乏理论和实践经验，遇到相关问题时不知所措。本节首先介绍排序的原理，然后通过一个实例介绍排序的监控和诊断方法，最后提出减少排序的建议，希望对大家有所帮助。

1. 排序的原理

正常情况下，DB2 排序发生在内存中，这块内存叫做排序堆，即 **SORTHEAP**。当需要排序的数据超出 **SORTHEAP** 大小限制时，就会发生排序溢出。溢出的数据会写到临时表中，这会产生更多的 I/O，因此对性能会有较大影响。

通过图 16.6 可知，DB2 的内存集包括实例内存集、数据库共享内存集、应用程序内存集和代理私有内存集等。内存池是从内存集中分配的。根据排序内存池的分配来源，分为私有排序和共享排序。私有排序是从代理私有内存集中分配的，而共享排序从数据库共享内存集中分配。



DB2 选择私有排序还是共享排序，是由 3 个排序参数决定的：**SHEAPTHRES**、**SHEAPTHRES_SHR** 和 **SORTHEAP**。在不同配置组合下，DB2 对排序内存的分配使用方式也大不相同，不同版本的 DB2 对排序内存的使用也存在较大差异。下面我们看一下这 3 个排序参数的用法。

- **SORTHEAP**: 数据库配置参数，指定为每个排序分配的最大内存大小，实际使用的大小是由优化器来决定的。如果表的统计信息不准确，会导致优化器对要使用的排序内存的大小估算不准，有可能分配比实际需要少的内存，导致不必要的排序溢出。这就提醒我们要经常使用 **runstats** 更新统计信息。

- **SHEAPTHRES_SHR**: 数据库配置参数, 该参数指定了数据库共享内存集中共享排序内存池的大小, 它限制了该数据库上的所有应用能达到的共享排序内存上限。在 DB2 8 版本中, 这个值是硬限制, 当达到此限制后, 请求排序的新应用会收到 SQL0955 (reason code 2) 错误。从 DB2 9.1 起, 这个参数改为软限制, 超过 SHEAPTHRES_SHR 的共享排序请求可以从数据共享内存集的溢出区 (Database overflow Buffer) 获得。
- **SHEAPTHRES**: 实例配置参数, 指定为本实例中所有私有排序分配的内存上限的软限制。当私有排序分配的内存达到了此限制, 新请求的私有排序的内存大小分配将会小于 sortheap 配置的大小。

简单地说, 每个排序是从 SORTHEAP 中分配的。如果使用私有排序, 那么允许分配的排序内存大小不能超过 SHEAPTHRES 实例参数; 如果使用共享排序, 允许分配的排序内存大小不能超过 SHEAPTHRES_SHR 数据库参数。

那么如何配置 DB2 使用共享排序还是私有排序呢? 从 DB2 9.1 开始, 如果将 SHEAPTHRES 实例参数设置为 0, DB2 将使用共享排序, 即排序内存从共享排序内存池中分配, 排序内存的最大限制由 SHEAPTHRES_SHR 决定。

2. 排序的监控

DB2 排序可以通过 SNAPSHOT 快照监控, 快照监控结果提供了很多关于排序的信息, 如总的排序次数、排序时间、排序溢出数量等。以下是数据库快照排序监控指标:

```
inst20@db2server:/data1/sh> db2 get snapshot for database on perfdb | more
```

```

      Database Snapshot
Total Private Sort heap allocated      = 9553
Total Shared Sort heap allocated      = 0
Shared Sort heap high water mark      = 0
Total sorts                           = 12936047
Total sort time (ms)                  = 19098348
Sort overflows                        = 83950
Active sorts                          = 1
... ..
Commit statements attempted           = 1130957
Rollback statements attempted         = 519

```

- **Total sorts**: 表示发生的总排序次数。
- **Total sort time(ms)**: 表示发生的总排序时间。
- **Sort overflows**: 表示发生的排序溢出次数。
- **Active sorts**: 表示监控时正在进行的排序次数。

前 3 个指标是自数据库启动以来的统计值, 最后一个指标是当前值。几个关键的指标如下:

- $\text{Sort overflows} / \text{Total sorts} * 100\%$ 表示排序溢出百分比, 通常情况下, 该值应该小于

3. 如果大于 3, 表示溢出的比例太高, 需要优化。

- `Total sorts time(ms)/Total sorts` 表示每次排序花费的时间 (毫秒), 对于交易系统来说, 该值最好小于 50ms。
- `Total sorts time(ms)/(Commit statements attempted + Rollback statements attempted)` 表示每个事务花费在排序上的时间。一个事务响应时间包含很多方面, 比如读/写时间、锁时间、排序时间、CPU 时间等。排序时间越少, 最终用户的响应时间也越快, 占用的 CPU 资源也越少。

上例中, 在 4 天的时间内共计发生了 12936047 次排序, 排序溢出比为 $83950/12936047=0.6\%$, 每个事务需要花费的排序时间为 $19098348/(1130957+519) = 16\text{ms}$ 。

除了数据库快照, 应用程序快照、动态 SQL 快照也包含一些排序信息。这对于我们定位排序根源有很大帮助。以下是动态 SQL 语句快照片段, 该语句平均每次执行需要 4 次排序, 而且 3 次发生溢出, 根据这些信息, 该语句很明显需要优化。

```
Number of executions           = 36
Number of compilations         = 1
Worst preparation time (ms)    = 18
Best preparation time (ms)     = 15
Internal rows deleted          = 0
Internal rows inserted         = 0
Rows read                     = 57462636
Internal rows updated          = 0
Rows written                   = 30062371
Statement sorts                = 144
Statement sort overflows       = 108
Total sort time                = 145700
Buffer pool data logical reads = 19546362
Buffer pool data physical reads = 8
Buffer pool temporary data logical reads = 4141609
Buffer pool temporary data physical reads = 250
Buffer pool index logical reads = 35513607
Buffer pool index physical reads = 1096
Buffer pool temporary index logical reads = 0
Buffer pool temporary index physical reads = 0
Total execution time (sec.ms)  = 355.390643
Total user cpu time (sec.ms)   = 164.663270
Total system cpu time (sec.ms) = 1.529099
Statement text                  = select * from ... (略)
```

3. 排序的优化

以上我们介绍了排序的监控, 当发现排序的监控指标超出期望值时, 就需要优化。影响排序的因素主要包括排序的行数、排序列的宽度和 `ORDER BY` 的列数等, 排序的行数乘以排序列的宽度决定了一次排序占用的空间, 因此, 应该尽量减少排序的行数(很多情况下, 排序列无法更改)。以下介绍一些常用的排序优化方法。

1) 参数调优

从 DB2 9.1 开始, SHEAPTHRES_SHR 实例参数的默认值为 0, 这表示排序内存从数据库共享内存区分配, 排序总量由 SHEAPTHRES 数据库参数控制。

那么, 如何设置 SORTHEAP 和 SHEAPTHRES 的大小呢?

首先 SORTHEAP 是每次排序能够分配的最大内存空间, 当排序的数值超过该值时, 就要发生排序溢出, SORTHEAP 的默认大小是 256 页, 即 1MB。假定, 要排序的数据每行是 1KB 大小, 那么每个 SORTHEAP 最多允许排序 $1\text{M}/1\text{K}=1024$ 行, 超出 1024 行数据, 排序将会溢出。

```
inst20@db2server:/data1/sh> db2 get db cfg for sample | grep -i sort
Sort heap thres for shared sorts (4KB) (SHEAPTHRES_SHR) = 5000
Sort list heap (4KB) (SORTHEAP) = 256
```

对于数据仓库系统来说, 一般行记录很长, 排序的行数也很多, 这是可考虑适当增大 SORTHEAP 大小, 但不要扩得太大。

SHEAPTHRES_SHR 参数, 用来限定所有排序可以占用的内存空间。该值的大小由两个指标来决定, 一个是 SORTHEAP, 另外一个同时排序的个数。但是, 同时排序的个数很难估计, 一般采用并发数来估计。以下的监控结果可知, 当时正在并发的应用数量大概是 10 个。

```
db2inst1@ibmswg01:~/> db2 get snapshot for database on prefdb
Appls. executing in db manager currently = 10
```

考虑到单个应用可能需要进行多次排序, 可设置:

```
SHEAPTHRES_SHR = SORTHEAP * 应用并发数 * 2
```

在 DB2 9.1 及以上版本, DB2 提供了自动调优功能 (STMM), 好处是 DB2 根据工作负载和资源情况, 决定一些内存的分配和回收, 比如缓冲池、排序堆、锁列表和 Package cache 大小, 当需要排序时, 可以从别的内存区借用, 当不需要时归还, 充分利用内存使用。对于运行稳定的系统, 可考虑启用 STMM, 设置这两个参数为 automatic 自动值。

```
inst20@db2server:/data1/sh> db2 update db cfg for perfdb using SORTHEAP automatic
inst20@db2server:/data1/sh> db2 update db cfg for perfdb using SHEAPTHRES_SHR
automatic
```

2) 逻辑/物理设计优化

调整参数有些情况下能解决问题, 但可能会隐藏问题, 因为调参只是尽量减少排序溢出的几率, 并不能减少排序的次数。因此, 作为一个优秀的 DBA, 需要挖掘排序的根源, 并给出解决方案。根源是什么? 尽管 Reorg、建索引等操作需要排序, 但几乎 99% 的概率是 SQL 语句引起的。在 SQL 语句无法更改的情况下, 作为 DBA, 能够做的就是物理/逻辑设计上做优化。

那么如何找到哪些语句发生了大量排序呢? 答案就是我们前面介绍过的动态 SQL 快照和应用快照监控。

当找到 SQL 语句后，就可以通过一些物理设计来优化 SQL，如索引、物化视图（MQT）等。当然，一些影响优化器的运维工作也必不可少，如 runstats、reorg、rebind 等操作。

索引是减少排序的利器，因为索引本身已经是经过排序的数状结构。创建合适的索引会大大减少、甚至避免排序。当 SQL 语句中出现以下操作时，可以考虑在相应的字段上创建索引：

- Order By/Group By 操作，如：

```
SELECT a,b,c FROM tab1 ORDER BY a,b
SELECT a,b,c FROM tab1 WHERE a=100 ORDER BY b
SELECT a,b, count(*),sum(*) FROM tab1 group by a,b
```

这时可考虑在 tab1(a,b)上建索引。

- Distinct 操作，如：

```
SELECT distinct a FROM tab1
```

可考虑在 tab1(a)上建索引，对于 distinct 的查询可避免排序。

- Hash Join 操作需要在 SORTHEAP 中建立哈希表，对于发生大量 Hash Join 的表，可考虑建立索引。
- Create index idx_1 on tab1(a) allow reverse scan

建索引时考虑用 Allow Reverse Scans 进行索引值逆向扫描，这也是 DB2 9.1 的默认选项。

3) 调优 SQL 语句

如果通过前两步仍然不能解决排序问题，那就需要对 SQL 语句本身进行逻辑调整和修改。比如：

- 能否在 SQL 语句中省略 ORDER BY, DISTINCT 等操作，如果无法省略，能否尽量减少排序的行数或列数，比如通过增加过滤条件等。
- 如果 UNION All 能满足需求，就避免用 UNION，因为 UNION All 不需要排序。
- 写优化的 SQL 语句，比如不要在排序字段上使用函数(因为函数无法走索引)。

4. 排序问题的诊断分析

下面我们通过一个详细的例子说明对排序问题的诊断分析过程。

1) 搭建实验环境

以下实验在 DB2 9.5 for Linux 下运行。首先创建一个 SORTDB 数据库，然后创建一个 EMPLOYEE 员工表，并插入 100 万行数据，其中 DEPT_ID 字段是随机产生的 1000 个部门编号，所有员工数据均匀分布在 1000 个部门里，每个部门大概包含 100 万/1000=1000 个员工。

脚本如下（以下命令存到一个脚本文件 create.ddl）：

```
--创建测试 database
```

```

create db sortdb@

--更改日志参数, 否则会在插入数据时出现 log full
update db cfg for sortdb using logfilsiz 2000 @
update db cfg for sortdb using logprimary 20@
update db cfg for sortdb using logsecond 30@

--日志参数的更改需要断掉连接才会生效
force applications all@

--连接数据库
connect to sortdb@

--创建 bufferpool
create bufferpool bp4k size 100000@

--创建 automatic stored tablespace
create tablespace emp_dms bufferpool bp4k@

--创建 table
create table db2inst1.employee
( emp_id int not null primary key,
  name char(20),
  dept_id int,
  salary decimal,
  address char(30),
  remark char(40)
) in emp_dms
@

--向表里插入 100 万条数据, 其中 dept_id 的范围为 0-1000, 其余几个 char 字段值都是根据 count 值产生
begin atomic
  declare count int default 1;
  while (count <1000000) do
    insert into db2inst1.employee values(
      count,
      '--emp' concat char(count),
      ceiling( (rand()*1000) ),
      20000.50,
      '-----address' concat char(count) concat '---',
      '-----remark' concat char(count) concat '----'
    );

    set count=count+1;
  end while;
end
@

--对表做 runstats

```

```
runstats on table db2inst1.employee @
```

执行 `db2 -td@ -f create.ddl` 命令创建脚本。

然后设置排序参数，`sortheap=250`，`sheapthresshr=1000`，并打开监控开关。

```
inst20@db2server:/backup> db2 update db cfg for sortdb using sheapthres_shr 10000
sortheap 250
DB20000I The UPDATE DATABASE CONFIGURATION command completed successfully.

inst20@db2server:/backup> db2 update dbm cfg using SHEAPTHRES 0
DB20000I The UPDATE DATABASE MANAGER CONFIGURATION command completed successfully.

inst20@db2server:/backup> db2 UPDATE DBM CFG USING DFT_MON_SORT ON
DB20000I The UPDATE DATABASE MANAGER CONFIGURATION command completed successfully.

inst20@db2server:/backup> db2stop force
06/03/2011 07:33:04 0 0 SQL1064N DB2STOP processing was successful.
inst20@db2server:/backup> db2start
06/03/2011 07:33:15 0 0 SQL1063N DB2START processing was successful.
```

然后查询 `dept_id<100` 的记录，并按 `dept_id` 排序，大概会有 9 万 9 千行记录返回。

```
inst20@db2server:/backup> db2 "SELECT dept_id, name FROM db2inst1.employee WHERE
dept_id<100 ORDER BY dept_id" | grep selected
99011 record(s) selected.
```

2) 对排序问题的诊断和分析

现在我们要通过数据库监控，查找排序问题的根源。

①数据库监控分析

通过监控结果可知，在共享内存区发生了一次排序，并且发生了排序溢出，排序时间是 204ms。以下监控是 SQL 语句完成后执行的，所以 Total Shared Sort heap allocated 的结果是 0，否则会显示当前分配的排序堆页数。因为是共享排序，所有与私有排序相关的参数都是 0。

```
inst20@db2server:/backup> db2 get snapshot for db on sortdb | more
Total Private Sort heap allocated      = 0
Total Shared Sort heap allocated      = 0
Shared Sort heap high water mark      = 250
Post threshold sorts (shared memory)  = 0
Total sorts                           = 1
Total sort time (ms)                  = 206
Sort overflows                        = 1
Active sorts                          = 0
```

然后监控缓冲池临时数据和临时索引的逻辑读/写页数，当排序溢出时，会溢出到临时表空间中。如果发现临时数据读写值较高，那么很可能是 SORT 引起的。

```
inst20@db2server:/backup> db2 get snapshot for database on sortdb|grep -i "temporary
```

```
data"
Buffer pool temporary data logical reads  = 1792
Buffer pool temporary data physical reads = 31
Buffer pool temporary index logical reads = 0
Buffer pool temporary index physical reads = 0
```

② 监控 SQL 语句

接着根据动态 SQL 语句快照，找到排序多、溢出多的 SQL 语句。我们会发现，该语句执行了一次，执行时间花费了 7.6 秒，Rows Read=1099018, Rows Written= 99011。排序发生了一次，溢出一次。可能有人有会比较奇怪，为什么 select 语句会产生 rows written 呢？这就是因为排序溢出写到临时表空间，产生了写 I/O，溢出的行数为 99011，正对应前面的查询结果。

```
Number of executions          = 1
Number of compilations        = 1
Worst preparation time (ms)   = 134
Best preparation time (ms)    = 134
Internal rows deleted         = 0
Internal rows inserted        = 0
Rows read                    = 1099018
Internal rows updated         = 0
Rows written                  = 99011
Statement sorts               = 1
Statement sort overflows      = 1
Total sort time               = 204
Buffer pool data logical reads = 30561
Buffer pool data physical reads = 2367
Buffer pool temporary data logical reads = 1792
Buffer pool temporary data physical reads = 1
Buffer pool index logical reads = 35
Buffer pool index physical reads = 27
Buffer pool temporary index logical reads = 0
Buffer pool temporary index physical reads = 0
Buffer pool xda logical reads = 0
Buffer pool xda physical reads = 0
Buffer pool temporary xda logical reads = 0
Buffer pool temporary xda physical reads = 0
Total execution time (sec.microsec)= 7.612814
Total user cpu time (sec.microsec) = 0.938412
Total system cpu time (sec.microsec)= 0.000000
Total statistic fabrication time (milliseconds) = 0
Total synchronous runstats time (milliseconds) = 0
Statement text                = SELECT dept_id, name FROM db2inst1.employee WHERE
dept_id<100 ORDER BY dept_id
```

当然，也可以使用 SYSIBMADM.SNAPDYN_SQL 管理视图找到排序最多的动态 SQL 语句。

③ SQL 语句分析

通过 explain 工具生成 SQL 语句的执行计划，并通过 db2exfmt 进行格式化。

```

inst20@db2server:~/sqllib/misc> db2 -tvf EXPLAIN.DDL
inst20@db2server:~> more query.sql
SELECT dept_id, name FROM db2inst1.employee WHERE dept_id<100 ORDER BY dept_id;

inst20@db2server:~> db2 set current explain mode explain
DB20000I The SQL command completed successfully.

inst20@db2server:~> db2 -tvf query.sql
SELECT dept_id, name FROM db2inst1.employee WHERE dept_id<100 ORDER BY dept_id
SQL0217W The statement was not executed as only Explain information requests are
being processed.  SQLSTATE=01604

inst20@db2server:~> db2exfmt -1 -d sortdb -o query_exfmt.out

```

然后打开 query_exfmt.out 文件，计划如下：

Access Plan:

Total Cost: 36010
Query Degree: 1

Rows

RETURN

(1)

Cost

I/O

|

98999.9

TBSCAN

(2)

36010

30763

|

98999.9

SORT

(3)

35270.1

30094.5

|

98999.9

TBSCAN

(4)

30134.7

29426

|

999999

TABLE: DB2INST1

EMPLOYEE

Q1

通过以上结果可知,在对表 EMPLOYEE 进行 TBSCAN 后,有一次 SORT 操作,然后对 SORT 的结果再次 TBSCAN。观察 TBSCAN(4), SORT(3)和 TBSCAN(2)这 3 个操作符的累计 I/O 次数,可以看出 I/O 在增长,再看 SORT 操作符的具体信息:

```

3) SORT : (Sort)
    Cumulative Total Cost:          35270.1
    Cumulative CPU Cost:            2.59546e+09
    Cumulative I/O Cost:            30094.5
    Cumulative Re-Total Cost:        0
    Cumulative Re-CPU Cost:          0
    Cumulative Re-I/O Cost:          668.5
    Cumulative First Row Cost:       35270.1
    Estimated Bufferpool Buffers:     30318

Arguments:
-----
DUPLWARN: (Duplicates Warning flag)
          FALSE
NUMROWS : (Estimated number of rows)
          99000
ROWWIDTH: (Estimated width of rows)
          32
SORTKEY : (Sort Key column)
          1: Q1.DEPT_ID(A)
SPILLED : (Pages spilled to bufferpool or disk)
          892
TEMPSIZE: (Temporary Table Page Size)
          4096
UNIQUE : (Uniqueness required flag)
          FALSE

Input Streams:
-----
    2) From Operator #4

        Estimated number of rows:      98999.9
        Number of columns:              2
        Subquery predicate ID:          Not Applicable

        Column Names:
        -----
        +Q1.NAME+Q1.DEPT_ID

Output Streams:
-----
    3) To Operator #2

        Estimated number of rows:      98999.9
        Number of columns:              2
  
```

Subquery predicate ID: Not Applicable

Column Names:

+Q1.DEPT_ID(A)+Q1.NAME

通过 SORT(3)排序操作，预计 892 页发生了排序溢出。

SPILLED : (Pages spilled to bufferpool or disk)
892

DB2 优化器预测的排序行数是 99000 行，每行宽度是 32 个字节，所以需要的排序空间至少是 $99000 \times 32 / 1024 = 3093k = 3.1M$ 左右，而 SORTHEAP 设定的排序内存是 $250 \times 4K = 1M$ ，所以发生了排序溢出。

3) 排序溢出处理

发生排序溢出时，第一个要考虑的就是调整 SORTHEAP 参数，本例中我们将 SORTHEAP 参数值由 250 页(1M)增加到 2500 页(10M)。为便于对比，我们把以前的监控结果清零。

```
inst20@db2server:~> db2 update db cfg for sortdb using sortheap 2500
DB20000I The UPDATE DATABASE CONFIGURATION command completed successfully.
inst20@db2server:~>
inst20@db2server:~>
inst20@db2server:~> db2 reset monitor all
DB20000I The RESET MONITOR command completed successfully.
```

然后再次执行 SQL 语句，并观察结果。没有发现排序溢出，并且排序时间是 162ms，小于之前的 206ms。

```
[db2inst1@ibmswg01]$ db2 get snapshot for db on sortdb | more
Total Private Sort heap allocated      = 0
Total Shared Sort heap allocated       = 0
Shared Sort heap high water mark      = 2322
Post threshold sorts (shared memory)   = 0
Total sorts                           = 1
Total sort time (ms)                   = 162
Sort overflows                         = 0
Active sorts                           = 0
```

再次查看 SQL 语句的执行计划，

Access Plan:

```
-----
      Total Cost:          30201.9
      Query Degree:        1

      Rows
      RETURN
      ( 1)
```

```

Cost
I/O
|
98999.9
TBSCAN
( 2)
30201.9
29426
|
98999.9
SORT
( 3)
30201.9
29426
|
98999.9
TBSCAN
( 4)
30134.7
29426
|
999999
TABLE: DB2INST1
EMPLOYEE
Q1

```

3) SORT : (Sort)

```

Cumulative Total Cost:      30201.9
Cumulative CPU Cost:       2.55223e+09
Cumulative I/O Cost:       29426
Cumulative Re-Total Cost:   30134.6
Cumulative Re-CPU Cost:    2.34872e+09
Cumulative Re-I/O Cost:    0
Cumulative First Row Cost:  30201.9
Estimated Bufferpool Buffers: 29426

```

Arguments:

```

DUPLWARN: (Duplicates Warning flag)
          FALSE
NUMROWS : (Estimated number of rows)
          99000
ROWWIDTH: (Estimated width of rows)
          32
SORTKEY : (Sort Key column)
          1: Q1.DEPT_ID(A)
TEMPSIZE: (Temporary Table Page Size)
          4096
UNIQUE : (Uniqueness required flag)
          FALSE

```

```

Input Streams:
-----
      2) From Operator #4

      Estimated number of rows:      98999.9
      Number of columns:             2
      Subquery predicate ID:         Not Applicable

      Column Names:
      -----
      +Q1.NAME+Q1.DEPT_ID

Output Streams:
-----
      3) To Operator #2

      Estimated number of rows:      98999.9
      Number of columns:             2
      Subquery predicate ID:         Not Applicable

      Column Names:
      -----
      +Q1.DEPT_ID(A)+Q1.NAME

```

对比前面的访问计划，可以看到 TBSCAN (4)，SORT(3)和 TBSCAN (2) 这三个操作符的累计 I/O 次数一直保持不变，SORT(3)操作中没有溢出。整个语句的 Cost 由 36010 降为 30201.9。

查看 SQL 语句执行时间，由 7.6 秒将为 6.97 秒。

4) 消除排序

尽管调整参数后，排序溢出消失，但速度仍然较慢。接下来要做的是如何消除排序，采用的方法是创建索引。

DB2 提供了一个很有效的索引推荐工具，db2advis。

```
db2advis -d sortdb -i advisesql.txt -n db2inst1 -o newindex.ddl
```

从 newindex.ddl 可以看到，推荐了如下索引：

```

-- LIST OF RECOMMENDED INDEXES
-- =====
-- index[1], 41.149MB
CREATE INDEX "DB2INST1"."IDX_EMP_DEPTID" ON "DB2INST1"."EMPLOYEE"
("DEPT_ID" ASC, "NAME" DESC) ALLOW REVERSE SCANS ;
COMMIT WORK ;
RUNSTATS ON TABLE "DB2INST1"."EMPLOYEE" FOR INDEX "DB2INST1"." IDX_EMP_DEPTID " ;
COMMIT WORK ;

```

创建索引：

```
db2 -tvf newindex.ddl
```

再次执行 SQL 语句，观察生成的执行计划，这是一个非常的高效的索引只读计划，没有任何排序。整个语句的 Cost 也大幅降为 1147.64，SQL 语句的执行时间变为 3.9 秒。

Access Plan:

```
-----
      Total Cost:          1164.35
      Query Degree:        1

      Rows
      RETURN
      ( 1)
      Cost
      I/O
      |
      98999.9
      IXSCAN
      ( 2)
      1164.35
      1043.44
      |
      999999
      INDEX: DB2INST1
      IDX_EMP_DEPTID
      Q1
```

16.3 KPI

在性能专题的最后阶段，我们要讲的就是关键性能指标。就像前文中所提高的，关键性能指标在很多时候是一种快速定位“可能的瓶颈”的重要手段。当一个富有经验的 DBA 观察一个系统时，他们可以很快地判断出什么地方可能是出现问题的入手方向。他们凭借的是什么呢？直觉、方法，再加上早已经深深印在脑海中的一些性能指标。

本节我们将从数据库、实例和操作系统等方面介绍一些重要的 KPI，希望对大家有所帮助。

16.3.1 缓冲池命中率 (bufferpool hit ratio)

来源：数据库快照

公式：(逻辑读—物理读)/逻辑读

指标：优秀>95%，良好>80%

这个也许是人们最熟悉的 DB2 性能指标之一。当系统性能下降的时候，甚至非专职 DBA 也会跳出来问一句：“数据库缓冲区命中率怎么样？”

没错，命中率是判定物理 I/O 频繁程度的一个最重要的指标之一，而且在很多优化不足的系统是一个最容易出现的问题。

从内存中访问数据可能仅需纳秒级，而从磁盘访问数据则需数毫秒。DB2 对数据的获取是通过缓冲池，如果数据已经缓存到缓冲池，就可通过缓冲池直接获取，这称为缓冲池命中 (hit)；相反，如果数据不在缓冲池，则需要从磁盘读到缓冲池，这称为缓冲池未命中 (hit miss)。

因此，命中率越高，代表着读取同样的数据时需要的 I/O 越少，性能就越好，因此广大 DBA 不惜将大部分内存分配给缓冲池以提高命中率。

对于 OLTP,一般要求数据 bufferpool 达到最少 90%的命中率。对于 OLAP，由于经常需要进行表扫描，所以不必追求很高的命中率，但临时数据和索引的命中率需要关注，因为仓库系统中一些复杂的 SQL 语句需要进行大量排序或哈希关联操作，而排序和哈希关联可能需要在临时表空间内完成。

可以通过数据库快照获取缓冲池相关的信息，几种常见的命中率如下。

数据库整体缓冲池命中率：

```
(1 -(Buffer pool data physical reads + Buffer pool temporary data physical reads +
Buffer pool index physical reads + Buffer pool temporary index physical reads) /
(Buffer pool data logical reads + Buffer pool temporary data logical reads + Buffer
pool index logical reads + Buffer pool temporary index logical reads) ) * 100
```

数据缓冲池命中率：

```
(1 -(Buffer pool data physical reads ) / (Buffer pool data logical reads) ) * 100
```

索引缓冲池命中率：

```
(1 -(Buffer pool index physical reads ) / (Buffer pool data index reads) ) * 100
```

临时空间缓冲池命中率：

```
(1 -(Buffer pool temporary data physical reads + Buffer pool temporary index physical
reads) / (Buffer pool temporary data logical reads + Buffer pool temporary index
logical reads) ) * 100
```

有一个问题需要格外注意，缓冲池命中率可能会蒙蔽我们的眼睛，我们经常发现很多系统命中率很高，但仍然存在严重的性能问题。

举例来说，有一张表 T1，大小为 200M，包含 100 万行数据。该表对应的表空间缓冲池大小为 1G。在 T1 表有一条 SQL 语句，带过滤条件，但过滤字段上没有索引，满足此查询条件的行数只有 2 行。

由于没有索引，DB2 通过表扫描将 100 万行数据全都缓存到缓冲池，每次查询时 DB2 在缓冲池里进行表扫描。由于每次都是逻辑读，缓冲池命中率高达 100%。但为了查找 2 行数据，每次都要在内存中对 100 万行数据进行全表扫描，效率是非常低的。

感兴趣的读者，可以创建一张表，在里面插入 1 万条记录，然后执行 “select * from t1 where col2=1000” 10 次，观察快照的结果，会发现只有第 1 次访问时进行了物理读，其余的 9 次访问都是从缓冲池中直接获取。

这样，缓冲池命中率很高，而且随着访问频度的增加，命中率会更高。但是从逻辑上就可以推断出，所有的访问都是扫描全表，并且 100 万行中的绝大部分都是无用的数据，真正选择的只有两行。所以，我们想要提升性能，不能从减少 I/O 上入手，而是应该从提高读取效率上着手：

```
Buffer pool data logical reads      = 2782
Buffer pool data physical reads    = 274
Commit statements attempted        = 10
Rollback statements attempted      = 0
Rows selected                      = 10
Rows read                          = 100004
```

在 col2 上建索引，然后观察结果，发现缓冲池逻辑读大大降低：

```
db2inst1@dpf1:~> db2 "create index i1 on t1 (col2 )"
db2inst1@dpf1:~> db2 "runstats on table db2inst1.t1 with distribution and detailed
indexes all "
db2inst1@dpf1:~> db2 reset monitor all
db2inst1@dpf1:~> db2 "select * from t1 where col2=1000 "
连续执行 10 次
db2inst1@dpf1:~> db2 get snapshot for db on db1

Buffer pool data logical reads      = 22
Buffer pool data physical reads    = 5
Buffer pool index logical reads    = 33
Buffer pool index physical reads   = 6
Commit statements attempted        = 10
Rollback statements attempted      = 0
Rows selected                      = 10
Rows read                          = 14
```

因此，单纯的缓冲池命中率的分析并不能涵盖所有的性能问题。从某种意义上来说，命中率只能代表性能问题中很小的一部分，而且在某些情况下，该数值过高反而可能代表一些潜在的性能问题。

16.3.2 有效索引读 ■ ■ ■

来源：数据库快照

公式：行读取/行选择（rows read/rows selected）

指标: OLTP $\leq$ 5, SAP 系统 $\leq$ 3

这个指标就是我们刚刚在命中率部分讨论的, 到底有多少行是属于有效的读取。为了获取一行数据, DB2 需要验证或读取多少行?

如果 DB2 不能根据有效的索引过滤结果集, 那么可能需要扫描大量的数据页才能找到满足条件的数据。

rows read 是读的行数, rows selected 是返回的结果集, 公式如下:

```
IREF(index read efficiency) = Rows read / Rows Selected (Fetched)
```

假如一张员工表有 10 万行数据, 根据员工号查询员工信息, 假定员工号唯一并且没有索引, 那么 rows read 是 10 万行, rows selected 是一行, IREF=100000/1=100000, 即为了查找一行数据, 需要扫描 10 万行数据, 这个效率是非常低的。如果在员工号字段上有唯一性索引, 则只需读取一行数据就可获取值, IREF=1/1=1, 效率非常高。

在 DB2 9.1 以上版本中, 可通过如下语句实现:

```
db2inst1@dpf1:~> db2 "select Rows_read / (Rows_Selected+1) as IREF from
sysibmadm.snapdb"

IREF
-----
5
```

IREF 当然越小越好, 如果是 5, 意味着找一行数据需要读取 5 行数据。

对于 OLTP 数据库, IREF 应当小于等于 5, 如果远远大于这个数值, 则意味着需要调优。

对于 OLAP 仓库系统, IREF 一般相对较高。值越高, 则消耗的 CPU 资源越高, 性能越差, 因此对此值的监控仍然必要。当 IREF 比较高时, 需要对 SQL 语句进行分析, 找出 IREF 比较高的 SQL 语句, 然后进行物理设计, 比如设计优良的索引等, 降低 IREF。

16.3.3 包缓存命中率 (package cache hit ratio) ■ ■ ■

来源: 数据库快照

公式: 1-包缓存插入数量/包缓存读取数量 (1-Package Cache Insert/Package Cache Lookup)

指标: 1, 或者能够长时间保持接近 1 的稳定数值

该指标表示有多少查询语句可以直接在包缓存中找到。

当一条查询被请求的时候, 数据库在将其编译之前, 首先要从包缓存中查找有没有已经被编译好的包可以直接运行。如果该包已经存在, 那么该 SQL 可以直接被运行而不用再次编译(不过取决于 REOPT 等参数, 在《DB2 数据库高级管理》中的优化器部分讨论)。

因此，我们的目标是，如果应用程序在运行一段时间后，绝大部分的语句都已经被缓存在包缓存中，那么我们可以节省很多 SQL 编译的时间与 CPU 消耗。

不过，正像大家所考虑的一样，在有些系统中该值不可能固定在很高的数值。有些系统在设计之初，并没有对 SQL 使用绑定变量。因此，每一个使用不同参数的 SQL，譬如“select * from t1 where c1=1”与“select * from t1 where c1=2”会被当做两个完全不同的语句编译（这是由编译器设计决定的，具体的原因及如何在不修改应用的前提下绕过这个问题的方法请参考优化器部分的讨论）。在这种情况下，系统几乎没有两个查询是完全一样的，这样就会导致每一个查询都要向包缓存中插入编译后的包，而且当缓存满了以后会替换缓存中显存的包，导致性能的降低。

为了减少编译时间，建议在使用动态程序(如 java 等)开发时指定 prepare statement，这样就只需编译一次。使用静态程序(嵌入 C 等)时，建议使用绑定变量。

16.3.4 平均结果集大小 ■ ■ ■

来源：数据库快照

公式：行选择/执行 Select SQL 的次数（Rows selected / Select SQL statements executed）

指标：OLTP≤10

平均结果集用来表示平均每条 Select SQL 语句返回的结果行数。对于 OLTP 系统来说，结果集一般很小，通常小于 10；而对于 OLAP 系统，结果集一般很大。平均结果集的计算公式如下：

Avg_Result_Set = Rows selected / Select SQL statements executed

通过快照可以得出这两个元素的值。rows selected 表示返回的结果集，Select SQL statements executed 表示查询语句的执行次数：

```
db2inst1@dpf1:~> db2 get snapshot for database on db1
...
Select SQL statements executed          = 2311
...
Rows selected                          = 5531

DB2 9 以上的版本，可通过管理视图获得同样结果
db2inst1@dpf1:~> db2 "SELECT ROWS_SELECTED / (SELECT_SQL_STMTS+1) AS Avg_Result_Set
from sysibmadm.snapdb"

AVG_ROWS
-----
      2
1 record(s) selected.
```

通过平均结果集大小，可以判断数据库类型。假设某一个数据库是 OLTP 交易型系统，但

发现平均结果集很大，这就说明应用程序有改善余地，比如可在 SQL 语句上做查询条件过滤并降低返回的结果集大小，而不是将所有数据取出，然后在应用逻辑上过滤。

16.3.5 同步读取比例 ■ ■ ■

来源：数据库快照

公式：异步读取/逻辑读取 $100 \times (1 - \text{Asynchronous read} / \text{Logical read})$

指标：OLTP $\geq 90\%$

DB2 的数据读取 I/O 主要包括异步读和同步读。如果采用高效的索引获取结果集时，DB2 将使用同步读 I/O 访问索引页和需要的数据页，当没有索引或物理设计不够有效时，DB2 将采用异步读 I/O 扫描索引或数据页。异步 I/O 是 DB2 通过预取（prefetcher）进程/线程执行的，当查询的结果集较大，DB2 认为顺序预取更有效率时就会触发预取请求，异步预取 I/O 的比例越高，则表示获取的数据量越大，性能就可能越差，而更高的同步读则表示索引的高效。

通过快照监控结果可知，并没有任何元素表示同步 I/O 读的页数，但提供了异步 I/O 读和所有 I/O 读的值，用所有 I/O 读减去异步 I/O 值就可以计算出同步 I/O 读的值，这个 KPI 的公式如下：

$$\text{SRP} = 100 - (((\text{Asynchronous pool data page reads} + \text{Asynchronous pool index page reads}) \times 100) / (\text{Buffer pool data physical reads} + \text{Buffer pool index physical reads}))$$

这个公式不仅适用于整个数据库异步读监控，也适用于表空间和缓冲池对象。

在 DB2 9.1 以上版本中，可通过如下语句实现：

```
db2inst1@dpf1:~> db2 "select 100 - (((pool_async_data_reads + pool_async_index_reads)
* 100) / (pool_data_p_reads + pool_index_p_reads + 1)) as SRP from sysibmadm.snapdb
where DB_NAME = 'DB1' "
```

```
SRP
-----
      1
1 record(s) selected.
```

在 OLTP 系统中，如果同步读高于 90%，则表明数据库使用了高质量的同步 I/O 获取结果集。如果低于 50%，则系统性能一般会较差，或者是响应时间慢，或者 CPU 利用率比较高，一般是由于没有进行很好的物理设计引起的，比如缺乏有效的索引。

在 OLAP 系统中，往往需要进行数据扫描，并返回大量的结果集，这时可能采用异步预取 I/O 效率更高。

16.3.6 数据、索引页清除 ■ ■ ■

来源：数据库快照

公式：异步写入/总写入（async writes/total writes）

指标：≥95%

该指标代表着页面清除进程（线程）是否能够有效地将脏页在后台刷入磁盘。

由于缓冲池的大小是有限的，一般来说数据库不可能把所有的数据都放入内存。这时，哪些数据需要驻留内存；哪些被更新的数据需要被写入磁盘，然后留出空间给其他数据，就是 DB2 缓冲池管理模块需要决定的。

当缓冲池中的被修改的数据页（脏页）与缓冲池总大小的比例超过一定阈值的时候（chngpgs_thresh），DB2 就会触发后台的页面清除进程/线程，将被选择的页（victim pages）以异步方式物理写入磁盘。

但是如果该清除机制触发得不够频繁，或者缓冲池太小使得系统无法有效地找到一个干净的页面，DB2 就会选择一个脏页，然后将它写入磁盘，然后读取另外一个页面进入内存，这种写入方式叫做同步写入。

可以想象，相对异步写入，同步写入会对数据的读取造成很大的性能问题。因而，该指标的用途就是监测异步写入与总写入的比例：

```
/home/db2inst1/temp $ db2 get snapshot for database on sample | grep -i "write"
Buffer pool data writes                = 542
Asynchronous pool data page writes     = 323
Buffer pool index writes               = 395
Asynchronous pool index page writes    = 357
Buffer pool xda writes                 = 0
Asynchronous pool xda page writes      = 0
Total buffer pool write time (milliseconds)= 659
Total elapsed asynchronous write time  = 111
```

16.3.7 脏页偷取（dirty page steal） ■ ■ ■

来源：数据库快照

公式：脏页偷取触发次数（dirty page steal cleaner triggers）

指标：非常低

正像上文中刚提到的，脏页偷取是一种对性能影响极大的操作。当系统中的脏页偷取过多的时候，意味着 DBA 需要让页清除器工作得更加卖力。

那么如何做到这点呢？如果我们发现系统中的页清除器一直很空闲，则可以通过调节 `softmax` 与 `chnngpgs_thresh` 来让它们忙起来。这两个参数都是控制何时触发页清除器的参数，其中 `softmax` 是按照缓冲池中 `MinbuffLSN` 与当前 `LSN` 之间的差距来计算何时需要触发，而 `chnngpgs_thresh` 则是计算缓冲池中脏页的数量与可用页面总数来进行计算。两者的作用同样都是触发页清除器，只不过从不同的角度计算而已。

但是如果我们通过 `db2pd -stack all` 抓取的 `stack` 发现所有的页清除器一直非常繁忙，但是无论如何刷新磁盘的速度也赶不上数据写入缓冲区速度，这时就需要增加页清除器的数量了。

默认情况下 `chnngpgs_thresh` 为 60，即缓冲池 60% 的页为脏页时，会自动触发页清除器。

曾经欧洲的一家大型企业使用单个缓冲区高达 300GB，为了能够让系统更有效地将数据均匀地写入磁盘，`chnngpgs_thresh` 在调优之后被设定为 5%。

16.3.8 缓冲区读写 I/O 响应时间 ■ ■ ■

来源：数据库快照

公式：缓冲区读取时间/物理读取次数

指标：1~10 毫秒

对数据和索引的读取都要先通过磁盘读入缓冲区，我们曾经提过，快照中的缓冲区读取时间并不包括内存读取。也就是说，该数值仅仅包括物理读取的时间。

可通过以下公式计算出完成一个物理读需要的平均时间：

```
Overall Average Read Time(ms) = (Total buffer pool read time (milliseconds) / (Buffer pool data physical reads + Buffer pool index physical reads + Buffer pool temporary data physical reads + Buffer pool temporary index physical reads))
```

对于现代存储系统来说，平均物理读/写时间一般在 1~10 毫秒左右（取决于存储系统的性能与缓存）。因此，如果我们监控的指标值超出此值，则需要结合操作系统 I/O 监控工具，调查 I/O 系统的瓶颈：

```
db2 "select tbspace_name, (POOL_READ_TIME / (POOL_DATA_P_READS + POOL_INDEX_P_READS + POOL_TEMP_DATA_P_READS + POOL_TEMP_INDEX_P_READS + 1)) as TSORMS from sysibmadm.snaptb space order by TSORMS desc fetch first 10 rows only"
```

这个公式可找出读时间最慢的表空间，然后评估这个表空间的设计是否为最佳。

当发现该值过高时（譬如超过 20 毫秒），一般来说系统会明显感觉到性能下降。这时，为了能够直观地证明存储的性能问题，可以通过 UNIX/Linux 中的 `dd` 工具向容器所在的文件系统（而不是容器文件）写入几千个数据页，计算平均写入时间（该数值为顺序写入），然后可以使用同样的工具从容器文件中随机读取几千个数据页，计算平均读取时间。

```
db2 "select tbspc_name, (POOL_READ_TIME / (POOL_DATA_P_READS + POOL_INDEX_P_READS +
POOL_TEMP_DATA_P_READS + POOL_TEMP_INDEX_P_READS + 1) as TSORMS from
sysibmadm.snaptbpc order by TSORMS desc fetch first 10 rows only"
```

这个公式可找出读时间最慢的表空间，然后评估该表空间的设计是否最佳。

16.3.9 Direct I/O 时间 ■ ■ ■

来源：数据库快照

公式：直接读取（写入）时间/直接读取（写入）次数

指标：1~10 毫秒

Direct I/O 是指直接从磁盘访问而不经缓冲区的 I/O，主要针对 LONG/LOB 数据的访问。

当一列被定义为 LONG/LOB，那么该列的数据则不存储在表的数据页内。而在数据页记录中该列所对应的位置，则是一些“指针”指向该数据真实所在的位置。

很多时候，这些大对象的数据可以非常大（比如视频），因此，对于 LONG 与 LOB 数据类型，所有的读取都是直接 I/O，并不通过缓冲池。

```
DRIOMS - The average time (ms) required to complete a Direct Read
DWIOMS - The average time (ms) required to complete a Direct Write

DRIOMS = Direct reads elapsed time (ms) / Direct Reads
DWIOMS = Direct write elapsed time (ms) / Direct Writes
```

16.3.10 直接 I/O 读取（写入）的次数 ■ ■ ■

来源：数据库快照

公式：直接读取（写入）次数/缓冲区读取（写入）次数

指标：非常低

我们已经知道了什么是直接 I/O，而且我们知道了直接 I/O 需要物理读，而不能在缓冲池中保存数据。因此，在应用程序的设计中我们希望尽量少用 Direct I/O。

通过对比 Direct I/O 与普通缓冲池读写的比例，我们可以知道到底有多少数据访问是读取大数据的。

除了有限的特定应用外（比如在线视频网站），大部分的应用程序都应该尽量避免频繁的大数据访问。一般来说，如果一列数据可以被定义为普通列的话，就尽量不要使用大对象。

当一定要使用大对象定义某列的时候，在应用的某些操作中不是真正需要访问该列时，请尽量在查询时不要指定大对象数据列，避免不必要的 Direct I/O。

16.3.11 编目缓冲区插入比例 ■ ■ ■

来源：数据库快照

公式：编目缓冲区插入次数/编目缓冲区查询次数 (catalog cache inserts/catalog cache lookups)

指标：0，或者在接近 0 的数值上长时间维持稳定

在系统的每一个分区中，**database heap** 中都会分配出一块空间用于缓冲编目表的信息。

由于在数据库的日常操作中，查询编目表是一个非常频繁的操作。譬如当用户从一个表读取数据的时候，系统就要查询编目表，理解该表在什么表空间、应该如何访问等信息。因此为了性能着想，DB2 在数据库栈内存中单独开辟了一块空间，用于存储编目信息。

但是如果用户有很多数据库对象，而该编目缓存的大小过小，则该内存无法容纳下所有的信息。那么当新的信息来临时，就会把一些不常用的信息替换出去。

不过可以想象，如果该替换经常发生，那么每次当系统想要查找编目数据时，就要从编目表空间中查找，这样会导致系统性能一定程度上的下降。

因此，一般建议将 **catalogcache_sz** 设置逐渐增大，直到系统中不再频繁出现编目缓冲区插入的操作。

16.3.12 排序指标 ■ ■ ■

1. 排序溢出比例

来源：数据库快照

公式：排序溢出次数/排序总数 (sort overflow/total sorts)

指标：OLTP：0，或者在接近 0 的数值上长时间维持稳定

前面我们已经详细讨论了数据库的排序，因此原理在这里不再赘述。

排序溢出就是当排序内存不够时，数据需要使用临时空间进行排序。

一般来说，我们希望数据尽可能在内存中完成。当我们发现大量的排序溢出时，就要看排序堆内存参数设置是否足够大。很多时候，排序的根源是由于 SQL 语句引起的。

2. 平均排序时间

来源：数据库快照

公式：排序总时间/排序总数 (total sort time/total sorts)

指标：远小于系统预期平均语句的执行时间

这个指标是理解平均每次排序所消耗的时间。

一般来说，一条语句的执行时间包括锁等待、数据读取时间（内存+I/O）加上排序时间（内存读取和内存排序都属于 **user CPU** 时间）。因此用数据库中平均的排序时间对比平均语句返回的速度，就可以大概估算出执行一条语句时有多少的时间用于排序。

作为 **DBA**，我们一般都希望该排序时间越少越好。如果该值过高，用户可以考虑优化查询与添加索引，尽量减少排序的消耗。

3. 平均每条交易的排序次数

来源：数据库快照

公式：排序总数/交易总数（ $\text{total sorts}/(\text{Commit statements attempted} + \text{Rollback statements attempted})$ ）

指标：OLTP<5

对于 **OLTP** 应用来说，由于每条交易的短小精干的特性，我们需要尽可能减少每条交易所需的排序数量。

大部分情况下，这种调优需要应用开发人员的配合，一方面在数据库中建立合适索引的同时，另一方面优化应用程序逻辑，减少 **SQL** 所需要排序的次数。在典型的 **OLTP** 系统中，尽量将每一条交易平均所需的排序数量维持在 5 以下。

16.3.13 基于事务的指标度量 ■ ■ ■

通过监控事务负载，我们可以了解业务的特点，比如每个事务包含多少条 **select** 语句、多少 **DML** 语句（**insert**、**update**、**delete**），每个事务需要的排序次数、每个事务需要读取的行数、每个事务返回的行数、每个事务需要的逻辑 **bufferpool I/O**。如果能够把每个事务的 **cost** 降低，那将会对性能有很大个改善。让我们一起来看下这些与事务相关的一些度量指标。

1. 总的事务数量

来源：数据库快照

公式：提交事务 + 回滚事务（ $\text{Commit statements attempted} + \text{Rollback statements attempted}$ ）

指标：无

前面我们已经介绍了事务的概念，事务是一个完整的工作单元，具有 **ACID** 特性。该指标可以直接反映出系统单位时间内的吞吐量高低。

可以通过以下公式计算数据库总的事务数量：

```
Total Number of Transactions (TXCNT): TXCNT = Commit statements attempted + Rollback
statements attempted
db2inst1@dpf1:~> db2 "select COMMIT_SQL_STMTS + ROLLBACK_SQL_STMTS as txn_counts
from sysibmadm.snapdb"
TXN_COUNTS
-----
5149
1 record(s) selected.
```

2. 每个事务包含的查询 SQL 语句数量

来源：数据库快照

公式：查询语句数量/交易总数 (Select SQL statements executed / (Commit statements attempted + Rollback statements attempted))

指标：OLTP ≤ 10

对于 OLTP 系统来说，每个事务执行的查询次数一般小于 10。如果事务太长，可能会造成一些锁等，影响并发性能。

```
db2inst1@dpf1:~> db2 "select decimal(SELECT_SQL_STMTS)/decimal(COMMIT_SQL_STMTS +
ROLLBACK_SQL_STMTS+1) as selpertxn from sysibmadm.snapdb "
SELPERTXN
-----
2.94
```

3. 每个事务包含的增删改语句数量

来源：数据库快照

公式：插入、更新与删除语句的数量/交易总数 (Update/Insert/Delete statements executed/(Commit statements attempted+Rollback statements attempted))

指标：OLTP ≤ 5

在 OLTP 系统中建议不要在同一条交易中使用过多的数据更改语句（插入、更新、删除），单条交易中过多的这类语句会造成大量的锁，容易引起锁等待甚至死锁，同时过长的交易可能会引起活动日志过长，导致日志空间占满。

```
db2inst1@dpf1:~> db2 "select decimal(UID_SQL_STMTS)/decimal(COMMIT_SQL_STMTS +
ROLLBACK_SQL_STMTS+1) as uidpertxn from sysibmadm.snapdb "
UIDPERTXN
-----
5.58
```

4. 每个事务返回的结果集行数

来源：数据库快照

公式：选择的行数/事务总数（Rows Selected/(Commit statements attempted + Rollback statements attempted)）

指标：针对 OLTP 业务，尽量靠近最终用户真正在终端上看到的結果集的数量

这个指标用来计算每个事务返回的平均结果集。这个值尽管无法控制，但如果发现该指标异常高，表明应用程序获取大量数据，可能的原因是在应用层进行数据过滤，可以考虑在数据库层增加过滤条件，减少返回到应用层的结果集。

```
db2inst1@dpf1:~> db2 "select decimal(rows_selected)/decimal(COMMIT_SQL_STMTS +
ROLLBACK_SQL_STMTS+1) as rowsselpertxn from sysibmadm.snapdb "
ROWSPERTXN
-----
17.11
```

5. 每个事务读的行数

来源：数据库快照

公式：读取的行数/事务总数（Rows Read/(Commit statements attempted + Rollback statements attempted)）

指标：OLTP 尽量靠拢前一个指标

该指标与上一个指标不同。在获取结果集的过程中，DB2 必须读取数据页，因此要读取的数据行数（rows_read）可能远远大于结果集（rows_selected）。

通过优化物理设计，比如设计好的索引，DBA 可以提高结果集与读取行数的比例。如果缺乏好的索引，DB2 必须读更多的行才能找到满足条件的数据。当应用了索引后，记得重新计算和评估这两个指标，验证读取的行数是否降低。

```
db2inst1@dpf1:~> db2 "select decimal(rows_read)/decimal(COMMIT_SQL_STMTS +
ROLLBACK_SQL_STMTS+1) as rowsreadpertxn from sysibmadm.snapdb "
ROWSREADPERTXN
-----
52.60
```

6. 每个事务需要的缓冲区逻辑读

来源：数据库快照

公式：逻辑读的总数/事务总数（(Buffer pool data logical reads + Buffer pool index logical reads)/(Commit statements attempted + Rollback statements attempted)）

指标：取决于业务量，在 OLTP 中尽量降低

DB2 对数据页和索引页的读取是通过缓冲区执行的。如果请求的数据不在 bufferpool 中，则首先将磁盘数据读到缓冲区。如果数据已经在缓冲区，则直接从缓冲区中获取，我们把这种

数据读叫做逻辑读。

那么大量的逻辑读为什么会对性能有影响呢？这是因为大量的逻辑读一般代表的是大量的数据扫描，通常是由于缺乏良好的物理设计造成的，所以尽管从缓冲区直接读取，但会消耗大量的 CPU 资源。

逻辑读和 CPU 资源消耗有很大关系，一个事务中执行的逻辑读越多，消耗的 CPU 资源就越多。

牢记一点：逻辑读与 CPU 消耗成正比。

为了降低该指标，需要找到那些开销高的 SQL 语句，比如 CPU 时间、排序时间、I/O 时间、读取的行数、写行数等。一旦找到，可以通过物理设计或对 SQL 语句进行调优的方法降低指数。

降低数据库整体逻辑 I/O 可从 SQL 语句着手，通过 `get snapshot for dynamic sql` 可监控每个 SQL 语句的逻辑读，然后对语句进行分析。也可以通过 `event monitor for statements` 进行监控。

16.3.14 检测索引页扫描 ■ ■ ■

来源：数据库快照

公式：逻辑索引读的总数/事务总数（`Buffer pool index logical reads / (Commit statements attempted + Rollback statements attempted)`）

指标：取决于业务量，在 OLTP 中尽量降低

The number of Bufferpool Index logical read per Transaction (BPLITX) : $BPLITX = \text{Buffer pool index logical reads} / \text{TXCNT}$

索引是 B+ 结构，包含根节点，中间节点和叶子节点，数据存在叶子节点，根节点和中间节点提供了遍历的路径。对索引的访问一般有两种遍历方式：一种是从 root 页开始读，然后访问中间节点，最后访问指向数据页 RID 的叶子节点。另外一种是不通过跟节点和中间节点，而直接遍历叶子节点获取需要的数据。

第一种方式是理想情况，一般每条 SQL 语句平均需要访问 3-4 个索引页。而第二种方式可能需要更多的索引页访问，需要消耗大量的 I/O 和 CPU 时间。

什么情况下会使用第二种访问方式呢？比如，在表 T1 (A,B) 列上建一索引，如果根据 B 字段执行查询: `select * from T1 where B='aaa'`，这时 DB2 优化器可能会通过索引扫描，遍历索引的叶子节点，查找满足 B 查询条件的值（一般情况下这种访问计划不会发生，因为大部分时候直接扫描表可能会更加有效，但是在某些特殊的设置中，如果优化器判定扫描所有索引节点更加有效的话，优化器可能会选择索引扫描）。

除了查询语句，更新和删除语句也应该充分利用索引。假如一个事务包含 10 个查询, 2 个修改操作，那么理想的索引页扫描数应该是：

$(10+2) * 4 * 1.5 = 72$ (4 表示 4 个索引页, 1.5 考虑了额外的开销)

如果实际环境中该指标值过大, 很可能是 DB2 在扫描索引页。

```
db2inst1@dpf1:~> db2 "select POOL_INDEX_L_READS/(COMMIT_SQL_STMTS+ROLLBACK_SQL_STMTS+1)
as BPLITX from sysibmadm.snapdb"

BPLITX
-----
      7

1 record(s) selected.
```

16.3.15 日志写入速度 ■ ■ ■

来源: 数据库快照

公式: 日志写时间/日志写次数 (Log write time (sec.ns)/Number write log IOs)

指标: <3 毫秒

日志写入的速度有时会对经常进行提交的应用程序性能产生决定性的影响。

在 mincommit 为 1 的情况下, 每一次的提交都会伴随着物理日志的写入。在一个频繁提交的系统中, 如果物理日志的写入速度过低, 会对性能产生非常大的影响。

在监测中, 可以使用数据库快照中 Log write time (sec.ns) 与 Number write log IOs 判断平均每个写日志 I/O 的时间消耗。如果发现该值超过 5 毫秒, 则说明该部分有待提高。

Log write time (sec.ns)	= 3823.0000000004
Number write log IOs	= 1184311

16.3.16 查询执行速度 ■ ■ ■

来源: SQL 快照

公式: 总执行时间/总执行数量 (Total execution time (sec.microsec) / Number of executions)

指标: OLTP 中 <1 毫秒

该指标代表给定查询的平均执行时间。对于典型的 OLTP 应用程序, 一般单条查询的时间在 1 毫秒以内。即使逻辑相对复杂, 也应该保持在 10 毫秒之内。对于运行时间超出 10 毫秒的纯 OLTP 查询, 需要进一步研究该 SQL 的访问计划并提升性能。

前面介绍一些常用的有代表性的数据库级别性能指标。当然, 还有很多语句级性能指标,

由于与应用程序与业务逻辑联系过于紧密，我们无法给出预期指标（譬如查询的每次执行所花费的 CPU 时间）。请读者在进行指标监控时根据自身的业务特点和平日间的数据收集，制定出适用于该系统的性能指标。

16.3.17 实例级性能指标 ■ ■ ■

1. 排序内存过量使用

来源：实例快照

公式：无

指标：排序内存高水位 < SHEAPTHRES

当排序内存超出实例排序内存的软上限后，系统会从溢出内存池分配一些内存，为请求的排序操作服务。

从数据库的可用性来说，这种现象是完全正常的。但是，如果系统经常需要进行大量的排序，实例排序内存软上限在每次大规模的排序请求时都会被超过，那么用户可以考虑提高软上限，避免系统一次又一次无意义地分配与释放内存。

2. 实例内存使用

（注：在 9.5 及以后的版本中，实例内存代表着该实例该分区所占用的全部内存。在 9.1 及以前的版本中，我们这里讨论的实例内存为实例内存+数据库内存+应用程序组内存）

来源：db2pd -dbptnmem

公式：当前使用 / 系统总物理内存

指标：在只运行一个实例单分区，并且没有其他应用程序运行的系统中≤90%

当实例的内存没有被妥善配置时，如果所提交使用的内存超出了系统物理内存的限制，则会引起换页。因而，为了确保换页不会发生，需要为操作系统内核预留出部分内存，一般我们将一个实例的总内存上限设置为物理内存的 80%~90%。

16.3.18 操作系统级指标 ■ ■ ■

1. LTG（Logical Track Group）大小

来源：lquerypv -M hdiskn

公式：无

指标：最大硬件可以支持的 LTG

当 LVM 接收到一个 I/O 请求时，系统将会把传入的 I/O 请求切分成一段一段的，每一段的大小就是这个 LTG，然后将被切分的请求传入磁盘。

在 AIX 5.2 中，内核所支持的 LTG 可以为 128KB、256KB、512KB 与 1024KB，而很多存储设备或磁盘则有自己的限额，对于现代的存储设备，大部分都可以支持超过 1MB 的 LTG。因此在 AIX 5.3 中，内核所支持的 LTG 被调整为最大 16MB。

为了提升 I/O 性能，一般建议将 LTG 调整为硬件所支持的最大尺寸：

```
varyonvg -M512K tmpvg
```

2. CPU 内核 / 磁盘比例

来源：操作系统与存储硬件信息

公式：存储系统中总物理磁盘数量 / 内核数量

指标：介于 6~20 之间

我们知道，很多系统中 I/O 都是制约性能的瓶颈。因此，将数据均匀地分布在多块物理磁盘中，能够有效地加快数据并行读取的性能。

一般来说，对于现代的服务器系统，很少有用户使用本地硬盘存储所有数据。很多时候数据存储在 SAN 或者 RAID 中，而存储设备则包含多个物理磁盘。

在系统的设计之初，一般的建议是一个 CPU 内核有 6~20 块物理硬盘相对应。过少的物理磁盘会对性能造成负面影响。

3. CPU 内核/内存比例

来源：db2pd -osinfo

公式：总内存/总 CPU (TotalMem/TotalCPU)

指标：4~8GB

在通常情况下，当一台系统中包含多个 CPU 内核的时候，每个 CPU 内核对应的内存数量应当在 4~8GB 之间。

过小的内存意味着数据库不能将足够多的数据放入内存，这样将会导致数据在内存和磁盘之间的频繁交换，致使 CPU 的处理能力不能够完全发挥。

同理，过大的内存意味着会有很多的数据需要处理，而有限的 CPU 数量会使 CPU 的负担加重。尽管在抛开业务逻辑后，内存的大小和 CPU 运算能力之间没有一个直接的运算公式，但是对经过了处理大量的性能负载问题分析后，关键性能指标建议每个 CPU 内核对应 4~8GB 内存比较合理。

16.4 小结



本章我们首先介绍了 DB2 优化器原理，然后通过实例介绍了性能调优的思路和方法。接着重点介绍了索引的使用和最佳实践，以及排序的根源及减少排序的思路和方法。

最后是一些 KPI 的介绍，KPI 是一些国内外性能分析专家总结的一些通用指标，目的是为了快速地判断系统是否存在异常。

16.5 判断题



(1) DB2 在进行访问计划的开销估算时，完全基于数据库对象的统计信息。

T: 正确

F: 错误

(2) 收集到当前的统计信息一定能够保证访问计划是最优的。

T: 正确

F: 错误

(3) DB2 支持 B+树索引与 Bitmap 索引。

T: 正确

F: 错误

(4) 语句中的排序一定可以通过增加合适的索引进行消除。

T: 正确

F: 错误

(5) KPI 适用于所有场合。

T: 正确

F: 错误

第五部分 DB2 问题诊断及安全



问题诊断

一般来说，系统的运维早已经有了成熟的脚本和软件，每天并不需要 DBA 手工运行无数的命令。在 DBA 的日常工作中，出问题的概率也不会问题很高，但是我们绝对不要小看问题诊断能力在工作中的重要性。因为问题往往是突发性的，如果不能在出现问题时快速诊断和解决，将会给生产造成重要影响。通常情况下，问题诊断和性能调优的能力高低，才能真正反映一个 DBA 的水平。

我们在上几章中已经介绍了性能监控与调优，本章不再重复。而锁相关问题则自成体系，已经在相关章节中进行了探讨。在这一章中，我们主要讨论几种典型的问题，以及对应的分析思路。

本章内容安排如下：

- 概述。
- 诊断日志分析。
- 宕机。
- 挂起。
- 错误信息分析。
- 分析数据收集工具。

17.1 概述



在列出问题类型之前，我们必须明确，世界上没有任何一本书能够告诉读者如何对所有类型的问题“一步一步”地进行问题诊断。本身问题诊断便是当系统发生意外的情况下如何处理，

如果有任何人能够把所有可能出现的“意外”都想到的话，那么意外便不是真正的“意外”了。

因此，在问题诊断的过程中，用户必须依靠自己的判断和直觉，通过分析各种相关的日志和数据，一步一步地接近问题的根源。

之所以我们要讨论几种不同类型的问题，是因为对于不同类型的问题，所使用的分析思路有着很大的区别。在本章中，我们主要讨论下列几个类型的问题：日志信息（应用程序与实例没有出错现象，但是日志中反映出一些错误）、宕机（实例 **crash**）、挂起（**hang**，在性能分析章节中讨论过，这里进一步说明）及错误信息（一些命令返回错误信息）。

17.2 日志信息错误

db2diag.log 文件是 DBA 的好朋友，DB2 会将大部分的错误信息都记录到该诊断日志文件中。

对于日志信息错误，从用户的角度看，数据库与应用程序一切运行良好（至少还能够工作，没有崩溃或者停止），但在日志中显示了一些与错误相关的信息。

在遇到这类问题时，我们第一个要问的问题就是，这个错误信息是否真的代表什么地方发生了错误，还是说 DB2 只是显示一些比较重要的不容忽视的正常信息。

例如下面的信息：

```
2011-02-18-13.30.42.616890-300 E3077A415      LEVEL: Warning
PID      : 47302258      TID : 1      PROC : db2agent (SAMPLE)
INSTANCE: db2inst1      NODE : 000      DB   : SAMPLE
APPHDL   : 0-409      APPID: *LOCAL.db2inst1.110218183018
AUTHID   : DB2INST1
FUNCTION: DB2 UDB, base sys utilities, sqliteDatabaseQuiesce, probe:1
MESSAGE  : ADM7506W Database quiesce has been requested.

2011-02-18-13.30.42.629345-300 E3493A431      LEVEL: Warning
PID      : 47302258      TID : 1      PROC : db2agent (SAMPLE)
INSTANCE: db2inst1      NODE : 000      DB   : SAMPLE
APPHDL   : 0-409      APPID: *LOCAL.db2inst1.110218183018
AUTHID   : DB2INST1
FUNCTION: DB2 UDB, base sys utilities, sqliteDatabaseQuiesce, probe:2
MESSAGE  : ADM7507W Database quiesce request has completed successfully.
```

这部分信息的级别是警告，但是是否意味着系统中真的出现了问题呢？答案是否定的。这条信息会在用户发出 **quiesce** 命令时显示，也就是用户手工挂起了这个数据库。

一般来说，在 DB2 中值得注意的错误类型是 **Severe** 和 **Error**。**Warning** 信息一般是指用户不应当忽略，或者当问题发生时用户需要查看的重要信息，而不是指系统真正存在问题，但是要注意，有一些警告信息本身可能无关痛痒，但是和一些其他信息结合在一起时就会意味着一些

问题。对于 Info 则是可以忽略，一般用来显示一些调错时需要的重要信息。

在阅读日志文件的时候，首先找到自己所关心的信息，然后根据每一条记录的时间戳与进程和线程 ID 向上翻阅日志得到该进程（线程）在这个事件中（相近的时间戳）第一个报告的问题。当我们找到了一个事件中问题起始的时间，我们就可以开始理解该条日志在说什么了。

对于一条 db2diag.log 日志，第一行主要包含日期与记录级别，第二行是进程 ID、线程 ID 和进程名，第三行为实例名和节点 ID。如果该记录对应某一特定应用，则第四行是应用程序句柄和 ID，第五行为其授权 ID；否则第四行为函数名和记录点。接下来是该记录的信息。

我们来尝试分析一下下面这条记录中的信息：

```
2010-12-10-02.00.19.631828-300 E102299A401      LEVEL: Info
PID      : 1028606          TID   : 1          PROC  : db2agent (SAMPLE) 0
INSTANCE: db2inst1        NODE   : 000          DB    : SAMPLE
APPHDL   : 0-3033          APPID: *LOCAL.db2inst1.101210070006
AUTHID   : DB2INST1
FUNCTION: DB2 UDB, database utilities, sqlubSetupJobControl, probe:1410
MESSAGE  : Starting an online db backup.
```

（1）首先，这条信息产生于 2010-12-10-02.00.19.631828，类型为“信息”，也就是不代表任何错误。

（2）进程 ID 为 1028606，线程 ID 为 1，进程名为 db2agent，数据库名（括号中的）为 SAMPLE。

（3）实例为 db2inst1，节点为 0，数据库名为 SAMPLE。

（4）应用程序句柄为 0-3033，ID 为 LOCAL.db2inst1.101210070006，也就是说该应用为本地图应用程序，启动实例为 db2inst1，连接时间为 2010 年 12 月 10 号 07 点 00 分 06 秒。连接授权用户为 DB2INST1。

（5）产生该记录的 DB2 函数名为 sqlubSetupJobControl，记录点为 1410。这部分信息主要是 IBM 的 DB2 支持专家用来确认问题发生的位置，对于用户来说意义不大。简单地说，DB2 中大部分函数都以 sql 起始，然后后面的一至两个字母代表着该函数在 DB2 中所属的模块，紧接着就是以大写字母开始的函数名称。而记录点则是每个函数中的唯一标识，用来帮助 IBM 的技术支持专家确定问题的出处。

（6）具体信息为在线数据库备份开始。

因此，我们通过以上的分析，可以看出，数据库 SAMPLE 在节点 0 中，曾经在 2010 年 12 月 10 号 07 点 00 分 06 秒由用户 DB2INST1 发起了在线数据库备份。

然后我们来看接下来的记录：

```
2010-12-10-02.29.22.307082-300 I102701A584      LEVEL: Error
PID      : 1274298          TID   : 1          PROC  : db2med.1028606.9 0
INSTANCE: db2inst1        NODE   : 000
```

```

FUNCTION: DB2 UDB, oper system services, sqlowrite, probe:200
MESSAGE : ZRC=0x850F000C=-2062614516=SQLO_DISK "Disk full."
          DIA8312C Disk was full.
DATA #1 : File handle, PD_TYPE_SQO_FILE_HDL, 8 bytes
0x0FFFFFFFFFFFFD5D0 : 0000 0003 0000 0080      ...
DATA #2 : unsigned integer, 8 bytes
1048576
DATA #3 : signed integer, 8 bytes
98304
DATA #4 : signed integer, 4 bytes
35

```

这条记录在 db2diag.log 中紧接着上一条。我们来分析这条错误信息的含义。该错误发生在 2010-12-10-02.29.22.307082, 也就是数据库在线备份开始的 29 分 03 秒之后, 进程 ID 为 1274298, 线程为 1, 而进程名则是 db2med.1028606.9, 分区为 0。

从进程名中我们可以看出, 该 db2med 是服务于进程 1028606 的, 而 1028686 则是我们刚才的那个发出在线备份开始信息的进程, 因此我们可以推断出, 该错误是在线备份所产生的。

紧接着, 我们能够看到, 写出该信息的 DB2 函数为 sqlowrite, 记录点 200, 信息为 SQLO_DISK, 即磁盘已满。而下面的具体的数据部分则没有具体信息, 一般来说是由 IBM 技术支持专家解读。这部分数据需要对照 DB2 源代码, 才能够了解每段数据所代表的具体含义, 因此不用深究。

那么从用户的角度, 我们可以知道, 在在线备份开始 29 分钟之后, db2med 进程报告在线备份磁盘空间满。

然后再继续向下看, 这时我们发现日志中还有很多其他 db2med 进程的信息 (每一个备份会启动若干个 db2med 进程), 因此如果我们想要知道 db2med.1028606.9 的信息, 则需要跟踪进程 1274298:

```

2010-12-10-02.29.22.488199-300 I132164A353          LEVEL: Warning
PID      : 1274298          TID : 1          PROC : db2med.1028606.9 0
INSTANCE: db2inst1          NODE : 000
FUNCTION: DB2 UDB, database utilities, sqluMCWriteToDevice, probe:956
MESSAGE : Media controller -- Disk full encountered on
          /udb/db2inst1/SAMPLE/udb10/backup

```

这条信息告诉我们造成磁盘满的路径, 即/udb/db2inst1/SAMPLE/udb10/backup。而对于这个错误, 用户需要做的就是增加该磁盘路径的空间。

17.3 宕机

宕机是一类比较严重的问题, 意味着 DB2 服务的意外中断。可能很多人下意识地认为, 宕机

就一定是 DB2 的 bug。这种说法也对也不对。从某种意义上说，DB2 应该尽量保证服务的不中断运行，理论上应该能够检测到不同类型的错误并给出相应的错误信息，而不是意外停止服务。

但是，对于一个数据库系统来说，比保障系统不中断运行更加重要的事情就是保证数据的完整性。如果任何操作可能会造成数据的损坏，当 DB2 无法有效地处理该问题，那么 DB2 就会在牺牲连续性的基础上，强行中断数据库服务，以保障数据的合法与完整性。

而另一种宕机则是非法内存访问。如果大家写过 C 的程序，都会明白当尝试访问一个非法内存地址时系统会发生什么。没错，就是信号 11 SIGSEGV，代表应用程序尝试访问一块没有被映射在进程空间中的内存。

一般来说，我们可以认为，第一种宕机可能是由于人为或者物理介质的损坏造成 DB2 主动停止服务；而第二种宕机则在大部分情况下都是 bug 所致，导致 DB2 内部处理时发生非法内存访问，以至于进程崩溃。

那么，在这一节，我们给出两个例子来说明这两种宕机情况。

1. 非法内存访问

在这类问题中，如果用户发现系统无缘无故地宕机，应该在 db2diag.log 中先搜索字符串 sigsegv。结果应该返回类似如下的信息：

```
2011-01-19-17.54.09.607488+000 I45688A517          LEVEL: Error
PID      : 1781762          TID : 1          PROC : db2agent (SAMPLE) 0
INSTANCE: db2inst1        NODE : 000          DB   : SAMPLE
APPHDL   : 0-420          APPID: 9.63.48.235.51598.110119175407
AUTHID   : SAMPLE_C
FUNCTION: DB2 UDB, base sys utilities, sqleagnt_sigsegvh, probe:1
MESSAGE  : Error in agent servicing application with coor_node:
DATA #1 : Hexdump, 2 bytes
0x0FFFFFFFFFFFF3E50 : 0000          ..
```

用刚才讨论过的方法分析该数据，可以看到数据库 SAMPLE 在 2011-01-19-17.54.09.607488 产生 sqleagnt_sigsegvh。

下一步我们要做的就是进入 db2dump 目录，找到所产生的 trap 文件。在 9.5/9.7 中，对应的文件在一个叫做 FODC 的目录中，可以在 db2dump 下搜索 *FODC*，将目录的时间戳和 db2diag.log 信息的时间对应，而 trap 文件则是包含 trap 字符串的文件名，搜索 *trap*.txt 可以找到。对于 8/9.1 来说，trap 文件在 db2dump 目录下，起始字母为 t，紧接着进程 ID：

```
U:\>dir t1781762*
Volume in drive U is sample
Volume Serial Number is 2B34-090C

Directory of U:

01/19/2011  05:54 PM          47,724  t1781762.000
              1 File(s)          47,724 bytes
```

```
0 Dir(s) 511,223,988,224 bytes free
```

在该文件中，查找字符串 **Signal #**，会出现类似下面的结果：

```
</Siginfo_t>
Signal #4 (SIGILL): si_addr is 0x0000000000000000, si_code is 0x0000001E
(ILL_ILLOPC:Illegal opcode.)
</SignalDetails>
```

一般来说，**Signal #4**、**#10** 或者 **#11** 都代表非法内存访问；而如果是 **#36**、**#21** 之类的，则说明 DB2 主动写入的信息。

对于非法内存访问的问题，从用户角度没有太多可以做的，需要做的就是尽快联系 IBM DB2 技术支持小组进行分析，同时笔者建议用户尽量使用最新的补丁包，避免遇到已经被修复的问题。

2. 数据损坏（包括数据文件与日志文件）

当数据损坏发生的时候，如果 DB2 检测到某一个数据页包含不合法的数据，或者所需要的日志文件不存在，那么 DB2 会主动停止数据库服务，以避免进一步的数据损坏。

用户必须区分该类型的宕机与内存非法访问类型的宕机。数据损坏或日志文件破坏所造成的宕机，大部分情况下都是由于存储问题或者人为原因所造成的。

对于数据页损坏的问题，类似下面的日志信息会在 **db2diag.log** 日志中显示：

```
2010-10-12-03.28.49.612975-300 I353077A2547 LEVEL: Severe
PID : 19575 TID : 1 PROC : db2agent (SAMPLE)
0
INSTANCE: db2inst1 NODE : 000 DB : ODC
APPHDL : 0-822 APPID: GAEB627.A53D.101012070528
AUTHID : DB2INST1
FUNCTION: DB2 UDB, buffer pool services, sqlb_verify_page, probe:3
MESSAGE : ZRC=0x86020001=-2046689279=SQLB_BADP "page is bad"
DIA8400C A bad page was encountered.
DATA #1 : String, 64 bytes
Error encountered trying to read a page - information follows :
DATA #2 : String, 23 bytes
Page verification error
DATA #3 : Page ID, PD_TYPE_SQLB_PAGE_ID, 4 bytes
301339
DATA #4 : Object descriptor, PD_TYPE_SQLB_OBJECT_DESC, 72 bytes
Obj: {pool:4;obj:149;type:1} Parent={3;66}
lifeLSN: 0000F4477218
tid: 0 0 0
extentAnchor: 28320
initEmpPages: 0
poolPage0: 28336
poolflags: 102
objectState: 27
lastSMP: 0
```

```

pageSize:                8192
extentSize:               16
bufferPoolID:             3
partialHash:              26542084
bufferPool: 0x0000000223919b00
DATA #5 : Bitmask, 4 bytes
0x00000002
DATA #6 : Page header, PD_TYPE_SQLB_PAGE_HEAD, 48 bytes
pageHead: {pool:29723;obj:0;type:0} PPNum:5 OPNum:240
  begoff:                 0
  datlen:                 0
  pagebinx:               1
  revnum:                 0
  pagelsn: 0300031EADB0  flag: 0
  signature:              0
  cbits1to31:             abcf04c4
  cbits32to63:            0
CALLSTCK:
[0] 0xFFFFFFFF7A8CAF48 __1cZsqlbLogReadAttemptFailure6FIpnJSQdDLB_OBJECT_DESC
IpnJSQdDLB_PAGE_ibLIpcpnMSQdDLB_GLOBALS__v_ + 0x150
[1] 0xFFFFFFFF7A8CEEC8 __1cQsqlb_verify_page6FpnJSQdDLB_PAGE_pnJSQdDLB_OBJECT
DESC_IIPnMSQdDLB_GLOBALS_pL_i_ + 0x498
[2] 0xFFFFFFFF7A8CC400 sqlbReadPage + 0xDD8
[3] 0xFFFFFFFF7A8ABB4 __1cTsqlbGetPageFromDisk6FpnLSQdDLB_FIX_CB_i_i_ + 0x2EC
[4] 0xFFFFFFFF7A8A5BD8 __1cHsqlbfix6FpnLSQdDLB_FIX_CB__i_ + 0x9A0
[5] 0xFFFFFFFF7C3DE184 __1cHsqlifix6FpnHSQdDLI_CB_pnOSQdDLI_PAGE_DESC_Ii_i_ +
0x64
[6] 0xFFFFFFFF7C41F78C __1cIsqlischa6FpnHSQdDLI_CB_pnLSQdDLI_SAGLOB_iI_i_ +
0x30AC
[7] 0xFFFFFFFF7C41E6B4 __1cIsqlischa6FpnHSQdDLI_CB_pnLSQdDLI_SAGLOB_iI_i_ +
0x1FD4
[8] 0xFFFFFFFF7C41DA30 __1cIsqlischa6FpnHSQdDLI_CB_pnLSQdDLI_SAGLOB_iI_i_ +
0x1350
[9] 0xFFFFFFFF7C41D8F4 __1cIsqlischa6FpnHSQdDLI_CB_pnLSQdDLI_SAGLOB_iI_i_ +
0x1214

```

该信息看起来很复杂，让我们来一步步地解读。

首先，问题发生的日期为 2010-10-12-03.28.49.612975，级别为严重（Severe），进程 ID 为 19575，数据库为 SAMPLE。

其次，问题发生时 DB2 的函数为 `sqlb_verify_page`，记录点为 3，从函数名称我们可以知道，这个函数是用来检验数据页（或者索引页）的。

从下面的信息，Data #1 显示错误原因：Error encountered trying to read a page 也就是读取页面的时候发生错误。Data #2 是相对详细的解释：Page verification error，即页校验错误。Data #3 是数据页 ID，这里是 301339，而 Data #4 是对象句柄的内存映像，其中我们可以看到该对象应该为表空间 4，对象 149，类型 1（即索引类型），父对象为表空间 3，对象 66（即表的对象）。

然后 Data #6 说的是在这个数据页上，其标识为表空间 29723，数据对象 0，类型 0，因此我们可以断定，该数据页完全损坏。当 DB2 将任何数据页写入磁盘之前，都要检测其合法性。因此，如果我们读到磁盘上的一个数据页的页头完全损坏，那么原因中 99% 的可能是发生在 DB2 之外，也就是磁盘或者存储系统的问题。

而要想抓到该完整的数据页，可以在 db2diag.log 中搜索 db2dart，会列出建议的命令，例如：

```
db2dart DBNAME /di /tsi 3 /oi 66 /ps 4089753p /np 1 /v y /scr n /rptn
iTsi3oi66pg4089753p+0.rpt
db2dart DBNAME /di /tsi 3 /oi 66 /ps 301339p /np 10 /v y /scr n /rptn
iTsi3oi66pg301339p+9.rpt
```

运行时需要把 DBNAME 换成真实的数据库名称。

对于索引损坏的问题，可以使用 db2dart/MI 参数，通过指定索引对象 ID，该参数可以将索引设置为不可用，然后在下次数据库启动时（或者第一次使用该索引时）自动重新创建。

对于上面的错误，用户可以使用如下命令将索引设置为不可用：

```
db2dart DBNAME /MI /TSI 4 OI 149
```

但是如果问题发生在数据上（Data #4 中的 type 为 0），那么用户需要联系 IBM 技术支持人员手工修复该表，或者恢复早先的数据库备份，回滚到最新时间戳。

对于另外一种损坏，也就是交易日志的损坏，相关的 db2diag.log 信息可能会多种多样。如果交易日志在数据库运行时被意外删除，可能会得到类似下面的日志信息：

```
2011-02-18-17.50.55.813824-300 I2284E460 LEVEL: Error
PID : 16998 TID : 47580792744256PROC : db2sysc
INSTANCE: db2inst1 NODE : 000
EDUID : 37 EDUNAME: db2loggw (SAMPLE)
FUNCTION: DB2 UDB, data protection services, sqlpgasn2, probe:1150
MESSAGE : ZRC=0x860F000A=-2045837302=SQLO_FNEX "File not found."
DIA8411C A file "" could not be found.
DATA #1 : <preformatted>
Log file: S0000001.LOG

2011-02-18-17.50.55.838576-300 I2745E413 LEVEL: Severe
PID : 16998 TID : 47580792744256PROC : db2sysc
INSTANCE: db2inst1 NODE : 000
EDUID : 37 EDUNAME: db2loggw (SAMPLE)
FUNCTION: DB2 UDB, data protection services, sqlpgasn2, probe:1160
RETCODE : ZRC=0x860F000A=-2045837302=SQLO_FNEX "File not found."
DIA8411C A file "" could not be found.
```

该信息说明日志文件 S0000001.LOG 无法被找到，读者可以用我们之前所讨论的方式自己尝试解读该日志信息。关于日志文件破坏的解决办法，我们在第 9 章节有详细解释。

17.4 挂起

挂起的意思就是数据库系统停止响应。一般来说有两种类型的挂起：一种是整个实例都停止响应，包括任何用户的输入，例如 GET SNAPSHOT 或者 LIST APPLICATIONS；另一种是某个应用程序挂起，而其他所有的连接保持正常。

对于挂起问题的分析思路，主要理解正在被挂起的进程（线程）到底在等待什么。

对于挂起问题，我们第一步要做的就是判断，到底是整个实例挂起，还是某一个应用挂起。

一般来说，我们可以用 LIST APPLICATIONS 命令判断是否是实例问题。当证明实例挂起，则可以用 db2pd 判断是否为操作系统的系统调用函数问题，还是 DB2 的 latch 问题。

大体上我们可以用图 17.1 所示的逻辑进行判断。

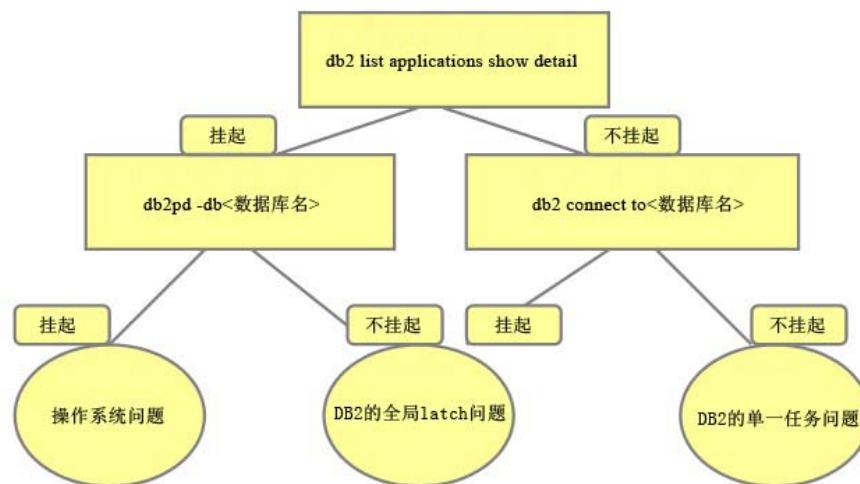


图 17.1 挂起问题诊断步骤

由于 db2pd 不需要等待任何 DB2 latch，所以如果 db2pd 命令依然挂起，则基本可以认为是操作系统问题。这时使用 procstack、pstack 或者 pdump.sh 等工具可以抓取该 db2pd 的堆栈，从而了解挂起在哪个操作系统函数中：

```

db2inst1:/DB2diag>sudo procstack 6901854
6901854: db2pd -alldbp -db adwprod -bufferpools
0x0900000000005ec10 __fcntl(??, ??, ??) + 0x260
0x0900000000005edac fcntl(0x300000003, 0xd0000000d, 0xfffffffffff9e88, 0x9001000a
090a9b0, 0x0, 0x10bc, 0x0, 0x0) + 0x44
0x090000000038ef978 sqloflock(0x300000100, 0x1000000000000001, 0x200000000000002) +
0xb8
0x090000000038ee6e4 sqllopenp__fdpr_2(0x0, 0x34c0000034c, 0x18000000180, 0xffffffff
fffffa520) + 0x64
  
```

```

0x090000000287ff9c
sqllex_write_log_record(char,char,SQLLEX_AUDIT_RECORD_T*,char*,unsigned int)(0x0,
0xffffffffffffaac8, 0x9001000a090a9b0, 0x1, 0x0) + 0x484
0x090000000287fa44
sqllex_write_log_record(char,char,SQLLEX_AUDIT_RECORD_T*,char*,unsigned
int)(??, ??, ??, ??, ??) + 0x80
0x0900000003658850 sqllex_aud_rec_func(char,int,unsigned int,short,SQLLEX_AUD_DATA_T*,
unsigned int*,sqlca*)__fdpr_2(??, ??, ??, ??, ??, ??, ??) + 0x69c
0x09000000031613d0 sqllex_get_user_authinfo(unsigned char,unsigned char,unsigned
char,int,SQLLEX_AUTHINFO_T*,sqlca*)__fdpr_1(0x1000000000000001, 0x0, 0x0, 0x0, 0x0,
0x0) + 0x138
0x00000001000009f1c pdProcessAuthorization(pdCB_t*)(??) + 0x234
0x000000010000048f4 main(??, ??) + 0x328
0x0000000100000318 __start() + 0x90

```

在这个输出中，最下面的函数代表最先被调用的，因此__start()作为进程的入口调用 main 函数，而 main 函数则调用 pdProcessAuthorization，然后该函数调用 sqllex_get_user_authinfo，依次类推。

最后，我们发现最上层的 DB2 函数为 sqloflock，它调用 fcntl 系统函数后，fcntl 调用__fcntl，然后造成挂起，很显然该函数为文件控制函数，因此解决思路应当着重考虑文件系统。

该挂起在解决了文件系统的 mount 问题后不再发生。

如果 db2pd 没有挂起，但是 list applications 命令停止响应，那么说明操作系统暂时没有证据可以证明其有问题，因此我们要从 DB2 进程（线程）栈入手分析。

使用 db2pd -stack all 可以对每一个进程（或者线程）发送 signal，然后在进程（线程）的 signal handler 之中会调用函数写出 trap 文件。通过分析每个 trap 文件中的堆栈信息和 latch 信息，可以分析出每个进程（线程）之间的等待关系。

既然是整个实例都在挂起，那么一定有一个全局 latch 阻挡了所有的请求。那么我们预期会在进程堆栈中看到很多的 latch conflict 信息。通过该 latch 的 ID，我们可以找到是谁持有这个 latch，然后分析该进程（线程）的状态。由于篇幅原因，这里不再举例。

而在最后一种情况，连接数据库与 list applications 都正常，但是系统中一个或者几个应用程序挂起，其状态一直在 UOW Executing 中，但是快照中逻辑读写与 CPU 使用都没有增长，这就是典型的应用程序挂起状态。这种情况类似全局挂起，我们需要通过快照中的应用程序 ID 对应到代理进程（线程）的 ID，然后同样用 db2pd -stack all 得到系统中所有进程（线程）的堆栈，然后从对应那个挂起的代理进程（线程）的堆栈入手。

譬如我们的挂起的代理线程中最上方的堆栈为：

```

__syscall(0x1,0x8a228,0x100d3689d48,0x3ffffe0000030000,0x2000454d8,0x80000000)
+ 0x1c

```

```

__1cRSQdDLO_SLATCH_CAS64LgetConflict6MkL_i_(0x100d3689d48,0x1,0x689e02f1100040,0
x3ffffe0000030000,0x46b0,0x8000) + 0xf0

__1cNsqlio_latch_nsDget6MkLkpkc1pnSQdDLO_LATCH_TABLE_kbkn0SQdDLO_LT_VALUES__i_(0
xcc,0x1,0xfffffffff7dcc80bc,0x2a3,0x201e74760,0x1) + 0xf8

__1cRsqlbSMSDirectRead6FpnRSQdDLB_DIRECT_IO_CB__i_(0xcc,0xfffffffff2cff43d0,0xffff
fffff7dcc80bc,0xfffff7c00,0xfffffffff7e5aa5a8,0xfffffffff13a93780) + 0x284

__1c0sqlbDirectRead6FpnRSQdDLB_DIRECT_IO_CB__i_(0xfffffffff2cff43d0,0x0,0x10010,0
x0,0x0,0x100e1bea5c8) + 0x138

__1c0sqldx_diskread6FpnJSQdDLDX_LWA_Iipc_i_(0xfffffffff2cff4798,0x1,0x100e1bea5c8
,0x2023d68e0,0x100e1bea5c8,0xfffffffff13a91380) + 0x168

__1cQsqldxReadLobData6FpnJSQdDLDX_LWA_pknMSQdDLDX_IOPARM_pc_i_(0xfffffffff2cff479
8,0xfffffffff2cff4668,0x8000,0x2c2,0xfffffffff547642c0,0x10) + 0x184

__1cMsqldxReadLob6FpnJSQdDLDX_LWA_pnISQdDLDX_LD_Iipc_i_(0xfffffffff2cff4798,0xffff
fffff54764170,0x10,0xfffffffff547642c0,0x0,0x0) + 0x368

__1c0sqldx_lob_read6FpnIsqeAgent_pnISQdDLD_TCB_IpIpnISQdDLDX_LD_pc_i_(0xfffffffff
54764178,0x0,0x10,0xfffffffff2cff4dec,0xfffffffff54764170,0xfffffffff547642c0) +
0x1a0

__1cMsqldFetchLFD6FpnIsqeAgent_HHCipnISQdDLD_LFD_iipipc_i_(0x2023d68e0,0x0,0x24,
0xfffffffff2cff4dec,0x2,0xfffffffff54764170) + 0x21c

__1cKsqlrlField0materializeLOB6MpnIsqlrr_cb_pnNSQdDLO_MEM_POOL_pnISQdDLD_CCB_pnK
SQdDLD_VALUE__i_(0xfffffffff54764080,0xfffffffff1a5718a0,0xfffffffff2cff4dec,0x7a00
0,0xfffffffff547640e0,0x0) + 0x138

```

而其 stack 文件中对应的 latch 信息为:

```

<LatchInformation>
Waiting on latch type: (SQLO_LT_SQLB_POOL_CB__readLatch) - Address: (100d3689d48),
Line: 675, File: sqlbfiles.C
</LatchInformation>

```

这个意思就是该代理线程在等待 latch 100d3689d48, 因此我们在目录中搜索:

```

U:\FODC_Hang_2011-02-05-11.19.38.431601>grep -i "100d3689d48" *.stack.txt | grep -i
"Holding"
19720.72.000.stack.txt:Holding Latch type: (SQLO_LT_SQLB_POOL_CB__readLatch) - A
ddress: (100d3689d48), Line: 2121, File: sqlbpacc.C
19720.72.000.stack.txt:Holding Latch type: (SQLO_LT_SQLB_POOL_CB__readLatch) - A
ddress: (100d3689d48), Line: 2121, File: sqlbpacc.C

```

可以发现是 19720.72.000 持有该 latch, 那么我们进入这个线程的堆栈, 得到:

```

<LatchInformation>

```

```
Holding Latch type: (SQL0_LT_SQLB_POOL_CB__readLatch) - Address: (100d3689d48), Line:
2121, File: sqlbpacc.C
</LatchInformation>
<StackTrace>
##### Object: /lib/sparcv9/libc.so.1
__syscall(0xffffffff5ea905c0,0xffffffff4effd154,0xffffffff5ea905c4,0x0,0x1,0xffff
ffff4effcf58) + 0x70
##### Object: /local/0/opt/IBM/db2/V9.5.FP2/lib64/libdb2e.so.1
__1cUSQdDLO_LIO_HANDLE_DATARsqloLioAIOCollect6MLpnXSQdDLO_LIO_COLLECT_STATUS_ppn
LSQdDLO_IO_REQdD__i_(0xffffffff4effff58,0x4,0xffffffff4effd718,0x8,0x18780000,0x
3) + 0x33c
sqloLioCollectNBLOCKS(0xffffffff4effd718,0x20,0x0,0xffffffff5ea90580,0x0,0x0) +
0x51c
__1cWsqlbClnrCollectSomeAIO6FpnMSQdDLB_CLNR_CB_L_v_(0x100d3684c50,0x20,0x0,0x37b
00ec,0x0,0x100d3684f80) + 0x84
__1cVsqlbClnrCollectAllAIO6FpnMSQdDLB_CLNR_CB__v_(0x100d3684c50,0x9ece8,0x37af6a
4,0x0,0x9ec00,0xffffffff7e5aa5a8) + 0x74
__1cQsqlbClnrFindWork6FpnMSQdDLB_CLNR_CB__i_(0x100d3684c50,0x12d8,0x100d3684c60,
0xffffffff7e5aa5a8,0x100d3684e60,0x100d3684ec0) + 0xc4c
__1cSsqlbClnrEntryPoint6FpCI_v_(0x18100000,0x100d3684d60,0x100d3684c50,0x2,0xffff
ffffffffffff,0x1001cd1c290) + 0xd0
sqloEDUEntry(0xffffffff613eb230,0xffffffff4effff58,0xa10612,0xffffffff7f020a48,0
x18780000,0xffffffff7e5aa5a8) + 0x3a4
##### Object: /lib/sparcv9/libc.so.1
```

很明显，我们在调用 `sqloLioAIOCollect` 时，该函数调用了系统调用（`__syscall`，并没有显示出是什么调用），不过根据函数名，应该可以想象是在进行 AIO 调用时的等待，而操作系统一直无法返回给 DB2。

因此，该问题依然是操作系统的文件系统 AIO 造成的挂起，而从 DB2 部分，如果想要绕过该问题则可以考虑设置：

```
db2set DB2LIOAIODISABLED=true
```

这样的话 DB2 就会避免使用操作系统 AIO 特性，虽然会对性能有一些影响，但是可以绕过一些对该特性支持不大好的平台（以上问题是在 SUN 中发现的）。

17.5 错误信息



错误信息是指当用户运行某些特定的命令（譬如说查询、备份或者 load），该命令返回一个错误代码。

这类错误可以说是日常工作最常见的问题，因此我们会用多个案例，从不同角度解释应该用什么样的思路分析这种问题。

一般来说，错误信息可以分为两大类，一类是 `SQLCODE`，也就是 DB2 返回的标准错误代

码，一般由引擎中的什么地方出现错误，然后将它映射到 `SQLCODE` 返回给用户。另一类则是其他的错误代码，比如一些 `DB2` 外围工具发生问题时，可能返回一些奇怪的数值，类似 1,2,3 之类的。

首先让我们来讨论 `SQLCODE` 错误。

17.5.1 `SQLCODE` ■ ■ ■

`SQLCODE` 是 `DB2` 的标准错误代码，可以在信息中心中查找到每一个 `SQLCODE` 所对应的含义。一般来说，使用命令“`db2"? <SQLCODE>"`”可以很轻易地得知该错误信息所代表的意义：

```
/home/db2inst1 $ db2 ? sql911
SQL0911N The current transaction has been rolled back because of
          a deadlock or timeout. Reason code "<reason-code>".

Explanation:
The current unit of work was involved in an unresolved contention
for use of an object and had to be rolled back.

The reason codes are as follows:

2 transaction rolled back due to deadlock.
68 transaction rolled back due to lock timeout.
72 transaction rolled back due to an error concerning a DB2 Data
Links Manager involved in the transaction.

Note: The changes associated with the unit of work must be
      entered again.
      The application is rolled back to the previous COMMIT.

User Response:
To help avoid deadlock or lock timeout, issue frequent COMMIT
operations, if possible, for a long-running application, or for
an application likely to encounter a deadlock.
Federated system users: the deadlock can occur at the federated
server or at the data source. There is no mechanism to detect
deadlocks that span data sources and potentially the federated
system. It is possible to identify the data source failing the
request (refer to the problem determination guide to determine
which data source is failing to process the SQL statement).

Deadlocks are often normal or expected while processing certain
combinations of SQL statements. It is recommended that you
design applications to avoid deadlocks to the extent possible.

For more detailed information about preventing deadlocks or lock
timeouts search the DB2 Information Center
(http://publib.boulder.ibm.com/infocenter/db2luw/v9) using
```

```
phrases such as "deadlock prevention", and terms such as
"deadlocks" and "lock timeouts".
```

```
sqlcode : -911
```

```
sqlstate : 40001
```

通过上面的例子，DB2 命令给出了该错误信息的详细解释。一般来说，通过阅读和理解该错误解释，加上在网络上搜索相关的错误信息，可以帮助用户解决绝大部分的错误。

但是并不是所有的错误都可以得到直观的解释。有一些错误，类似 SQL1042 或者 SQL901，它们所代表的意义便不是非常明确：

```
SQL1042C An unexpected system error occurred.
```

```
SQL0901N The SQL statement failed because of a non-severe system
error. Subsequent SQL statements can be processed.
(Reason "<reason>".)
```

一般来说，SQL1042 和 SQL901 错误都是当发生了一些 DB2 无法简单将其归结为某一类型的问题时所给出的错误代码，而解决这类问题则需要更进一步的分析。

对于这种能够反复重现，并且给出特定错误代码的问题，分析的思路永远是先从 db2diag.log 入手。

由于很多用户并不定期清理 DB2 日志，因此当问题发生后，用户首先应该做的就是运行：

```
db2diag -A
```

该命令将当前的 db2diag.log 改名并且压缩，然后用户可以重新运行命令得到错误信息，查看是否有新的 db2diag.log 生成。如果有新的文件生成，则说明其中很有可能包含该错误相关的信息。

首先我们从一个简单的例子开始：

```
/home/db2inst1 $ db2 backup db sample online to /home/db2inst1/temp
SQL2048N An error occurred while accessing object "4". Reason code: "6".
```

当我们使用 BACKUP 命令备份数据库时，SQL2048N 发生，我们需要明白为什么发生这个问题。

那么我们可以使用 db2 “? SQL2048N”得到相关信息：

```
/home/db2inst1 $ db2 "? sql2048n"
```

```
SQL2048N An error occurred while accessing object "<object>".
Reason code: "<reason-code>".
```

```
Explanation:
```

```
An error occurred while accessing an object during the processing
of a database utility. The following is a list of reason codes:
```

1 An invalid object type is encountered.

...

6 The object being accessed is a table space and either the table space is in such a state that the operation is not allowed or one or more containers of the table space is not available. (LIST TABLESPACES will list the current table space state.) Examples of such states are: quiesced, offline.

7 A delete object operation failed.

8 Trying to load/quiesce into a table that is not defined on this partition.

The utility stops processing.

User Response:

1 Ensure that "<object>" is of valid type.

...

6

The table space may be offline. Attempt to determine the underlying problem and correct it. Some examples of problems are: the filesystem is not mounted, which you can fix by mounting the filesystem then altering the table space switch to online, or table space files have been deleted, which you can fix by performing a restore operation.

The table space may be quiesced. Use LIST TABLESPACES to check the table space state. Use QUIESCE RESET OR QUIESCE TERMINATE to make the table space available. Note that the userid holding the quiesce may be needed to perform the QUIESCE REST OR TERMINATE operation.

既然命令中已经提示了 reason code 为 6，那么我们着重研究 6 所代表的含义。

在错误解释中，6 代表给定的表空间状态不对。那么我们可以连接到数据库并且检查表空间 4 的状态：

Tablespace ID	= 4
Name	= IBMDB2SAMPLEXML
Type	= Database managed space
Contents	= All permanent data. Large table space.

```

State                                = 0x0080
  Detailed explanation:
    Roll forward pending
Total pages                          = 8192
Useable pages                        = 8160
Used pages                          = 0
Free pages                          = 0
High water mark (pages)             = 0
Page size (bytes)                   = 4096
Extent size (pages)                 = 32
Prefetch size (pages)               = 32
Number of containers                 = 1

```

很显然，该表空间处于 **Roll forward Pending** 状态，因此可能由于先前发生了表空间恢复后并没有运行 **roll forward** 命令导致，因此我们需要对该表空间进行 **roll forward** 以更改该状态，然后进行备份。

这个例子是一个非常直观的问题，用户可以在给出的信息中轻易地判断出问题的出处。

但是并非所有的情况都能够如此直观，让我们来再看一个相对有点难度的例子：

首先我们来对某一个数据库做重定向恢复：

```

(db2inst1@db2host1) /home/db2inst1/temp $ db2 restore db sample into sampl redirect
SQL1277W A redirected restore operation is being performed. Table space
configuration can now be viewed and table spaces that do not use automatic
storage can have their containers reconfigured.
DB20000I The RESTORE DATABASE command completed successfully.

```

```

(db2inst1@db2host1) /home/db2inst1/temp $ db2 "set tablespace containers for 5 using
(file '/home/db2inst1/temp/ts2/data' 1000)"
DB20000I The SET TABLESPACE CONTAINERS command completed successfully.

```

```

(db2inst1@db2host1) /home/db2inst1/temp $ db2 restore db sample continue
SQL1277W A redirected restore operation is being performed. Table space
configuration can now be viewed and table spaces that do not use automatic
storage can have their containers reconfigured.
DB20000I The RESTORE DATABASE command completed successfully.

```

```

(db2inst1@db2host1) /home/db2inst1/temp $ db2 connect to sampl
SQL0752N Connecting to a database is not permitted within a logical unit of
work when the CONNECT type 1 setting is in use.

```

这是怎么回事？不是明明说“The RESTORE DATABASE command completed successfully.”了吗？

开启另外一个窗口，尝试连接数据库：

```

(db2inst1@db2host1) /home/db2inst1/sqlllib/db2dump $ db2 connect to sampl
SQL1119N A connection to or activation of database "SAMPL" cannot be made
because a previous restore is incomplete. SQLSTATE=57019

```

看来恢复过程中出现了问题，但是我们并没有看到太多相关的错误信息。

怎么办？我们按照刚才提到的思路，首先备份 db2diag.log 文件：

```
(db2inst1@db2host1) /home/db2inst1/temp $ db2diag -A
db2diag: Moving "/home/db2inst1/sqlllib/db2dump/db2diag.log"
to      "/home/db2inst1/sqlllib/db2dump/db2diag.log_2011-02-20-10.04.53"
```

然后重新做一遍 restore continue:

```
(db2inst1@db2host1) /home/db2inst1/temp $ db2 restore db sample continue
SQL1277W A redirected restore operation is being performed. Table space
configuration can now be viewed and table spaces that do not use automatic
storage can have their containers reconfigured.
DB20000I The RESTORE DATABASE command completed successfully.
```

然后看一下 db2diag.log 中有什么信息：

```
2011-02-20-10.07.48.248041-300 E7542A1013          LEVEL: Warning
PID      : 1454084          TID : 1          PROC : db2agent (SAMPL)
INSTANCE: db2inst1        NODE : 000        DB   : SAMPL
APPHDL   : 0-67           APPID: *LOCAL.db2inst1.110220145541
AUTHID   : DB2INST1
FUNCTION: DB2 UDB, database utilities, sqludCheckRedirectedStatus, probe:1162
MESSAGE : SQL1277W A redirected restore operation is being performed. Table
          space configuration can now be viewed and table spaces that do not
          use automatic storage can have their containers reconfigured.
DATA #1 : SQLCA, PD_DB2_TYPE_SQLCA, 136 bytes
sqlcaid : SQLCA    sqlcab: 136    sqlcode: 1277    sqlerrml: 0
sqlerrmc:
sqlerrp : sqludChe
sqlerrd : (1) 0x00000000    (2) 0x00000000    (3) 0x00000000
          (4) 0x00000000    (5) 0x00000000    (6) 0x00000000
sqlwarn : (1)    (2)    (3)    (4)    (5)    (6)
          (7)    (8)    (9)    (10)   (11)
sqlstate:
```

```
2011-02-20-10.07.48.248262-300 I8556A771          LEVEL: Warning
PID      : 1454084          TID : 1          PROC : db2agent (SAMPL)
INSTANCE: db2inst1        NODE : 000        DB   : SAMPL
APPHDL   : 0-67           APPID: *LOCAL.db2inst1.110220145541
AUTHID   : DB2INST1
FUNCTION: DB2 UDB, database utilities, sqludCheckRedirectedStatus, probe:1165
DATA #1 : <preformatted>
Insufficient space in tablespace TS1; you must have at least 4960
usable pages. (The "usable pages" total does not include pages used
internally by DB2, so the value specified on the SET TABLESPACE
CONTAINERS operation should be increased by one extent per container.
Based on the latest SET TABLESPACE CONTAINERS values specifed, the
tablespace should have 4992 pages in total.)
```

“Insufficient space in tablespace TS1; you must have at least 4960 可以看到 usable pages.”。也就是说，我们的 TS1 所指定的空间太小，需要最少 4960 可用页，即 4992 页的表空间。

那么我们来再一次 SET TABLESPACE CONTAINERS:

```
(db2inst1@db2host1) /home/db2inst1/temp $ db2 "set tablespace containers for 5 using
(file '/home/db2inst1/temp/ts2/data' 5000)"
DB20000I The SET TABLESPACE CONTAINERS command completed successfully.
(db2inst1@db2host1) /home/db2inst1/temp $ db2 restore db sample continue
DB20000I The RESTORE DATABASE command completed successfully.
(db2inst1@db2host1) /home/db2inst1/temp $ db2 connect to sampl
SQL1117N A connection to or activation of database "SAMPL" cannot be made
because of ROLL-FORWARD PENDING. SQLSTATE=57019
```

这次又出了一个新的错误：SQL1117N。

我们来看一下这个错误又是什么：

```
(db2inst1@db2host1) /home/db2inst1/temp $ db2 ? sql1117n
```

```
SQL1117N A connection to or activation for database "<name>"
cannot be made because of ROLL-FORWARD PENDING.
```

Explanation:

The specified database is enabled for roll-forward recovery and it has been restored but not rolled forward.

No connection was made.

Federated system users: this situation can also be detected by the data source.

User Response:

Roll forward the database or indicate that you do not wish to roll forward by using the ROLLFORWARD command. Note that if you do not roll forward the database, the records written since the last backup of the database will not be applied to the database.

Federated system users: if necessary isolate the problem to the data source rejecting the request (see the problem determination guide for procedures to follow to identify the failing data source) and take recovery action appropriate to that data source to bring the data source to a point of consistency.

sqlcode : -1117

sqlstate : 57019

意思是说我们需要 rollforward 这个数据库，那么我们来演示一下：

```
(db2inst1@db2host1) /home/db2inst1/temp $ db2 rollforward db sampl complete
```

Rollforward Status

```

Input database alias           = sampl
Number of nodes have returned status = 1

Node number                   = 0
Rollforward status           = not pending
Next log file to be read      =
Log files processed           = -
Last committed transaction    = 2011-02-20-14.55.21.000000 UTC

```

```

DB20000I The ROLLFORWARD command completed successfully.
(db2inst1@db2host1) /home/db2inst1/temp $ db2 connect to sampl

```

Database Connection Information

```

Database server               = DB2/AIX64 9.1.8
SQL authorization ID          = DB2INST1
Local database alias          = SAMPL

```

现在看到了如何用 db2diag.log 诊断一些并不是非常直观的问题。但是 db2diag.log 是否是万能药呢？答案是否定的。

17.5.2 db2trc ■ ■ ■

有些时候，甚至 db2diag.log 都不能告诉我们太多信息，那么我们就要寻求别的工具的帮助了。这个工具就是 db2trc，也就是 DB2 跟踪工具。

db2trc 工具在 DB2 中非常强大，不过想要读懂这个工具所产生的文件也是非常辛苦的。db2trc 工具在设计时主要是为了 DB2 实验室的开发人员所准备的，很多情况下需要对照 DB2 的源代码进行调试。一般情况下，db2diag.log 足以应付 99% 以上的问题，而对于剩下的那 1%，用户可以给 IBM 技术支持中心打电话获得帮助。而如果用户想要看一看 db2trc 中都有什么东西，我们在这里进行一些简单的介绍。

实际上在前文中的性能章节我们已经给了一个使用 db2trc 的案例，不过只是走马观花。这里让我们来仔细讨论一下 db2trc 的功能及其设计理念和使用方法。

db2trc 在设计之初主要是用来跟踪 DB2 中函数调用的关系。譬如说我们有一个程序 fibonacci.c，实现“斐波那契数列”计算：

```

#include <stdio.h>
FILE *ftrace;
int fibonacci(int n)
{
    int a = 0;
    int b = 1;
    int sum;
    int i;

```

```

fprintf(ftrace, "fibonacci entry\n");
fprintf(ftrace, "fibonacci probe 1: %d\n", n);
for (i=0;i<n;i++)
{
    printf("%d\n",a);
    fprintf(ftrace, "fibonacci probe 2: a=%d, b=%d, sum=%d\n", a,b,sum);
    sum = a + b;
    a = b;
    b = sum;
    fprintf(ftrace, "fibonacci probe 3: a=%d, b=%d, sum=%d\n", a,b,sum);
}
fprintf(ftrace, "fibonacci exit\n");
return 0;
}

int main ()
{
    int n;
    if(!(ftrace=fopen("trace.out", "w")))
    {
        printf("Failed to open trace file\n");
        exit(1);
    }
    fprintf(ftrace, "main entry\n");

    printf("\nHow many numbers of the sequence would you like?\n");
    scanf("%d",&n);

    fprintf(ftrace, "main probe 1: %d\n", n);
    fibonacci(n);
    fprintf(ftrace, "main exit\n");
    return 0;
}

```

在我们编译完并且执行后，我们有如下输出：

```

(db2inst1@db2host1) /home/db2inst1/temp $ cc test.c
(db2inst1@db2host1) /home/db2inst1/temp $ ./a.out

How many numbers of the sequence would you like?
5
0
1
1
2
3

```

首先我们给定一个 5 作为输入，然后结果显示 0,1,1,2,3。

而我们自己写的跟踪文件 trace.out 则包含：

```

main entry
main probe 1: 5
fibonacci entry
fibonacci probe 1: 5
fibonacci probe 2: a=0, b=1, sum=0
fibonacci probe 3: a=1, b=1, sum=1
fibonacci probe 2: a=1, b=1, sum=1
fibonacci probe 3: a=1, b=2, sum=2
fibonacci probe 2: a=1, b=2, sum=2
fibonacci probe 3: a=2, b=3, sum=3
fibonacci probe 2: a=2, b=3, sum=3
fibonacci probe 3: a=3, b=5, sum=5
fibonacci probe 2: a=3, b=5, sum=5
fibonacci probe 3: a=5, b=8, sum=8
fibonacci exit
main exit

```

通过这个文件，我们可以理解程序的处理流程，也就是 `main` 函数首先在记录点 1 中得到输入 5，然后调用 `fibonacci` 函数一次，在函数中的第一个记录点为 5，也就是 `n` 的值。等进入了循环，程序会打印记录点 2,3 各 5 次，每一次都显示 `a,b` 和 `sum` 的数值。

使用这个信息，我们可以在程序发生异常时将结果映射到源代码中，发现可能存在的问题。这个也就是 `db2trc` 的设计思路。当然，`db2trc` 要远比 `fprintf` 复杂得多，但是其基本的理念没有改变，就是为了跟踪程序的流程和一些重要的临时变量而设计的。

那么读者可能开始想了，既然是内部使用的，我们为什么还要费心思去看。

没错，一般来说用户是不需要分析 `trace` 文件的，就算花时间去分析，大部分情况也会毫无头绪。就好像只给用户 `trace.out`，如果没有源代码的话谁知道 `a`、`b`、`sum` 都代表什么？

这里提 `db2trc` 的意义并不是给出一个详细的分析教程，而是给愿意继续钻研的读者一个选择，可以做到比阅读 `db2diag.log` 更进一步的分析工作。

`db2trc` 的使用方法是，在重现某一问题之间，要打开跟踪，然后重现问题，然后关闭跟踪，然后格式化二进制文件到 `FLW` 与 `FMT` 文件中。其中 `FLW` 为流程文件，就是每一个函数的调用关系；而 `FMT` 为格式化文件，是 `FLW` 每一个函数中所记录的具体的临时变量信息。

一般来说，在分析的过程中需要结合两者进行分析。

打开跟踪的方法一般有 3 类，第一类是写入文件。这样的好处就是可以抓到从跟踪打开到关闭之间所有的记录点，但是缺点就是对性能影响太大，这个方法只能用于测试系统，在生产系统切记不要使用这种方法。

第二类是写入内存。这个方法相比起第一种，会对性能造成相对较小的影响（但是依然会有大约 20% 以上的 CPU 损耗），缺点就是如果指定的缓冲区大小满了以后，可能会丢失之前的信息。这种方法一般用于生产系统，在万不得已的时候使用 `trace`，一定要使用内存方式收集数据。

第三种是性能统计数据，也就是并不记录每一个函数之间的调用关系，而是记录每一个函数的调用次数和进入/退出之间所消耗的总时间。该方法对性能产生的影响极为微弱，一般用户系统中发生严重的性能问题时，对每个函数的运行时间进行细致的了解。该用法对于用户来说用处不大，需要 IBM 技术支持专家在需要时收集该信息并对文件进行细致的解读，因此在本文中不予讨论。

第一种方法的格式为：

```
db2trc on -t -f <跟踪文件名>
<执行重现问题的命令>
db2trc off
db2trc flw -t <跟踪文件名> <跟踪文件名.flw>
db2trc fmt <跟踪文件名> <跟踪文件名.fmt>
```

而第二种方法的格式为：

```
db2trc on -t -l 128M
<执行重现问题的命令>
db2trc dmp <跟踪文件名>
db2trc off
db2trc flw -t <跟踪文件名> <跟踪文件名.flw>
db2trc fmt <跟踪文件名> <跟踪文件名.fmt>
```

两者的区别在于开启与关闭时的参数。写入内存的选项需要指定 `-l` 和内存池的大小，然后再关闭之前需要将该内存写入的跟踪文件；而直接写入文件的选项就可以直接向文件中写入。

在本文的案例中，为了简便起见，我们使用第一种方法。不过用户切记在生产系统中一定不要使用文件跟踪方式，以免对性能造成大幅度的影响。

我们使用一个笔者曾经给出结论的案例：

```
用户使用 32 位 Linux 操作系统，每天不定时在连接数据库时会出现 SQL1084C 的错误：
[db2inst1@localhost ~]$ db2 connect to suzhou
SQL1084C Shared memory segments cannot be allocated.  SQLSTATE=57019

-- 日志如下：
2010-11-05-15.35.28.449778+480 E8617819G952    LEVEL: Warning
PID      : 4708                      TID   : 2949639056   PROC  : db2sysc 0
INSTANCE: db2inst1                  NODE  : 000          DB    : SUZHOU
APPHDL   : 0-79                      APPID: 192.168.6.248.61993.10110507352
AUTHID   : L_SZ_V5
EDUID    : 20                        EDUNAME: db2agent (SUZHOU) 0
FUNCTION: DB2 UDB, base sys utilities, sqlLocalDatabase::FirstConnect, probe:100
MESSAGE  : ZRC=0x850F0005=-2062614523=SQLO_NOSSEG
          "No Storage Available for allocation"
          DIA8305C Memory allocation failure occurred.
DATA #1 : String, 299 bytes
Failed to allocate the desired database shared memory set.
The configured DATABASE_MEMORY plus desired overflow may have
exceeded INSTANCE_MEMORY or the maximum shared memory on the system.
Attempting to start up with a smaller overflow allowance.
```

```
Desired database shared memory set size is (bytes):
DATA #2 : unsigned integer, 4 bytes
1782579200
```

这个错误就是说，当数据库想要分配一块 1782579200 字节的内存时失败。

在 db2diag.log 中是唯一的结论，很难更加深入一步地去了解为什么失败。

我们检查了内核参数与当前内存剩余，发现一切指标正常。

因此我们抓 db2trc，其中错误信息点（在 FLW 文件中搜索 SOLO NOSEG）得到：

```
6062          0.015829000 | ||||| | | | | | | | | | | | sqloGetSharedMemoryFromOs  
entry [eduid 869 eduname db2agent]  
6063          0.015830000 | ||||| | | | | | | | | | | | sqloMemCreateSingleSegment  
entry [eduid 869 eduname db2agent]  
6064          0.015845000 | ||||| | | | | | | | | | | | sqloMemCreateSingleSegment  
exit  
6065          0.015845000 | ||||| | | | | | | | | | | | sqloGetSharedMemoryFromOs  
data [probe 2020]  
6066          0.015846000 | ||||| | | | | | | | | | | | sqloMemAttachToSegments  
entry [eduid 869 eduname db2agent]  
6067          0.015851000 | ||||| | | | | | | | | | | | sqloMemAttachToSegments  
SYSTEM ERROR [probe 100] [ ZRC = 0x850F0005 = -2062614523 = SQLO_NOSEG]  
6068          0.015853000 | ||||| | | | | | | | | | | | pdLogSysRC entry  
[eduid 869 eduname db2agent]  
6069          0.015854000 | ||||| | | | | | | | | | | | pdIsDiagLevelOk  
entry [eduid 869 eduname db2agent]  
6070          0.015855000 | ||||| | | | | | | | | | | | pdIsDiagLevelOk  
data [probe 10]  
6071          0.015856000 | ||||| | | | | | | | | | | | pdIsDiagLevelOk  
data [probe 20]  
6072          0.015857000 | ||||| | | | | | | | | | | | pdIsDiagLevelOk  
data [probe 500]  
6073          0.015858000 | ||||| | | | | | | | | | | | pdIsDiagLevelOk  
exit  
6074          0.015861000 | ||||| | | | | | | | | | | | sqloMemAttachToSegments  
SYSTEM ERROR [probe 200]  
6075          0.015861000 | ||||| | | | | | | | | | | | pdLogSysRC exit  
6076          0.015862000 | ||||| | | | | | | | | | | | sqloMemAttachToSegments  
exit [rc = 0x850F0005 = -2062614523 = SQLO_NOSEG]
```

```
6063          entry DB2 UDB SQO Memory Management sqloMemCreateSingleSegment fnc  
(1.3.129.43.0)  
pid 6827 tid 1061153680 cpid 25803 node 0 sec 0 nsec 15830000  
eduid 869 eduname db2agent  
bytes 48
```

```
Data1      (PD_TYPE_SET_SIZE,4) Current set size:  
1782644736  
Data2      (PD_TYPE_UINT,4) unsigned integer:  
0  
Data3      (PD TYPE HEXINT,4) Hex integer:
```

```

0x000007C1
Data4      (PD_TYPE_BITMASK,4) Bitmask:
0x02000000

6066      entry DB2 UDB SQO Memory Management sqloMemAttachToSegments fnc
(1.3.129.45.0)
pid 6827 tid 1061153680 cpid 25803 node 0 sec 0 nsec 15846000
eduid 869 eduname db2agent
bytes 12

Data1      (PD_TYPE_PTR,4) Pointer:
0x00000000

6067      SYSTEM ERROR DB2 UDB SQO Memory Management sqloMemAttachToSegments fnc
(5.3.129.45.0.100)
pid 6827 tid 1061153680 cpid 25803 node 0 sec 0 nsec 15851000 probe 100
Error ZRC = 0x850F0005 = -2062614523 = SQL0_NOSEG
Func.Called: shmat
System Errno: 12

bytes 44

Data1      (PD_TYPE_PTR,4) Pointer:
0x00000000
Data2      (PD_TYPE_OSS_MEM_SET_ID,4) Memory set ID:
0880 4F14          ..0.

Data3      (PD_TYPE_UINT,4) unsigned integer:
0

```

这一部分说明，当前的共享内存创建已经成功了，但是调用 `shmat` 将共享内存连接到进程时失败。这时基本上能说明一个问题了，就是当时 32 位的进程中不存在足够的连续空间用来连接到共享内存。

因此我们的思路应该转向进程内存映像，即 `pmap`，正常情况下我们有：

40010000	4	-	-	-	----	[anon]
40011000	1852	-	-	-	rwX--	[anon]
40300000	1188	-	-	-	rwX--	[anon]
40429000	860	-	-	-	----	[anon]
4063d000	1024	-	-	-	rwX--	[anon]
4083d000	4100	-	-	-	rwX--	[anon]
40cd6000	4	-	-	-	rwX--	[anon]
40d3d000	4	-	-	-	rwX--	[anon]
40d3e000	1740864	-	-	-	rwXs-	[shmid=0x4fa30006]
ab14e000	40064	-	-	-	rwXs-	[shmid=0x4fa28005]
ad86e000	1024	-	-	-	rwX--	[anon]
ad96e000	40	-	-	-	r-x--	IBMOsauthserver.so
ad978000	8	-	-	-	rwX--	IBMOsauthserver.so
ad97a000	788	-	-	-	r-x--	libcrypto.so.0.9.7
ada3f000	64	-	-	-	rwX--	libcrypto.so.0.9.7
ada4f000	12	-	-	-	rwX--	[anon]
ada52000	64	-	-	-	r-x--	libiclib.so
ada62000	4	-	-	-	rwX--	libiclib.so

```
ada63000      4      -      -      - r-x-- locale-archive
```

可以看到, 40d3e000 1740864 - - - rwxs- [shmid=0x4fa30006]这一段就是正常情况下的数据库内存。通过计算 $1740864 * 1024 + 0x40d3e000 = 0xAB14E000$, 我们发现它刚好等于下一段共享内存的起始地址 ab14e000, 这一段很显然也就是应用程序组内存, 占用到 0xAD86E000 地址。

而当分配失败的时候, 该内存看起来为:

```
40d17000      24      -      -      - rwx-- [ anon ]
ad82e000      128     -      -      - rwxs- [ shmid=0x6a858006 ]
ad84e000      128     -      -      - rwxs- [ shmid=0x6a850005 ]
```

也就是我们所拥有的空间仅为 $ad82e000 - 40d17000 = 1823567872$ 字节, 通过观察当时的 db2trc, 发现该空间略小于(差几 K 字节)数据库所需要的内存大小。

那么很明显, 是由于我们的内存空间不足引起的。而关于那个 128KB 的内存到底是什么, 通过对比当时的 db2pd 输出, 可以确定这两个内存为 LCL 内存, 也就是本地连接所使用的共享内存。

因此, 想要解决这个问题, 我们有以下几个方法:

- ①使用 64 位系统。
- ②对数据库保持持续的连接, 不要经常使数据库停止。
- ③手工 activate 数据库。
- ④每次 activate 数据库之前确保所有的 db2bp 进程被终止。

17.5.3 strace ■ ■ ■

而另一种更为底层的跟踪方法是使用 OS 跟踪命令, 例如 strace。

我们在这里给出一个案例但不进行深入讨论, 因为其理论类似于 db2trc。

```
用户执行 db2start 时发生 SQL1042 错误:
-bash-3.2$ db2start
SQL1042C An unexpected system error occurred.  SQLSTATE=58004

db2diag.log 中唯一的信息是:
2011-01-10-11.30.30.883268+480 I149139853E337      LEVEL: Severe
PID      : 6934      TID  : 47545288465088PROC : db2start
INSTANCE: db2inst1      NODE : 000
FUNCTION: DB2 UDB, base sys utilities, sqlestrt.C::main, probe:32
DATA #1 : Hexdump, 4 bytes
0x00007FFF86134E54 : EEFB FFFF      ...
```

对于这个问题，我们通过使用操作系统的 `strace` 跟踪 `db2start`，下面是其中一小部分相关的信息：

[illegible]

```
32556) = 32556
17209 --- SIGSEGV (Segmentation fault) @ 0 (0) ---
```

首先，第一部分：

```
17208 execve("/home/db2inst1/sqlllib/adm/db2start", ["db2start"], [/* 24 vars */])
= 0
17208 brk(0) = 0x9dd000
17208 mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x2b6f9ebd4000
17208 uname({sys="Linux", node="szv-dev-pyd-rh1", ...}) = 0
```

这部分说的是 db2start 命令开始，进程号为 17208。

然后该进程执行了一段时间以后，调用了 clone 系统函数创建一个 17209 进程：

```
17208 msgget(IPC_PRIVATE, IPC_CREAT|IPC_EXCL|0701) = 13238274
17208 clone(child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0x2b6faa36eb50) = 17209
17209 close(0 <unfinished ...>
17208 wait4(17209, <unfinished ...>
17209 <... close resumed>) = 0
```

在下一段跟踪中，我们看到这个线程调用 execve 执行 db2chkau 程序：

```
17209 rt_sigprocmask(SIG_UNBLOCK, ~[RTMIN RT_1], NULL, 8) = 0
17209 geteuid() = 507
17209 execve("/home/db2inst1/sqlllib/security/db2chkau", ["db2chkau", "13205505"],
[/* 24 vars */]) = 0
17209 brk(0) = 0x1ebaa000
17209 mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x2acf5dc8a000
17209 uname({sys="Linux", node="szv-dev-pyd-rh1", ...}) = 0
```

此时我们可以看到打开了一个 /lib64/libnss_winbind.so.2 的库，作为安全验证第三方软件：

```
17209 mmap(NULL, 69490, PROT_READ, MAP_PRIVATE, 4, 0) = 0x2acf69b69000
17209 close(4) = 0
17209 open("/lib64/libnss_winbind.so.2", O_RDONLY) = 4
17209 read(4,
"\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\320\31\0\0\0\0\0"... , 832) = 832
17209 fstat(4, {st_mode=S_IFREG|0755, st_size=23712, ...}) = 0
```

然后打开一个套接字，作为文件句柄 4：

```
17209 close(4) = 0
17209 munmap(0x2acf69b69000, 69490) = 0
17209 lstat("/tmp/.winbindd", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
17209 lstat("/tmp/.winbindd/pipe", {st_mode=S_IFSOCK|0777, st_size=0, ...}) = 0
17209 socket(PF_FILE, SOCK_STREAM, 0) = 4
17209 fcntl(4, F_GETFL) = 0x2 (flags O_RDWR)
17209 fcntl(4, F_SETFL, O_RDWR|O_NONBLOCK) = 0
17209 fcntl(4, F_GETFD) = 0
17209 fcntl(4, F_SETFD, FD_CLOEXEC) = 0
17209 connect(4, {sa_family=AF_FILE, path="/tmp/.winbindd/pipe"...}, 110) = 0
```

过了一会，该进程得到信号 11，也就是非法内存访问：

```
17209 mmap(NULL, 159744, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x2acf69d86000
17209 select(5, [4], NULL, NULL, {5, 0}) = 1 (in [4], left {5, 0})
17209 read(4, "oradba admins\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"...
157060) = 124504
17209 select(5, [4], NULL, NULL, {5, 0}) = 1 (in [4], left {5, 0})
17209 read(4,
"\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"...
32556) = 32556
17209 --- SIGSEGV (Segmentation fault) @ 0 (0) ---
```

在 `sigsegv` 之前的最后一步操作是从套接字中读取信息，而该套接字的 `connect` 指定的 `path` 为 `path="/tmp/.winbindd/pipe"`，也就是说该套接字操作与第三方验证软件有关。因此我们下一步要做的就是从操作系统层面停止使用这个验证软件，然后 DB2 实例就可以成功启动了。

17.6 分析数据收集工具

当我们在分析数据时，出于各种各样的原因，有可能 DBA 并不能直接登录出现问题的系统（譬如公司的安全规定之类的）。这时我们需要一些工具来收集尽可能多的有用的数据，以供我们线下分析。

在这一节中，我们将会介绍一些数据收集工具，以快速地收集分析数据。本节中将会介绍 `db2support`、`db2fodc`、`db2service.perf1` 与 `db2service.mem1` 工具，分别对应着不同类型的数据收集。

1. db2support

`db2support` 工具也许是最常用的系统信息收集工具了。如果读者曾经联系过 IBM 技术支持小组，应该有过收集 `db2support` 文件的经验。

一般来说，`db2support` 的收集命令为：

```
db2support . -d <数据库名> -c -g -s
```

而当数据库无法连接时（或者 `hang`），则使用：

```
db2support . -g -s
```

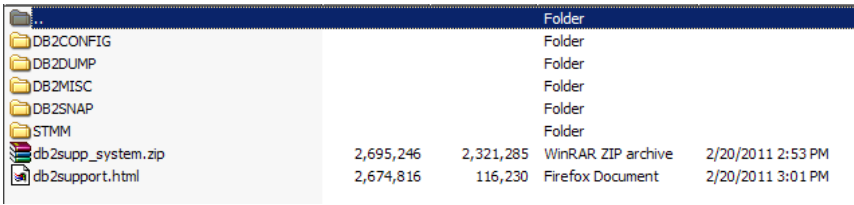
对于优化器相关的问题，当确定某一个 SQL 运行缓慢时，可以使用：

```
db2support . -d <数据库名> -cl 1 -sf <SQL 文件名>
```

注意在 `<SQL 文件名>` 中，出现问题的 SQL 需要以分号结尾。

这个命令会在当前目录产生一个 `db2support.zip` 文件，其中包含非常多的实用信息。

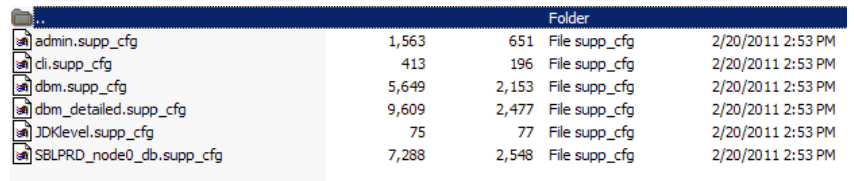
例如，如图 17.2 所示为使用了 -d -c -g -s 收集信息。



Folder			
DB2CONFIG			Folder
DB2DUMP			Folder
DB2MISC			Folder
DB2SNAP			Folder
STMM			Folder
db2supp_system.zip	2,695,246	2,321,285	WinRAR ZIP archive 2/20/2011 2:53 PM
db2support.html	2,674,816	116,230	Firefox Document 2/20/2011 3:01 PM

图 17.2 db2support 目录结构

其中，DB2CONFIG 目录主要包含一些 DB2 设置相关的信息，如图 17.3 所示。



Folder			
admin.supp_cfg	1,563	651	File supp_cfg 2/20/2011 2:53 PM
cli.supp_cfg	413	196	File supp_cfg 2/20/2011 2:53 PM
dbm.supp_cfg	5,649	2,153	File supp_cfg 2/20/2011 2:53 PM
dbm_detailed.supp_cfg	9,609	2,477	File supp_cfg 2/20/2011 2:53 PM
JDKlevel.supp_cfg	75	77	File supp_cfg 2/20/2011 2:53 PM
SBLPRD_node0_db.supp_cfg	7,288	2,548	File supp_cfg 2/20/2011 2:53 PM

图 17.3 db2support DB2CONFIG 目录

DB2DUMP 目录包含 db2diag.log 文件，如图 17.4 所示。



Folder			
db2diag.log	150,209,857	6,161,889	Text Document 2/20/2011 3:01 PM

图 17.4 db2support DB2DUMP 目录

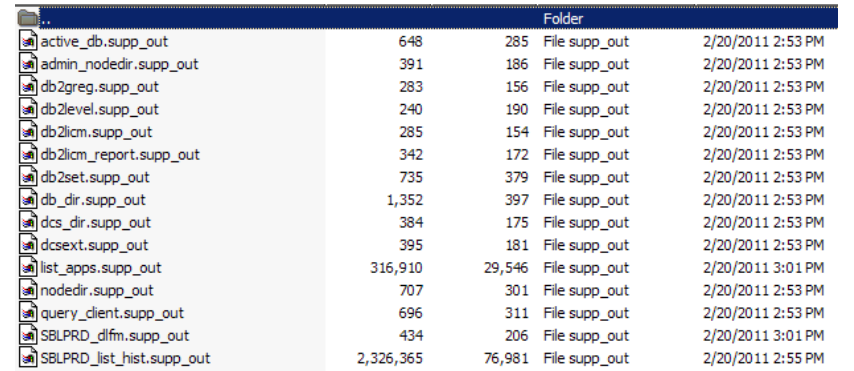
DB2MISC 目录主要是日志和 sqllib 目录结构，如图 17.5 所示。



Folder			
db2_log_directory.txt	6,620	848	Text Document 2/20/2011 2:53 PM
db2_sqllib_directory.txt	28,796	3,991	Text Document 2/20/2011 2:53 PM

图 17.5 db2support DB2MISC 目录

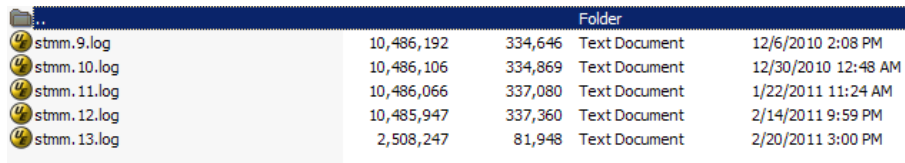
DB2SNAP 是一些快照相关信息（或者有关 list applications 之类的），如图 17.6 所示。



Folder			
active_db.supp_out	648	285	File supp_out 2/20/2011 2:53 PM
admin_nodedir.supp_out	391	186	File supp_out 2/20/2011 2:53 PM
db2greg.supp_out	283	156	File supp_out 2/20/2011 2:53 PM
db2level.supp_out	240	190	File supp_out 2/20/2011 2:53 PM
db2licm.supp_out	285	154	File supp_out 2/20/2011 2:53 PM
db2licm_report.supp_out	342	172	File supp_out 2/20/2011 2:53 PM
db2set.supp_out	735	379	File supp_out 2/20/2011 2:53 PM
db_dir.supp_out	1,352	397	File supp_out 2/20/2011 2:53 PM
dcs_dir.supp_out	384	175	File supp_out 2/20/2011 2:53 PM
dcsext.supp_out	395	181	File supp_out 2/20/2011 2:53 PM
list_apps.supp_out	316,910	29,546	File supp_out 2/20/2011 3:01 PM
nodedir.supp_out	707	301	File supp_out 2/20/2011 2:53 PM
query_client.supp_out	696	311	File supp_out 2/20/2011 2:53 PM
SBLPRD_difm.supp_out	434	206	File supp_out 2/20/2011 3:01 PM
SBLPRD_list_hist.supp_out	2,326,365	76,981	File supp_out 2/20/2011 2:55 PM

图 17.6 db2support DB2SNAP 目录

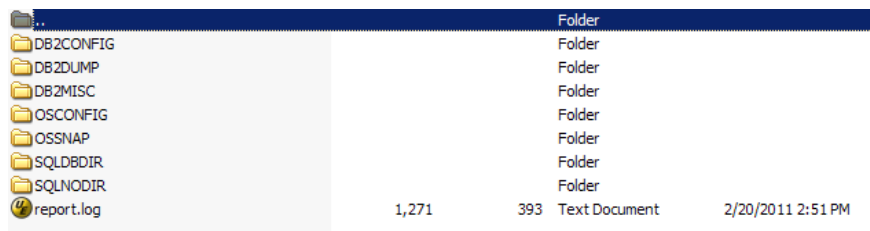
STMM 目录，顾名思义，包含 STMM 日志信息，如图 17.7 所示。



File Name	Size	Type	Created
stmm.9.log	10,486,192	Text Document	12/6/2010 2:08 PM
stmm.10.log	10,486,106	Text Document	12/30/2010 12:48 AM
stmm.11.log	10,486,066	Text Document	1/22/2011 11:24 AM
stmm.12.log	10,485,947	Text Document	2/14/2011 9:59 PM
stmm.13.log	2,508,247	Text Document	2/20/2011 3:00 PM

图 17.7 db2support STMM 目录

如果使用了 -s 参数，则会收集系统信息，这些信息存在于 db2support.zip 包中的 db2supp_system.zip 文件中，如图 17.8 所示。



File Name	Size	Type	Created
DB2CONFIG		Folder	
DB2DUMP		Folder	
DB2MISC		Folder	
OSCONFIG		Folder	
OSSNAP		Folder	
SQLDBDIR		Folder	
SQLNODIR		Folder	
report.log	1,271	Text Document	2/20/2011 2:51 PM

图 17.8 db2support db2supp_system.zip 目录

其中比较重要的目录包括 DB2DUMP、OSCONFIG 与 OSSNAP。这里的 DB2DUMP 包括除了 db2diag.log 文件与 core 之外的所有 db2dump 目录下的信息，包括 FODC、dump、trap 文件等，在宕机问题中非常重要。而 OSCONFIG 与 OSSNAP 则包含操作系统设置与一些快照信息，对于了解系统配置有很大的帮助。

最后，在 db2support.zip 包中，最重要的文件 db2support.html 包含了该实例与数据库（如果使用 -d 参数）配置、实例版本、db2nodes.cfg 等重要的实例信息，可以帮助用户在最短时间内了解系统的配置情况。

2. db2fodc

在 9.5 版本之前，当系统发生故障时，用户一般都需要尝试重现问题，然后在问题发生的过程中使用一些自己写好的脚本收集数据。

但是，对于很多企业来说，重现问题也就意味着业务的中断。因此，db2fodc 就是为了解决这个问题被创造出来的。db2fodc 工具的全称为 DB2 first occurrence data collection，即第一次问题发生时的数据收集。

譬如说，用户在下午的业务高峰期发现系统 CPU 占用很高，如果用户是一个非常有经验处理性能问题的专家，那么应该知道收集多次快照与进程堆栈信息是处理该类似问题的唯一途径。但是并非所有的用户都有过大量的经验，因此，当用户不明确需要收集什么样的数据时，可以简单地运行 db2fodc -perf 就可以收集与性能相关的数据了。

在 db2fodc 中，包括了类似性能、挂起等不同类型的问題。


```

run_snapshot      6
run_stacktrace    5
run_db2perfcount  2
run_db2trc        1

```

Data Collection Start: Tue Feb 22 08:23:51 EST 2011

```

-----
at 1 seconds: snapshot - Executing
at 1 seconds: vmstat - Executing
at 1 seconds: iostat - Executing
at 2 seconds: snapshot - Finished
at 80 seconds: stacktrace - Executing
at 116 seconds: stacktrace - Finished
at 159 seconds: snapshot - Executing
at 159 seconds: snapshot - Finished
at 238 seconds: stacktrace - Executing
at 278 seconds: stacktrace - Finished
at 317 seconds: db2perfcount - Executing
at 327 seconds: db2perfcount - Finished
at 396 seconds: snapshot - Executing
at 396 seconds: snapshot - Finished
at 475 seconds: stacktrace - Executing
at 515 seconds: stacktrace - Finished
at 554 seconds: snapshot - Executing
at 555 seconds: snapshot - Finished
at 633 seconds: stacktrace - Executing
at 673 seconds: stacktrace - Finished
at 712 seconds: snapshot - Executing
at 712 seconds: snapshot - Finished
at 791 seconds: db2trc - Executing
at 801 seconds: db2trc - Finished
at 870 seconds: db2perfcount - Executing
at 880 seconds: db2perfcount - Finished
at 949 seconds: stacktrace - Executing
at 989 seconds: stacktrace - Finished
at 1002 seconds: iostat - Finished
at 1002 seconds: vmstat - Finished
at 1028 seconds: snapshot - Executing
Finished all snapshots
at 1028 seconds: snapshot - Finished

```

/home/db2inst1/sqllib/bin/db2cos_perf Finished. Exiting at Tue Feb 22 08:42:17 EST 2011...

```

Output                                     directory                                     is
/home/db2inst1/sqllib/db2dump/FODC_Perf_2011-02-22-08.23.35.824594
Open db2fodc.log in that directory for details of collected data

```

3. db2service.perf1

刚才我们提到了，db2fodc 存在于 9.5 及以后的版本。那么对于 8/9.1 的用户，如果发生了性能或者挂起的问题，有没有简单的脚本可以运行呢？

正如性能章节中所提到的，对于一般的长时间性能数据收集，每小时一次或者半小时一次的快照会非常有效。但是当问题突然发生的时候，我们需要提高数据收集的频率做到更高的采样，然后还需要一些其余类似跟踪、进程栈等信息。

这类问题发生时所捕捉的数据，在 9.5 以后可以使用 db2fodc 收集，但是在 9.5 之前，就需要一些独立的脚本程序了。所幸的是，IBM 提供了这种脚本的下载，用户可以到 DB2 支持中心的首页搜索 db2service.perf1（如图 17.9 所示）：

http://www-947.ibm.com/support/entry/portal/Overview/Software/Information_Management/DB2_for_Linux,_UNIX_and_Windows

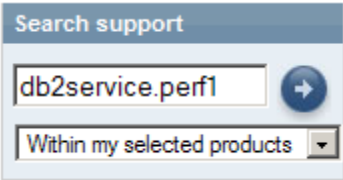


图 17.9 db2service.perf1 下载

表 17.1 是结果所对应的链接。

表 17.1

Windows	http://www-01.ibm.com/support/docview.wss?uid=swg21324857
HP-UX	http://www-01.ibm.com/support/docview.wss?uid=swg21324853
Solaris	http://www-01.ibm.com/support/docview.wss?uid=swg21324850
Linux	http://www-01.ibm.com/support/docview.wss?uid=swg21322385
AIX	http://www-01.ibm.com/support/docview.wss?uid=swg21324849

对于不同的平台，用户可以使用不同的脚本收集性能数据。

一般说来，该脚本所收集的数据类似 db2fodc，但由于主要为了 8/9.1 设计，在 9.5/9.7 中不能针对线程模型收集所有有用的信息，因此在 9.5/9.7 中，用户需要使用 db2fodc，而不是 db2service.perf1。

4. db2service.mem1

对于性能相关信息，IBM 同样也有一个现成的数据收集脚本，即 db2service.mem1。但是由

于各种原因，该脚本并没有网络上的下载。

当用户需要收集内存相关信息时，请联系 IBM 技术支持人员获得针对当前平台的脚本程序。

17.7 IBM 服务支持体系

□ -----

不知道各位是否寻求过 IBM 电话支持？如果购买了 IBM support 服务，当出现问题的时候，可以向 IBM 寻求帮助，即开 PMR(Problem Management Record)。国内 DB2 服务流程如下：

(1) 打 800 电话：8008101818，根据电话提示，输入相应的产品支持按键。接线员核对 PPA 信息（即客户名称和购买的服务号，每个购买 IBM 电话支持的客户，IBM 授予一个 PPA 号码），当核对无误后，转给 DB2 一线技术支持，或称 Level 1。

(2) Level 1 工程师与客户进行沟通，详细了解出现的问题，并给出建议（如果能给的话）。当需要更多信息时，会请客户收集。必要时，有时 Level 1 会协助开一个 PMR 号，用于追踪问题。

(3) 当 Level 1 对问题进行初步诊断后，如果可以解决，会直接反馈给客户。如果问题比较棘手，就可能将问题升级到 Level 2 支持小组。大部分情况，Level 2 不会直接和客户沟通，而是通过 Level 1。当然，有些复杂的问题，也不排除 Level 2 直接和客户交流。

(4) 当 Level 2 发现问题可能是 DB2 的 bug 时，就会转向 Level 3，Level 3 根据情况进行错误修复。

(5) 国内 800 的正常工作时间是早 8:30-晚 5:30。对于紧急的 case，比如宕机，IBM 提供 7*24 小时支持。

当然，800 是远程服务，而且价格不便宜，可以根据实际情况向 IBM 或第三方服务公司购买一些现场服务，以便当出现紧急问题时，可以获得快速响应。

17.8 小结

□ -----

在本章中，我们主要讨论了几种不同类型问题的分析思路，其中包括日志信息、宕机、挂起及命令错误信息。

针对不同类型的错误，其信息的收集与分析思路也有着很大的不同。我们并不能使用分析宕机的思路来研究挂起的问题，也不能用分析死锁的思路来研究错误信息问题。

因此，在分析问题之初，就要根据问题的类型使用不同的思路来细化，如果让我们最后做一个总结，则可以认为：

- 日志信息错误：从错误点起始向前，根据时间与 PID/TID 得到该进程（线程）的第一个问题点，然后根据整体框架去理解问题的根源，千万不要仅仅依赖一两条错误信息匆忙下结论。
- 宕机：要从错误点起始，找到第一个造成宕机或者坏页的日志，如果是非法内存访问问题，则需要查看进程栈；如果是坏页，则需要恢复或者手工修复。
- 挂起：检查进程线程堆栈，主要的思路是要弄明白当前系统进程线程之间的等待关系。
- 命令错误：通过 SQLCODE、db2diag.log 和 db2trc 深入问题。

17.9 判断题

（1）阅读 DB2 日志时，所有的 WARNING 信息都需要额外关注。

T: 正确

F: 错误

（2）出现问题时，最好的办法就是重启机器。

T: 正确

F: 错误

（3）不一定所有的问题都会显示在 db2diag.log 日志中。

T: 正确

F: 错误

（4）系统挂起时，DB2 会自动在后台收集诊断信息。

T: 正确

F: 错误

（5）开启 DB2 跟踪会对性能产生极大的影响。

T: 正确

F: 错误



数据库安全

不论从技术上还是公司的管理上来看，系统安全都可以说是一个自成体系的领域。在很多公司，系统安全部门可能完全独立于数据库管理或是开发部门，这种系统安全部门的职责是监控整个公司管理流程与 IT 系统设施，确保没有技术上或者人为的安全漏洞。

本章主要介绍 DB2 提供的安全机制：认证、权限控制和审计机制，并通过几个案例介绍了安全权限和审计的实施步骤。本章内容安排如下：

- 安全概述。
- 认证机制。
- 权限控制。
- 审计机制。
- DB2 安全最佳实践。

18.1 安全概述

安全是个很大的课题，本章我们只讨论 DB2 数据库安全。数据是企业的生命线，特别是银行、保险、通信等一些行业，数据库里保存了大量的客户信息，假如这些关键信息被某些不法分子获取，并对数据进行修改、删除、复制、传播等，将给企业和个人造成巨大损失。

数据安全威胁不仅来自外部，也可能来自内部，正所谓日防夜防家贼难防。在国内某些企业，安全意识比较薄弱，安全机制还不够健全，某些人的权限很大，假如没有机制记录他们的操作行为，而仅凭自觉和道德约束，难免会出现重要数据被更改的情况(比如在自己的存款账户上加几个 0)。制定严密的可操作的安全策略是每个企业必须严肃对待的一件事情。

在安全领域，有个 4A 理论：Authentication(认证)、Authorization(授权)、Account(用户/账号)和 Audit(审计)。认证是第一道屏障，通过验证用户身份，判断是否可以通过；权限是指用户认证通过后，能够执行什么操作，比如有的用户有读权限，有的用户可以增删改；审计类似飞机的黑匣子，能够记录一些重要的操作，是内控的有效手段（但审计永远都是事后审计，只能追溯问题原因）。

DB2 提供了一个非常好的安全模型支持 4A 理论。接下来我们将详细介绍 DB2 提供的认证、权限和审计机制。

18.2 认证机制

认证是验证用户身份的过程，可以把数据库想象为一个房子，认证（身份信息）就是房子的钥匙，只有通过认证，才能进入房子，否则就被挡在外面。DB2 支持几种形式的认证：操作系统用户名/密码认证、Kerberos 认证和第三方安全插件。

图 18.1 是当客户端发出 `db2 connect to testdb user aa using password` 命令时，服务端进行认证的过程：

- ①在数据库连接时提供用户名/密码，这些信息会提交给数据库服务器；
- ②数据库服务器将用户名/密码传递给安全插件，安全插件调用操作系统 API 验证用户名/密码是否正确；
- ③操作系统验证用户名/密码，并将结果返回给安全插件，安全插件再返回给数据库服务器；
- ④如果安全检查返回的信息表明用户名/密码没有通过操作系统认证，则返回错误信息给连接用户。如果用户名/密码正确，将会加载这个用户所属的组信息，并通过 `syscat.dbauth` 验证该用户/组/PUBLIC 是否有连接权限，如果有，则允许连接。

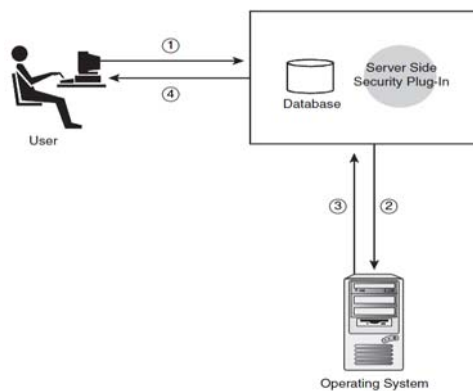


图 18.1 DB2 认证过程

认证方法的配置是通过实例参数 `AUTHENTICATION` 控制的，缺省值是 `SERVER`，如果希望更严密的认证控制，可以使用 `SERVER_ENCRYPT` 或 `DATA_ENCRYPT`。

```
inst20@db2server:~> db2 get dbm cfg | more
Database manager authentication (AUTHENTICATION) = SERVER
```

需要再次强调的是，DB2 连接时使用的用户名必须是操作系统用户，DB2 自身并不能创建用户。

18.3 权限控制

通过认证仅仅是过了第一道门槛，进来后能做什么就要看权限了。权限是保护数据的最主要机制之一，权限的设置需要遵循以下两个原则。

- **最小权限原则：**最小权限是指访问者只能访问职责范围内的必要数据，而不能超越，这样就将权限控制在很小的粒度分析，降低了风险。举例来说，`EMPLOYEE` 表包含了企业的员工基本信息和工资信息，从安全和隐私的角度，可以创建一个视图，这个视图只能访问查询者本人的信息，而无法访问其他人的数据。另外，可以将表的某些字段更新权限赋给某些用户/组，而不是整张表的更新权限。
- **职责分离原则：**该原则的道理很简单，就是一个事务/操作交给不同的人来处理，每个人负责独立的一部分，而不是整体，这样可以防止权力过分集中。比如，国内很多 IT 部门已经将系统管理员（SA）和 DBA 职责分离、将 DBA 和安全管理分离。对于涉及很多部门的应用来说，进行明确的职责分离更加重要。职责分离需要从整个公司的安全制度和法规来考虑。

以上两个原则应该作为安全权限控制的基本准则，在制定安全策略时，应该坚决贯彻。很多时候，安全问题并非技术问题，更重要的是公司流程和制度建设，以及完善的安全管控体系。

DB2 的权限大致分为两类：一类为管理类权限（`Authority`），针对实例和数据库层的权限控制；另一类为对象特权（`Privilege`），针对数据库对象，如表空间、模式、表、索引、存储过程等细粒度授权。

18.3.1 管理权限 ■ ■ ■

管理权限包括实例级和数据库级权限，图 18.2 为管理权限的层次关系，其中包含 4 个实例级权限，分别是系统管理权限（`SYSADM`）、系统控制权限（`SYSCTRL`）、系统维护权限（`SYSMAINT`）和系统监控权限（`SYSMON`）。这 4 个权限也可以理解为 4 种管理角色。

- `SYSADM` 是最高层的管理权限，能够控制实例下创建的各种资源，也用于对数据库的 `SECADM` 权限进行授权/撤销。相当于 UNIX/Linux 的 `root` 用户，高高在上，无所不能。

- SYSCTRL 拥有最高层的系统控制权限，能够执行实例、数据库和对象操作，但无法查看数据。
- SYSMOINT 拥有第二层的系统控制权限，用于执行数据库上的维护操作，但无法创建新的数据库，也无法查看数据。
- SYSMON 是底层的系统控制权限，主要用来监控实例和数据库，无法操作数据库对象，也无法查看数据。

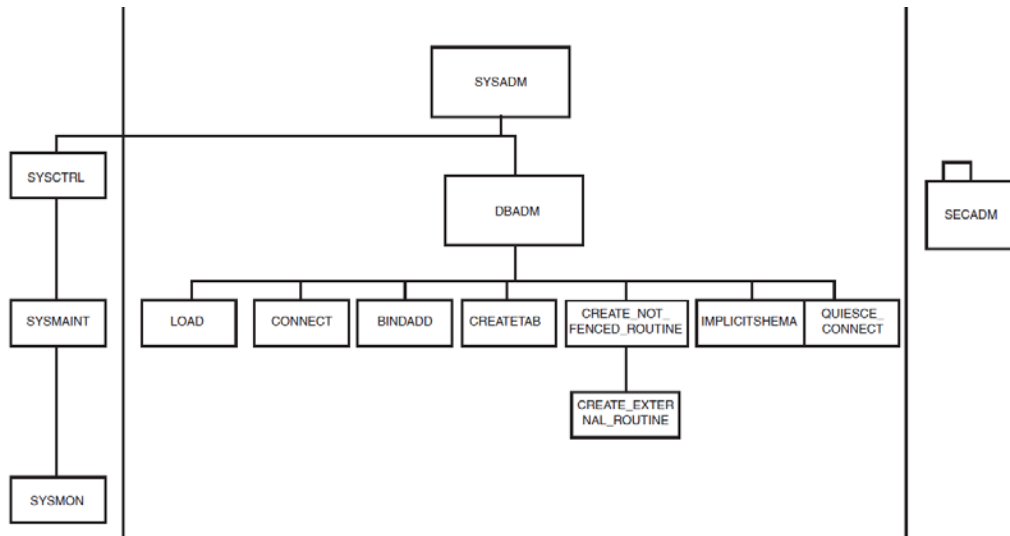


图 18.2 DB2 管理权限层次关系

这几个实例级权限的设置需要系统管理员的配合，因为每个参数值代表一个操作系统组（group），系统用户/组的创建和管理一般是系统管理员控制的。当确定了 group 后，DBA 就可以通过更改实例参数配置权限。需要注意一点，更改实例参数的时候，DB2 并不会检查该组是否存在，如果不存在，实例的启停可能会出现权限问题，更改参数后，需要重启实例才会生效。下例中，DB2ADMIN_GRP 组的所有用户均有 SYSADM 权限。

```

G:\Documents and Settings\db2admin>db2 update dbm cfg using SYSADM_GROUP
DB2ADMIN_GRP
SYSADM group name          (SYSADM_GROUP) = DB2ADMIN_GRP
SYSCTRL group name         (SYSCTRL_GROUP) =
SYSMAINT group name        (SYSMAINT_GROUP) =
SYSMON group name          (SYSMON_GROUP) =
  
```

在 UNIX/Linux 平台下，刚创建实例后，SYSADM_GROUP 的值是实例所属的主组（primary group）。其余几个参数的值均为 NULL。

在 Windows 平台下，在默认情况下，SYSADMIN_GROUP 是 NULL 值，这时本地管理组 Administrators 下的所有用户拥有 SYSADM_GROUP 权限。

数据库级别权限

数据库级别的权限包括 DBADM、SECADM、LOAD、CONNECT、CREATETAB、BINNADD、

CREATE_EXTERNAL_ROUTINES、CREATE_NOT_FENCED_ROUTINE、IMPLICIT_SCHEMA 和 QUIESCE_CONNECT。

- DBADM 是针对每个数据库的最高管理权限，该权限可以控制数据库下的所有对象，也可以访问数据。
- SECADM 权限主要针对数据库的安全，设置安全相关的对象，如安全策略、安全标签、安全标签组件等，SECADM 权限必须由 SYSADM 用户授予。从职责隔离的原则考虑，建议不要将 DBADM 和 SECADM 权限赋给同一人。如果一定要交给同一人管理，也要通过 AUDIT 做审计。
- CONNECT 是指认证通过后，有权限连接到数据库。
- BINDADD 允许用户创建 package。
- CREATETAB 允许用户创建新表。
- IMPLICIT_SCHEMA 允许用户隐式创建不存在的模式。
- LOAD 允许用户调用 LOAD 工具。

数据库级别的授权是通过 SQL 语句 grant 命令实现的，可以授权给 user，也可以授权给 group 或者 PUBLIC。以下是一些数据库权限的例子：

```
Connect to sample          -- 需先通过实例用户连接数据库
Grant dbadm on database to user test  -- 将 dbadm 权限赋给 test 用户
Grant connect, bindadd, createtable on database to user test1;  -- 将 connect、bindadd、
createtab 赋给用户 test1
```

DB2 提供了一个命令可以获取当前连接的用户所拥有的管理权限。

```
inst20@db2server:~> db2 get authorizations

Administrative Authorizations for Current User

Direct SYSADM authority          = NO
Direct SYSCTRL authority         = NO
Direct SYSMAINT authority        = NO
Direct DBADM authority           = YES
Direct CREATETAB authority       = YES
Direct BINDADD authority         = YES
Direct CONNECT authority         = YES
Direct CREATE_NOT_FENC authority = YES
Direct IMPLICIT_SCHEMA authority = YES
Direct LOAD authority            = YES
Direct QUIESCE_CONNECT authority = YES
Direct CREATE_EXTERNAL_ROUTINE authority = YES
Direct SYSMON authority          = NO

Indirect SYSADM authority        = YES
Indirect SYSCTRL authority       = NO
Indirect SYSMAINT authority      = NO
Indirect DBADM authority         = NO
```

Indirect CREATETAB authority	= NO
Indirect BINDADD authority	= YES
Indirect CONNECT authority	= NO
Indirect CREATE_NOT_FENC authority	= NO
Indirect IMPLICIT_SCHEMA authority	= NO
Indirect LOAD authority	= NO
Indirect QUIESCE_CONNECT authority	= NO
Indirect CREATE_EXTERNAL_ROUTINE authority	= NO
Indirect SYSMON authority	= NO

18.3.2 对象特权 ■ ■ ■

管理类权限比较高，一般只有 DBA 才可拥有。对于其他用户，比如应用用户，只需要对某些表的访问权限，这时只需在对象级别授权即可。图 18.3 展示了 DB2 的各种对象特权，从左到右的图标含义依次为：数据库对象、特权，椭圆形图标代表查看特权的系统表。以模式对象来说，它有 Alterin、Createin 和 Dropin 三种特权，模式特权的授权信息存放在 syscat.schemaauth 系统表中。

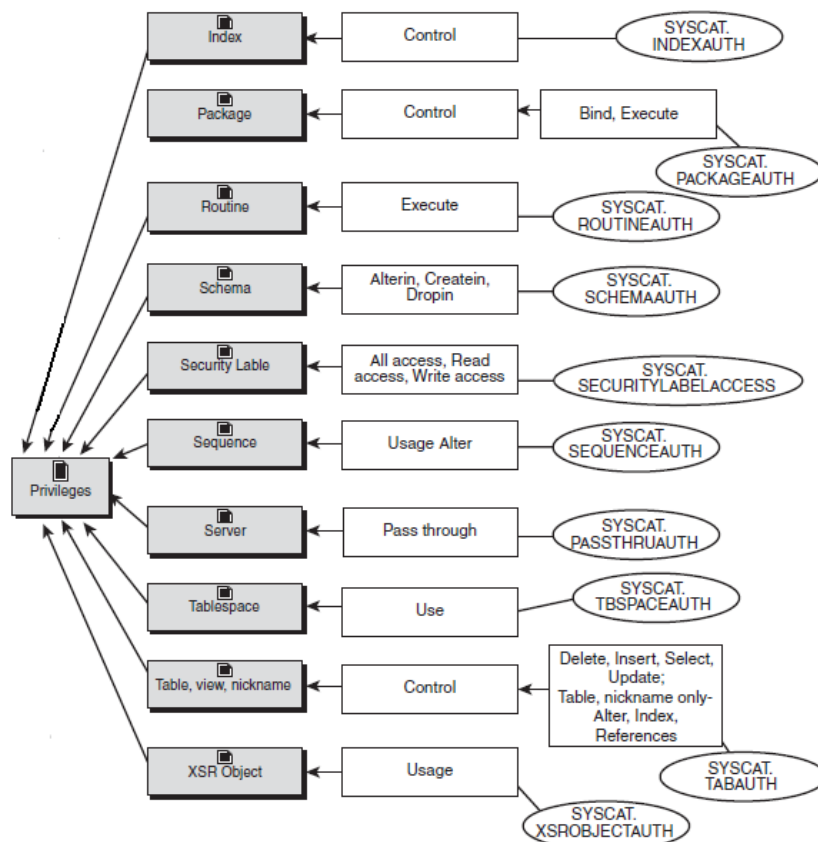


图 18.3 数据库对象特权

对象授权/回收的命令是通过 `grant/revoke` 实现的，以表授权为例：

```
inst20@db2server:~> db2 grant select,update,insert,delete on table inst20.employee
to user inst21
DB20000I The SQL command completed successfully.
```

DB2 是否支持对某个模式下的所有表授权？这是个非常普遍的问题，遗憾的是，DB2 并不支持。也就是说，DB2 只能对每张表分别授权，一个通常的做法是写脚本批量处理(从 `SYSCAT.TABLES` 视图获取表名)。但当新增了表后，仍然需要为新增表授权。在安全允许的范围时，可考虑为相应的用户授予 `DBADM` 权限，这样该用户自动拥有对象特权。

特权可以授予操作系统的某个组或用户。除此之外，DB2 还包括一个特殊的伪组，即 `PUBLIC` 组，任何用户都属于这个组。当创建数据库时，DB2 会缺省的把一些数据库权限赋给 `PUBLIC`，如 `CONNECT`、`BINDADD`、`CREATETAB` 和 `IMPLICIT_SCHEMA` 等。同时，也会赋予一些缺省的对象特权给 `PUBLIC`，如系统表的访问权限等。

下列语句查看 `PUBLIC` 的数据库权限：

```
inst20@db2server:~> db2 "select substr(grantee,1,32) as grantee, GRANTEETYPE,
BINDADDAUTH, CONNECTAUTH, CREATETABAUTH, DBADMAUTH, IMPLSCHEMAAUTH, LOADAUTH from
syscat.dbauth where GRAN-TEE='PUBLIC' "
```

GRANTEE	GRANTEETYPE	BINDADDAUTH	CONNECTAUTH	CREATETABAUTH	DBADMAUTH	IMPLSCHEMAAUTH	LOADAUTH
PUBLIC	G	Y	Y	N	Y		N

1 record(s) selected.

下列语句查看 `PUBLIC` 的对象特权限：

```
inst20@db2server:~> db2 "SELECT substr(OBJECTSCHEMA,1,32) as objectschema,
substr(OBJECTNAME,1,32) as objectname , OBJECTTYPE, PRIVILEGE FROM
SYSIBMADM.PRIVILEGES WHERE AUTHID = 'PUBLIC' "
```

在生产系统中，`PUBLIC` 拥有的这些权限可能造成一些安全隐患，根据最小权限原则，建议将这些权限取消，需要 `DBADM` 或 `SYSADM` 权限用户执行。

```
inst20@db2server:~> db2 revoke connect, bindadd, createtab, implicit_schema on
database from PUBLIC
```

从 DB2 9 开始，在创建数据库时提供了一个 `RESTRICTIVE` 选项，这意味着 DB2 不再把相关权限授予 `PUBLIC`。可通过以下参数查看数据库是否使用了该选项。

```
inst20@db2server:~> db2 get db cfg for sample |grep -i restrict
Restrict access = NO
```

18.3.3 权限设计案例 ■ ■ ■

某商业银行网银项目由 A 公司和 B 公司协作开发，A 负责网银本身应用，B 负责用户认证部分，两部分各有一个 DB2 数据库。在规划时，客户要求将这两个库放在同一个实例中，从运维和划分责任的角度考虑，要求每家公司创建和维护自己的库，并且不允许访问对方的库。同时，为了防止运维和诊断过程中的误操作，要求创建一个用户，该用户可以读取这几个数据库的表数据，但不能对数据进行增、删、改。

基于这个需求，通过实例用户 db2inst1 创建了两个数据库，并为每个库创建一个用户，将 DBADM 数据库管理权限赋给该用户。

```
--利用 db2inst1 创建了两个数据库 RADB 和 UIBS
CREATE DATABASE RADB ON '/database' ALIAS RADB USING CODESET UTF-8 TERRITORY CN
CREATE DATABASE UIBS ON '/database' ALIAS UIBS USING CODESET UTF-8 TERRITORY CN

--使用 root 用户创建两个用户：rauser 和 euser
mkgroup id=931 ragroup
mkuser id=731 pgrp=ragroup groups=ragroup home=/home/rauser rauser
mkgroup id=911 ebankgrp
mkuser id=711 pgrp=ebankgrp groups=ebankgrp home=/home/euser euser

-- 将 RADB 数据库的 DBADM 权限赋给 rauser
CONNECT to RADB user db2inst1 using password
Grant dbadm on database to rauser

-- 将 UIBS 数据库的 DBADM 权限赋给 euser
CONNECT to UIBS user db2inst1 using password
Grant dbadm on database to euser
```

DBADM 权限赋给相应的用户后，就可以通过该用户连接数据库，并创建表结构。

接着创建第三个用户 readuser，由于 DB2 无法批量授权，可通过脚本读取 syscat.tables 获得表名，然后写成脚本。

```
-- 使用 root 用户创建 readuser
mkgroup id=951 readgrp
mkuser id=751 pgrp=readgrp groups=readgrp home=/home/readuser readuser

--创建授权脚本 grantread.sh
DB=$1
DB_USER=$2
DB_PWD=$3

db2 connect to $DB user $DB_USER using $DB_PWD
db2 "select 'grant select on table ' || rtrim(tabschema) || '.' || tabname || ' to
readuser;' from syscat.tables where
tabschema='$DB_USER' " > readauth_radb.sql
```

```
grep grant readauth_radb.sql > radb_read.sql
db2 -tvf radb_read.sql
```

问题 1: 如何查看某个用户拥有哪些权限?

这个问题是许多人比较关心的, 比如某员工离职或换部门, 需要回收他的权限。在回收权限前, 首先要找到该用户, 然后查看他的实例权限、数据库权限和对象特权。步骤如下:

首先查看当前数据库有哪些用户, authidtype 的字段值含义: G=group,R=role,U=user。

```
-- 首先查看数据库有哪些用户
G:\Documents and Settings\db2admin>db2 "select char(authid,40) as authid ,
authidtype from sysibmadm.AUTHORIZATIONIDS "
```

AUTHID	AUTHIDTYPE
PUBLIC	G
SYSROLE_AUTH_DBADM	R
SYSROLE_AUTH_EXPLAIN	R
SYSROLE_AUTH_SECADM	R
SYSROLE_AUTH_SQLADM	R
SYSROLE_AUTH_WLMADM	R
SYSROLE_PRIV_AUDIT_ARCHIVE	R
SYSROLE_PRIV_AUDIT_LIST_LOGS	R
DB2ADMIN	U

9 条记录已选择。

然后查看这个用户是否有管理权限, 可以查看前述讲的 4 个实例参数查看所在的组中是否包含该用户。

接着查看该用户的数据库权限, 可以查看 SYSCAT.DBAUTH 视图:

```
-- 查看用户拥有什么数据库权限
G:\Documents and Settings\db2admin>db2 "select char(grantee,20) as grantee,
granteetype, BINDADDAUTH, CONNECTAUTH, CREATETABAUTH, DBADMAUTH, SECURITYADMAUTH
from syscat.dbauth "
```

GRANTEE	GRANTEETYPE	BINDADDAUTH	CONNECTAUTH	CREATETABAUTH	DBADMAUTH	SECURITYADMAUTH
DB2ADMIN	U	N	N	Y	Y	
PUBLIC	G	Y	Y	N	N	

2 条记录已选择。

最后查看该用户的特权:

```
-- 查看用户拥有哪些对象特权
G:\Documents and Settings\db2admin>db2 "select char(authid,25) as authid, authidtype,
```

```
PRIVILEGE, char(OBJECTNAME) as objectname from sysibmadm.privileges where
authid='DB2ADMIN' "
```

AUTHID	UTHIDTYPE	PRIVILEGE	OBJECTNAME
DB2ADMIN	U	DELETE	HMON_ATM_INFO
DB2ADMIN	U	DELETE	HMON_COLLECTION
DB2ADMIN	U	DELETE	POLICY
DB2ADMIN	U	DELETE	STMG_DBSIZE_INFO
DB2ADMIN	U	ALTER	ACT
DB2ADMIN	U	ALTER	ADEFUSR
DB2ADMIN	U	ALTER	CATALOG
DB2ADMIN	U	ALTER	CL_SCHED
DB2ADMIN	U	ALTER	CUSTOMER
...			

问题 2: 查看某个对象曾经授权给谁?

对于数据库权限和对象特权信息, 都记录在数据字典中, 要查看某个对象的权限, 只需到相应对象授权视图中过滤即可。

```
G:\Documents and Settings\db2admin>db2 list tables for schema syscat | find /i "AUTH"
COLAUTH          SYSCAT          V      2011-02-20-23.00.08.171008    -- 列权限
DBAUTH           SYSCAT          V      2011-02-20-23.00.08.593008    -- 数据库权限
INDEXAUTH        SYSCAT          V      2011-02-20-23.00.08.890008    -- 索引权限
LIBRARYAUTH      SYSCAT          V      2011-02-20-23.00.09.203008    -- 类权限
MODULEAUTH       SYSCAT          V      2011-02-20-23.00.09.234008    -- 模块权限
PACKAGEAUTH      SYSCAT          V      2011-02-20-23.00.09.359016    -- 包权限
PASSTHRUAUTH     SYSCAT          V      2011-02-20-23.00.09.421007    -- passthru
权限
ROLEAUTH         SYSCAT          V      2011-02-20-23.00.09.468007    -- 角色权限
ROUTINEAUTH      SYSCAT          V      2011-02-20-23.00.09.484017    -- 例程权限
SCHEMAAUTH       SYSCAT          V      2011-02-20-23.00.09.593017    -- 模式权限
SEQUENCEAUTH     SYSCAT          V      2011-02-20-23.00.09.796008    -- Sequence
权限
SURROGATEAUTHIDS SYSCAT          V      2011-02-20-23.00.09.890008    --
TABAUTH          SYSCAT          V      2011-02-20-23.00.09.921008    -- 表权限
TBSPACEAUTH      SYSCAT          V      2011-02-20-23.00.09.968017    -- 表空间权限
VARIABLEAUTH     SYSCAT          V      2011-02-20-23.00.10.078008    -- 变量权限
WORKLOADAUTH     SYSCAT          V      2011-02-20-23.00.10.171025    -- 负载权限
XSROBJECTAUTH    SYSCAT          V      2011-02-20-23.00.10.312016    -- XML 对象权
限
```

以下是查找 EMPLOYEE 表曾经赋给哪些用户什么权限:

```
G:\Documents and Settings\db2admin>db2 "select char(grantee,30) as grantee,
char(tabname,20) as tabname, controlauth, alterauth, deleteauth,
insertauth,selectauth,updateauth from syscat.tabauth where tabname='EMPLOYEE' "
```

GRANTEE	TABNAME	CONTROLAUTH	ALTERAUTH	DELETEAUTH	INSERTAUTH	SELECTAUTH	UPDATEAUTH
---------	---------	-------------	-----------	------------	------------	------------	------------

DB2ADMIN	EMPLOYEE	Y	G	G	G	G	G
----------	----------	---	---	---	---	---	---

18.4 审计机制

随着视频技术的发展，很多商场、高档住宅小区都配置了视频监控系统，用于实时监控和记录每天发生的情况，当出现问题时，可调出监控录像寻找线索、查明真相，视频监控已经成为现代社会重要的安保手段。DB2 的审计机制提供了与视频监控系统类似的功能，它可以把一些重要的操作记录下来，比如非法登录、更改权限、每天执行的 SQL 语句等，这样当出现问题时就可以分析审计日志，找到罪魁祸首。据统计，多达 80% 的数据泄露是由员工、业务伙伴或拥有对数据库的合法访问权的个人执行的。这些人获得对数据库的访问权后，便尝试读取他们本来不能访问的数据。这些类型的数据泄露很容易从审计数据中发现。

那么是不是把审计机制交给 DBA 就可以了？绝不是这么简单，除了技术层面，更重要的是安全控制流程、规划等方面，得到企业管理层和业务部门的支持。在技术层面实施之前，需要考虑：

- 通过审计，期望得到什么好处？
- 哪些数据库需要做审计？
- 需要审计哪些事件？审计 DB2 实例启停？审计对象创建或删除操作？审计权限变更？审计认证？审计所有 SQL 语句？是否需要每个表都做审计？
- 为什么要实施审计？是公司内控需求，还是有敏感数据？
- 谁对审计结果负责？谁定期查看审计结果？多长时间查看一次？

不应该把这些问题都丢给 DBA，对于大企业来说，这些问题应该是安全部、业务部门、管理部门和 IT 部门经过仔细讨论决定的。一旦需求明确，DBA 就可以通过相应工具进行配置、测试，并对审计结果进行初步分析。DB2 的审计功能会原系统造成一定影响，需要进行仔细的性能测试对比。

在 9.1 版本中，实例和数据库的审计都是通过系统管理员（拥有 SYSADM 权限用户）通过 db2audit 配置的。到了 V9.5，实例和数据库可以分开审计，实例级审计仍然由系统管理员配置，但数据库级别的审计是由安全管理员（拥有 SECADM 权限用户）通过新的 AUDIT 命令实现的，两个审计结果分别写到不同的日志文件中，但都由 db2audit 抽取和解析。在 9.5 版本中，audit 的性能与以前版本也有了很大的提升，对原有系统的影响更小。

以下以 9.5 版本为例，介绍数据库级的配置步骤。

（1）修改 AUDIT_BUF_SZ 实例参数大小，否则会严重影响性能：

```
$ db2 update dbm cfg using AUDIT_BUF_SZ 128
$ db2stop force
```

```
$ db2start
```

(2) 将 SECADM 权限授权给安全审计员 (本例是 SAUSER), 并用该用户登录数据库, 创建容纳审计数据的表 (通过<inst_home>/sqllib/misc/db2audit.ddl 脚本创建):

```
db2inst1@dpf1:~> db2 connect to sample
Database Connection Information
Database server          = DB2/LINUX 9.5.1
SQL authorization ID     = DB2INST1
Local database alias     = SAMPLE

db2inst1@dpf1:~> db2 grant secadm on database to SAUSER
DB20000I The SQL command completed successfully.

D:\>db2 connect reset
DB20000I SQL 命令成功完成。

db2inst1@dpf1:~> db2 connect to sample user sauser using password

Database Connection Information

Database server          = DB2/LINUX 9.7.1
SQL authorization ID     = SAUSER
Local database alias     = SAMPLE

db2inst1@dpf1:~/sqllib> cd misc
db2inst1@dpf1:~/sqllib/misc> db2 -tvf db2audit.ddl
...(创建了 8 张表, 审计数据将写到这 8 张表中)
```

(3) 用 SAUSER 用户创建 audit policy, 在 policy 中, 配置了几种审计事件, 然后启用审计策略。审计策略可以针对单个数据库、单个表、单个用户的操作:

```
db2inst1@dpf1:~> more auditpolicy.ddl
create audit policy auditdb
    categories audit status both,
                secmaint status both,
                sysadmin status both,
                objmaint status both,
                checking status failure,
                validate status failure,
                execute WITH DATA status both
    error type audit
;
db2inst1@dpf1:~> db2 -tvf auditpolicy.ddl
DB20000I The SQL command completed successfully.

db2inst1@dpf1:~> db2 audit database using policy auditdb # 使用以上策略审计数据库
DB20000I The SQL command completed successfully.
db2inst1@dpf1:~> db2 audit table testt1 using policy auditdb # 使用以上策略审计表
DB20000I The SQL command completed successfully.
```

```
db2inst1@dpf1:~> db2 audit SYSADM using policy auditdb      # 使用以上策略审计 SYSADM
```

(4) 如果我们需要查看审计日志, 需要通过下面两个命令归档并抽取出感兴趣的全部或部分审计日志 (用 SYSADM 权限用户执行):

```
db2inst1@dpf1:~/sqlllib/security> mkdir auditarchive
```

归档审计文件, 将其写入刚创建的 auditarchive 目录下:

```
db2inst1@dpf1:~/sqlllib/security> db2audit flush
AUD00000I Operation succeeded.

# 归档数据库审计, 将其写入 auditarchive 目录
db2inst1@dpf1:~/sqlllib/security> db2audit archive database sample node 0 to
/home/db2inst1/sqlllib/security/auditarchive/

Node      AUD      Archived or Interim Log File
      Message
-----
      0 AUD00000I db2audit.db.SAMPLE.log.0.20110311040552

AUD00000I Operation succeeded.
```

从归档日志文件中抽取审计内容:

```
db2inst1@dpf1:~/sqlllib/security> db2audit extract delasc to
~/sqlllib/security/auditdelasc/ from files
~/sqlllib/security/auditarchive/db2audit.db.SAMPLE.log.0.20110311040552

AUD00000I Operation succeeded.
```

检查生成的目录, 发现为每个分类生成一个 DEL 文件:

```
db2inst1@dpf1:~/sqlllib/security/auditdelasc> ls -alt
total 284
drwxr-xr-x 2 db2inst1 db2iadm1 4096 2011-03-11 04:09 .
-rw-rw-rw- 1 db2inst1 db2iadm1 0 2011-03-11 04:09 audit.del
-rw-rw-rw- 1 db2inst1 db2iadm1 115498 2011-03-11 04:09 auditlobs
-rw-rw-rw- 1 db2inst1 db2iadm1 0 2011-03-11 04:09 checking.del
-rw-rw-rw- 1 db2inst1 db2iadm1 0 2011-03-11 04:09 context.del
-rw-rw-rw- 1 db2inst1 db2iadm1 147281 2011-03-11 04:09 execute.del
-rw-rw-rw- 1 db2inst1 db2iadm1 232 2011-03-11 04:09 objmaint.del
-rw-rw-rw- 1 db2inst1 db2iadm1 302 2011-03-11 04:09 secmaint.del
-rw-rw-rw- 1 db2inst1 db2iadm1 0 2011-03-11 04:09 sysadmin.del
-rw-rw-rw- 1 db2inst1 db2iadm1 0 2011-03-11 04:09 validate.del
drwxr-xr-x 5 db2inst1 db2iadm1 4096 2011-03-11 04:08 ..
```

如果不是系统管理员而是安全审计员, 可以使用 SYSPROC.AUDIT_ARCHIVE 和 SYSPROC.AUDIT_DELIM_EXTRACT 两个存储过程替代 db2audit archive 和 db2audit extract 命令:

```
db2inst1@dpf1:~> db2 "call SYSPROC.AUDIT_ARCHIVE('/home/db2inst1/sqlllib/security
```

```

/auditarchive', -1) "

  DBPARTITIONNUM PATH          FILE          SQLCODE  SQLSTATE  SQLERRMC
  -----
  -----
                0          /home/db2inst1/sqllib/security/auditarchive
db2audit.db.SAMPLE.log.0.20110311060907      0 -      -      -

db2          "call          SYSPROC.AUDIT_DELIM_EXTRACT
(NULL, '/home/db2inst1/sqllib/security/auditdelasc', '/home/db2inst1/sqllib/securi
ty/auditarchive', 'db2audit.db.SAMPLE.log.0.20110311060907', '') "

Return Status = 0

```

(5) 对审计数据进行分析。为了便于分析，可将以上的 DEL 数据导入到之前创建的 8 张表中：

```

db2 "load from /home/db2inst1/sqllib/security/auditdelasc/validate.del OF DEL
MODIFIED BY DELPRIORITYCHAR LOBSINFILE INSERT INTO sauser.validate"
db2 "load from /home/db2inst1/sqllib/security/auditdelasc/context.del OF DEL
MODIFIED BY DELPRIORITYCHAR LOBSINFILE INSERT INTO sauser.context"
db2 "load from /home/db2inst1/sqllib/security/auditdelasc/audit.del OF DEL MODIFIED
BY DELPRIORITYCHAR LOBSINFILE INSERT INTO sauser.audit"
db2 "load from /home/db2inst1/sqllib/security/auditdelasc/checking.del OF DEL
MODIFIED BY DELPRIORITYCHAR LOBSINFILE INSERT INTO sauser.checking"
db2 "load from /home/db2inst1/sqllib/security/auditdelasc/sysadmin.del OF DEL
MODIFIED BY DELPRIORITYCHAR LOBSINFILE INSERT INTO sauser.sysadmin"
db2 "load from /home/db2inst1/sqllib/security/auditdelasc/objmaint.del OF DEL
MODIFIED BY DELPRIORITYCHAR LOBSINFILE INSERT INTO sauser.objmaint"
db2 "load from /home/db2inst1/sqllib/security/auditdelasc/secmaint.del OF DEL
MODIFIED BY DELPRIORITYCHAR LOBSINFILE INSERT INTO sauser.secmaint"
db2 "load from /home/db2inst1/sqllib/security/auditdelasc/execute.del OF DEL
MODIFIED BY DELPRIORITYCHAR LOBSINFILE INSERT INTO sauser.execute"

```

然后需要针对一些常见的审计需求和场景，为审计部门或 IT 管理部门生成有用的报告。常见的几种审计场景如下。

1. 审计失败的登录尝试或对未授权的数据的访问

审计失败的登录尝试对于防止未授权的访问十分关键，发现这方面的趋势有助于防止入侵。如果要审计所有失败的登录尝试，可以使用 VALIDATE 审计事件创建一个审计策略。可通过如下语句进行审计。

清单 1 （分析那些失败的验证尝试）：

```

db2inst1@dpf1:~/sqllib/security/auditdelasc> more xx.del
Select timestamp, appname, appid, userid, execid
from sauser.validate
where status < 0
order by execid, userid, timestamp;

```

清单 2 的查询将返回一个报告，该报告显示曾经尝试访问他们无权访问的对象的所有用户。通过检查 `authid`，可以确定用户向操作系统进行认证时所使用的 ID。应用程序名称可以帮助确定用户如何访问 DB2，对象名称可以帮助查明用户试图访问的特定对象。如果看上去是可疑的活动，那么可能需要审计 `authid`，查找可疑行为的更多模式。

清单 2（分析未经授权的访问）：

```
db2inst1@dpf1:~/sqlllib/security/auditdelasc> more yy.del
Select timestamp, appname, userid, authid, objschema, objname, objtype
from sauser.checking
where status < 0
order by userid, authid, timestamp;
```

2. 审计超级用户

有些法规要求审计系统管理员或安全管理员。审计部门会关心这些用户修改或运行了哪些对象，以及这些用户可能授予任何特权。审计者可能还想知道这些用户是否修改过任何审计配置。为满足这些审计需求，需要在审计策略中指定以下选项：AUDIT、EXECUTE、OBJMAINT、SECMAINT 和 SYSADMIN。创建好审计策略后，可以对要审计的权限 SYSADM、SYSMAINT、DBADM、SECADM 或特定的用户发出 AUDIT 语句。清单 3 用于查询数据库中授予或撤销的所有安全特权。

清单 3（分析特权的管理）：

```
Select authid, event, grantor, grantee, granteetype objtype, objschema, objname,
status from audit.secmaint
order by grantor, grantee;
```

3. 审计敏感数据

审计部门最常见的要求是审计敏感数据的能力，这个要求通常牵涉到员工或客户的数据隐私权，可使用审计跟踪。

为了审计敏感数据，需要创建一个审计策略，以确保在某些表上执行的所有 SQL 都被捕捉。这需要审计策略指定 EXECUTE 选项。创建好审计策略后，可以指出要用审计语句来审计哪些表。激活审计语句后，在指定的表上执行的 DML 语句将被记录下来。如果要生成一个显示在特定表上执行的所有 DML 语句的报告，可以使用清单 4 中的查询。

清单 4（分析某张表的所有访问）：

```
Select timestamp, authid, appname, rowsmodified, rowsreturned, stmttext, event
from sauser.execute
order by timestamp, userid;
```

18.5 DB2 安全最佳实践



安全是一个大课题，涵盖了从系统、应用、数据库等多个方面，是个繁杂的系统工程，以下是安全权限的一点最佳实践：

- 首先需要制定一个可操作的流程化的安全体系。这个体系的建设需要得到公司领导层的大力支持，否则在实施时会遇到很多阻力。安全体系的建设需要各部门的协调配合，如安全管理部、IT 运维中心、应用部门等。
- 权限的设置应遵循职责分离和最小权限原则，避免权力集中。
- 使用加密的认证方式，确保客户机和服务器之间的数据传输安全。
- 撤销 PUBLIC 隐式的权限和特权。
- 当员工离职或更换部门时，应回收他的所有权限，包括隐式权限。
- 注意操作系统的用户管理。
- 审计应该成为安全计划中的重要组成部分，并且需要定期或随机进行审计检查，防范风险。

18.6 其他安全技术增强



在最近几个版本，DB2 的安全机制得到了很大增强。比如 LBAC（Label Based Access Control），基本标签的访问控制，提供了更加细粒度的权限管理，可以将权限控制在行和列上，但实施起来相对比较复杂，适用于对权限要求非常细分的客户。另外，9.5 版引入了角色（role）的概念，实现了基于角色的访问控制，这样权限的管理更加精细。

18.7 小结



数据是企业的生命线，确保数据安全是任何一个企业都必须面对的重要课题。安全规划的制定和实施需要得到企业管理层和各部门的大力支持和配合，职责分离和最小权限原则应该作为安全规划的两个基本原则。DB2 提供了一个完整的安全框架：认证是第一道防线，用于识别用户的身份；权限是第二道屏障，包括管理权限和细粒度对象特权；审计则是防攻击、防非法授权、安全内控的有效手段，审计事件的制定和实施需要得到公司审计部门的大力支持，另外，审计可能会对性能造成影响，需要事先在测试机上进行性能对比测试。

18.8 判断题



- (1) DB2 维护着自己的用户列表。

T: 正确

F: 错误

(2) 一般建议用户将 **SYSADM** 权限赋予所有应用程序。

T: 正确

F: 错误

(3) 管理员可以对单独用户指定权限，也可以对用户组指定权限。

T: 正确

F: 错误

(4) 用户权限的优先权高于用户组权限。

T: 正确

F: 错误

(5) 管理员可以指定某用户能够在给定时段访问特定的数据表。

T: 正确

F: 错误

18.9 参考文献



《Understanding DB2 9 Security》

《IBM Information Center》

《Understanding DB2 Learning Visually With Examples 2nd Edition》