Amazon全五星评价畅销书，由拥有20余年工作经验的资深数据库专家撰写，权威性毋庸置疑

详尽阐述了100余条SQL语句优化的技巧和最佳实践，以及编写高性能SQL语句的标准和原则，包含大量案例，为解决各种复杂的DB2性能问题提供了解决方案

DB2 SQL
Tuning Tips for z/OS Developers

DB2 SQL性能调优秘笈

（美） Tony Andrews 著
陈勇 杨健康 译

NLIC2970843976

机械工业出版社
China Machine Press

DB2 SQL Tuning Tips for z/OS Developers

DB2 SQL性能调优秘笈

IBM PRESS

一本能帮助DBA和开发人员全面改善DB2数据库性能的调优秘笈，涵盖DB2 V9和V10

好的数据库性能往往是通过SQL代码进行优化而得到的。本书不仅为我们奉上了100多个与SQL代码优化相关的技巧和最佳实践，而且还深刻阐述了SQL编码的标准和原则，以及SQL语句调优的方法——15步调优法，可以说是无价之宝，几乎适用于任何DB2应用的优化。所有开发人员和DBA都可以通过本书的内容构建并优化DB2应用，以得到最优的性能。

本书的作者Tony Andrews即大名鼎鼎的“Tony the Tuner”，在数据库领域奋斗了20余年，在DB2性能调优方面积累了非常丰富的经验。在本书中，Tony清楚地告诉我们如何去应对各种与DB2相关的性能问题，如何去发现并消除性能上的瓶颈，而且还提供了大量经验性的解决方案。

本书中所讲解的技术从未在其他书中出现过，而且一般的DB2课程中也很少涵盖。

本书主要内容：

- 帮助开发人员了解在解决查询或程序中的性能问题时如何做，以及从哪里做起
- 提供大量编程和SQL编码示例
- 建立保证高性能SQL的标准和原则
- 实现高效的代码走查，确保遵循标准
- 关注耗费资源最多的少量SQL语句
- 找出能提供最大好处的简单解决方案
- 重写查询谓词使之更高效，从而优化性能
- 更好地理解SQL优化和Runstats统计
- 寻找机会，找到比优化工具更好的方法来调整代码，使之更高效
- 用COBOL应用优化SQL代码
- 高效地检查数据、行或表的存在性的技巧
- 使用Runstats的最新功能优化数据访问路径

PEARSON

上架指导：计算机/数据库

ISBN 978-7-111-42502-1

9 787111 425021 >

定价：39.00元

客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259
投稿热线：(010) 88379604

数字阅读：www.hzmedia.com.cn
华章网站：www.hzbook.com
网上购书：www.china-pub.com

DB2 SQL
Tuning Tips for z/OS Developers

DB2 SQL性能调优秘笈

（美） Tony Andrews 著
陈勇 杨健康 译

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

DB2 SQL性能调优秘笈 /（美）安德鲁（Andrews，T.）著；陈勇，杨健康译. —北京：机械工业出版社，2013.5
（华章程序员书库）
书名原文：DB2 SQL Tuning Tips for z/OS Developers

ISBN 978-7-111-42502-1

I. D… II. ①安… ②陈… ③杨… III. 关系数据库系统 IV. TP311.138

中国版本图书馆CIP数据核字（2013）第101045号

这是一本不可多得的 DB2 数据库性能调优秘笈，由拥有 20 余年 DB2 工作经验的资深数据库专家撰写，Amazon 全五星评价畅销书。本书不仅详尽阐述了 100 余条 SQL 语句优化的技巧和最佳实践、编写高性能 SQL 语句的标准和原则，以及 DB2 数据库性能优化的“15 步法”，而且还包含大量案例，为解决各种复杂的 DB2 性能问题提供了解决方案。

全书共 7 章：第 1 章总结了 116 条优化 SQL 语句的技巧和最佳实践；第 2 章讲解了 SQL 语句中提示的编写方法和技巧；第 3 章讲解了编写高质量 SQL 语句需要遵守的 SQL 标准和原则；第 4 章介绍了 SQL 程序走查；第 5 章用 2 个实例介绍了如何检查存在性；第 6 章介绍了 Runstats，用来得到一些统计信息；第 7 章讲解了 DB2 性能优化的“15 步调优法”，被誉为 DB2 性能调优领域的无价之宝，适用于各种情况下的性能调优问题。本书最后还有 2 个附录，分别介绍了谓词的重写示例和 DB2 SQL 中的术语。

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：关 敏
北京京师印务有限公司印刷
2013年6月第1版第1次印刷
145mm × 210mm · 5.375印张
标准书号：ISBN 978-7-111-42502-1
定 价：39.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 8837964 68995259 读者信箱：hzjsj@hzbook.com

译 者 序

拿到这本书，你可能会有这样的疑惑：这么一本薄薄的小书能告诉我们什么？真的能像作者所说的那样是一本不可多见的调优技巧秘笈吗？要回答这些疑问，最好的办法就是翻开书，自己研读，自己给出答案。

也许你在为应用的调优一筹莫展，也许你在惊异于调优专家的妙手回春，也许你在四处探求应用调优的秘诀，也许你有志于调优却不知从何做起……你手上的这本书会帮你解除疑惑，带你步入调优世界。

这本书的作者是著名的“Tony the Tuner”——Tony Andrews，他在 DB2 领域奋战了 20 多年，现在将他积累的经验凝结成书中的 100 多个 SQL 优化技巧，利用这些技巧，我们可以用最佳的方式构建和优化 DB2 应用。通过大量示例，他为我们展示了 DB2 SQL 调优中的种种问题和解决方案，强调了通常被忽略但意义重大的诸多方面，比如建立标准和原则，实现代码走查、重写查询谓词等等。相信这本书能帮你拨开谜团，享受成功调优的乐趣。

本书由陈勇、杨健康主译，程芳、吴忠望、张练达、陈峰、江健、姚勇、张莹参与了全书的修改整理，并完善了关键部分的翻译。全体人员共同完成了本书的翻译工作。不过由于水平有限，译文肯定有不当之处，敬请批评指正。

前　言

大多数关系型数据库调优专家都认为，对于众多访问关系型数据库的应用来说，性能问题主要是因为程序编写不当，或者 SQL 编写有问题。业内专家还指出，80% 的响应时间问题都应由性能差的 SQL 负责。我个人也非常赞同这种观点。我曾为很多 IT 工作室提供过性能和调优咨询服务，在所有这些工作室中，大多数性能问题都与修改应用和 SQL 代码直接相关，或者与增加和修改索引有关。正是因为这个原因，我一直在努力让开发人员了解各种 SQL 编程方法以及相关的性能问题。我认为应该让更多的开发人员了解如何阅读和分析 DB2 Explain 输出。另外，我相信对于每一个涉及 RDMS 的大型开发项目而言，项目组里都应该配备一个 SQL 技术专家。IT 行业中的 SQL 开发人员并不少，不过从我的经验来看，我发现其中只有不到 10% 的人真正了解 SQL 编程中存在的性能问题，或者真正知道该如何修正这些问题。如果有一位 SQL 技术专家坐镇，很多性能和逻辑问题就能在迁移到生产版本之前及早发现。

本书的出发点是为需要对 SQL 语句或程序调优的开发人员提供一个参考指南。一旦发现程序或应用运行很慢，大多数开发人员都会仓促地归咎于网络、数据库、系统或者过大的事务量等，把它们当作罪魁祸首。不过，很多情况下程序运行速度慢的直接原因其实在于他们的代码。希望本书能为这些开发人员提供帮助，使他们在迫不及待地给 DBA 或其他人打电话求助之前，能够先自己尝试来解决或改善遇到的性能问题。

还有另外一种情况：尽管 SQL（以及 SQL Explain）看起来还不错，但实际效率很差。针对这些情况，本书还提供了一些“调优

技巧”，这些技巧会采用与优化工具选用方法不同的方式对 SQL 进行调整优化。业界很多专家就经常使用这些技巧采用不同方式对原先性能表现很差的 SQL 语句进行优化，使它能更快地执行。有些情况下，修正性能问题的关键在于运行时间，这些调优技巧对于这种情况尤其适用。

在本书中，我总结了我在众多 IT 工作室所采用的一组 SQL 标准和原则。如果你的 IT 工作室还没有建立任何 SQL 标准，可以把我提供的这些标准作为不错的起点。很多工作室都会针对他们的应用另外增加更多原则。

有了标准和原则还不够，还需要确保遵循这些原则。我见过很多这样的 IT 工作室，开发人员向我展示了他们的 SQL 编程、COBOL 编程、Java 编程等等标准，但是并没有建立质量保证机制来确保这些标准确实得到实施。所有将进入生产阶段的程序都要经过某种代码走查或审查，以确保遵循标准，另外要保证所写的 SQL 确实高效。审查可以确保程序和 SQL 逻辑正确，还可以确保程序的设计符合需求。本书中专门有一章列出了完成 SQL 编程代码走查时需要询问和检查的各个问题。代码走查需要花一点时间，不过与可能在生产阶段暴露的性能或逻辑问题相比，代码走查绝对是值得的。

很多情况下，开发人员希望对 SQL 代码调优，但是不知道从哪里做起。我会告诉他们，首先要查看各个查询中的所有谓词，尽可能编写得更为高效。本书提供了一个附录，其中列出了性能不好的 SQL 谓词，以及如何更高效地重写这些谓词。开发人员要知道一个谓词是可索引谓词还是不可索引谓词，另外要知道它是 Stage 1 谓词还是 Stage 2 谓词，这很重要。这些问题将在本书中详细讨论。

与其说性能是一个 DB2 问题，还不如说它是一个关系型数据库问题。开发人员必须注意如何建立查询结构，以及如何基于这些查询设计应用代码。数据库分析人员和数据库建模人员必须注意如

何设计一个数据库应用。他们需要花一些时间做充分的分析。性能取决于环境、应用和需求。另外，对于性能来说，没有最好，只有更好。尽管大多数调优都在操作系统级和数据库级进行，但是实际上对应用代码调优才能获得最大的性能收益。如果应用需要使用SQL 从数据库中获取数据，这一类应用就非常适合进行调优。由于很少的 SQL 语句可能会占用大多数资源，所以 SQL 调优通常能得到显著的性能提升。不过，SQL 调优有时可能相当复杂。本书会提供一个起点，力求简单，使开发人员能轻松上手，让 SQL 驱动的程序和应用更高效地执行。

免责声明

本书中给出的调优技巧和建议是我个人的观点，这些都是根据我多年来完成 DB2 应用设计、编程和调优总结得出的。其中一些调优技巧可能并不完全反映 IBM（以及相关组织或这个领域的专家）的看法和观点。这些调优技巧基于我个人的经验积累而成，在我参与的各个应用中频繁使用，使应用得到更好的性能。这 100 多个技巧中，每一个技巧我都曾在对 DB2 SQL 应用调优时使用过，力图使查询和程序更高效地执行。

谈到一种方法好用或不好用时，很多人都会说“看情况”，相信每个人对此都不陌生。一定要谨记这一点。不要机械地照搬本书中的各个技巧，想当然地认为性能会立即改善。这些技巧是为了给开发人员指明方向，提供一些思路，以便他们改进查询或程序。所有人都要自己完成独立测试，验证本书中一些说法的正确性，只有在验证了这些说法确实正确的前提下，才能依此做出判断和决策。

致谢

我要向这些年来与我共事过的广大开发人员和 DBA 致以深深

的谢意。我是程序设计走查和程序代码走查的忠实拥护者。就我个人而言，多年来与其他同事坐在一起讨论、采纳他们关于程序改善的建议，期望改进得到最好、最高效的代码而顺利进入生产阶段，这些让我学到了很多很多。多年前，我们在一起开发一些庞大的应用时，他们就催促我把经验总结下来。后来我不断在会议上或者通过邮件为我们的开发小组提供关于编码技术的技巧，以改善他们的查询和程序性能。最后，终于有一天他们都劝说我应把这些技巧汇集成一本书。

我要特别感谢完成技术编辑的 Chuck Kosin 和 David Simpson。他们是我认识的 DB2 领域最棒的技术人员，他们能加入这个项目真是我的荣幸。非常感谢他们给出的大量评论和建议，正是这些意见让这本书更出色。我认识 David 已经有 3 年多了，我们曾在 Themis Inc 共事过。说心里话，他是我在这个行业见过的知识最渊博、最有经验的 DB2 专家之一。Chuck 有非常深厚的技术背景，之前就曾做过技术编辑，他与 Craig Mullins 共同编辑了《DB2 Developer's Guide》。

另外，非常感谢 IBM 出版社和我一同完成这本书的工作人员，感谢他们的理解和耐心。特别要感谢 Mary Beth Ray、Steven Stansel、Christopher Cleveland 以及参与这本书第一版的所有编辑和制作人员。要按照规定的所有格式和风格汇集成这样一本书，对我来说真不是一件容易的事。我发自内心地认为确实需要更高的 Word 处理水平才能胜任这个工作。

要感谢我任职的 Themis 公司（位于新泽西州的韦斯特菲尔德），感谢大家支持我完成这个项目。在我多年的培训和咨询生涯中，这是我遇到和共事过的水平最高、沟通最顺畅的一群技术人员。有了他们的协助，紧跟最新的版本与时俱进也成了一件乐事，出现问题时他们的第一手经验真是无价之宝。

另外，最重要的是，谢谢我亲爱的妻子 Jan。是她帮助我建立信心、消除恐惧，鼓励我把在这么多项目中学到和实现过的方法公

开出版，与大家分享。

如果你对本书有任何问题或评论，可以与我联系（tandrews@themisinc.com）。也可以写信给出版社转交给我。真诚期待你们的宝贵建议，实际上早在多年前做程序走查时我就已经感受到他人意见的重要性，从那时起就不敢再自以为是，衷心欢迎你的意见和反馈。本书的目的就是帮助广大开发人员了解各种高效 DB2 SQL 编程的方法。

目　录

第 1 章
SQL 优化技巧宝典 100+

开发人员编写 SQL 代码时，为了确保得到更好的性能，需要采取一些措施，下面会列出其中最有效的一些做法。在关系型数据库管理系统（relational database management system, RDBMS）中，通常 90% 的运行时问题源于 10% 的应用问题。与以往相比，如今的 DB2® 优化工具在选择正确的访问路径来满足 SQL 需求方面已经更胜一筹，不过仍不能保证百分之百正确。目前的所有调优工作都与 CPU 使用、I/O 和并发直接关联。本章的大部分内容都会针对 SQL 编码，不过所有内容都与性能和调优有关。

本书中给出的技巧和提示是为开发人员提供的一般最佳实践，适用于大多数 RDBMS。在这些数据库系统中，可能有一些基于成本的优化工具，它们会查看所有可用的选项，构建一个 SQL 查询集，并尽其所能给出最高效的访问路径。对于不同的优化工具，可用的选项可能或多或少，不过一般而言，本书中的技巧可以应用于任何数据库管理系统。

程序编写不当或者 SQL 语句编写不正确，这些通常就是导致性能低下的罪魁祸首。看起来学习和掌握 SQL 语言很容易，不过在性能方面，DB2 优化工具处理 SQL 语言时存在很多低效问题。开发人员、测试人员、DBA 和业务分析师等都能从本书中获益，因为本书特别针对如何编写合适的 SQL 来提高效率提供了很多技巧。

本书的重点是增强开发人员对调优方法的认识，更多地了解 IBM®DB2 关系型数据库环境中如何改善性能和调优。本书条理清

楚、通俗易懂，提供了大量开发技巧、建议和 SQL 编码示例，所有这些的最终目的都是为了得到更好的性能。

这里的 SQL 编码示例会用到一些 DB2 示例表，这些示例表在各个 DB2 网站中都能找到，另外在所有 IBM DB2 手册中都有提到。

导致应用、程序或查询性能低下最常见的原因是什么？

- 数据库设计不当
- 应用设计不当
- SQL 查询结构不合理
- 编目统计不当 / 不充分
- 系统类资源（如缓冲区池、RID 池和记录日志）分配不当

以下是解决未能合理优化的一些方法：

- 用不同的方法重写查询
- 采用不同方式重写查询中的谓词
- 改变数据库对象的设计
- 改变数据库分区的设计
- 为已经存在的表增加一个分区策略
- 管理、修改或增强编目统计，以涵盖非标准统计
- 修改系统资源

要记住，DB2 优化工具只能做到：

- 接受 SQL 查询
- 验证语法
- 确认 DB2 编目有效性
- 检查查询中对象的编目统计
- 对所有不同的访问路径计算成本
- 选择成本最低的访问路径
- 计算分区值，确定满足 SQL 数据请求所需的目标分区

作为开发人员，你应该掌握这个领域的大部分内容，这很重要，这样才能在分析性能问题时对调优有所认识。优化工具只能处

理编目中的信息，另外只能处理如何建构和设计查询及程序。

一旦找出性能问题，解决这个问题的最佳方法是什么？如前所述，需要对很多方面进行分析，而且其中很多都由开发人员掌控。关于要查找什么以及要做些什么，作为开发人员我们了解得越多，我们在公司里的价值就越高，地位就越重要。

接下来会为开发人员提供 100 多个技巧，在对特定的查询或程序调优时，这些技巧可以帮助你正确地考虑和分析。最前面的 20 到 25 个技巧可能是最常用的，其余的技巧没有特定的顺序，不过关于 DB2 V9 和 V10 的技巧会放在最后介绍。

1. 去除在谓词列上编写的任何标量函数

肯定有办法重写谓词，使得谓词列上不再使用函数。要记住：在 SQL 语句的 Select 部分对列编写标量函数是完全可以的，不过如果在 Where 部分中对列使用函数，就会自动将谓词变成不可索引的谓词，并成为一个 Stage 2 谓词。例如：

```
SELECT EMPNO, LASTNAME
FROM EMP
WHERE YEAR(HIREDATE ) = 2005
```

应当写为：

```
SELECT EMPNO, LASTNAME
FROM EMP
WHERE HIREDATE BETWEEN '2005-01-01' and '2005-12-31'
```

如上所示重写这个语句之后，DB2 可以选择使用列 HIREDATE 上的索引（如果存在这样一个索引）。但是如果在谓词中使用了 YEAR 函数，DB2 就无法使用该列的索引了。

DB2 V9 中，在列上创建索引时可以使用函数。V9 称之为表达式索引（Index on Expression）。用函数和 / 或表达式创建这种表达式索引时，DB2 会查看索引，如果它与 SQL 语句中的具体 SQL 函数和 / 或表达式匹配，就会使用这个索引。需要注意，即便如此，

仍然要尽可能像上面那样重新编写查询。维护更多的索引可能成本很高，因为这样一来 DBA 需要管理的索引也会更多，而且它们会增加插入、删除和一些更新的成本。例如：

```
CREATE INDEX XEMP4 ON
 EMP (YEAR(HIREDATE ) )
 USING STOGROUP ...
 PRIQTY ... SECQTY ...
```

2. 去除在谓词列上编写的任何数学运算

肯定有办法重写谓词，使得谓词列上不再使用数学运算。在 SQL 语句的 Select 部分对列应用数学运算是完全可以的，不过如果在 Where 部分中对列使用数学运算，就会自动将谓词变成不可索引的谓词。例如：

```
SELECT EMPNO, LASTNAME
FROM EMP
WHERE SALARY * 1.1 > 50000.00
```

应当写为：

```
SELECT EMPNO, LASTNAME
FROM EMP
WHERE SALARY > 50000.00 / 1.1
```

如上所示重写这个 SQL 语句之后，DB2 可以选择使用列 SALARY 上的索引（如果存在这样一个索引）。但是如果这个语句编写为直接对 SALARY 列应用数学运算，DB2 就无法使用该列的索引了。一定要做到：列本身（不加数学运算）放在操作符的一边，而所有计算都放在操作符的另一边。

在 DB2 V9 中，在列上创建索引时可以使用数学运算。

3. SQL 语句的 Select 部分只写必要的列

如果 Select 部分包含不需要的列，优化工具会选择 Indexonly = 'N'，

这会强制 DB2 必须进入数据页来得到所请求的特定列，这就要求更多的 I/O 操作。另外，由于这些多余的列可能是某些排序的一部分，这样一来，就需要创建和传递一个更大的排序文件，相应地会使排序的成本更高。对于排序来讲，其规模大小会直接影响成本。随着排序规模变大，成本也会更昂贵。

如果查询中涉及多个表，指定多余的列还会对优化工具选择何种联接类型产生影响。目前，z/OS®DB2 中有 4 种联接类型（嵌套循环联接、合并扫描联接、复合联接和星型联接），DB2 LUW 中还有一种哈希联接。优化工具会根据不同的原因选择各种联接类型。如果包含了根本不会用到的多余的列，优化工具就无法在联接处理中做出最佳选择。

4. 尽可能不用 Distinct

大多数情况下，Distinct 函数都会导致对最终结果集完成一次排序，因此，这就成为成本最昂贵的排序之一。Distinct 一直是 SQL 语言中成本最高的函数之一。不过，对于 DB2 V9，优化工具会尽量利用索引来消除为确定唯一性所带来的排序，其方法类似于目前用 Group By 语句完成优化时的做法。不过，实际上不必在 SQL 中使用 Distinct，完全可以使用其他方式重写查询来得到同样的结果，这样做往往更为高效。开发人员现在都很喜欢用 Distinct，很多人在所有语句上都会加上 Distinct 来确保不出现重复。不过这种代码的效率很低。在对应用调优时，我首先要做的事情之一就是审查源代码，找出所有包含 Distinct 的语句，了解这些查询是否确实会得到重复的结果，查看这个 Distinct 是否会导致一个排序，然后重写这些语句。通过对每晚批量作业的全面分析，就能很容易地大幅减少批量处理周期时间。不过要记住，如果查询中包含 Distinct，只要不会因此导致执行一个排序，这个查询可能也很高效。

要从结果集消除重复，可以尝试下面的做法：

- 使用 Group By（V9 之前），这会利用关联索引来消除为确定唯一性所导致的排序。
- 使用一个 In 或 Exists 子查询重写查询。如果某个表可能导致重复（由于是一对多关系），但是这个表中并不包含结果集将返回的数据，这种情况下这种方法就很适用。

例如，给定当前参与项目的一组员工。其中很多员工可能同时参加了多个项目，不过我们希望他们只出现一次。以下查询：

```
SELECT DISTINCT E.EMPNO, E.LASTNAME
FROM EMP          E,
     EMPPROJACT   EP
WHERE E.EMPNO = EP.EMPNO
```

可以重写为：

```
SELECT E.EMPNO, E.LASTNAME
FROM EMP          E,
     EMPPROJACT   EP
WHERE E.EMPNO = EP.EMPNO
GROUP BY E.EMPNO, E.LASTNAME
```

也可以重写为：

```
SELECT E.EMPNO, E.LASTNAME
FROM EMP  E
WHERE EXISTS
         (SELECT 1
          FROM   EMPPROJACT  EP
          WHERE E.EMPNO = EP.EMPNO)
```

还可以重写为：

```
SELECT E.EMPNO, E.LASTNAME
FROM EMP  E
WHERE E.EMPNO IN
         (SELECT EP.EMPNO
           FROM   EMPPROJACT  EP)
```

5. 尽量将 In 子查询重写为 Exists 子查询

In 和 Exists 子查询可以生成同样的结果，不过它们的做法截然不同。通常它们的表现各有优劣，这取决于实际的数据分布。例如：

```
SELECT E.EMPNO, E.LASTNAME
FROM EMP E
WHERE E.EMPNO IN
      (SELECT D.MGRNO
        FROM DEPARTMENT D
        WHERE D.DEPTNO LIKE 'D%')
```

也可以写为：

```
SELECT E.EMPNO, E.LASTNAME
FROM EMP E
WHERE EXISTS
        (SELECT 1
          FROM DEPARTMENT D
          WHERE D.MGRNO = E.EMPNO
            AND D.DEPTNO LIKE 'D%')
```

这些相关子查询和非相关子查询有不同的处理优势。现在 DB2 V9 可能会把子查询转换为它认为更高效的类型，特别是当一个子查询无法转换为联接时，V9 就会转换子查询类型。DB2 可能选择将子查询转换为联接来处理重复结果。这样一来，在查看 DB2 Explain 时，可能看不到之前编写的那个子查询，而是会看到一个联接或者另一种类型的子查询，这可能会让人有些困惑。

DB2 的做法是根据成本来关联、解除关联或转换为一个联接。IBM Data Studio 工具可以显示优化工具完成的查询转换。

6. 确保宿主变量定义为与列数据类型匹配

如果一个列定义为一个小整数，那么与它比较的宿主变量也应当声明为相同的定义（例如 COBOL 中声明为 S9(4) comp）。这在 DB2 V8 和 V9 中已经有显著改进，在这两个版本中，DB2 会更高效地处理进行比较的不同数值数据类型和不同字符串。DB2 中可

定义的数据类型并非在所有开发语言中都可用，在过去这会带来一些问题。不过从一般经验来看，将数据类型声明为与列定义匹配，可以确保最高的性能和最大程度的优化。例如，如果一个列定义为整数数据类型，那么谓词中包含比较值的宿主变量也应该定义为整数（而不是小整数、小数、浮点数，等等）。

COBOL程序员应当尽可能利用已创建的DCLGEN宿主变量来确保完全匹配。DCLGEN表示声明生成器（Declarations Generator），这是COBOL以及很多其他语言生成DB2 SQL数据结构的一个便利工具。对于COBOL，它会生成一个表声明（EXEC SQL DECLARE TABLE），另外会生成对应表中各列的宿主变量的COBOL定义。

7. 由于优化工具处理“或”逻辑可能有问题，所以尽量采用其他方式重写

尽可能分离谓词。下面的例子可以说明这一点：

```
SELECT DEPTNO, DEPTNAME
FROM DEPT
WHERE (ADMRDEPT   = 'E01'
                AND DEPTNAME LIKE 'BRANCH%')
   OR (DEPTNO =   'D01'
                AND DEPTNAME LIKE 'BRANCH%')
```

也可以写作：

```
SELECT DEPTNO, DEPTNAME
FROM DEPT
WHERE (ADMRDEPT = 'E01' OR
             DEPTNO =   'D01')
  AND  DEPTNAME LIKE 'BRANCH%'
```

这里的关键是：可以看到这两个例子有相同的逻辑。

很多情况下，这会使优化工具从不可索引谓词转移到可索引的谓词，或者可能使优化工具选择一个多索引处理访问路径，也

可能不再选择多索引处理。用“或”（OR）逻辑连接两个谓词会创建非布尔项谓词，这意味着即使某个谓词为 false，也不能因此完全消除一行，所以通常最好避免用 OR 连接谓词。也许一个谓词为 false，但另一个可能是 true。相比之下，使用“与”（AND）逻辑来连接谓词通常更为高效。不过，在这种情况下，它能为优化工具提供更多选择。

如果谓词用“OR”连接，首先要把谓词写为与表中最大数目的行匹配，这很重要。因为一旦某一行满足条件，OR 列表中的谓词计算就会结束。

用 OR 逻辑连接简单谓词时，得到的复合谓词会在简单谓词第 1 阶段或第 2 阶段中相对高的阶段完成计算。例如：

```
SELECT EMPNO, LASTNAME, SALARY
FROM EMP
WHERE  WORKDEPT = 'D01'          -- Stage 1 Indexable
   OR EDLEVEL <> 16              -- Stage 1 Non Indexable
```

由于第二个谓词是一个 Stage 1 不可索引谓词，而另一个简单谓词是一个 Stage 1 可索引谓词，所以整个复合谓词会成为一个 Stage 1 不可索引谓词：

```
SELECT EMPNO, LASTNAME, SALARY
FROM EMP
WHERE  WORKDEPT = 'D01'          -- Stage 1 Indexable
   OR YEAR(HIREDATE ) = 1990     -- Stage 2
```

由于第二个谓词是一个 Stage 2 不可索引谓词，而另一个简单谓词是一个 Stage 1 可索引谓词，所以整个复合谓词是一个 Stage 2 不可索引谓词。

8. 确保所处理的表中数据分布和其他统计信息正确并反映当前状况

可以在各个指定表和关联索引上执行 Runstats 工具，这样就能

确保所处理的表的数据分布和其他统计信息正确并且反映当前的最新状况。这个工具会在系统编目表中加载某个表和/或关联索引特性的有关信息。另外它还会提供数据分布信息，优化工具在选择访问路径时会查找这些信息。下面是 Runstats 工具提供的部分信息：

- 表大小（行数）
- 索引列的基数
- 某些列按值统计的行频率信息
- 表文件的物理特性
- 分区表按分区统计的信息

如果没有加载表统计信息，或者表统计信息已经过时，优化工具可能会选择并不特别适用于这些表的访问路径。这可能会影响性能。例如，如果一个表包含 500 万行，但 Runstats 是在表包含 20 万行时运行的，那么 DB2 仍然会认为这是一个小表（只有 20 万行数据），而对它包含 500 万行一无所知。

在系统编目表中，有一个名为 STATSTIME 的列，它会提供对这个表最后一次运行 Runstats 的准确日期和时间。如果这个列包含的值为 0001-01-01-00.00.00.000000，则说明对这个表没有执行过 Runstats，或者 Runstats 信息已经重置。对于 DB2 LUW，如果从未运行 Runstats，STATS_TIME 将为 null（关于 Runstats 的更多信息请参见第 6 章）。

下面是 Runstats 要更新而且优化工具要查看的一些系统编目表：

- SYSIBM.SYSCOLDIST
- SYSIBM.SYSCOLDISTSTATS
- SYSIBM.SYSCOLSTATS
- SYSIBM.SYSCOLUMNS
- SYSIBM.SYSINDEXES
- SYSIBM.SYSINDEXPART
- SYSIBM.SYSINDEXSTATS

- SYSIBM.SYSTABLES
- SYSIBM.SYSTABLEPART
- SYSIBM.SYSTABLESPACE
- SYSIBM.SYSTABSTATS

会对各个表生成以下统计信息：

- 所有表和索引都会有基本统计信息。
- 要为 Where 和 Order By 语句中常用的列生成统计信息。对于 V8，运行 Runstats 时可以使用 Column (All) 作为参数。
- 要为数据偏斜（不均匀）的列生成频率值统计信息。
- 要为所有关联的（两个或多个）列生成关联统计信息。
- 要为所有存在区间偏斜和区间谓词的列生成直方图统计信息。

9. 尽可能用 UNION ALL 取代 UNION

大多数开发人员会在需要时写 UNION，这往往会导致执行一个排序来消除重复。不过，不同的查询之间通常并不存在重复，所以 SQL 语句应该写为 UNION ALL，从而去除这种“做无用功”的排序。例如：

```
SELECT DEPTNO
FROM DEPT
WHERE MGRNO in ('000010', '000020', '000030')

UNION

SELECT DEPTNO
FROM PROJ
WHERE PROJNO = 'PA1000'
```

如果有 UNION 语句，DB2 最后会自动地执行一个排序来消除所有 DEPTNO 重复。不过，有很多这样的查询，在不同的查询之间可能根本不会出现重复。在这种情况下，如果写 UNION，尽管实际上没有重复记录，DB2 还是会执行一个无用的排序来查找重复。

另外，开发人员通常会把 UNION 作为一个安全网，以备万一出现重复。这并不是一个好的编码实践做法。开发人员应该充分了解他们的数据及关系，以确认是否确实会出现重复。写 DISTINCT 时也是这样。另外对于 V9 INTERSECT/INTERSECT ALL 和 EXCEPT/EXCEPT ALL 语句也同样如此。如果要消除重复，这些 INTERSECT 和 EXCEPT 语句也会执行排序。你要根据数据来确定具体需要哪一个。

10. 考虑使用硬编码还是使用宿主变量

要使用硬编码而不是使用宿主变量，如果程序员在 SQL 语句中使用了宿主变量字段，这一点就尤其重要。大多数 COBOL 程序都写为静态 SQL 程序而不是动态 SQL 程序。对于这些 COBOL 程序，执行 DB2 Bind 时，直至运行时之前优化工具都无法知道宿主变量中的具体值。所以，优化工具为 SQL 语句选择访问路径时，它会有一些默认的判定规则。

例如，假设一个表包含 100 万行。在这个表中，列 Status_Cd 上有一个索引。对这个表执行典型的 Runstats 时，优化工具会知道状态码（Status_Cd）有 3 个不同的值。运行一次特殊的 Runstats 之后，会为这个列指定频率值统计信息，DB2 就会了解到以下数据分布：

- 状态码值 A 包含 50% 的数据。
- 状态码值 B 包含 45% 的数据。
- 状态码值 C 包含 5% 的数据。

COBOL 程序包含以下 SQL 语句，其中程序中设置的值总是 C：

```
SELECT COL1, COL2, COL3
FROM TABLE
WHERE STATUS_CD = :WS-STATUS_CD
```

如果使用宿主变量，绑定时优化工具要确定一个访问路径，此

时它并不知道要处理什么值，不过它知道在这 100 万行中有 3 个不同的状态码值。其默认规则就是不论运行时是什么值，都大致访问 1/3 的数据。考虑到要访问 1/3 的数据，很多情况下它会选择对数据完成表空间扫描。

如果 COBOL 程序包含以下 SQL 语句：

```
SELECT COL1, COL2, COL3
FROM TABLE
WHERE STATUS_CD = 'C'
```

通过硬编码写入‘C’，优化工具可以进一步浏览系统编目表，查看到只有 5% 的数据的 STATUS_CODE 列值为 C，由于这个原因，选择访问路径时它可能会选择使用索引。

代码中要尽量使用硬编码值，这并不是一个标准，甚至不能算是 SQL 的编码实践原则。不过，很多情况下，通过使用硬编码值可以显著改变所选择的访问路径，从而使性能大为改善。如果某些列值的数据分布很不均匀，这种做法就尤其适用。要记住，通常需要运行一个特殊的 Runstats 来得到额外的分布统计信息。这些信息称为频率值统计信息。如果不能在程序中写入硬编码值，可以考虑使用 REOPT 绑定参数，它会在运行时使用宿主变量中的值对查询执行一次“再优化”。不过，要当心 REOPT，因为它会在程序执行时对每一个 SQL 语句进行再优化，这可能会增加执行时间，以至于抵消基于值再优化所节省的时间。

11. 尽可能减少 DB2 的 SQL 请求

性能调优中通常都要尽量减少对 DB2 的 SQL 请求，特别是批量程序，因为与其他程序相比，批量程序会处理更多的数据。每次向数据库管理器发送一个 SQL 调用都会带来开销，包括向 DB2 发送 SQL 语句的开销，以及从操作系统的一个地址空间到另一个地址空间以便 DB2 执行的开销。所以，一般来讲开发人员需要尽量

减少：

- 打开 / 关闭时间游标的次数。
- 随机的 SQL 请求数（在 DB2 监视器中称为同步读）。

很多开发人员会采用过程方式考虑和编写代码。利用 RDBMS 开发应用并编写 SQL 时，开发人员需要更多地从关系角度来考虑。也就是说，他们需要考虑“怎样才能向数据库管理器发送最少的 SQL 语句来得到所需的全部数据，而不是做这么多的过程调用？”对于 DB2 来说，尽管执行一条 Select 语句可能效率很高，但是向 DB2 发送这个语句本身还是会带来开销。

一方面要尽量把所有调用都放在一个游标中，另一方面还要保证能够维护和理解逻辑，这二者之间存在一个平衡。不过要记住，如果将程序中的 DB2 调用减少 10% 或者更多，程序绝对会运行得更快。以下原则会对你很有帮助：

- 编写一个多表联接，而不是把它分解为单个游标。你可能会先建立一个驱动游标（Driver Cursor），然后在获取记录行时分解和打开 / 关闭其他游标。这种做法很不好，不要采用这种方式编写代码。正确的做法是，应当把所有工作都写在一个语句中。
- 编写外联接，而不是随机地执行存在性检查。然后再在另一个表的一个列中检查 null 来确定存在性。
- 要想只返回存在的行，应当编写 exists 子查询，而不是随机地执行单独的存在性检查查询。
- 在 Select 中根据需要指定列格式，而不是执行其他 SQL 语句来达到这个目的。可以在 SQL 语句的 Select 部分中对列应用必要的 SQL 标量函数，为它指定所需的格式。
- 用一个语句得到所需的全部日期信息，而不要发送多个不同的语句。

例如：

```
SELECT CURRENT DATE + 7 DAYS,
       CURRENT DATE - 7 DAYS,
       LAST_DAY(CURRENT DATE)
FROM SYSIBM.SYSDUMMY1
```

12. 尽量将区间谓词重写为 Between 谓词

假设一个查询性能不佳，它的主要筛选谓词之一是一个区间谓词，如下所示：

```
WHERE HIREDATE  > :HV-DATE
```

如果数据库管理器没有使用 HIREDATE 列上的索引，你可能希望尽量让它使用这个索引，那么可以如下编写代码：

```
WHERE HIREDATE  BETWEEN  :HV-DATE and :HV-DATE2
```

只是要确保将第二个变量设置为某个极值，并将其包含在一个宿主变量字段中（这里设置为‘9999-12-31’）。注意，只有在使用宿主变量（静态 SQL 中）或参数标记（动态 SQL）时这种方法才奏效。优化工具很聪明，它知道实际上硬编码的‘9999-12-31’不会改变逻辑。不过，如果它看到一个变量，在运行时之前它都不知道变量的值，只能在设置值之前完成优化（或准备）。所以，这会哄骗优化工具认为 BETWEEN 谓词会筛选更多记录行（实际上并非如此）。

尽管不能保证，不过优化工具处理 Between 谓词和处理区间谓词的做法确实不同，它会采用不同的方式计算谓词的筛选率。开发人员可以大胆尝试，很多情况下这会改变优化路径。

说明： 如果 SQL 来自动态 SQL 语句，这种方法就不可行了，它只适用于来自静态 SQL 语句的程序（如 COBOL binds 或 Java ™ prepares）。对于动态 SQL 语句，直到执行时 DB2 才能知道宿主变量中的具体值是什么。

如果把代码从

```
WHERE HIREDATE  > ?
```

改为

```
WHERE HIREDATE  BETWEEN ? and '9999-12-31'
```

访问路径不会有任何改变，这是因为，利用硬编码的变量，DB2知道所写的是一个最大值，它会根据这个值完成优化。对于DB2来说，这与区间谓词没有任何差别。

说明：如果是硬编码的变量，与非硬编码变量相比，DB2能了解到更多信息。有时硬编码对于选择访问路径很有好处；不过有时则可能没有任何帮助。

不过，如果数据很不均匀（如本例），对优化工具来说就确实需要硬编码。

另外，DB2 V9特别针对区间偏斜或区间谓词（如Between、Like、>=和<=）中常用的列引入了直方图统计。

13. 考虑使用全局临时表

如果所处理的数据会一直保留在应用的逻辑工作单元中，可以考虑使用全局临时表（global temporary table，GTT），而不要反复建立或物化数据。如果在程序执行期间多次获取或物化同样的数据，可以把这些数据在一个GTT中加载一次，然后在代码的其他地方引用这个表。

例如，一个程序有多个游标，每个游标包含多个表联接。另外，每个游标都包含同样的表Table1，而这恰好是各个查询的驱动表。处理各个游标时，可能必须从数据库获取相同的行来完成Table1处理。这样一来，就会导致对相同数据额外的I/O处理。

所以，更好的做法是先获取Table1数据，把它插入到一个GTT中，然后在各个游标中引用这个GTT。这会消除对Table1的

重复 I/O。GTT 也很适合创建一个概要数据表，可以在逻辑工作单元中的其他查询中引用。

警告： 有两类 GTT：一类是 DBA 创建的 GTT，另一类是应用代码中声明的 GTT。使用 DBA 创建的全局表时要特别当心，因为它们不能有索引。把这些表加入一个多表联接时，可能会导致性能问题。不过，如果在应用代码中声明一个全局表，在其中加载数据之前，应用代码可以在这个表上创建索引。如果是这样，会随着数据的插入收集动态统计信息，而且由于可以使用这个索引和统计信息，联接通常会更高效。另外，要记住由于应用代码中声明的全局表有更大的灵活性，所有引用这些全局表的 SQL 语句会作为动态 SQL 执行，而引用 DBA 创建的全局表的 SQL 则作为静态 SQL 执行。

下面的例子展示了代码中如何声明和创建一个全局临时表。在应用代码中声明全局临时表时，所有者必须使用 SESSION：

```
DECLARE GLOBAL TEMPORARY TABLE   SESSION.TEMP_EMP
  (EMPNO       CHAR(6)              NOT NULL,
   FIRSTNME    VARCHAR(12) NOT NULL,
   MIDINIT     CHAR(1)              NOT NULL,
   LASTNAME    VARCHAR(15) NOT NULL,
   WORKDEPT    CHAR(3),
   PHONENO     CHAR(4)
  )
 ON COMMIT DROP TABLE      or
 ON COMMIT DELETE / PRESERVE ROWS
;

CREATE INDEX SESSION.EMPX1 ON SESSION.TEMP_EMP
       (LASTNAME  ASC)
 ;

INSERT INTO SESSION.TEMP_EMP
 SELECT EMPNO,
        FIRSTNME,
        MIDINIT,
```

```
        LASTNAME,
        WORKDEPT,
        PHONENO
FROM EMP
;

SELECT *
FROM SESSION.TEMP_EMP
 ;
```

14. 优先使用 Stage 1 谓词而不是 Stage 2 谓词

为了获取数据，DB2 使用了一种两阶段架构。数据管理器（Data Manager）称为第一阶段，即 Stage 1，照惯例这个阶段会应用简单的 SQL 谓词，另外可以使用索引来获取数据。有时，Stage 1 谓词也称为“可求值”（sargable）谓词，这是 IBM 的说法，表示可搜索参数（searchable argument）。如果数据管理器无法处理这个谓词，会把它传递到 DB2 的关系型数据服务（relational data service，RDS）部分，作为一个 Stage 2（剩余）谓词。

Stage 1 谓词的性能要优于 Stage 2 谓词，因为不同阶段间的数据传递在列值级进行。如果行在第 1 阶段已经限定，就会减少向第 2 阶段传递的行。索引可以进一步缩小搜索的范围，并减少处理。要确保 SQL 查询只请求真正需要的列，这样还可以限制返回结果集所需的处理量。

要尽量编写 Stage 1 谓词，而且应当是可索引的谓词。一般来讲，Stage 2 谓词的性能表现往往不太好，会消耗额外的 CPU 资源。可以参考 IBM 性能手册（V9 和 V10 可以查看“性能监视与调优”，V10 还可以参考“管理性能”），以确定谓词是第 1 阶段谓词还是第 2 阶段谓词，检查时要确保 DB2 版本正确。每个新版本通常都会把很多 Stage 2 谓词改为 Stage 1 谓词来得到更好的效率。一些 DB2 Explain 工具会把各个 SQL 语句中的谓词分为 Stage 1 谓

词和 Stage 2 谓词。

IBM 新推出的 Data Studio 是一个很棒的工具，可以用来解释查询，因为它能指出谓词是第 1 阶段还是第 2 阶段。这个工具可以从 IBM 网站免费下载，另外还可以购买升级版。对于开发人员来说，免费版本已经很完美了，它能够对外提供原本仅 IBM 可见的更多优化工具信息，这是前所未有的。

谓词可以分为以下几类：

- Stage 1 可索引谓词和 Stage 1 不可索引谓词
- Stage 2 不可索引谓词
- Stage 3 应用筛选谓词

第 1 阶段处理很了解查询中的对象，可以利用索引来完成处理。第 2 阶段会处理函数和表达式，不过不能直接访问表和 / 或索引。DB2 从查询中取出所有 Stage 1 谓词，处理这些谓词来得到数据，再将数据传递到第 2 阶段做进一步处理。Stage 3 谓词是返回到应用的谓词，由应用确定是否保留这一行。这些 Stage 3 谓词并不算是严格意义上的一类 DB2 谓词，不过有时会用来表示根据程序代码中 If 逻辑应用筛选逻辑来确定某一行是否已处理。

处理 Stage 1 可索引谓词总能得到最优的筛选性能。不过，有些情况下，尽管一个谓词列为 Stage 1 可索引谓词，但并不这样处理。例如，如果一个谓词列为 Stage 1 谓词，但是它的筛选出现在联接处理之后，这个谓词就会变成 Stage 2 谓词。另外还有一些类似的情况，即取决于 Stage 1 谓词如何使用或者它与什么进行比较，这个谓词可能会变成一个 Stage 2 谓词。

对于不可索引谓词，由于它们所采用的编写方式，不允许 DB2 使用相关列上的索引进行处理。Stage 2 谓词总是在数据获取之后应用，而不是在数据获取期间应用。很多 Stage 2 谓词可以采用不同方式重写，以便改写为 Stage 1 谓词。

Visual Explain 工具（V8）或 IBM Data Studio tool（V9）的当

前版本将查询中的所有谓词分为几类：匹配索引谓词、扫描索引谓词、Stage 1 谓词和 Stage 2 谓词。这个工具很棒，可以给出直观显示，使你能看出是否存在 Stage 2 谓词来进行分析和重写。V9 包含了 Visual Explain，这是其优化服务中心（Optimization Service Center）工具或 IBM Data Studio 工具的一部分。

附录 A 提供了一些例子，介绍了如何重写谓词。另外可以参见附录 B 中的 Stage 1 和 Stage 2 定义。

除此以外，还可以利用网站 www-306.ibm.com/software/data/db2/zos/v9books.html。其中提供了 IBM V8、V9 和 V10 手册的相应标签。另外可以在网上搜索“谓词处理小结”了解有关内容。

15. 记住（某些）谓词的顺序很重要

编写谓词时，要按表、按谓词类型从最受限到最不受限来编写。编写一个有多个谓词的 SQL 语句时，首先要写从结果集中筛选大多数数据的谓词作为第一个谓词。通过按从最受限到最不受限的顺序排列谓词，后面的谓词筛选的数据会更少。

默认地，DB2 优化工具会根据某些谓词类型选择它想最先处理的谓词。不过，如果查询包含多个谓词，而且这些谓词都是同一类谓词，就会按照编写的顺序来执行这些谓词。正是因为这个原因，正确地排列谓词的顺序很重要，要把筛选最多数据的谓词放在谓词列表的最前面。

这不算是一个能显著改善性能的调优技巧，无法帮助查询从几小时优化到几分钟或者从几分钟优化到几秒，不过，对 SQL 语句调优时，每一个小小的改善都会有帮助。

DB2 总是先应用索引谓词（而不论这些谓词的具体位置是什么）。接下来，不论按什么顺序编写，都会按以下顺序应用 Stage 1 不可索引谓词：

1. Equal 谓词
2. Range 谓词
3. In 列表
4. Like 谓词

最后再应用 Stage 2 谓词。

16. 多个子查询排序

一个 SQL 语句包含多个子查询时，要应用以下规则：

1. 子查询是完全关联或完全不关联时，就要按照最受限的放在最前面的顺序来编写，因为它们会按其编写的顺序进行处理。

2. 如果一个 SQL 语句同时包含关联和非关联子查询，不论查询中的实际顺序是怎样的，总是先执行非关联子查询。

3. 开发人员可以选择编写相同类型的子查询，并按他们选择的顺序排列谓词表，有时这会在运行时有不同的表现。与常规谓词相比，在 SQL 语句中从最受限到最不受限的顺序编写子查询对性能会有更大的影响，这在调优技巧 #15 中也有说明。

下面是非关联子查询的一个例子：

```
SELECT E.EMPNO, E.LASTNAME
FROM EMP E
WHERE E.EMPNO IN
      (SELECT D.MGRNO
        FROM DEPT D
        WHERE D.DEPTNO LIKE 'D%')
```

以下是关联子查询的一个例子：

```
SELECT E.EMPNO, E.LASTNAME
FROM EMP E
WHERE EXISTS
        (SELECT 1
          FROM DEPT D
          WHERE D.MGRNO = E.EMPNO
            AND D.DEPTNO LIKE 'D%')
```

17. 索引关联子查询

使用关联子查询处理一个 SQL 语句时有几点需要注意：

- 可能要执行多次关联子查询来满足 SQL 请求。考虑到这一点，必须使用一个索引来处理子查询以减少表空间扫描。
- 如果关联子查询会执行成千上万甚至上百万次，最好使用 Indexonly = Yes 确保使用一个索引来执行子查询。这可能需要对索引有所改变，或者要对一个已经存在的索引进行调整。

例如：

```
SELECT E.EMPNO, E.LASTNAME
FROM EMP E
WHERE EXISTS
            (SELECT 1
             FROM DEPT D
             WHERE D.MGRNO = E.EMPNO
               AND D.DEPTNO LIKE 'D%')
```

在这个例子中，会对 EMP 表中各个不同的 EMPNO 执行子查询。这个子查询会使用一个索引（MGRNO 索引），不过接下来必须在表的数据文件中检查谓词 D.DEPTNO LIKE ‘D%’。

每次执行这个子查询时，可能会发生两个 I/O：一个是对索引文件的 I/O，另一个对应表的数据文件。这个子查询只执行 32 次，所以影响不会太大，不过如果外查询向子查询提供数百万个值，那么这些额外的 I/O 就会急剧累加。对于这种情况，我建议修改 MGRNO 上的索引，增加列 DEPTNO，这样索引就同时包含这两个列（MGRNO, DEPTNO），从而能消除第二个可能出现的数据文件 I/O（和 / 或进一步的获取页面 I/O）。对于子查询处理，DB2 Explain 将把优化显示为 Indexonly = Yes。

对于执行上百万次的子查询，这种方法会显著影响运行时性能。

18. 了解 DB2 Explain 工具

每个 DB2 SQL 开发人员都应当知道对一个查询执行 DB2 Explain 后如何分析得到的 Explain 信息。这是一个非常有用的工具，可以指出优化工具为一个特定查询选择的访问路径。市场上有很多不同的 Explain 工具，每个 DB2 工作室都有能力执行 Explain。所以要熟悉你的工作室使用什么工具执行 Explain。DB2 Explain 会回答以下基本优化问题（当然还不只这些）：

- 涉及表上使用的索引吗？或者是否选择了一个表空间扫描？
- 具体选择了哪些索引？
- 采用什么顺序处理表？
- 涉及排序吗？
- 为什么会有排序？
- 选择了哪一个联接方法？
- 是否对视图或嵌套表表达式（nested table expression）或者公共表表达式（common table expression）进行物化？
- 能否只访问索引文件来满足查询，或者是否 DB2 还必须访问表的数据文件？
- 每个谓词的筛选率是什么？
- 有没有 Stage 2 谓词？

IBM 提供了一些很棒的工具，可以对外提供一个查询的 DB2 优化信息。开始是在 V8 中提供了 Visual Explain 工具，后来在 V9 中提供了 Optimization Service Center 和 IBM Data Studio 工具。这些工具采用一种可视化 Explain 的形式对外提供优化 Explain 信息。我推荐 IBM 的 Data Studio 工具，因为 V10 中已经废弃了 Optimization Service Center。与以往执行常规 Explain 相比，使用 Visual Explain 会得到更多的优化信息。它还提供了一个统计建议工具，如果它认为可能有帮助，还会推荐一些额外的查询统计信息。

19. 使用工具进行监控

几乎每个 DB2 工作室都有一个完成 DB2 监控的工具，可以进一步提供给定 SQL 语句或程序的有关细节。有时，查找性能问题时要想提供更特定的指导，只能利用这个监控工具。每个 DB2 SQL 开发人员都应该熟悉他所在的工作室安装的 DB2 监控工具。这些工具有助于回答以下问题：

- CPU 用于哪些方面的处理（DB2、存储过程、应用代码，等等）？
- 每个 SQL 语句执行多少次？
- 排序花费多长时间？
- 哪个 SQL 语句耗费时间最长？
- 是否存在锁定和并发问题？
- 要得到所需的数据，涉及哪些 I/O 成本？
- 有哪些同步和异步处理成本？

最常用的 z/OS DB2 监控工具包括：OMEGAMON®、TMon、MainView、Apptune、Insight for DB2 以及 DB2PM。最常用的 DB2 LUW 监控工具包括 IBM 的 Optim™ Query Tuner、Precise for DB2、Quest、DBI Software、Embarcadero 的 Performance Analyst 以及 ITGAIN。

20. 采用提交和重启策略

每一个 DB2 SQL 批量程序开发人员都应该在需要修改数据的程序中加入一种提交策略。还可以更好，IT 工作室应该有一个要求所有开发人员都遵守的提交策略。如果未能成功提交，这可能导致锁定、并发和性能问题。如果一个程序正在修改数据，而另外某个程序或进程试图读取或修改同样的数据，这个程序或进程就可能被锁定。除了提交策略，在异常终止的情况下还必须有重启策略，

使得程序能够知道重启时从哪里继续。

SQL 提交（commit）会应用已发生的所有 SQL 修改（截至最后一次提交），并使这些数据修改永久保留。提交时还会释放所持有的锁。如果另一个程序试图处理某些相同的数据，但这些数据已锁定，DB2 会把这个程序置于一个等待队列中，直到锁释放，或者已经过去一定的时间。一般工作室会要求 DB2 将请求在队列中保留 30 到 45 秒，超过这个时间就会向请求者发回一个 -911 SQL 码。DB2 就是通过这种方式告诉程序或进程：它已经努力尝试得到所请求的数据，但是发现另一个进程在指定的等待时间内持有这个数据的锁。另外还有一个 zParm IRLMRWT，可以用来控制线程到期之前等待的秒数，默认为 60 秒。工作室通常会把它降低为某个更合适的值。

建议维护一个 DB2 重启表，其中包括包名、最后提交的键值和提交频率。每次提交之后程序会更新这个表，并检查提交频率。开发人员无需改变代码就能够直接修改这个表中的程序提交频率，甚至在程序运行期间也可以修改。

21. 实现优良的索引设计

要创建合适的索引，确实需要了解驱动特定应用的 SQL，从而能最充分地理解访问模式。不过很多情况下，甚至还未创建任何 SQL 就要求 DBA 创建索引。这就意味着，要在对 SQL 一无所知的情况下创建索引来优化 SQL 性能，这确实是一个难度很大的任务。如果不了解驱动应用的 SQL 语句以及每个语句执行的频率，将很难创建合适的索引。对应用调优时，其中一部分工作就是随着应用开发的进行或者在做全面测试时创建和 / 或修改索引。

创建索引时有一些基本规则：

- 按工作负载索引，而不是按对象。

- 根据常用查询谓词构建索引。
- 一定要索引使用最多的查询。
- 为所有主键和外键列创建索引。
- 对 ORDER BY、GROUP BY 或 DISTINCT 子句中频繁使用的列创建索引，以避免排序。
- 考虑增加列加重一些索引的负担，以鼓励只能使用索引（index-only）访问。
- 对需要确保唯一性的列创建索引。
- 特别注意哪个索引应当定义为聚簇索引。见本章后面的调优技巧 #57。
- 如果索引包含多个列，要明智地选择列的顺序，这一点很重要。一般来讲，索引的第一列应当是一个高基数列。
- 对于 WHERE 子句中用 AND 连接并频繁结合使用的列，使用组合索引会很有好处，这样可以避免维护多个索引。

说明： 检查驱动应用的SQL，并查看所有插入、更新和删除活动。这些语句对索引会有很大影响。如果必须以某种方式修改这些索引的列，由于DB2要保持索引是最新的，这就会引入额外的开销。对于每个插入和删除，DB2不仅要对这个数据执行相应活动，还必须对每个索引执行相应的动作，从而导致更大开销。所以倘若在一个表上执行一个Insert语句，这个表中定义了5个索引，就会完成6次修改来满足插入请求，这包括对基本数据文件的一次插入，另外还要对各个索引文件分别做一次插入。

22. 避免与非列表达式不一致

编写包含非列表达式（non-column expression）的谓词时，有一点很重要，即要确保各个表达式的结果与所比较的列有相同的定义。

优化工具首先确定等号左边列的数据类型（例如，smallint），然后计算右边表达式结果的最大规模。如果 smallint 定义能满足结果的最大规模，那么这个谓词就是一个可索引的 Stage 1 谓词。如果最大规模过大，smallint 定义无法满足，或者结果是一个小数（这就要求为这个比较完成数据转换），这样谓词就是 Stage 2 不可索引谓词。

优化工具可以完成向下类型强制转换，从而将列与更小的值比较，但是（目前还）不能完成向上强制转换，无法将列与一个更大的值比较。对于 V8 或更高版本，这通常不是问题，因为 DB2 已经改进了与不同数据类型数值的比较，不过保证表达式与列定义一致是一个好习惯，这样可以考虑到那些可能还未涵盖的情况。

例如：

```
WHERE EDLEVEL = 123.45 * 12
```

应该写为：

```
WHERE EDLEVEL = SMALLINT(ROUND(123.45*12,0))
```

首先会执行乘法，取整，然后转换为一个 smallint。将 smallint 类型的列 EDLEVEL 与这个值比较，可以确保这是一个 Stage 1 可索引谓词。

23. 所有筛选逻辑放在应用代码之外

通常最好把所有必要的筛选逻辑都作为谓词写在 SQL 语句中。不要漏掉某些谓词，让数据库管理器获取多余的行，然后再通过程序逻辑检查来消除 / 避开这些多余的数据行（有些人把这称为 Stage 3 处理）。

首先要把所有逻辑和筛选谓词放在 SQL 语句中，以获取应用所需要的那些数据行——重申一句，只是应用实际需要的那些行。应当利用数据库管理器引擎的强大能力来完成筛选工作。只有当性

能出现问题，而且所有其他办法都不奏效时才可以考虑利用应用代码完成筛选。

在使用一个 DB2 卸载工具将数据表转储到平面文件中，并由一个程序使用这个文件来处理数据时，也可以采用这种方法。不过要把这作为最后一道防线，只有当所有其他调优手段都无效时才考虑这样做。

24. 确保涉及 Min 和 Max 的子查询谓词要处理可能返回 Null 的情况

编写谓词时，如果其中包括一个选择最小值（Min）或最大值（Max）的子查询，要注意子查询返回的值可能为 null。例如：

```
SELECT E.EMPNO, E.LASTNAME
FROM EMP E
WHERE E.HIREDATE <=
            (SELECT MAX(T2.HIREDATE)
             FROM TABLE2 T2
             WHERE ....
                AND ....)
```

由于返回值可能为 null，这会增加一些处理开销。

开发人员应该养成习惯在用到 Min、Max、Sum 和 Avg 函数的查询上加上一个 COALESCE、VALUE 或 IFNULL 函数。如果没有发现满足条件的数据行可以用来计算这个函数，DB2 会返回 null 作为结果。

不过，使用 Count 函数时，如果没有满足条件的行来计算统计结果，DB2 会返回 0。

要让这个谓词更为高效，可以重写为在子查询中应用 VALUE、COALESCE 或 IFNULL。需要说明的是，目前 IFNULL 函数只是一个 z/OS 函数。这 3 个函数完成的工作都是一样的，也就是说，它们会把可能的 null 值替换为一个预定义的默认值。要确保已经

定义了一个默认值。在这个例子中，指定了‘0001-01-01’来确保如果没有找到一个最大值则不返回任何数据行。例如：

```
SELECT E.EMPNO, E.LASTNAME
FROM EMP E
WHERE E.HIREDATE  <=
  (SELECT COALESCE(MAX(T2.HIREDATE ), '0001-01-01')
    FROM TABLE2 T2
    WHERE .....
       AND .....     )
```

通过应用COALESCE函数，DB2可以保证不会从子查询返回一个Null。它可能返回一个具体的HIREDATE值，也可能返回0001-01-01。

25. 如果查询只选择数据，一定要把游标处理写为For Fetch Only（只获取）或For Read Only（只读）

DB2获取进程中有一种做法称为块获取（block fetching）。块获取可以显著减少通过网络发送的消息数，这种方法只适用于那些不更新或不会删除数据的游标。通过使用块获取，DB2会把一个SQL查询获取的行归组放入一个很大的行块（这个行块可以放在一个消息缓冲区中）。然后DB2通过网络传输这个行块，而不必为每一行分别发送一个单独的消息。增加这个语句还有另外一个好处，这样可能会减少锁定开销（如果有相应的代码）。

要让DB2充分利用块获取进程，DB2必须确定游标不能用于更新或删除。要指示游标不会修改数据，最容易的办法就是为查询增加For Fetch Only或For Read Only子句。如果没有这个语句，DB2会把游标定义为一个不明确的游标，这就会关闭块获取。

指定For Read Only还可以改善获取操作的性能，因为DB2会避免排他锁（即X锁），保证所选的数据不会被修改，从而避免某

些类型的死锁。

通过 For Fetch Only，DB2 可以了解到以下方面：

- 结果集是只读的。
- 不允许定位更新和删除。
- 游标不会得到 U 锁或 X 锁。

26. 避免只是为了帮助确定代码逻辑应当执行更新还是插入而从表中选择一行

要从表中选择一行需要对 DB2 做一次调用。如果有一个处理记录的程序，要根据各个记录执行一个 SQL UPDATE 或 SQL INSERT，它就需要知道在一般执行期间的更新数和插入数，这很重要。你可能并不希望首先用 SQL SELECT 语句来确定表中是否已经存在某一行。如果程序在运行期间处理的 SQL UPDATE 多于 SQL INSERT，那么程序逻辑就应该跳过最初的 SQL Select，而直接执行 SQL Update。如果 SQL Update 语句得到一个 +100 Not Found（未找到）错误，则执行 SQL Insert。

如果程序执行的插入多于更新，则跳过 SQL Select 而直接执行 SQL Insert。如果 SQL INSERT 语句接收到一个 -803 Duplicate Insert（重复插入）错误，则执行 SQL UPDATE。

要当心 -803 错误存在很大开销，如果遇到太多这种错误就会影响性能。因此应当注意一定要了解插入和更新的记录有多少。

注意 DB2 V9 引入了一个 SQL MERGE 语句，这个语句会更为高效。SQL Merge 可以指定在匹配和不匹配条件下分别做什么（分别完成 UPDATE 和 INSERT），这样一来，两种情况都可以在同一个 SQL 语句中加以处理（有关内容参见本章后面的调优技巧 #82）。

27. 避免只是为了得到更新值而从表中选择一行

开发人员在执行 SQL UPDATE 之前没有必要先获取和处理数据。要避免这一点，需要对 DB2 多做一个调用。

例如，假设一个员工的工资要提高 10%。可以选择这个员工的工资，将这个值增加 10%，然后在 SQL UPDATE 语句中使用这个新值。实际上，这个 Update 语句应该这样写：

```
UPDATE  EMP
SET SALARY = SALARY * 1.1
WHERE EMPNO = ?
```

28. 利用动态 SQL 语句缓存

对于 Java .NET 和很多其他可以执行动态 SQL 语句的语言，DB2 会在运行时（即准备语句时）确定该语句的访问路径（优化）。由于要在运行时优化，可能会使其性能比静态 SQL 语句的性能差。不过，如果一个应用经常执行相同的 SQL 语句，那么它可以使用动态 SQL 语句缓存来减少准备这些动态语句的次数。

一些工作室把动态 SQL 缓存设置为 On（即打开动态 SQL 语句缓存），如果是这样，已准备好的语句优化会存储在系统缓存中，以后执行相同语句时可以使用（可能由同一个进程执行，也可能由另一个进程基于一个不同的值来执行）。不过，除了宿主变量中的值可以不同，要执行的所有后续 SQL 语句必须与缓存的 SQL 语句完全匹配。要记住，所有后续语句必须提供相同的用户 ID。如果用户 A 和用户 B 都运行相同的 SQL 语句，第一次执行时它们会分别完成一个完整的准备（而不能使用另一个用户缓存的语句）。

所以，对于可以执行动态 SQL 语句的语言，要利用参数标记（?）准备 SQL 语句，这很重要。如果将值移入宿主变量，然后通过引用宿主变量来准备语句，就会使用具体值完成优化。在这种情况下，以后每次使用不同的值执行这个语句时，都会另外完成一次

语句准备，因为每次语句都与之前有所不同。这样还会占满缓存，导致其他共享缓存的应用性能下降。

DB2 V10 现在已经有所改进，即使缓存中的准备语句是使用硬编码直接量建立的，对于其他包含不同值的查询，DB2 V10 现在也能重用缓存中的这个准备语句。V10 把动态语句缓存称为直接量替换（literal replacement）。由于 DB2 能够识别出一个 SQL 与之前的 SQL 相同（只是直接量值有所不同），这样就能大大减少实现语句重用所需的工作。如果一个应用在 SQL 语句中使用了大量直接量，就可以采用很多方法来启用这个特性。

可以修改 SQL PREPARE 语句，在其中包含新的 ATTRIBUTES 子句，并指定 CONCENTRATE STATEMENTS WITH LITERALS，如下例所示：

```
SET :HV1 = 'CONCENTRATE STATEMENT WITH LITERALS';
PREPARE STMT
  ATTRIBUTES :HV1
  FROM HV-SQL-TEXT;
```

接下来，修改 JSS 驱动程序来包含 enableliteral-replacement=YES 关键字（在数据源或连接属性中指定）。然后修改 ODBC 初始化文件，设置在 z/OS 中启用 LITERAL REPLACEMENT，这会使得通过 ODBC 提交到 DB2 的所有 SQL 启用直接量替换。

通过使用参数标记，可以提供一个更高的动态语句缓存命中率，不过使用直接量通常能提供更好的访问路径。因此，是否使用这个方法要根据动态 SQL 调用的性能特性做出权衡。

29. 避免使用 Select *

对于一次性的查询来说，使用 Select * 没有任何问题，不过要避免将它作为应用开发代码的一部分。程序员可能在代码中编写

SQL 来选择数据，并放入宿主变量字段。使用 Select * 时，必须为表中的每一列分别写一个宿主变量。

倘若表少了一列，或者表中增加了一个新列呢？如果发生这种情况，就需要调整源代码。为了避免这种可能性，首先就不要使用 Select *。

开发人员通常并不需要得到所查询的表中的每一列。如果选择了实际不会用到的额外的列，可能会导致优化和效率问题（参见本章前面的调优技巧 #3）。

30. 当心可以为 Null 的列，还要当心 SQL 语句可能从数据库管理器返回 Null

大多数语言都能很好地处理 null，不过 COBOL 在这方面很成问题。如果 SQL 或 COBOL 代码未能正确地处理 null，COBOL 程序会得到一个 -305 SQLCODE，这通常会让它进入一个错误例程。

不过，不论是什么开发语言，如果没有建立适当的逻辑来考虑可能出现 null 的情况，就会出现逻辑错误。以下是可能出现 null 的一些情况：

- 表定义中某些列定义为 null
- 没有 Else 的 Case 语句
- 左、右和完全外联接
- 选择 Min、Max、Avg 和 Sum
- 数学运算涉及可为 null 的列

COBOL 开发人员要尽可能使用 VALUE、IFNULL 或 COALESCE 标量函数避免 null 进入程序。这样 COBOL 开发人员就可以不必编写相应代码而在每次执行 SELECT 或 FETCH 之后都检查 null 指示符。不过，有时相对于列中的所有其他值，null 值有特定的含义。在这些情况下，就会需要一个 null 值，而在使用上

述标量函数之一时指定某个默认值。

开发人员必须了解这一点，这很重要。由于开发人员不知道可能会有 null，因此没有编写相应的代码，太多的事件报告和异常情况都是由此导致的。

31. 尽量减少执行打开和关闭游标的次数

打开和关闭游标的开销远远大于单例 SQL SELECT 语句的开销。如果大多数情况下 SQL SELECT 语句只返回一行数据，就没有必要执行游标处理。

在这些情况下，应当建立适当的逻辑来首先执行一个 SQL SELECT 语句，并查看返回一行还是多行。如果一个 SQL SELECT 语句返回多行，还会返回一个 -811 SQLCODE。

如果返回了 -811，可以根据这个特定的返回码再执行游标处理。倘若按这种方式编写程序逻辑，则只在返回多行时才会执行游标处理，而所有其他情况下只会执行一个简单的 Select 语句。

需要注意，DB2 返回的 -811 错误以及所有其他错误都会带来开销。如果首先执行 SQL SELECT 时大多数情况下都会返回一个 -811，那么这个处理的运行时间反而会有不必要的增加。

32. SQL 中要避免非逻辑

在 SQL 语言中非（Not）逻辑是非常低效的。例如，请看下面的例子：

```
SELECT EMPNO, LASTNAME
FROM EMP
WHERE NOT HIREDATE    < '2000-01-01'
```

对一个列应用非逻辑（Not）时，会把这个 SQL 谓词降级为不可索引的谓词。要尽量以正面（而非负面）的方式编写 SQL 谓词，

如下所示：

```
SELECT EMPNO, LASTNAME
FROM EMPLOYEE
WHERE  HIREDATE  >= '2000-01-01'
```

33. 使用关联 ID 来保证更好的可读性

有些人编写的 SQL 语句很不清楚，或者不包含关联 ID，处理这些 SQL 语句很让人头疼。如果查询只包含两个表，这可能并没有太大问题，不过如果 SQL 语句中涉及更多的表，就会让人很费解了。开发人员通常使用表名作为其关联 ID，不过这样需要编写很多代码，特别是现在表名甚至可以包含 128 个字符。

使用关联 ID 很重要，这些关联 ID 应当能清楚地指示它们所表示的表。不要使用通用的 A、B、C、D，等等。例如，你可能使用 E 表示 EMP 表，使用 D 表示 DEPT 表，如下：

```
SELECT E.EMPNO, E.LASTNAME
FROM EMP E
WHERE EXISTS
        (SELECT 1
         FROM DEPT D
         WHERE D.MGRNO = E.EMPNO
           AND D.DEPTNO LIKE 'D%')
```

有一些网站可以帮你“格式化”SQL 代码，使之更为清晰。可以访问 Google，键入“SQL Formatter”（SQL 格式化工具），你会搜索到一些网站，可以在这些网站中粘贴一个 SQL 语句，网站会把它进行处理，适当地设置对齐和缩进，使它可读性更好。另外，IBM Data Studio 工具也包含一个查询格式化特性。

联接谓词中的所有关联子查询中都应当编写关联 ID，这一点尤其重要。有时，取决于子查询中 join 语句中是否有关联 ID，有可能会有不同的逻辑。

34. 保证表和索引文件合法而且有组织

DBA 要完成很多任务，其中之一就是维护表和索引文件。数据和索引都包含在自己的文件中。随着对这些文件执行越来越多的活动，它们会变得没有组织、很零碎，或者过于冗长。

DB2 提供了很多工具来帮助维护物理文件。如果这些工具使用得当，可以显著改善性能。清理数据和索引文件最常用的工具是 REORG。REORG 工具可以用来重新组织 DB2 表空间和索引，相应地提高访问这些对象的效率。需要定期地重新组织，以确保数据处于一种最优的方式以利于完成后续的访问。

这是 DBA 通常要完成的诸多任务之一，大多数情况下他们都能做得很好，会适当地进行监控并执行 REORG 工具来保证文件的有效性。不过，如果性能有问题，还应该对这个方面再做检查。

索引也是文件，需要进行维护。在索引文件上执行 REORG 时，会完成碎片空间回收以及其他一些处理。如果索引没有组织、没有条理，这往往是导致应用出现性能问题的一个主要原因，另外执行 REORG 可以提高效率，还可能减少文件中的索引层次。

如果你使用的是 V9，可以检查 SYSIBM.SYSTABLESPACESTATS 和 SYSIBM.SYSINDEXSPACESTATS 表中的数据。这些表中的列名都很直观，含义很清楚。这些实时统计表会定期更新，有助于确定什么时候需要重新组织对象。

35. 充分利用 Update Where Current of Cursor 和 Delete Where Current of Cursor

有些语言会提供其他语言没有的一些选项。其中一个选项就是在游标处理期间执行一个 Update 或 Delete 语句。执行这些语句有两种方法。一种方法是针对游标中刚获取的键字段执行 Update 或

Delete 语句[⊖]。这通常称为定位更新（positioned update）。另一种方法是利用游标定位，并执行 Update 或 Delete，这种方法高效得多。

很多情况下，游标处理默认为只读游标，开发人员可能无法利用 Update Where Current of Cursor。典型的只读游标是游标定义有一个 Order By 语句。不过基于动态可滚动游标的优点，即使有 Order By 语句，Delete Where Current of Cursor 仍能执行（有关内容参见本章后面的调优技巧 #43）。

用 For Update Of 指定游标时会有一些锁定问题，可能会影响所涉及数据的并发处理。不过，如果不用考虑锁定，利用 Update Where Current of Cursor 或 Delete Where Current of Cursor 声明游标就会得到更快的处理。

36. 使用游标时，利用多行获取、多行更新和多行插入来使用 ROWSET 定位和获取

DB2 V8 开始支持多行获取、更新和插入处理。DB2 之前的版本只允许程序在游标处理过程中一次处理一行。由于一次能获取、更新或插入多行，这就能减少网络通信量，相应地还能降低与各 DB2 调用有关的其他成本。

从游标获取数据时，开发人员可以编写代码用一个 FETCH 语句一次获取多行，比如 100 行，对应各个宿主变量将数据获取到一个数组中。需要说明，目前看来一次 100 行是最高效的阈值；多于或小于 100 都会使效率有所降低。建议先从一次获取、插入或更新 100 行开始尝试，然后再测试其他行数。这样做可以使运行时间平均减少 35%。进一步的详细内容和代码示例参见 IBM DB2 手册。

对于本地应用，通常使用这些多行更新语句，数据库访问会更少。对于分布式应用，使用这些多行语句可以减少网络操作，并得

⊖ 实际修改在基表上完成。——译者注

到性能的显著提升。

为了让分布式应用利用多行处理，需要根据所使用的IBM数据服务器驱动程序的类型和版本更新一些属性和设置（有关的代码示例参见本章后面的调优技巧 #46、#47 和 #48）。

37. 了解锁定隔离级别

DB2 提供了 4 个锁定隔离级别：可重复读（Repeatable Read，RR）、读稳定性（Read Stability，RS）、游标稳定性（Cursor Stability，CS）和未提交读（Uncommitted Read，UR）。每个隔离级别允许用户和应用控制一个工作单元中持有读（共享）锁的个数和时间。基于特定应用的需求设置适当的隔离级别时，可以最小化锁资源，从而提高用户 / 程序并发性。来看下面的例子：

```
SELECT LASTNAME, EMPNO
FROM EMP
WHERE LASTNAME LIKE 'S%'
WITH UR
```

RR 表示同一个查询可以在同一个工作单元中执行多次，每次查询的结果都相等（可重复）。在这种情况下，将在各行或各页上设置一个共享锁并一直保留，直到查询或逻辑工作单元结束。所访问的所有行或页都被锁定，即使它们并不满足谓词也会被锁定。对于表扫描，这会涵盖表中的每一行 / 页。对于不处理表扫描的其他查询，会涵盖满足 SQL 语句谓词规则的所有行或页。在上面的例子中，这包括所有包含以 S 开头的姓的行或页。

使用 RR 时，所有共享锁会一直持有，直到完成提交。这些共享锁能有效地避免执行提交前任何进程对任何行 / 页的更新、插入或删除（X 锁）。

说明： 目前市场上大多数查询工具都将其默认隔离级别设置为 RR，这并不好。如果用户、分析人员、开发人员和其他人员每天

> 经常查询数据，在这些环境下这种设置就会带来一些问题。有时用户会离开他们的工作站，在此期间，在后台运行的查询会对获取的数据应用锁并一直保持。这是产生很多 -911 SQLCODE 错误的常见原因。

RS 与 RR 很相似，只不过还允许其他用户插入数据。有时这可能会锁定更多行 / 页，因为即使进入 Stage 2 处理来进一步检查谓词，也会在数据上加锁。如果有一个 Stage 2 谓词，即使数据不满足谓词规则，仍然会加 RS 锁并一直持有。

CS 会对所处理的各行或页设置一个共享锁，游标移到另一行或页时，它会释放这个锁。所以在任何时刻，某一行或一页数据上只会持有一个锁。显然这会支持很好的并发性和一定程度的数据完整性。如今 IT 工作室中几乎所有批量 COBOL 程序都会使用锁定参数 CS。这是因为，当这些程序执行游标处理时，它们不需要重读所处理的数据。查询在游标间移动时会释放共享锁，因此查询处理各个当前行或当前页时可以保证数据完整性。通过将这个绑定参数和另一个绑定参数 Currentdata(No) 结合，可以完全避免锁定。结合使用这两个绑定参数时，DB2 可以测试一行或页是否已经提交其中的数据，如果是，DB2 就不必得到任何锁。

UR 表示对这个查询处理的任何行或页都不设置共享锁，即使其他进程持有所获取数据上的锁，也对这个查询没有影响。使用 UR 可以提高效率，因为这样能减少总的处理时间。不过，使用 UR 存在一个问题：如果另外某个进程对所获取的数据应用了更新，在这个进程执行提交之前 UR 会从缓冲区返回更新后的数据。如果出于某种原因该进程将其更新回滚，那么这个 UR 进程就会更新根本没有提交的数据。

尽管存在这个问题，即有可能选择根本未提交的数据，不过很多情况下仍然可以使用 UR：

1. 数据仓库查询。在这些环境下通常不会发生更新。

2. OLAP/OLTP 处理。如果大多数更新处理都在每晚批量处理中完成，那么将所有 OLAP/OLTP 处理设置为 UR 可以自动提供更好的性能和响应时间，因为不会发生任何锁定。

3. 表很少改变。编码表和参考表就是这种例子。

4. 查询工具。 全天执行动态查询时，如果使用 UR，就不会因为在数据上设置共享锁而干扰正常的生产处理。

38. 了解 Null 处理

开发人员（特别是 COBOL 程序员）需要知道并掌握 null 处理。COBOL 语言不如其他语言那样能够很好地处理 DB2 返回的 null。如果一个 COBOL 程序从 DB2 接收到一个 null，而且程序没有使用 null 指示符或利用 VALUE、IFNULL 或 COALESCE 来处理 null，那么 DB2 会返回一个 -305 错误。从定义来讲，null 值表示一个未知值。

例 1：

```
SELECT EMPNO,
SALARY + BONUS + COMM AS TOTAL_PAY
FROM EMP
```

如果 Salary、Bonus 或 Comm 中任意一列为 null（或是一个未知值），那么返回的 Total_Pay 就是 null（或未知值）。例 1 应当写为：

```
  SELECT EMPNO,
         COALESCE (SALARY,0)   +
         COALESCE (BONUS, 0)   +
         COALESCE (COMM,  0)   AS TOTAL_PAY
FROM EMP
```

例 2：

```
  SELECT  AVG(SALARY)
  FROM EMP
  WHERE  WORKDEPT = 'XYZ'
```

如果没有找到任何行，返回的答案就是null。例2应该如下编写。不过如果将默认值设置为0，要确保实际的工资平均值不能是0：

```
SELECT
COALESCE(AVG(SALARY),0)
FROM EMP
WHERE  WORKDEPT = 'XYZ'
```

例3：

```
SELECT EMPNO, SALARY, BONUS, COMM
FROM EMP
WHERE BONUS <> COMM
```

如果某一列有一个值，而另一列为null，这并不说明它们不相等，也不能作为结果在结果集中返回。如果其中一列有值而另一列没有值，而且开发人员希望它们因为不相等而作为结果集的一部分，查询应该如下编写：

```
SELECT EMPNO, SALARY, BONUS, COMM
FROM EMP
WHERE  COALESCE(BONUS,1) <>  COALESCE(COMM,0)
```

不过，需要注意，这个谓词现在变成一个Stage 2谓词。如果这两列都可以包含null值，这个谓词也可以正常工作。

或者对于V8，可以像这样编写代码：

```
SELECT EMPNO, SALARY, BONUS, COMM
FROM EMP
WHERE BONUS IS DISTINCT FROM COMM
```

需要指出，这个语法更简单，更直接，不过也是一个Stage 2谓词。COBOL程序员需要知道结果集中什么时候可能返回null，应当尽量利用COALESCE或VALUE标量函数消除null。如果结果为null，而DB2查找null指示符来放入-1值时，会出现-305 SQL错误。使用VALUE、IFNULL或COALESCE函数时，并不需要null指示符。

39. 编程时要考虑性能

程序员为用户组开发程序和应用时，要谨记两大目标：

- 得到正确的请求数据结果
- 尽可能快地返回结果

很多情况下，程序员可能会忽略第二个目标。他们可能不知道如何让程序更快地运行，而把性能不好的原因归咎于环境的其他部分（数据库、网络、TCP/IP，等等），或者认为相对于数据量而言，处理数据所用的时间并不多。他们需要确保其程序尽可能高效地运行。如果不确定，就应该咨询同事，或者更好的做法是购买这本书。这本书中的很多调优技巧会为他们提供一个不错的起点。有一点程序员一定要知道，即运行期间会处理多少数据。程序会处理十万行数据还是一千万行数据？各个输出中指出了多少更新、插入、删除、选择、打开游标和提交？

如果一个程序性能不佳，首先要问的第一个问题通常是“处理了多少数据？”。每个程序应当在某处指出所处理的数据量，并在某个地方记录或显示这个信息，以供其他人参考。

40. 让 SQL 来处理

关系式编程和过程式编程之间是有区别的。很多情况下开发人员可能会像下面这样做：

- 分解联接或游标。在 DB2 之外联接表通常不是明智或高效的 SQL。
- 执行单独的 Exists 逻辑。
- 为代码描述执行单独的 Select。
- 执行自己的取整逻辑。
- 执行自己的截断逻辑。

- 执行自己的连接逻辑。
- 在代码中执行自己的数学运算、串函数，等等。

开发人员应当充分利用 DB2 SQL 标量函数得到他们实际需要的数据来完成输出。在 SQL 的 Select 子句中使用标量函数并不会导致运行时间增加，不过，在谓词列上使用标量函数确实可能带来严重的性能问题。开发人员应当首先在一个查询工具中编写查询，得到实际需要的数据，然后再把这个查询转移到他们的程序中。大多数程序都应该只是简单的 SQL 获取和代码移动，之间不再执行其他例程。使用函数可以大大减少代码，这意味着执行和测试的代码都会更少。

应当尽量在一个 SQL 语句中得到所需的全部数据（即使它需要一个 5 表或 10 表的联接），只有当性能有问题时才进行分解。不要只是为了检查存在性或者为了得到代码描述而执行单例 Select 语句（即只返回一行的 Select）。要在 SQL 语句中建立这个逻辑。

41. 使用 Lock Table

开发人员经常编写执行多个 SQL UPDATE、INSERT 和 / 或 DELETE 语句的程序。这些语句执行时，通常会出现一个 DB2 锁。锁对于整体性能和运行时有很大影响，所以出现的锁越多，进程执行的时间就越长。DBA 往往会设定一些阈值，保证开发人员不要处理太多的锁。他们可能设置一个阈值选项，将锁增强为表空间锁使程序继续，或者设置一个阈值让程序提前终止。

如果在处理之前写一个 Lock Table 语句，这会在表级发生一个排他锁。由于这个锁比行级或页级层次更高，因此能减少更新、插入或删除上单个的锁。这样一来，程序可以更快地执行，不过在程序运行期间整个表都会被锁定。

例如，Lock Table 可能会帮助程序将执行时间限制在 10 分钟

以内而不是 30 到 40 分钟，因为这里只有一个表空间锁，而不再是单个的行或页锁。不过在这 10 分钟内，所有其他人都无法处理这个表，除非他们使用了 With UR。

下面是使用 Lock Table 语句的一个例子：

```
LOCK TABLE EMP IN EXCLUSIVE MODE
LOCK TABLE  EMP PART 3  IN EXCLUSIVE  MODE
```

如果一个程序要设置锁，这些锁可能会影响表中 25% 或更多的数据，那么使用 Lock Table 语句尤其高效。要记住，在表上加一个排他锁有助于减少运行时间，不过会排斥别人，不允许他们访问这个表。如果对并发性有要求，这种方法可能就不适用了。

42. 考虑 OLTP 前端处理

在可能返回多行的前端 OLTP 程序之外执行 SQL 语句时，可以考虑使用 Optimize for n Rows 语句来影响优化工具，让它根据行数选择最佳的访问路径。如果所需的行数远远小于可能返回的总行数时，这种做法就尤其适用。例如：

```
SELECT EMPNO, LASTNAME,  WORKDEPT, SALARY
FROM EMP
WHERE  WORKDEPT > ?
OPTIMIZE FOR 14 ROWS
```

如果你很清楚不论返回多少行数据，屏幕上只能显示 100 行，那么 SQL 语句就应该指定 Fetch First 100 Rows Only（有关内容见本章后面的调优技巧 #65）。例如：

```
SELECT EMPNO, LASTNAME,  WORKDEPT, SALARY
FROM EMP
WHERE  WORKDEPT > ?
FETCH FIRST 100 ROWS ONLY
```

这个语句通常会在优化中禁用 List Prefetch（列表预取），这会使优化工具在访问数据之前绕过索引 RID 排序。通常不鼓励顺序预取、多索引访问和混合联接。

列表预取特性通常都很不错，因为RID排序有助于减少获取数据时的Getpage请求数。不过，大多数OLTP进程不会处理太多数据，所以如果增加这个排序，可能意味着会带来额外的运行时间，而且这个排序对其处理并没有太大帮助。让OLTP进程绕过这个RID排序可能更为高效。

应用Optimize for n Rows不会影响SQL语句的逻辑。在SQL语句中使用OPTIMIZE FOR n ROWS时，就是要告诉优化工具要处理多少行。优化工具可以由此做出判断，以根据这个数量来确定最高效的方法。

DB2使用OPTIMIZE FOR n ROWS子句来选择访问路径，尽量减少获取前面几行的响应时间。这说明，DB2会尽可能避开涉及排序的访问路径。如果指定一个不等于1的n值，DB2会根据成本选择一个访问路径，而不一定避开排序。值为1时，这种方法对于优化路径的选择影响最大。

如果DB2必须在返回第一行之前收集整个结果集，DB2就会忽略这个子句。代码中包含Distinct、Group By或Union时就是如此。这个子句对于必须物化的视图、表表达式和子查询可能也没有什么作用。

43. 考虑使用动态可滚动游标

可滚动游标（scrollable cursor）是V7开始引入的。DB2 V8又引入了动态可滚动游标，可以进一步提高性能。

动态可滚动游标支持直接访问基表中的数据，而不必使用一个声明的临时表（物化）。每次获取都会返回最新的数据，对所有插入、更新和删除操作都敏感（这意味着不会出现使用可滚动游标时存在的删除孔或遗漏更新问题）。使用常规的可滚动游标时，需要创建临时表，而且从游标得到的选择数据要物化到表中。这就存在

开销。如果能直接处理这些数据，而无需 DB2 创建一个临时文件、加载这个文件，然后处理这个文件，就会使性能大幅提升。例如：

```
DECLARE CURSOR1 SENSITIVE DYNAMIC SCROLL   CURSOR FOR
    SELECT C1, C2
    FROM T1;
```

通过使用一个动态可滚动游标，可以获取新插入的行，但是不能获取已经删除的行。相反，使用一个静态可滚动游标时，将无法获取新插入的行，已经删除的行会在游标的结果表中指示为删除孔（hole）。由于使用动态可滚动游标的关联 Fetch 语句在基表上执行，所以这个游标不需要临时结果表。

将一个游标定义为 SENSITIVE DYNAMIC 时，不能在该游标的相应 Fetch 语句中指定 INSENSITIVE 关键字。

使用动态可滚动游标时可以得到一些性能收益：

1. 有时，使用动态可滚动游标不用将数据物化到工作文件。如果 SQL 语句中没有写 Order By 子句，或者如果优化工具可以使用一个关联索引满足指定的 Order By，就不会发生物化。

2. 为一个 SENSITIVE DYNAMIC 游标指定 Order By 子句时，如果这个 Order By 能够由一个现有索引完全满足，DB2 就会选择一个索引访问路径。不过，用 Order By 声明的动态可滚动游标不能使用 Update Where Current of Cursor，但可以使用 Delete Where Current of Cursor。有 Order By 的动态滚动游标不会生成一个工作文件，因此不能排序。所以 Order By 必须有一个相应的支持索引来避免排序。这可能会得到不同的性能特性（有时好，有时不好），因为本质上讲它会强制避开排序。

处理一个有 Order By 的不可滚动游标时，会自动使游标成为只读游标，并不支持 Update 或 Delete Where Current of Cursor 语句。不过，如果把一个游标定义为 Sensitive Dynamic Scroll，则允许进程执行 Delete Where Current of Cursor 语句。如果使用 Update 或 Delete Where Current of Cursor 完成更新或删除而不是执行单个

的 Update 或 Delete 语句，这样往往会让进程更快地执行。

44. 利用物化查询表改善响应时间（只适用动态 SQL）

物化查询表（materialized query table，MQT）实际上就是一个视图，其数据物化并保存到一个表中。所以可以使用 MQT 把一个视图转换为一个表，这对于很多复杂的查询很有好处（特别是那些完成汇总和分析的查询）。当然，这里还涉及一些管理问题，不过可以使用 MQT 创建非规范化或聚集结构来改善查询性能，同时保持基表仍为完全规范化表。

使用 MQT 的主要好处是，优化工具能够识别一个查询可以利用 MQT，而不用对源表进行查询。有时会把源表和相应逻辑替换为已存在的汇总 MQT。

MQT 可以显著改善查询的性能，尤其是对复杂查询。如果优化工具确定一个查询或查询的某一部分可以使用一个 MQT 处理，优化工具可能会重写这个查询来利用 MQT。这称为自动查询重写（automatic query rewrite，AQR）。

视图和 MQT 都是基于查询定义的。只要引用一个视图，该视图所基于的查询就会运行。不过，MQT 实际上会将查询结果存储为数据，所以可以直接使用 MQT 中的数据而不是底层表中的数据。

为什么要使用 MQT？它们对于改善性能极有帮助。对非规范化或聚集数据结构（如很多 MQT）编写 SQL 查询时，可以节省 I/O，这会直接带来性能收益。

下面是对 MQT 的一个概要总结：

- MQT 是表，通常包含推导得到的信息。
- MQT 可能存储预计算或汇总的数据。
- MQT 可能由成本昂贵的 SQL 推导得出，如联接或复杂聚集。

- 利用 AQR，DB2 会考虑使用一个 MQT 而不是 SQL 查询中的底层表。
- MQT 查询只重写动态 SQL 查询。
- MQT 是物理表，可以直接处理。
- 预计算的数据会外部导出到一个 MQT 中。
- 可以在 MQT 上创建索引。

下面给出一个例子：

```
CREATE TABLE DEPT_SUM_MQT_TABLE AS
(SELECT  WORKDEPT, AVG(SALARY) AS AVG_SAL,
 AVG(BONUS) AS AVG_BONUS
 AVG(COMM)  AS AVG_COMM,
 SUM(SALARY) AS SUM_SAL,
 SUM(BONUS) AS SUM_BONUS
 SUM(COMM)  AS SUM_COMM
 FROM EMP
 GROUP BY  WORKDEPT)
 DATA INITIALLY DEFERRED REFRESH DEFERRED
 MAINTAINED BY SYSTEM
 ENABLE QUERY OPTIMIZATION;
```

说明： 可以为 Create 语句的引用（Deferred）和维护（Maintained）部分指定不同的值，对于 z/OS 和 LUW，它们也是不一样的。DB2 LUW 中 Maintained by System 以及 Refresh Immediate 表示系统会在发生任何插入、更新或删除（可能影响 MQT 中数据）时保持这个表与底层表一致。有关的完备说明请参见 IBM 手册。遗憾的是，对于 z/OS，Refresh Immediate 不起作用。要刷新 MQT 表，唯一的办法就是手动执行一个刷新或者定义触发器。

指定了 Enable Query Optimization 时，如果一个查询编写如下：

```
SELECT SUM(BONUS)
FROM EMP
WHERE  WORKDEPT = 'A00'
```

优化工具实际上会把它重写为：

```
SELECT AVG_BONUS
FROM DEPT_SUM_MQT_TABLE
WHERE  WORKDEPT = 'A00'
```

DB2 重写并从 MQT 总结表选择数据，这比从基表收集所有‘A00’行、对 Bonus 列汇总，然后再返回结果的做法高效得多。通过处理 MQT，对于部门 A00 表中会有一行已经包含对 Bonus 列的汇总，这就减少了 I/O 量。

说明： 只有当查询是动态查询（而不是静态查询），而且查询中不包含参数标记时，优化工具才会选择 MQT。要确保将特殊寄存器 CURRENT REFRESH AGE 设置为 ANY。ANY 表示重写时考虑所有 MQT，而 0 表示重写时不考虑 MQT。

45. 结合 Select 的 Insert

要想插入数据然后选择刚才插入的结果，现在使用一个语句就可以做到，而不用向 DB2 发出多个调用。这也称为关系式编程而不是过程式编程。在过程式编程中，开发人员要执行 Insert 语句，然后再向 DB2 发回另一个 SELECT 语句来查看插入的结果。关系式编程则要考虑如何得到所需的结果，同时尽可能减少对 DB2 做出的 SQL 调用。使用结合 Select 的 Insert 是一个很棒的技术，可以在一个插入进程中从 DB2 创建或修改的列获取程序后续处理所需的值。

例如，可以编写一个结合 Select 的 Insert 语句来获取以下的值：

- 由 DB2 自动赋值的标识列或序列值。
- 开发人员不知道的用户自定义的默认值和表达式。
- 触发器修改的列，多次插入时取决于具体的值可能会有所不同。
- 自动赋值的 ROWID 和 CURRENT TIMESTAMP。

例如：

```
SELECT EMPNO, LASTNAME, MIDINIT, FIRSTNME,
       DEPTNO, SALARY, BONUS, COMM
FROM FINAL TABLE
  (INSERT   (EMPNO, LASTNAME, MIDINIT, FIRSTNME, DEPTNO)
   VALUES ('123456', 'SMITH', 'A', 'JOE', 'A00')
  )
```

这里的“Final Table”是SQL DB2保留字，要让这个语句正常工作必须写上这个保留字。在这个例子中可以看到，这里选择了多个列，其中一些并未出现在Insert语句中。这可能是通过触发器推导出的列，或者是默认值，也可能是自动赋值的序列数。

这个技术在插入父/子表时尤其有用。父表会先插入，如果DB2自动赋一个序列数或标识值作为主键的一部分，那么在所有子表插入中也要使用这个值。由于能立即获得这个值来完成插入，这样就不必再向DB2发出调用来得到序列数或标识值了。

46. 充分利用多行获取

现在可以用一个Fetch语句从DB2获取多行（使用数组）。多年来，对于DB2 SQL，一直都不允许在SQL语句中使用数组。这种情况直到V8才有所改变。如今，通过对所选择的各个列使用数组（在COBOL工作存储空间中定义），向DB2发出一个Fetch语句可以一次返回多个数据行。然后程序可以使用自己的数组逻辑来处理这些行。为此DECLARE CURSOR SQL语句增加了一个新的关键字。

WITH ROWSET POSITIONING会在游标打开时告诉DB2：这个游标允许多行处理。即使定义了游标支持多行处理，仍然可以选择一次获取一行。这将是一个单行构成的行集。多行获取可以提高吞吐量，减少数据库访问调用，并减少网络操作。已经证实这样可以减少CPU开销。大多数测试表明性能平均会有35%的提升。目前，每次获取10行时开始有性能收益，每次获取100行时达到峰

值。如果一次获取超过 100 行，性能收益又会回落。因此开发人员应该从 100 行开始尝试。

例如：

```
01  HV-NUM-ROWS                PIC S9(4) COMP.
01  HOST-VARIABLES.
   05  HV-EMPNO-ARRAY   PIC X(6) OCCURS 100 TIMES.
   05  HV-LASTNAME-ARRAY                OCCURS 100 TIMES.
         49  HV-LASTNAME-LENGTH  PIC S9(4) COMP.
         49  HV-LASTNAME-TEXT   PIC X(25).
   05  HV-SALARY-ARRAY   PIC S9(7)V9(2)
                                        OCCURS 100 TIMES.
   05  HV-BONUS-ARRAY    PIC S9(7)V9(2)
                                        OCCURS 100 TIMES.
   05  HV-BONUS-NI-ARRAY PIC S9(4) COMP
                                        OCCURS 100 TIMES.
DECLARE EMP_CSR CURSOR WITH ROWSET POSITIONING
 FOR  SELECT EMPNO, LASTNAME,  SALARY, BONUS
 FROM EMP
 WHERE EMPNO > '000050'
OPEN EMP_CSR ;

MOVE +100 TO HV-NUM-ROWS

FETCH NEXT ROWSET FROM EMP_CSR  FOR :HV-NUM-ROWS ROWS
INTO :HV-EMPNO-ARRAY,
       :HV-LASTNAME-ARRAY,
       :HV-SALARY-ARRAY,
       :HV-BONUS-ARRAY   :HV-BONUS-NI-ARRAY
```

这里为要选择的每一列和 null 指示符建立了数组，并执行 FETCH 将数据获取到这些数组中。SELECT 中的每一列都必须有自己的数组，还要有关联的 null 指示符数组。通过 V8 新引入的 SQL 语句 GET DIAGNOSTICS，DB2 可以提供需要的所有信息（如行数和 SQLCODE），来帮助了解 DB2 在数组中放入的内容。有关的更多信息参见 IBM 手册。所有工作室的 SQL 编程标准和准则中都应该有这样一个 SQL 标准。我建议开发人员使用 COALESCE 或 VALUE 标量函数，而不是编写 null 指示符。

接下来程序会使用自己的逻辑循环处理这些数组并处理数据。DB2 SQLCA 通信域包括一个 SQLERRD (3) 字段，其中包含每次获取实际返回的行数。

47. 充分利用多行插入

在 V8 中，可以用一个 Insert 语句向 DB2 发送存放在宿主变量数组中的多个 Insert 语句。对于 DB2 for z/OS，要使用 SQL 将数据行插入表中有两种方法。可以使用一个带 VALUES 子句的 Insert 语句一次插入一行，或者通过一个 Insert 一次插入多行（其中包括选择另一个表的 Select 子句）。到目前为止，之前一直没有办法用一个语句插入多行。

DB2 V8 可以使用数组，与获取处理类似，可以用一个 Insert SQL 语句向 DB2 表中插入多行。分布式应用绝对支持多行插入，应该充分加以考虑以提升插入操作的整体性能。大多数测试表明多行插入平均能使性能提升 20%。

例如，可以为每一列和 null 指示符建立数组，加载数组，然后向 DB2 发送一个 Insert 语句。要记住，如果任何项有任何错误，则需要一个 DB2 SQL Get Diagnostics 命令来查看哪些项有错误。有关的更多信息请参见 IBM 手册。

例如：

```
01   HV-NUM-ROWS                          PIC S9(4) COMP.
01   HOST-VARIABLES.
     05  HV-EMPNO-ARRAY PIC X(6)  OCCURS 100 TIMES.
     05  HV-LASTNAME-ARRAY OCCURS 100 TIMES.
       49  HV-LASTNAME-LENGTH  PIC S9(4) COMP.
       49  HV-LASTNAME-TEXT       PIC X(25).
     05  HV-SALARY-ARRAY       PIC S9(7)V9(2) OCCURS 100 TIMES.
     05  HV-BONUS-ARRAY    PIC S9(7)V9(2)  OCCURS 100 TIMES.
     05 HV-BONUS-NI-ARRAY PIC S9(4) COMP OCCURS 100 TIMES.
```

程序首先加载数组，其中包含要插入的数据，然后执行以下

SQL 语句：

```
MOVE +100 TO HV-NUM-ROWS.
INSERT INTO EMP
  (EMPNO, LASTNAME, FIRSTNME,
    MIDINIT, SALARY, BONUS)
VALUES (:HV-EMPNO-ARRAY,
        :HV-LASTNAME-ARRAY,
        :HV-SALARY-ARRAY,
        :HV-BONUS-ARRAY:HV-BONUS-NI-ARRAY)
FOR :HV-NUM-ROWS
ATOMIC
```

这里有一个新关键字 ATOMIC，它告诉 DB2 如果插入中出现错误该怎么做。编写一个多行插入语句时，可以指定 ATOMIC（默认）或 NOT ATOMIC CONTINUE ON SQLEXCEPTION。ATOMIC 会打开“全有或全无”开关。这说明，作为数组一部分的任何插入如果失败，那么这个 SQL 语句的所有插入（甚至那些已经成功完成的插入）都要回滚。如果指定 NOT ATOMIC CONTINUE ON SQLEXCEPTION，那么出错行之前的所有成功插入的行会保留，它们不会回滚。INSERT 还会继续尝试插入数组中剩余的行。只是有错误的那些行不会插入。SQL GET DIAGNOSTICS 语句可以用来确定哪些行未能成功插入。

对于动态 SQL，开发人员必须在准备（PREPARE）时指定 For Multiple Rows，另外要在执行（EXECUTE）时对变量列表指定 For N Rows 和 Using。有关的更多信息请参见 IBM 手册。

48. 充分利用多行更新

现在可以利用宿主变量数组通过一个 Update 语句在 DB2 中更新多行。多行更新可以提高吞吐量，减少数据库访问调用数，并减少网络操作。已经证实这样可以减少 CPU 开销。这里的性能收益包括多行获取所得到的效率提升，另外可能还有通过一个 Update

语句更新整个行集的效率提升。

不过要注意执行哪一个更新：

- Update Where Current of Cursor 会更新数组中的每一行。
- Update Where Current of Cursor for Row 12 of Rowset 会更新数组中的第 12 行。

例如，下面的例子会一次获取 100 行，并一次更新所有这 100 行，让工资分别增加 10%：

```
DECLARE EMP_CSR  CURSOR WITH HOLD WITH
  ROWSET POSITIONING FOR
 SELECT EMPNO, SALARY, BONUS
 FROM  EMP
 WHERE EMPNO > '000040'
 FOR UPDATE OF SALARY;

OPEN EMP_CSR ;

MOVE +100 TO HV-NUM-ROWS
FETCH NEXT ROWSET FROM EMP_CSR FOR :HV-NUM-ROWS ROWS
INTO :HV-EMPNO-ARRAY,
     :HV-SALARY_ARRAY,
     :HV-BONUS-ARRAY :HV-BONUS-NI-ARRAY ;

UPDATE EMP
SET SALARY = SALARY * 1.1
WHERE CURRENT OF EMP_CSR;

CLOSE CURSOR2;
```

说明： 只要每一行都以相同的方式更新，这会是一个很棒的特性。不过，如果获取的各行的工资会以不同方式更新该怎么办呢？对于这种情况，行集中的各行必须单独地更新。例如，如果把一个 100 行的行集获取到宿主变量数组中，每一行要按不同的公式调整工资，那就必须单独地更新每一行。

例如，要以不同方式更新各行：

```
MOVE SQLERRD(3) TO WS-FETCHED-CNT
```

```
PERFORM VARYING SUB1 FROM 1 BY 1 UNTIL
   SUB1 >  WS-FETCHED-CNT
     MOVE HV-SALARY-ARRAY (SUB1) TO HV-SALARY
     UPDATE EMP
     SET SALARY = :HV-SALARY
     WHERE CURRENT OF EMP_CSR  FOR ROW :SUB1
        OF ROWSET
END-PERFORM
```

49. 充分利用多行删除

现在可以通过一个 Delete 语句利用宿主变量数组在 DB2 中删除多行。多行删除可以提高吞吐量，减少数据库访问调用数，并减少网络操作。已经证实这样可以减少 CPU 开销。多行删除的做法与多行更新类似。这里的性能收益包括多行获取所得到的效率提升，另外可能还有通过一个 Delete 语句删除整个行集的效率提升。

不过要注意执行哪一个删除语句：

- DELETE WHERE CURRENT OF EMP_CSR 删除行集数组中的每一行。
- DELETE WHERE CURRENT OF EMP_CSR FOR ROW 12 OF ROWSET 删除行集数组中的第 12 行。

50. 在 Select 子句中使用标量全选

很多情况下，SQL 开发的输出需要结合详细数据和聚集数据。编写这种 SQL 有很多种方法，不过如果利用 V8 提供的标量全选(scalar fullselect)，只要使用了索引，就能提供另外一种非常高效的选择。假设需要一个报告来得到员工工资以及员工所在部门的平均工资。这就需要有详细数据以及按部门聚集的平均工资。例如，可以使用新增加的标量全选：

```
SELECT E1.EMPNO, E1.LASTNAME,
       E1.WORKDEPT, E1.SALARY,
```

```
        (SELECT AVG(E2.SALARY)
         FROM EMP E2
         WHERE E2.WORKDEPT = E1.WORKDEPT)  AS DEPT_AVG_SAL
FROM EMP E1
ORDER BY E1.WORKDEPT, E1.SALARY
```

在 V8 之前，这个逻辑必须使用一个 SQL 嵌套表表达式来编写（见下面的代码），这要求联接之前先物化 X 表。在 V8 中，嵌套表表达式现在可以重写为公共表表达式（对表 X 使用 With 表达式）。下面通过几个例子来说明以前如何编写这个查询，以及使用公共表表达式如何编写这个查询。使用标量全选并不一定是更好的选择，不过利用这个特性可以为开发人员提供另一种选择，允许他们采用一种不同的方式编写解决方案，这可能会得到不同的优化，有可能运行得会更快。

下面是原先的嵌套表表达式：

```
SELECT E.EMPNO,     E.LASTNAME,
       E.WORKDEPT,      E.SALARY,      X.DEPT_AVG_SAL
FROM EMP E,
     (SELECT WORKDEPT, AVG(E2.SALARY) AS  DEPT_AVG_SAL
      FROM EMP E2
      GROUP BY WORKDEPT) AS X
WHERE E.WORKDEPT = X.WORKDEPT
ORDER BY E.WORKDEPT, E.SALARY
```

以下是新的公共表表达式：

```
WITH X AS
(SELECT WORKDEPT, AVG(E2.SALARY) AS
 DEPT_AVG_SAL
 FROM EMP E2
 GROUP BY WORKDEPT)

SELECT E.EMPNO,     E.LASTNAME,
       E.WORKDEPT,      E.SALARY,      X.DEPT_AVG_SAL
FROM EMP E,
         X
WHERE E.WORKDEPT = X.WORKDEPT
ORDER BY E.WORKDEPT, E.SALARY
```

51. 在动态 SQL 中充分利用 REOPT ONCE 和 REOPT AUTO，在静态 SQL 中充分利用 REOPT VARS 和 REOPT ALWAYS

通过使用 V8 和 V9 提供的新的绑定参数（V8 中为 Once，V9 中为 Auto），可以对动态 SQL 进一步优化。这些参数告诉 DB2 在运行时要根据所使用的宿主变量值重新优化。

REOPT ONCE 告诉 DB2 只在运行时根据宿主变量中的第一组值重优化一次。为第一组变量选择的访问路径会存储在动态语句缓存中，对于所有后续的语句准备和执行，DB2 都会使用这个访问路径，而不论宿主变量的值是什么。这样可以避免以后每次执行 SQL 语句时都再次重优化。

现在 V9 中还提供了 REOPT AUTO，指示 DB2 只有当宿主变量值发生改变，而且新值非常重要以至于有必要选择一个不同的访问路径时才进行重优化。列包含的数据分布很不均匀时，REOPT AUTO 参数尤其有用。

REOPT VARS 和 REOPT ALWAYS 用于静态 SQL，因为 DB2 不会缓存静态计划或包。这些参数的作用是一样的，它们告诉 DB2 每次接收到一个 SQL 语句时使用宿主变量中当前的值重优化。必须考虑到运行时重优化多个 SQL 语句可能带来的开销。更新的选择是 REOPT ALWAYS，REOPT VARS 比较老，不过作为“退路”仍然可以使用。建议使用新的 REOPT ALWAYS。

有时一个程序需要分解为两个单独的程序，其中一个利用这些 REOPT 选项尽量减少开销。

有些列包含不均匀的数据分布，程序经常要接收这样一些列的值。例如，可能要根据接收到的 Status_Code 值执行一个 SQL 语句，其中某个 Status_Code 值可能在表中 80% 的数据中出现。其他 Status_Code 值则覆盖其余 20% 的数据。

对于这些特殊的情况，DB2 应当在运行时进行重优化，从而查看所处理的值，并允许优化工具相应地选择一个合适的访问路径。

说明： 只有当 Status_Code 列上生成了频率值 Runstat 统计信息时，运行时重优化才会有帮助。这些统计信息会告诉 DB2 表中有哪些不同的值以及这些值相应数据行的百分比。如果没有这些特殊的统计信息，REOPT 优化与使用宿主变量时没有任何区别。REOPT 对于区间谓词也很有效（即使没有提供额外的统计信息）。还有一种情况下开发人员要让 DB2 知道宿主变量中的值，即在 SQL 中硬编码写入值或者将 SQL 语句写为动态 SQL 语句。

52. 标识易失表

VOLATILE 表 DDL 关键字标识一种特殊的表，这些表有时为空，有时则包含大量数据。这个关键字指示优化工具在完成 SQL 操作时要尽可能在这个表上使用索引访问，这对于通常选择表扫描的表尤其有帮助（因为统计信息太少，不能很好地支持表扫描）。不过要注意，使用 VOLATILE 时可能会禁用列表预取和另外一些优化技术。有时开发人员会处理一些表，他们要删除表中的数据，再重新加载来完成某个进程，每一次在表中加载的行数可能变化很大。这会带来一个问题，因为这些表需要一些数据统计信息，但是数据会变化。SAP 和 PeopleSoft 应用经常包含很多类似这样的表，利用这种改进可以更高效地进行处理。另外，对于包含临时工作表的 ERP 和 CRM 软件包，对这些表指定这个选项时，这些软件包会有更好的表现。要访问这些易失表，优化工具更倾向于索引处理（而不是表扫描，不论是否有统计信息）。

如果表为空或者只有几行数据时计算统计信息，则当表包含很

多行时这些统计信息可能不再正确。在 Create 或 Alter Table 语句上使用 VOLATILE 关键字可以标识数据列存在波动的表。

优化工具使用 VOLATILE 关键字来确定是否使用索引（如果存在一个索引）。如果表为空，或者只包含少量数据，性能会稍有下降，因为优化工具总会使用索引，即使此时表扫描可能是更好的选择。

对于 PeopleSoft 和 SAP 中使用的很多临时表，应在这些表定义中增加 VOLATILE 关键字。

VOLATILE 表关键字要作为 Create Table DDL 和 Alter Table DDL 的一部分。

例如：

```
CREATE TABLE EMP_TEMP
 (EMPNO    CHAR(6)   Not Null,
  LASTNAME VARCHAR(15)   Not Null )
VOLATILE
```

53. 使用 ON COMMIT DROP 改进

ON COMMIT DROP 是一个 DTT 改进，会在提交时自动删除 DTT。如果只需要在应用进程生命期中存储数据，而不需要共享表定义，就应当使用 DTT。这个表定义只在应用进程运行时才存在。DB2 会对声明的临时表完成有限的日志和锁定操作。这种改进对于分布式应用和存储过程尤其重要，因为游标关闭或者提交时可能会完成清理。

下面给出一个例子：

```
DECLARE GLOBAL TEMPORARY TABLE
SESSION.EMP_TEMP1   (COL1 CHAR (20)
                     COL2 CHAR (10)
                     COL3 SMALLINT )
ON COMMIT DROP TABLE
```

如果某个表在一个工作单元中只处理一次，开发人员就应该考

虑这样编写代码。当然还有另外一些选择，如果提交之后还要重用一个表，有时其他做法的性能可能更好，如利用 ON COMMIT DELETE ROWS 或 ON COMMIT PRESERVE ROWS。

54. 使用多个 Distinct

在 V8 中，对指定的列应用 Distinct 时，可以在一个 SQL 语句中得到多个不同的值。这样可以自动提升性能，减少对 DB2 的调用。

下面给出一个例子：

```
SELECT SUM (DISTINCT SALARY),
       AVG(DISTINCT BONUS)
FROM EMP;
```

下面给出另一个例子：

```
SELECT COUNT(EMPNO), COUNT(DISTINCT(WORKDEPT)),
       COUNT(DISTINCT(JOB))
FROM EMP
```

55. 充分利用反向索引扫描

DB2 V8 提供了反向索引扫描的功能。反向索引扫描（Backward index scanning）可以改善包含 Order By column DESC 子句的 Select 语句的性能，因为这样可以减少 DB2 排序。另外，反向索引扫描功能可以减少对降序索引的需求，因为 DB2 现在使用升序索引反向扫描。

如果满足以下条件，DB2 就可以使用一个索引完成反向扫描：

- 定义索引的列与 Order By 子句指定的列完全相同，或者索引列包括 Order By 子句中指定的列作为前几列，后面还有其他列。
- 对于 Order By 子句中的每一列，索引中指定列的顺序与 Order By 子句中指定的顺序正好相反。

对于 DB2 以前版本创建的计划或包，需要为静态 SQL 语句重新绑定计划或包来利用这种改进。对于 V10，所有 SQL 都在包中，所以没有必要重新绑定计划。

56. 当心 Like 语句

编写 Like 语句来检查 Begins with 逻辑时，这个 Like 语句是一个可索引的 Stage 1 谓词。由于知道搜索串最前面是 % 或 _，可以避免 DB2 使用匹配索引，并避免进一步导致一个索引扫描或表空间扫描。

在下面的例子中，DB2 不会选择匹配索引处理，尽管 LASTNAME 列确实有一个索引。这被认为是一种包含逻辑（Contains logic），尽管这不是一个可索引的谓词，但它是一个 Stage 1 谓词：

```
SELECT EMPNO, LASTNAME, SALARY
FROM EMP
WHERE LASTNAME LIKE '%AND%'
```

在下面的例子中，DB2 会选择匹配索引处理：

```
SELECT EMPNO, LASTNAME, SALARY
FROM EMP
WHERE LASTNAME LIKE 'AND%'
```

57. 正确地设置聚簇索引

聚簇索引确定了表空间中行如何物理排序（聚簇）。对于某些操作，聚簇索引会提供显著的性能提升，特别是在需要获取多行的情况下，聚簇索引会很有帮助。例如，分组和排序操作以及除了相等比较之外的其他比较都能由聚簇索引获益。

表空间文件中数据的顺序由创建索引时是否指定 Cluster=Yes 参数来确定。如果定义的索引都没有指定这个参数，则默认为创建

的第一个索引。更确切地讲，这是数据库描述文件中使用的索引链中的第一个索引（通常是主键索引）。运行一个表空间 REORG 工具时，DB2 会查看相关索引来查找有聚簇参数的索引。这个索引的列确定了重组过程中放回表空间文件的数据顺序。

确定应当将哪个索引定义为聚簇索引时，要考虑以下方面：

- 需要分析数据如何处理，根据你的分析来指定聚簇索引，这一点非常重要，特别是在批量程序中，这是因为批量程序处理的数据最多。这样做可以尽量减少所需的 I/O 和 Getpage 数来得到更好的性能。应当定义聚簇索引序列来支持大容量的数据处理。
- 允许 DB2 默认地使用聚簇索引存在一个问题：如果索引被删除然后重新创建，索引链上的第一个索引可能不再是所创建的第一个索引，现在另外一个索引可能会成为下一次运行 REORG 时的聚簇索引。
- 要根据应用通常的处理方式组织表空间中的数据。这会对性能产生巨大影响，特别是对于大容量的批量处理。
- 查看一个表在应用中如何加入多表联接。这还有可能指示出一个表应当如何聚簇。

例如，假设目前 EMP 表按 EMPNO 排序，从 000001 开始，到 999999 结束。不过，假设大多数操作都是用 WORKDEPT 来处理，比如很多报告和结果屏幕都按 WORKDEPT 分组、按 WORKDEPT 排序、要得到 WORKDEPT 的平均值和汇总结果、按 WORKDEPT 给出员工列表，等等。执行这些查询时，表空间文件上将主要是同步处理，因为数据并没有按 WORKDEPT 排序。如果在 WORKDEPT 上建立聚簇索引，下面的语句就能从中受益：

```
SELECT EMPNO, LASTNAME, SALARY
FROM EMP
WHERE  WORKDEPT = 'D11'
```

D11 部门的员工在表空间文件中会分散存储，因为目前数据是按 EMPNO 排序的。这就会导致获取所请求的数据时会有大量 Getpage 请求。如果有 10 个 WORKDEPT = 'D11' 的员工，就可能导致 10 个 Getpage 请求。

不过假设数据按 WORKDEPT 聚簇。在这种情况下，所有这 10 个员工都可能在物理文件的同一个数据页上，这样一来，要获取所请求的数据，可能只会有一个 Getpage 请求。这个小小的改变可以使这个查询的 Getpage 请求数减少 90%。所以一定要了解你的数据总体处理情况，这很重要，特别是批量处理，另外要让表空间文件中的数据按正确的顺序存放以便于处理。

如果 EMP 表主要根据 WORKDEPT 列联接，那么按 WORKDEPT 聚簇排序也很有帮助：

```
SELECT D.DEPTNO, D.DEPTNAME,
E.EMPNO,  E.LASTNAME
FROM DEPT D,
EMP   E
WHERE D.DEPTNO = E.WORKDEPT
```

58. 必要时使用 Group By 表达式

在 V8 中，DB2 允许在 Group By 子句中加入表达式。而在 V8 之前，如果需要通过应用标量函数、数学运算、级联或者采用某种特殊方式对某些数据分组，那么 SQL 必须创建并物化一个嵌套表表达式。在 Group By 子句中直接编写表达式会更为高效。例如，假设希望提供以 A、B、C、D 等开头的部门的员工人数列表。

以下是从前的做法：

```
SELECT X.DEPT_GROUP, COUNT(*) AS #_EMPS
FROM
  (SELECT SUBSTR(WORKDEPT,1,1) AS DEPT_GROUP
   FROM EMP) AS X
GROUP BY X.DEPT_GROUP
```

下面给出新方法：

```
SELECT SUBSTR(WORKDEPT,1,1) AS DEPT_GROUP
      ,COUNT(*) AS #_EMPS
FROM EMP
GROUP BY SUBSTR(WORKDEPT,1,1)
```

59. 当心表空间扫描

如果发现 SQL 执行期间出现表空间扫描该怎么办？表空间扫描是一个红色警报，一定要仔细研究。有时表空间扫描可能比使用索引更好，不过一般情况下，大多数查询都应当对所处理的所有表使用索引处理。

可以利用以下列表来确定 DB2 优化工具为什么会选择表空间扫描来得到访问路径：

- 表可能很小，DB2 认为表空间扫描会比索引处理更快。
- 可能由编目统计得出表很小，或者表可能根本没有统计信息。
- 根据谓词，DB2 可能认为查询要获取大量数据，所以需要一个表空间扫描。
- 谓词可能只允许 DB2 选择一个非聚簇索引，这会带来太多随机 I/O，相应地可以减少使用表空间扫描的预测成本。
- 表空间文件或索引文件的物理结构可能很成问题，需要重新组织。
- 查询中的谓词可能与表上的任何可用索引都不匹配。
- 谓词编写有问题，不可索引。

60. 不要问你已经知道的信息

这听上去可能很显然，不过大多数程序员都会经常违反这个规则。例如：

```
SELECT EMPNO, LASTNAME, SALARY
```

```
FROM EMP
WHERE EMPNO = '000010'
```

这里的问题是Select列表中包含了EMPNO。你已经知道EMPNO会等于值000010，因为这正是Where子句要告诉DB2的。不过，由于Where子句中有EMPNO列表，所以即使Select部分中没有列出，DB2也会“负责任地”获取这一列。这会降低性能，特别是需要排序时这会让性能大幅下降，因为很多情况下选择的所有列都要完成排序。

61. 注意查询中的表顺序

编写内联接时，表顺序并不重要。DB2允许优化工具选择它认为最高效的表顺序来完成处理。优化工具会根据对各个表应用的谓词来选择表的顺序。优化工具会把筛选量最大的表选作为第一个表，这样能减少其他表的I/O。检查性能时要特别注意这一点。DB2 Explain工具会显示优化工具为特定查询处理所选择的表顺序。

编写外联接时，只需要确保有正确的起始表，有时这也称为驱动表（driver table）。这是DB2展开工作的起点，它会由这个表开始对查询中的其他表执行所有外联接逻辑。

例1：在这个例子中，Department表是这个语句中的起始（驱动）表：

```
SELECT D.DEPTNO, D.DEPTNAME, D.MGRNO,
       E.LASTNAME, P.PROJNO, P.PROJNAME
FROM DEPT   D LEFT JOIN
     EMP    E   ON D.MGRNO = E.EMPNO
              LEFT JOIN
     PROJ   P   ON P. = D.DEPTNO
```

它与下面语句的逻辑相同：

```
SELECT D.DEPTNO, D.DEPTNAME, D.MGRNO,
       E.LASTNAME, P.PROJNO, P.PROJNAME
FROM DEPT   D LEFT JOIN
```

```
        PROJ   P   ON P.DEPTNO = D.DEPTNO
                 LEFT JOIN
        EMP    E   ON D.MGRNO = E.EMPNO
```

例2：在这个例子中，Department表将是这个语句中的起始（驱动）表：

```
SELECT D.DEPTNO, D.DEPTNAME, D.MGRNO,
       E.LASTNAME
 FROM EMP   E   RIGHT JOIN
      DEPT  D ON D.MGRNO = E.EMPNO
              RIGHT JOIN
      PROJ P ON P.DEPTNO = D.DEPTNO
```

它与下面的语句有相同的逻辑：

```
SELECT D.DEPTNO, D.DEPTNAME, D.MGRNO,
       E.LASTNAME
FROM PROJ P     RIGHT JOIN
    DEPT   D ON D.DEPTNO = P.DEPTNO
             RIGHT JOIN
    EMP   E     ON E.EMPNO = D.MGRNO
```

说明： 在这些例子中，最重要的是外联接的起始表。SQL语句中加入左联接的另外两个表的顺序并不重要，除非第3个表要联接到第2个表（第2个表联接到第1个表）。

62. 使用左外联接而不是右外联接

编写外联接逻辑时，要保证逻辑正确，编写左外联接还是右外联接并不重要，只要求有正确的起始（驱动）表（参见调优技巧#61）。除了起始驱动表的位置之外，左外联接和右外联接没有任何区别。例如，下面这两个查询是一样的，它们会生成完全相同的结果，而且这两个查询都将DEPT作为驱动表。[⊖]

例1：Department表将作为起始（驱动）表：

```
SELECT D.DEPTNO, D.DEPTNAME, D.MGRNO,
```

⊖ 原文此处为EMP表，有误。——译者注

```
        E.LASTNAME
FROM DEPT   D LEFT JOIN
     EMP    E   ON D.MGRNO = E.EMPNO
```

例 2：Department 表也将作为起始（驱动）表：

```
SELECT D.DEPTNO, D.DEPTNAME, D.MGRNO,
        E.LASTNAME
FROM  EMP    E   RIGHT JOIN
      DEPT   D   ON D.MGRNO = E.EMPNO
```

左联接左边的表就是驱动表，而右联接右边的表也为驱动表。

在这些例子中，有意思的是，实际上 DB2 优化工具会在执行前执行任一右外联接查询并将其重写为左外联接查询。这一点可以利用 DB2 Explain 工具来了解，即在 Explain 中查看 JOIN_TYPE 列。

对很多开发人员来说，选择左外联接还是右外联接可能会让人困惑。应用中很多查询写为左外联接，而另外很多查询写为右外联接，这会让人很迷惑。甚至在同一个查询中表之间可能同时存在左外联接和右外联接，这更让人迷惑不解。

DB2 开发人员应当只写左外联接。左外联接更直接，因为驱动表总是写在最前面，而且后续所有与它联接的表会也会写为左外联接，这样更容易理解，也更可读。

63. 检查不存在的行

要编写逻辑来确定一个表中的哪些行在另一个表中不存在，有两种常用的方法。一种方法是编写外联接逻辑，然后从替换 null 的表（null supplying table，非驱动表）检查 Where COLx IS NULL（COLx 可以是非驱动表中的任何列，不过如果检查联接中的列自然最好）。另一种方法是编写 Not Exists 逻辑。下面两个例子都会返回 DEPT 表中职位不是经理的员工。

例 1：

```
SELECT E.EMPNO, E.LASTNAME
```

```
FROM EMP    E   LEFT JOIN
      DEPT D   ON D.MGRNO = E.EMPNO
WHERE D.MGRNO IS NULL
```

例 2：

```
SELECT E.EMPNO, E.LASTNAME
FROM EMP   E
WHERE NOT EXISTS
     (SELECT 1
      FROM DEPT D
      WHERE D.MGRNO = E.EMPNO)
```

这两种方法在逻辑上是相同的，不过它们的表现有很大差别。第二种使用 Not Exists 的方法通常更高效。如果利用外联接逻辑查找不存在的行，会在联接发生之后才应用谓词 WHERE D.MGRNO IS NULL。这样一来，构建结果集时，无论是不是经理，所有数据行都会包含在结果集中，只是在联接之后再将“经理”行删除。第二种方法会在处理过程中执行筛选，结果集中只写那些不是经理的员工。取决于所处理的数据量，这一点差异在处理中会对性能产生巨大影响。有时 LEFT JOIN 可能会超越 NOT EXIST，特别是在大多数左边行都匹配时。NOT EXISTS 是一个 Stage 2 谓词，null 检查是一个 Stage 1 谓词（也是一个联接后谓词）。这里的重点是要考虑实现同一逻辑有多种方法。此外还有第三种方法，可以用一个非关联子查询编写 NOT IN 逻辑。这种方法通常效率最低，但也不尽然。

64. 使用存储过程

存储过程是在数据库环境中运行的程序。存储过程可以用 SQL、C、Java、COBOL 等语言编写，它们的目的之一就是帮助减少网络通信量。存储过程通常写为包含多个 SQL 数据操纵语言（Data Manipulation Language，DML）语句，另外还包括一些过程逻辑结构，如循环和 if/else 语句。因此，从概念上讲，存储过程类

似于“小”程序，可以为多个应用提供有用的、数据库密集的业务服务。很多应用（相对于 DBMS 本身而言，其中一些是远程应用）会用一个 SQL CALL 语句调用存储过程。

存储过程可能需要参数提供输入（IN）、输出（OUT），或者同时指定输入和输出（INOUT）。另外，存储过程可能返回一个或多个结果集。

要记住，调用存储过程存在开销，另外要知道，如果存储过程中只有一条语句，这就没有任何意义，除非编写这个存储过程的目的只是为了将执行的 SQL 作为包的一部分而取代动态 SQL。如果网络通信量显著减少，使用存储过程就能得到更大收益。存储过程包含多个 SQL 语句时就是如此。

如果确实有必要使用存储过程，可以考虑使用 DB2 V9 的原生 SQL 过程来提高性能。包含 SQL 的外部存储过程需要为用户程序提供一个完整的语言环境，而且外部程序能够返回这个环境来加载包、指定任务控制块，并执行 SQL 语句。当到来的存储过程请求排队等待工作负载管理器调度执行时，DB2 线程可能会挂起一段时间。

利用原生 SQL 过程，在处理调用语句时，DB2 线程只需切换包，它会直接运行这个过程而不用排队。局部变量使用的存储空间在基址寄存器（bar）以上，会由高效的算法来管理。

编写 DB2 V9 原生存储过程时参数语言设置为 SQL，另外必须使用 SQL PL 语言将逻辑写为 Create Procedure 语句的一部分。

使用原生存储过程还有一个更重要的原因：从一个分布式环境调用这样一个过程时，如果存在 zIIP CPU 引擎，这个存储过程就是符合 zIIP 的。这样就能使一些应用节省大量 CPU 成本，这也是最近人们纷纷编写原生存储过程的主要原因。另外我们已经看到，并不是所有原生存储过程的表现都优于外部存储过程。

65. 不要只是为了排序而选择某一列

DB2 中，没有必要只是为了排序而选择某一列。因此，在这个例子中，如果最终用户不需要 D.LOCATION 的值，就不必选择这一列。从 Select 列表删除一些项可以避免不必要的处理。现在已经不再需要选择 Order By 或 Group By 子句中使用的列。例如：

```
SELECT D.DEPTNO, D.DEPTNAME, E.LASTNAME
FROM DEPT   D  LEFT JOIN
     EMP    E  ON D.MGRNO = E.EMPNO
WHERE D.DEPTNO IN (:HV-DEPTNO1, :HV-DEPTNO2)
ORDER BY D.LOCATION, D.DEPTNO
```

在这个 SQL 语句中可以看到，它会根据 D.LOCATION 排序，但是 LOCATION 并不是 Select 子句的一部分。

66. 尽可能限制结果集

如果要从一个结果集获取最多的数据行，而且行数已知，就应当使用 Fetch First n Rows Only 子句。这个子句会调用一个快速隐式关闭游标，限制从结果集返回的行数。处理完第 n 个结果行时，缓冲池会迅速释放这些数据页。

不要把它与 Optimize For n Rows 子句混淆。Optimize For n Rows 子句并不调用快速隐式关闭游标，它会一直保持锁定，继续获取，直到隐式或显式关闭游标。与之相反，Fetch First n Rows Only 不允许获取第 n+1 行，从而会得到一个 +100 SQLCODE。

67. 批量删除时充分利用 DB2 V8 的改进 DISCARD 功能

程序需要定期删除表中的大量数据行。这可能是一个每周、每月、每季度或每年都要做的工作，为了减少并发问题，通常会安排

在某天的某个时段进行。开发人员会尽其所能地尽量减少这些问题，他们可能将游标声明为 With Hold，并频繁地执行提交。这会有帮助，不过 DB2 还存在一个日志问题，这可能会影响性能。如果程序可能删除成千上万甚至上百万行，DB2 日志可能会填满，这会导致 OLTP 事务和批量处理出问题。

对于这个例子，可以利用 V8 DISCARD 功能来解决，让 DBA 使用 SHRLEVEL CHANGE（也称为 ONLINE REORG）建立一个 REORG 作业。这个 ONLINE REORG 允许其他人读写访问数据，与此同时它还能卸载和重新加载表中的数据。结合 REORG 使用 DISCARD 可以指定卸载时要删除的行，并把它们放在一个删除文件中。

说明：要确保指定 LOG NO 作为 REORG 参数。如果要删除的某一行在重新组织期间恰好被另一个进程更新，这个 REORG 作业就会终止，返回码为 8。

另外要注意，DISCARD 的删除规则应当很简单（不能访问其他表，不能有关联子查询，等等）。如果这个表是其他表的父表，可能还存在引用完整性（referential integrity，RI）问题。

关于联机重新组织的更特定的信息请参见 IBM DB2 手册。

68. 充分利用 DB2 LOAD 工具完成批量插入

程序需要定期地在表中插入大量数据行。要得到最佳性能，应当将要插入的这些记录写到一个文件中，然后 DB2 LOAD 工具使用这个文件把这些记录加载到表中。这通常比程序直接加载格式化数据页更高效，而且这样可以避免单行插入处理的大部分开销（例如，基本上能消除日志记录）。另外，如果机器有多个处理器，加

载工具可以更好地利用多处理器的并行性。

可以使用 LOAD 工具将数据批量加载到 DB2 表。在这个过程中，LOAD 会完成所有必要的数据转换（例如，字符转换为日期格式）和错误处理（例如，拒绝有重复键的记录）。舍弃的输入记录会写至一个顺序文件，可供以后检查来确定拒绝这些记录的原因。如果使用了 LOG NO 选项，LOAD 工具会生成一个至少为 04 的返回码，因为在 LOAD 之后还需要一个 COPY。

另外，如果需要并发性，使用 LOAD SHRLEVEL CHANGE 选项来加载可能很合适。尽管没有常规的 LOAD 那么高效，但比程序插入的效率高。

69. 注意视图、嵌套表表达式和公共表表达式的物化

有时 DB2 优化工具会把嵌套表表达式和公共表表达式合并到一个查询中，而不是先执行和物化数据。一般来说，如果表达式包含一个 Group By、Distinct 或 Union，这个表达式就会物化到一个工作文件表中。发生这种物化时，会增加这个工作文件的创建和加载开销，而且这个工作文件表还可能联接到其他表。有时这可能会有麻烦，因为物化表不包含索引。如果程序或查询包含视图或表达式，可以执行一个 DB2 Explain。如果视图或表达式的名字作为表名出现在 DB2 Explain 输出中，说明 DB2 完成了物化。一种可能的调优方法是将物化的输出放在一个基表中以便处理，然后引用这个基表来取代原来的视图或表达式。有些情况下，如果输出是聚集的数据，可以用标量全选重写查询（参见本章前面的调优技巧 #50）。还有一种可能的做法，可以把表转变成一个 GTT，再为物化数据定义一个索引（参见本章前面的调优技巧 #13）。

在后面的例子中，如果视图名 EMP_VIEW1、嵌套表表达式名

E1 或公共表表达式名 DEPT_SAL_TABLE 在 DB2 Explain 输出中作为表名出现，就说明发生了物化。

例 1，视图：

```
CREATE VIEW EMP_VIEW1 AS
  SELECT WORKDEPT, AVG(SALARY) AS AVG_SAL
  FROM EMP
  GROUP BY WORKDEPT
 ;
SELECT E.EMPNO, E.SALARY, E.WORKDEPT, E1.AVG_SAL
FROM EMP E,
     EMP_VIEW1   E1
WHERE E.WORKDEPT = E1.WORKDEPT
ORDER BY E.WORKDEPT, E.EMPNO
;
```

例 2，嵌套表表达式：

```
SELECT E.EMPNO, E.SALARY, E.WORKDEPT, E1.AVG_SAL
FROM EMP E,
     (SELECT WORKDEPT, AVG(SALARY) AS AVG_SAL
      FROM EMP
      GROUP BY WORKDEPT) AS E1
WHERE E.WORKDEPT = E1.WORKDEPT
ORDER BY E.WORKDEPT, E.EMPNO
 ;
```

例 3，公共表表达式：

```
 WITH DEPT_SAL_TABLE  AS
  (SELECT WORKDEPT, AVG(SALARY) AS AVG_SAL
   FROM EMP
   GROUP BY WORKDEPT)
 SELECT E.EMPNO, E.SALARY, E.WORKDEPT, E1.AVG_SAL
 FROM EMP E,
      DEPT_SAL_TABLE E1
WHERE E.WORKDEPT = E1.WORKDEPT
ORDER BY E.WORKDEPT, E.EMPNO
```

例 4，标量全选：下面使用标量全选重写这个查询，可以得到与前面 3 个例子相同的结果，而且没有发生物化，因为平均工资会在处理期间计算。这并不是说这个例子一定比前面的做法好，不过有时可能确实如此，而且这为开发人员提供了另外一种可以得到相

同结果的不同方法。是否使用取决于具体情况……

```
SELECT E.EMPNO, E.SALARY, E.WORKDEPT,
               (SELECT  AVG(SALARY)
                FROM EMP E2
               WHERE E.WORKDEPT = E2.WORKDEPT) AS AVG_SAL
FROM EMP     E
ORDER BY E.WORKDEPT, E.EMPNO
;
```

70. 考虑压缩数据

DBA可能把数据压缩到一个表空间或分区中，为此会在CREATE TABLESPACE或ALTER TABLESPACE中指定COMPRESS YES，然后运行LOAD或REORG工具。很多情况下，使用COMPRESS子句可能会显著减少存储数据所需的磁盘空间，不过得到的压缩率取决于数据本身的特性。压缩会把所有数据转换为变长的数据行。实际上这可能会增加每个数据页上的行数，相应地会在获取数据时减少I/O量。存储数据时会有一些处理开销，另外从存储空间获取数据时也存在一些开销。数据解压缩的开销要远远小于加载和插入数据时压缩数据带来的开销。压缩可能会使应用或查询的性能显著提高，这取决于对表数据完成的压缩量。在大多数工作室中，压缩操作会交给辅处理器来完成，这样就能消除相应的处理开销。利用压缩数据，应用可以得到：

- 更高的缓冲池命中率。由于每页上有更多的数据行，这就允许在缓冲池中保留更多的数据。
- 更少的I/O。
- 更少的Getpage操作。
- 更高效的日志记录。发生SQL改变（插入、删除或更新）时，会把相应的数据记入日志，这些数据也会被压缩。因此，应用完成插入和删除时的日志会减少，不过更新时的日志量可能不定。

DSN1COMP 工具会估计某个表的数据压缩情况。

有些情况下，特别是对于大量更新的数据，使用压缩数据会导致 Getpage 请求、锁请求和同步读 I/O 数增加。如果一个表的数据页包含压缩的定长行，几乎没有或者已经没有空闲空间，更新的行就可能会重新分配到另一页。这个问题可以通过增加页的空闲空间来缓解。

71. 考虑并行性

DB2 计划从一个分区表空间中的一个表或一个索引访问数据时，可能会启动多个并行操作。对于数据或处理器密集的查询，响应时间可能会显著减少。IBM z/OS 环境的大多数工作室都会有一个包含多个处理器的计算机。因此，可以通过充分利用 DB2 的能力，从分区表空间中的一个表或一个索引访问数据时启动多个并行操作，这样能显著减少数据或处理器密集查询的响应时间。除非启用并行处理，否则查询无法利用并行性。下面展示了如何对静态 SQL 启用并行处理：

```
DEGREE(ANY) as a Bind parameter
```

或

```
DEGREE(ANY) on a REBIND
```

对动态 SQL 启用并行处理的方法如下：

```
SET CURRENT DEGREE='ANY'
```

并行操作往往至少涉及分区表空间中的一个表。对于很大的分区表空间，如果 I/O 和中央处理操作能够并行完成，表空间扫描会有最大的性能提升。不过分区表和非分区表都可以利用并行性。

很多使用 V8 的工作室可能等到生产阶段才考虑并行性。还有一些工作室似乎不愿意考虑并行性，因为如果不加适当的控制，程序有可能耗尽各个 CPU 处理器的所有可用资源。

大多数程序采用并行模式时都能运行得更快，特别是那些需要处理大量数据的作业。可以请求你的 DBA 提供帮助。

如果查询要处理跨多个表分区的大量数据，并行性对这些查询的性能会有显著影响。运行时间可能会大大减少。有 3 种常见的并行性：

- Query I/O 并行。 DB2 将 SQL 处理分解为一个查询的并发 I/O 请求，并将数据放回缓冲区。这对于大规模 I/O 密集的查询很有帮助。
- Query CP 并行。 DB2 将大查询分解为较小的部分，然后并发地运行这些部分。例如，在处理一个两表联接的同时，并发地对第三个表进行排序，并准备与之联接。
- SYSPLEX 并行。 DB2 可以把一个查询分摊到数据共享组其他 DB2 成员可用的多个处理器上运行。

72. 让 STDDEV、STDDEV_SAMP、VAR 和 VAR_SAMP 函数远离其他函数

DB2 执行包含聚集函数的查询时，很多函数聚集可以在数据获取时计算，而不是在获取之后再计算。不过 STDDEV、STDDEV_SAMP、VAR 和 VAR_SAMP 这 4 个聚集函数总是在数据获取之后才计算。一些简单的聚集函数（如 Min、Max、Sum、Avg 和 Count）通常在数据获取过程中计算，除非它们包含在以下 4 个函数中：

- STDDEV 返回一个标准偏差。
- STDDEV_SAMP 计算一个样本标准偏差。
- VAR 返回一个方差。
- VAR_SAMP 计算一个样本方差。

73. 考虑使用 ROWID 数据类型 (V8) 或 RID 函数 (V9) 直接访问行

如果表中某一列定义为一个 ROWID 数据类型，这个 ROWID 值会隐式地包含行的位置。如果在后续更新或删除的搜索条件中使用这个 ROWID 值，DB2 可以直接导航到那一行，而绕过索引处理或表空间扫描。这种直接行访问（direct row access）速度很快。要充分利用这个特性，必须先将 ROWID 值选择到一个宿主变量，然后在后续的 Update 或 Delete 语句中引用这个宿主变量。如果在更新或删除之前已经选择了一些行，就推荐这种做法。例如，如果客户需要修改一个订单，通常首先要获取订单信息来查看，然后完成更新或删除。如果当前更新或删除进程不需要先选择数据，就可以不做替换。例如：

```
CREATE TABLE EMP
  (EMPNO      CHAR(6)    NOT NULL,
   EMP_ID      ROWID     NOT NULL GENERATED ALWAYS,
   LASTNAME CHAR(25) NOT NULL,
   ....
   ....  )

SELECT EMP_ID, LASTNAME, ...
INTO :HV-EMPID-ROWID-VALUE,
     :HV-LASTNAME
FROM EMP
WHERE EMPNO = '000010'

UPDATE EMP
SET SALARY = SALARY * 1.1
WHERE EMPID = :HV-EMPID-ROWID-VALUE
```

对于 COBOL，工作存储空间中的数据类型是：

```
10 EMP-ID          USAGE SQL TYPE IS ROWID
```

对于其他应用，声明的变量是：

```
EXEC SQL BEGIN DECLARE SECTION;
  SQL TYPE is ROWID HV-EMPID-ROWID-VALUE;
```

```
    SHORT                HV-EMPNO;
    CHAR                 HV-LASTNAME[30];
    DECIMAL(7,2)       HV-SALARY;
EXEC SQL END DECLARE SECTION;
```

尽管 DB2 可能想使用直接行访问，但具体环境可能会使 DB2 在运行时并不使用直接行访问。例如，在更新或删除发生之前选择的数据行可能已经移动。发生这种情况的原因有可能是执行了一次 DB2 重组，或者另一个进程删除并重新插入了这一行，或者对一个变字符列的更新可能会将它移出当前页。

编写应用时，还应当在 SQL 语句中包含一个键值，以处理可能无法使用直接行访问的情况。例如，如果 EMPNO 上有一个索引，DB2 可以在直接访问失败时使用索引访问：

```
UPDATE EMP
SET SALARY = SALARY * 1.1
WHERE EMPID = :HV-EMPID-ROWID-VALUE
  AND EMPNO = '000010'
```

有关的更多详细内容参见 IBM 手册。

现在 DB2 V9 会在请求时获取每一行的 ROWID，而不必另外增加一列。然后可以用同样的方式在更新和删除时使用这个 ROWID（只要它不是 In 列表中的一个 RID 值）。例如：

```
SELECT RID(EMP)   ,LASTNAME, ...
INTO :HV-EMPID-ROWID-VALUE,      --Bigint Definition
     :HV-LASTNAME
FROM EMP
WHERE EMPNO = '000010'
```

74. 用真实统计和一定的数据测试查询以反映性能问题

大多数组织都至少有 3 个数据库环境：开发环境、测试环境和生产环境。程序员使用开发数据库环境来创建和测试应用，在迁移到生产环境之前，会在测试环境中由程序员和用户做更严格的检查。

在测试环境或质量保证环境中测试一个 SQL 语句或程序时，要尽量确保测试数据库包含有能够反映生产数据库的数据。如果用非真实数据来测试 SQL 语句，在生产环境中使用这些 SQL 语句时可能会有不同的表现。为了确保完成严格的测试，测试环境中的数据分布应当与生产环境中的数据分布很相近。很多工作室可能没有这么奢侈能够将生产数据完全复制到测试环境，因为很多表实在太过庞大，或者可能包含需要消费者确认的数据，而开发人员可能没有权限看到这些数据。对于这些工作室来说，要克服这个问题，他们的做法是从系统编目表（SYSTABLE、SYSCOLUMNS、SYSCOLDIST、SYSINDEXES 等）复制生产数据统计信息。这样至少可以让测试表反映相同的统计信息，SQL 或程序就能够在绑定 / 准备优化期间创建一个适当的访问路径，它应该能反映迁移到生产阶段后的表现。这是一种很好的做法，不过有时尽管访问路径看起来不错，但是查询或程序实际运行的时间可能远远大于预期。另外需要指出，生产环境和测试环境中的 RID 池、排序池、处理器类型和处理器个数可能不同，这些变化也会对优化路径的选择带来一些影响。

测试环境中应该有一定的数据来反映这种情况：即尽管 Explain 访问路径看起来不错，但性能并不好。例如，一个程序的优化结果可能显示了一个很好的访问路径，可以在 10 分钟内完成运行。不过对于只有几十万行的表来说，10 分钟的处理时间可能太长了。迁移到生产环境中运行时，这个运行时间还会更长。正是因为这个原因，开发人员应该在程序的最后显示处理量，这样其他人就能准确地看到执行期间完成了多少处理和 I/O。然后可以将这些数与运行时间和 CPU 时间比较，确定他们的程序是否确实能高效地运行。

要记住，除了数据统计信息外，还有另外一些因素可能会导致测试环境和生产环境之间不同的访问路径。比如缓冲池大小、RID

池大小和 CPU 都会有影响。RID 池尤其麻烦，测试环境和生产环境间不同的 RID 池大小可能导致很多不同的优化选择。

75. 在 WHERE 子句中指定前导索引列

对于一个复合索引，只要一个查询的 WHERE 子句中指定了这个索引的前导列，该查询就能使用这个索引，并执行匹配处理，如下面这个查询：

```
Index = (Part_Num, Product_Num)

SELECT *
FROM PARTS_TABLE
WHERE PART_NUM = 100;
```

另一方面，下面这个查询不会使用这个复合索引作为匹配索引，不过优化工具仍然可能在处理中选择使用这个索引，尽管必须执行一个索引文件扫描：

```
SELECT *
FROM PARTS_TABLE
WHERE PRODUCT_NUM = 5555;
```

V8 Visual Explain（V9/V10 使用 IBM Data Studio）将特定于一个索引的谓词划分为匹配谓词和扫描谓词。匹配谓词（matching predicate）是特定于索引前导列的谓词。扫描谓词（screening predicate）则是可以应用到索引列而不是匹配谓词的那些谓词。

很多开发人员会像下面这样重写查询，即在索引的第一列上应用一个谓词，认为 DB2 会利用这个索引，实际上它并不会完成任何筛选：

```
MOVE ZERO to HV_PART-NUM
SELECT *
FROM PARTS
WHERE PART_NUM > :HV-PART-NUM
AND PRODUCT_ID = 5555;
```

在这个查询中，假设 PART_NUM 列总有一个大于 0 的值。这

个语句的 Explain 可能看起来不错，因为它可能会显示一个匹配索引进程，但是实际上这个处理与之前的索引扫描并无不同，这是因为第一个谓词实际上没有做任何筛选，DB2 必须扫描整个索引，查看每一项。这与索引扫描的处理完全相同。DB2 Explain 中索引扫描显示为索引处理（Index Processing），但是没有匹配列（匹配列为 0）。

76. 尽可能使用 WHERE 而不是 HAVING 完成筛选

要尽一切可能避免结合使用 HAVING 子句和 Group By 来筛选数据。有时只使用 WHERE 子句就可以排除一些数据行，而不需要编写 HAVING 子句。HAVING 子句可以在 GROUP BY 处理之后用来筛选组，而 WHERE 子句会在处理期间筛选数据。下面第一个语句会让 DB2 收集所有不同的 WORKDEPT 值，利用一个排序对它们分组，然后清除 WORKDEPT=‘A00’以外的所有其他组。这会变成一个两段进程，还需要完成一个 DB2 排序；而第二个查询只收集特定于 WORKDEPT =‘A00’的数据，再对工资汇总。这会消除很多排序和 I/O 时间。

例如：

```
SELECT WORKDEPT,
SUM(SALARY)
FROM EMP
GROUP BY WORKDEPT
HAVING WORKDEPT = 'A00';
```

不过，可以如下重写这个查询来充分利用索引，消除所有其他分组，这样就能减少排序时间：

```
SELECT WORKDEPT,
SUM(SALARY)
FROM EMP
WHERE WORKDEPT = 'A00'
GROUP BY WORKDEPT
```

77. 尽可能考虑 Index Only 处理

有时开发人员编写的查询需要执行某个列聚集函数（有 4 个这样的聚集函数：Sum、Avg、Min 和 Max）。如果在索引中再增加一个额外的列，这些聚集函数可以得到一些很好的结果。例如，有以下查询和关联索引，Index1 = (YEAR, PERIOD)：

```
SELECT SUM(JRNL_AMT)
FROM LEDGER_TABLE
WHERE YEAR = 2008
  AND PERIOD IN (1,2,3)
```

这个查询会优化匹配 Index1 上的两列（Year 和 Period），不过接下来必须到表空间文件中收集所有满足条件的数据页，以便对 JRNL_AMT 列完成汇总。有时这可能导致索引文件和数据文件之间大量的 I/O。在财务处理中，通常需要按月、按季度和按年汇总大量数据。如果这是一个需要在不同年份和时段经常执行的查询，其目的只是为了对某一列完成汇总（JRNL_AMT），要对这个查询调优，让它执行得更快，一种方法就是在 Index1 中增加 JRNL_AMT 列。这样索引就会变成（YEAR, PERIOD, JRNL_AMT）。这对于非聚簇索引尤其有帮助。

DB2 能很好地识别索引中的 JRNL_AMT 列（尽管这一列上并没有特定的谓词），它会选择仅索引处理（index only）。之前 DB2 需要访问表空间文件来得到 JNRL_AMT，相应地会带来大量 I/O 成本，现在通过在索引中增加一列，这些 I/O 成本都将消除。这会大大提升性能，将显著改善响应时间。这并不是说所有索引文件中都要增加额外的列，使所有进程都优化为仅索引处理，不过有些特殊情况下这确实很有意义。如果列 JRNL_AMT 是一个频繁更新的列，那么这可能不是一个得到更快响应时间的好办法。索引列的更新会带来大量索引处理开销。

78. DB2 V9 中表达式上的索引

在 DB2 V9 中，可以使用 Create Index 语句在 SQL 表达式上创建索引。如果创建索引时指定了 UNIQUE 选项，可以对索引中存储的值保证唯一性，而不是保证原始列值的唯一性。例如，如果一个应用包含很多类似下面的查询：

```
SELECT EMPNO, LASTNAME, ...
FROM EMP
WHERE YEAR(HIREDATE )  = ?
```

这样一来，DB2 不会选择 HIREDATE 上的索引（尽管这一列上确实存在一个索引）。在 V9 中，现在可以使用下面的索引：

```
CREATE INDEX EMPX3 ON EMP
  YEAR(HIREDATE ) ...
```

不过，DB2 不一定总能使用 Index On 表达式。例如，如果查询包含多个外联接、物化视图或物化表达式，DB2 就不能使用 Index On 表达式。

没有充分利用 Index On 表达式的查询通常可以重写，以使 DB2 选择 Index On 表达式。例如以下查询：

```
SELECT E.EMPNO, E.LASTNAME, ...
FROM EMP         E LEFT OUTER JOIN
     DEPT  D ON E.EMPNO = D.MGRNO
               LEFT JOIN
     PROJ P ON P.DEPTNO = D.DEPTNO
WHERE YEAR(HIREDATE )  = ?
```

可以重写为：

```
SELECT E.EMPNO, E.LASTNAME, ...
FROM
    (SELECT EMPNO, LASTNAME
     FROM EMP
     WHERE YEAR(HIREDATE) = ?) as E
                    LEFT OUTER JOIN
     DEPT  D ON E.EMPNO = D.MGRNO
                    LEFT OUTER JOIN
     PROJ  P ON P.DEPTNO = D.DEPTNO
```

在外联接之外的单独的查询块中指定谓词 YEAR(HIREDATE) = ? 时，DB2 能够利用 Index On 表达式。

要记住，这个谓词很容易重写为：

```
WHERE HIREDATE  BETWEEN DATE(? CONCAT '-' CONCAT '01-01')
                    AND DATE(? CONCAT '-' CONCAT '12-31')
```

不过还有一些包含表达式的谓词并不能重写为更高效的谓词。下面就是这样一个例子：

```
WHERE SALARY + BONUS + COMM > 100000.00
```

79. 考虑 DB2 V9 Truncate 语句

在 DB2 V9 中，如果一个进程要执行批量删除，就应该考虑使用新增的 Truncate 语句。Truncate 类似于批量删除，不过它能避开与表关联的所有删除触发器。表中的行不会物理删除或记入日志，这样进程能非常快地运行。这个语句会清空表数据，还可以选择清空其存储空间，并忽略遇到的所有删除触发器。这是一个清理测试环境表的非常方便的工具，需要快速清理一个表时也可以使用这个工具。

对一个表使用 Truncate 时，这个表可以是任何表空间类型，它可以是一个基表，也可以是一个声明的临时表。例如：

```
TRUNCATE TABLE TEST_TABLE
  REUSE STORAGE
  IGNORE DELETE TRIGGERS
  IMMEDIATE;
```

在这个例子中：

- Reuse Storage 告诉 DB2 清空已经分配的存储空间，但是仍保留分配的空间。另一种选择（默认选择）是 DROP STORAGE。这会告诉 DB2 释放已经分配的存储空间，使这些空间变为可用。
- Ignore Delete Triggers 告诉 DB2 不执行任何删除触发器。

- Immediate 告诉 DB2 立即执行，这样一来，这个进程就不能取消。指定 Immediate 时，会对底层文件做一个 Delete/Redefine 处理。

80. 在子查询中使用 DB2 V9 Fetch First 和 Order by

当只需要一个很小的子集时，有些查询会执行并返回大量数据行。获取一个完整的结果集效率会很低。可以在 Select 语句中指定 Fetch First n Rows，现在甚至可以在子查询中指定 Fetch First n Rows。例如：

```
SELECT EMPNO, LASTNAME, SALARY
FROM EMP
WHERE EMPNO IN
   (SELECT MGRNO
     FROM DEPT
     FETCH FIRST 3 ROW ONLY) ;
```

下面的例子会找出工资排名前十的员工中的部门经理：

```
SELECT MGRNO, DEPTNO
FROM DEPT
WHERE MGRNO IN
   (SELECT EMPNO
     FROM EMP
     ORDER BY SALARY DESC
     FETCH FIRST 10 ROWS ONLY) ;
```

81. 利用 DB2 V9 乐观锁定

在处理中，有时可能要先获取（或选择）一行，最后再完成一个更新，不过这个进程希望确保在获取和执行更新之间不会有其他进程更新该数据。一个应用获取一行时，会得到共享级锁，然后立即释放。DB2 提供了一些方法可以确保不释放锁，但是这会削弱并发性（参见本章前面的调优技巧 #37）。那么，如何确保对某一行执行 Update 语句时这一行未被其他进程改变？

V9 中增加了一个新特性，支持自动生成一个时间戳列，并由 DB2 填充和维护。DB2 会在插入时自动为各行生成时间戳，行中任何列出现更新时则修改这个时间戳。下面是一个例子：

```
CREATE TABLE EMP
 (EMPNO  CHAR(6)  NOT NULL,
  .....
  .....
  EMP_UPD  TIMESTAMP NOT NULL
  GENERATED BY DEFAULT
  FOR EACH ROW ON UPDATE
  AS ROW CHANGE TIMESTAMP)
)
```

建议指定 Generated by Default，因为这样可以提供灵活性，如果需要还可以覆盖生成的值。

由于定义表时指定了 ROW CHANGE TIMESTAMP，所以在查询时可以使用这个时间戳。例如，可以用来查找某个日期以来更新过的所有行。

下面的例子会选择一行，然后在处理中更新这一行。由于定义了 EMP_UPD 列，这就很容易确保在之前的 SELECT 和后来的 UPDATE 处理之间未发生任何改变。

```
SELECT LASTNAME, EMP_UPD
INTO :HV-LASTNAME, :HV-EMP-UPD
FROM EMP
WHERE EMPNO = :HV-EMPNO
```

先在之前的 Select 中选择这一列，然后在后面的 Update 语句中使用。如果 UPDATE 返回 +100，这说明在选择这一行和执行更新之间的这一段时间内这一行已被另一个进程修改。下面的例子将更新之前选择的行，并在 WHERE 逻辑中使用了 EMP_UPD 值：

```
UPDATE EMP
SET SALARY = :HV-SALARY
WHERE EMPNO = :HV-EMPNO
  AND EMP_UPD =  :HV-EMP-UPD-TIMESTAMP
```

82. 使用 DB2 V9 MERGE 语句

Merge 是 DB2 新增的一个 SQL 语句，它能将针对相同目标表的条件 INSERT 和 UPDATE 操作结合在一个语句中。 在应用处理中，有时需要从表中选择一行来确定接下来要执行 INSERT 还是 UPDATE。有了 MERGE 语句，通过一个 SQL 语句就能达到这个目的，而无需再对 DB2 执行最初的 SELECT 请求。如果要处理一行，请考虑下面的例子：

```
MERGE INTO DEPT D
  USING (VALUES ('Q99', 'Q DEPT', '000010', 'A00', 'CA') )
    AS NEWDEPT  (DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, LOCATION)
  ON D.DEPTNO = NEWDEPT.DEPTNO
 WHEN MATCHED  THEN
    UPDATE SET D.DEPTNAME = NEWDEPT.DEPTNAME,
               D.MGRNO    = NEWDEPT.MGRNO,
               D.ADMRDEPT = NEWDEPT.ADMRDEPT,
               D.LOCATION = NEWDEPT.LOCATION
 WHEN NOT MATCHED  THEN
   INSERT (DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, LOCATION)
   VALUES (NEWDEPT.DEPTNO, NEWDEPT.DEPTNAME,
         NEWDEPT.MGRNO,  NEWDEPT.ADMRDEPT, NEWDEPT.LOCATION)
```

现在如果程序想知道每次将这个语句发送到 DB2 之后实际上执行了哪个语句（INSERT 还是 UPDATE），该怎么做呢？这个语句需要与 Select ... From Final Table 语句结合。请看下面的例子：

```
SELECT UPSERT_IND FROM FINAL TABLE

 (MERGE INTO DBTHM40.DEPT D
   INCLUDE (UPSERT_IND CHAR(1))
  USING (VALUES ('Z99', 'Z DEPT', '000010', 'A00', 'NJ') )
    AS NEWDEPT  (DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, LOCATION)
  ON D.DEPTNO = NEWDEPT.DEPTNO
  WHEN MATCHED  THEN
    UPDATE SET D.DEPTNAME = NEWDEPT.DEPTNAME,
               D.MGRNO    = NEWDEPT.MGRNO,
               D.ADMRDEPT = NEWDEPT.ADMRDEPT,
               D.LOCATION = NEWDEPT.LOCATION,
               UPSERT_IND = 'U'
```

```
WHEN NOT MATCHED  THEN
  INSERT (DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, LOCATION, UPSERT_IND)
  VALUES (NEWDEPT.DEPTNO, NEWDEPT.DEPTNAME,
          NEWDEPT.MGRNO,  NEWDEPT.ADMRDEPT,
          NEWDEPT.LOCATION, 'I')
)
```

注意语句中增加的 Include，它定义了一个指示符，这个指示符会设置为 U 或 I，并随 Select * 返回的所有其他列一起返回给语句。如果只需要这个指示符，可以把它改为 Select Upsert_Ind From Final Table。所包含的列可以是任何列名。

在多行处理中也可以使用这个 SELECT FROM MERGE 语句，为此要为涉及的每一列定义一个数组。如果使用多行处理，则不必发送单个语句来执行，可以发送一个语句一次执行多个操作。下面是一个 COBOL 例子，这里没有给出所有变量的定义：

```
01    WS-DEPT-ARRAY.
  05  WS-DEPTNO   PIC X(03)  OCCURS 100 TIMES.
  05  WS-DEPNAME  ....       OCCURS 100 TIMES.
  05  WS-MGRNO    ....       OCCURS 100 TIMES.
........

MERGE INTO DEPT D
  USING (VALUES (:WS-DEPTNO, :WS_DEPTNAME, :WS_MGRNO, .....)
        FOR 100 ROWS )
    AS NEWDEPT  (DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, LOCATION)
  ON D.DEPTNO = NEWDEPT.DEPTNO
 WHEN MATCHED  THEN
    UPDATE SET D.DEPTNAME = NEWDEPT.DEPTNAME,
               D.MGRNO    = NEWDEPT.MGRNO,
               D.ADMRDEPT = NEWDEPT.ADMRDEPT,
               D.LOCATION = NEWDEPT.LOCATION
 WHEN NOT MATCHED  THEN
   INSERT (DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, LOCATION)
   VALUES (NEWDEPT.DEPTNO, NEWDEPT.DEPTNAME,
           NEWDEPT.MGRNO,  NEWDEPT.ADMRDEPT,
           NEWDEPT.LOCATION)
NOT  ATOMIC CONTINUE ON SQLEXCEPTION
```

83. 了解 DB2 NOFOR 预编译选项

使用游标处理时，通常需要对从游标获取的行完成更新。根据定义，有时游标是只读的，无法进行定位更新。这会使程序执行单个的 Update 语句，因而不如定位更新那么高效。DB2 提供了一个 NOFOR 预编译选项，利用这个选项，可以绕过只读问题，使用定位更新而不必编写 For Update Of 子句。

NOFOR 预编译指令指示要完成一个定位更新，而无需相应的 For Update Of 语句。STDSQL(YES) 选项就表示 NOFOR，不过默认为 STDSQL(DB2)。对只读游标不允许使用 For Update Of 子句。这个子句对于临时表也不合法。

获取数据时，NOFOR 仍然会导致 U 锁。它还允许在处理只读游标时使用定位更新。如果没有 For Fetch Only，游标会变得模糊，不支持块获取，不过由于能执行定位更新，因此能缓解这个问题。

84. 考虑 Select Into 中使用 Order By

大多数情况下，Select 语句都比游标处理更高效。如果只需要一行，但是逻辑会返回多行，则开发人员的做法通常是从 DB2 返回所有行，然后在返回第一行之后停止获取。考虑下面的例子：

```
SELECT EMPNO, LASTNAME
INTO :HV-EMPNO, :HV-LASTNAME
FROM EMP
WHERE  WORKDEPT = 'A00'
ORDER BY LOCATION
FETCH FIRST 1 ROW ONLY
```

DB2 V7 提供了一种简便的方法来限制 Select 语句的结果，即使用一个新子句：Fetch First n Rows 子句。指定 Fetch First n Rows 子句时，DB2 会限制 Select 语句获取和返回的行数。

DB2 V8 允许同时使用 Order By 和 Fetch First。

85. 尽量编写布尔项谓词

布尔项谓词是指用与（And）逻辑连接的谓词，所以对于某一行如果一个谓词计算为 false，就会使整个 Where 子句计算为 false。对于单索引访问可以考虑使用布尔项谓词，而非布尔项谓词最好在多索引访问时使用（参见本章前面的调优技巧 #7）。

以下查询包含非布尔项谓词：

```
SELECT EMPNO, LASTNAME, SALARY
FROM EMP
WHERE (LASTNAME > 'SMITH')
OR (LASTNAME = 'SMITH' and FIRSTNME > 'BOB')
```

可以重写这个谓词来包含布尔项谓词，如下所示，这样一来，DB2 可以更好地利用 LASTNAME 上的索引。下面的逻辑与非布尔项谓词例子中的逻辑完全相同：

```
SELECT EMPNO, LASTNAME, SALARY
FROM EMP
WHERE  LASTNAME >= 'SMITH'
  AND (LASTNAME >  'SMITH'   OR
        (LASTNAME = 'SMITH'
         AND FIRSTNME > 'BOB')
       )
```

86. 编写传递闭包

DB2 优化工具通常会把接收到的最初查询转换为一个重写的新查询（已转换）。优化工具可能会增加一个冗余的谓词（如果它确定可以通过传递闭包来应用这个谓词）。传递闭包是指：

```
If D.DEPTNO = E.WORKDEPT and D.DEPTNO = "A00'
 Then E. E.WORKDEPT must also = 'A00'
```

例如：

```
SELECT D.DEPTNO, DEPTNAME, E.LASTNAME
FROM DEPT D,
     EMP  E
WHERE D.DEPTNO = E. WORKDEPT
```

```
  AND D.DEPTNO   = 'A00'
```

DB2 优化时会在内部为这个查询生成以下冗余谓词：

```
AND E. WORKDEPT = 'A00'
```

这会为优化工具提供更多选择来完成筛选、建立表访问序列和索引，这是因为两个表在列 DEPTNO 上都有一个索引。

说明： 如果一个语句中包含与一个视图的联接，这个视图中包含 Union，而且从这个表到这个视图可能有谓词传递闭包，那么 DB2 不会应用这个传递闭包谓词。你必须自己编写相应的代码。

V9 的传递闭包以 <COLA op value> 类型的谓词出现，但是 Like 谓词、In 谓词和 capitalize 子查询不能用于传递闭包。不过，V10 现在已经可以在 In 谓词上使用传递闭包。

```
SELECT D.DEPTNO, DEPTNAME, E.LASTNAME
FROM DEPT D,
     EMP  E
WHERE D.DEPTNO = E. WORKDEPT
  AND D.DEPTNO  LIKE  'A%'
```

然而，DB2 不会为查询生成一个冗余谓词。所以作为开发人员，对你来说，最好编写自己的传递闭包谓词。这会为优化工具提供更多选择。

```
SELECT D.DEPTNO, DEPTNAME, E.LASTNAME
FROM DEPT D,
     EMP E
WHERE D.DEPTNO = E. WORKDEPT
  AND D.DEPTNO  LIKE  'A%'
  AND E. WORKDEPT LIKE 'A%'
```

87. 避免用 Order By 排序

利用与 Order By 列匹配的索引的前导列，DB2 可以避免因 Order By 带来的排序。如果选择的索引对应单个表，或者对应联接

处理中选择的第一个表上某个索引的前导列，就能避免这种排序。例如：

```
SELECT EMPNO, LASTNAME, SALARY
FROM  EMP
ORDER BY LASTNAME,  WORKDEPT

SELECT EMPNO, LASTNAME, SALARY
FROM  EMP
WHERE LASTNAME = 'SMITH'
ORDER BY LASTNAME,  WORKDEPT
```

如果索引中前导列与 Order By 匹配，或者如果查询包含索引前导列上的一个相等谓词，DB2 就会避开排序。如果选择了这个索引，这两个例子都能避免排序。

在调优中，如果采用一种不同的方式重写查询，通常会改变优化路径。开发人员可以为查询增加一个 Order By 子句，使优化工具选择另一个不同的索引来完成处理。很多情况下，还会创建新索引与应用中最常用的 Order By 匹配，其目的只是为了在这些查询中避免排序。

88. 尽可能使用联接而不是子查询

一般来讲，联接要优于子查询，特别是在没有引入重复行时。利用联接，优化工具处理数据会有更多的选择，而且可以引入传递闭包谓词。

如果引入了重复行，则需要一个 Distinct 或 Group By 来消除重复，这通常会导致排序。尽管联接可以得到更好的优化，但可能因为排序开销而抵消。所以如果有重复，就应当从联接转换为子查询处理，这样可能会有更好的表现（参见本章前面的调优技巧 #4）。

89. 当心 Case 逻辑

DB2 SQL 提供了很多函数以便开发人员管理数据。一般来说，

DB2 中的函数处理比应用编程进程中的处理成本更昂贵。如果在查询的 Select 部分中对某一列加上函数，那么对于返回的每一行都要处理这一列。为便于编写代码，减少开发时间，并保证可移植性，可以在 SQL 语句中写函数。不过如果需要最优的性能，则要把函数逻辑移到应用代码中。

Case 表达式开销相当大。如果要编写 Case 表达式，要按从最受限到最不受限的顺序编写 When 逻辑。例如：

```
SELECT EMPNO, LASTNAME, SALARY,
       CASE WHEN SALARY < ...
            WHEN SALARY > ...
            ELSE
       END
FROM EMP
```

如果大多数行都满足 Salary < 50000.00，就应该最先写这个逻辑。

90. 在 Order By 子句中充分利用函数

可以在 SQL 语句的 Order By 部分使用函数，通常这样可以消除多余的编码和额外的处理。

例 1：按 HIREDATE 在一周中的哪一天返回数据：

```
SELECT EMPNO, HIREDATE , DAYOFWEEK(HIREDATE)
FROM  EMP
WHERE .........
ORDER BY DAYOFWEEK(HIREDATE)
```

例 2：按 HIREDATE 在一周中的哪一天排序。

```
SELECT EMPNO, LASTNAME, HIREDATE, DAYOFWEEK(HIREDATE)
FROM  EMP
WHERE .........
ORDER BY DAYOFWEEK(HIREDATE)
```

91. 了解你的 DB2 版本

DB2 经过 V8、V9，现在发展到 V10，分别提供了不同的模式

级别（兼容性模式、启用新函数模式和新函数模式），应当知道你使用的子系统运行的是哪个 DB2 版本，这很重要。很多工作室会使用不同版本的不同子系统，特别是在测试环境中通常都有不同的版本。开发人员必须了解这一点，这非常重要，这样可以减少他们的困惑，比如有时遇到的某个错误看似是一个 SQL 错误，但实际上它是一个与版本有关的问题，可能是因为没有使用正确的版本来处理某个特定的函数。

要确定 DB2 z/OS 的版本，可以选择以下方法：

1. 从 DB2I 菜单选择 DISPLAY GROUP。
2. 执行以下新的 V8 GET VARIABLE 语句：

```
SELECT GETVARIABLE('SYSIBM.VERSION')
FROM SYSIBM.SYSDUMMY1
```

返回的串包括以下内容：

- PPP 为产品串，设置为 DSN。
- VV 是两位的版本标识符，如 09。
- RR 是两位的发行标识符，如 01。
- M 是一位的维护级别标识符，如 5。

可以在你当前使用的子系统中执行这些方法。要记住，各个子系统可能有不同的版本号和发行号。

92. 了解日期运算

DB2 可以加减 Date、Time 和 Timestamp 值和列。这些在 SQL 中称为“标示时段”（labeled duration）：年（Year）、月（Month）、日（Day）、小时（Hour）、分钟（Minute）、秒（Second）和毫秒（Microsecond）。

从一个日期减去另一个日期会返回一个类型为 Decimal(8,0) 的日期时段。例如，返回时段 00100802 等于 10 年 8 个月零 2 天。例如：

```
SELECT DATE('2011-03-01') - DATE('2009-06-14')
FROM SYSIBM.SYSDUMMY1
```

返回 010817，这表示 1 年 8 个月 17 天。

如果想要分解时间段，可以如下编写代码：

```
SELECT YEAR(DATE('2011-03-01')  - DATE('2009-06-14')),
       MONTH(DATE('2011-03-01') - DATE('2009-06-14')),
       DAY(DATE('2011-03-01') - DATE('2009-06-14'))
FROM SYSIBM.SYSDUMMY1
```

这会返回 01 08 17, 即 1 年 8 个月 17 天，不过年、月、日在不同的列中。

93. 了解大容量插入选择

如果一个进程要完成大量列插入，不必逐个插入后提交，与单个插入相比，还有很多更高效的选择。下面来看有哪些选择。

选择 1：结合 Select 的 Insert。尽管这与单个插入很类似，不过这种选择具有单个插入所没有的一些好处：

- 可以尽量减少与 DB2 之间的来回调用次数。
- 对于本地操作，通过在插入时一次传递多行，可以减少所需的调用。
- 对于分布式操作，可以尽量减少每个 DB2 调用相关的网络操作。

例如：

```
INSERT INTO OWNER1.EMP
   SELECT * FROM OWNER2.EMP
   WHERE .....
```

选择 2：多行插入。这个选择对于远程进程尤其适用，可以减少网络通信量（参见本章前面的调优技巧 #47）。

选择 3：使用 DB2 Load 工具完成插入。这种选择有两个主要优点：

- DB2 直接加载数据页，避免了单独行处理的大部分开销（例如，基本能消除日志记录）。

• Load 可以更好地利用并行性。

选择 4: 在一个表上设置追加处理。在表上设置追加处理时，DB2 会绕过一般的插入处理，即试图根据聚簇索引键建立行序列。相反，它只是把行增加到表或分区的最后。聚簇顺序没有任何影响，因为 DB2 只是将行增加到表空间的最后。在 V7 和 V8 中，可以通过设置表空间选项 PCTFREE 0 FREEPAGE 0 MEMBER CLUSTER 打开这个特性。在 V9 中，只需在创建表时指定选项 APPEND YES。

这个选择有很多好处：

• DB2 不必进行搜索来维护聚簇顺序。

• 这样意味着更少的锁定。只是在表空间最后的新数据页上会发生锁定。

• 可以通过 ALTER 命令打开或关闭。完成大量插入处理时可以打开这个特性，而在 OLTP 日常处理时将它关闭。

需要注意，这里的选择 4（即追加选择）能更快地完成插入处理，但是要以数据组织和聚簇为代价。这可能要求对表或分区表空间执行更频繁的重新组织。对于审计记录表来说这可能很合适，因为这里 Select 性能相对来讲并不重要，而 Insert 性能则是重中之重。对于审计记录，可能甚至不需要额外的重新组织。

94. 了解 Skip Locked Data（V9）避免锁定

一般地，运行进程来创建报告或者得到数据时，这些数据往往可能被锁定，因为可能正在进行其他更新处理。如果对一个 SQL 语句应用 SKIP LOCKED DATA，这就允许 DB2 跳过被其他事务锁定的那些行，而不会被阻塞。很多进程中，只需要在某个特定时刻显示数据，而对你的应用来说，跳过一些数据是可以接受的。

可以在 Select、Select Into 和 Prepare 语句上指定 SKIP LOCKED

DATA，也可以在带搜索的 Update 和 Delete 语句上使用。这个特性只与 CS 和 RS 锁定模式兼容，与 UR 或 RR 并不兼容。

这种方法的好处是可以改善性能，因为不再有锁定等待时间。不过要记住，如果锁定级别是页级，就会跳过整个数据页，尽管另一个进程可能仅仅更新了该页上的一行。

例如：

```
SELECT *
FROM EMP
WHERE .........
ORDER BY  ........
SKIP LOCKED DATA
```

95. 对输入流排序

对于处理大量数据的批量操作，要将处理的输入数据按一种有意义的方式组织（或排序），这很重要。数据要根据表的聚簇顺序排序。如果要处理多个表，最好让所有表都以相同的顺序聚簇。有时这可能意味着在处理之前先对输入数据预排序。

对输入流排序有很多好处：

- DB2 可以避开动态预取，而且很多 I/O 可以是 ASYNC（异步）而不是 SYNC（同步）。
- DB2 可以引入索引后备（Index Lookaside）处理，这允许 DB2 反复访问索引文件中的项，而不必重新搜索。这会大大减少 Getpage 请求（相应地也会减少 CPU 时间）。如果输入文件要按某个索引排序，该索引将用于查找、更新或删除，这就很适用。这里可以选择按索引排序，或者按表中数据的聚簇顺序排序。前者可以提供后备（Lookaside）的好处，后者能提供预取（Prefetch）的好处。如果使用的索引同时也是聚簇索引，那么这个处理将“鱼与熊掌兼得”，同时得到这两方面的好处。

- DB2 不必再返回数据页来完成处理，因为有序的数据可以保证进程在这个数据文件中移动，可能会重新应用锁。

96. 如果需要真正的唯一性，可以使用 V8 Generate_Unique 函数

在某些表中，随着数据行的插入，时间戳列有时并不能提供足够的精度（毫秒数）来保证每个插入行真正唯一。DB2 V8 有一个新函数，可以使用这个函数确保表中各行的唯一性。通过这个函数生成的各个后续值都比前一个值大。

使用这个函数与使用特殊寄存器 Current Timestamp 有所不同，对于多行插入或带全选的 Insert 语句，这里会为所插入的各行分别生成一个唯一的值。

Generate_Unique 函数返回一个 13 字节的位串，定义为 Char(13) for Bit Data。例如：

```
CREATE TABLE ORDER_TBL
 (ORDER_ID          CHAR(13) FOR BIT DATA,
  ORDER_DATE        DATE,
  ORDER_LOC         CHAR(03) );

INSERT INTO ORDER_TBL
  VALUES(GENERATE_UNIQUE(), CURRENT DATE, 'NY');
```

要查看所生成的这一列的值（采用一种统一时间内部形式存储），可以使用 Timestamp 函数：

```
SELECT TIMESTAMP(ORDER_ID), ORDER DATE, ....
FROM ORDER_TBL;
```

V10 现在能够在最多 12 毫秒内定义表中的列，这种情况下这个特性也会有帮助（参见本章后面的调优技巧 #114）。

97. 了解声明临时表的新选项

在 V8 之前，声明的临时表会一直保留，直到处理结束，或者

在执行提交时删除。现在声明全局临时表时有了一些新的选项：

- ON COMMIT DELETE ROWS。如果没有定义为 With Hold 的打开游标，表中的行会被删除。
- ON COMMIT PRESERVE ROWS。这个选项会保留行，但是不允许线程重用，因为线程包含一个活动的声明表。
- ON COMMIT DROP TABLE。如果没有定义为 With Hold 的打开游标，表会被删除。

全局临时表用于存储中间数据，这些中间数据可能在整个逻辑工作单元中或者在应用结束之前引用。例如：

```
DECLARE GLOBAL TEMPORARY TABLE TEMP_EMP
(EMPNO   CHAR(6),
 LASTNAME VARCHAR(25),
 WORKDEPT CHAR(3)
 )
ON COMMIT DELETE ROWS
;
```

98. 执行 Get Diagnostics 时需要注意

可以使用 Get Diagnostics 语句来得到之前执行的一个 SQL 语句的有关信息。这对于多行处理尤其有用。现在我们知道 SELECT、INSERT 和 UPDATE 的多行处理可以大大节省 CPU 开销。不过，Get Diagnostics 本身可能是一个开销很大的语句，所以要注意不要太频繁地使用。

开发人员应该尽量首先使用从 SQLCA 得到的信息，然后如果需要再执行 Get Diagnostics。SQLCA 中最常用的一个字段是 SQLERRD(3)，其中包含每个多行获取操作中实际获取的行数。这对于收到一个 +100 的多行获取尤其有用。+100 意味着数据结束，不过行集里可能还有很多行需要处理。

99. 适当地对 In 列表排序

SQL 语句中通常都有 In 列表谓词。要按宿主变量或值的个数建立这个列表，将最常用的值放在列表最前面。这会尽量减少列表值的检查。这对于那些 Not In 值列表尤其重要，因为要让 DB2 确定一行的值是否在一个列表中，必须检查完整的值列表。如果最常用的值放在列表最前面，就可以尽量减少这种检查。

100. 结合 Select 的 Update 和 Delete（V9）

现在完全可以在执行一个插入或删除的同时选择和返回数据。如果需要写一个审计记录，显示操作之前 / 之后的信息，这个特性就很有帮助。这就是所谓的关系式编程而不是过程式编程。在过程式编程中，开发人员要执行一个 Select 语句得到当前数据（即老数据），然后再执行 Update 或 Delete 语句。结合 Select 的 Update 或 Delete 是一个很棒的技术，可以在更新或删除之前获取当前值，从而能同时得到操作之前和操作之后的值。

从下面这个例子中可以看到，在更新之前选择了一些列，并返回这些数据：

```
SELECT SALARY, BONUS
FROM OLD TABLE
 (UPDATE EMP
   SET SALARY = :HV1-SALARY,
       BONUS = :HV2-BONUS
    WHERE EMPNO = :HV-EMPNO)
```

Old Table 是 SQL DB2 保留字，要保证这个语句正常工作，必须写上这个保留字。

下面这个例子显示了可以在删除之前选择列并返回这些列：

```
SELECT LASTNAME, FIRSTNME, WORKDEPT
FROM OLD TABLE
  (DELETE FROM EMP
    WHERE EMPNO = :HV-EMPNO)
```

101. 只在必要时执行 SQL 语句

你可能会惊奇地发现，程序经常会发出一些不再需要执行（或者从来不需要执行）的 SQL 语句。开发人员编写代码时，都应该想想下面这些问题：

- “我需要这些数据吗？”
- “这个语句必要吗？”
- “这些数据可以从之前的处理得到吗？”
- “在这个程序中之前读过这些数据吗？”
- “上一步读取的数据还在使用吗？”

应当去除那些不必要的 SQL 语句，这对于调优很有意义，千万不可小觑。向 DB2 发出的 SQL 语句越少，性能就越好（参见本章前面的调优技巧 #11）。

102. 充分利用内存中的表

程序可能会访问一些表来完成代码验证、获取代码描述或者完成转换。这些表往往是一些小表，会在批量处理中读取多次，并通常会加载到前端界面的下拉框中。这些表一般也不会有太多改变。

可以把这些表加载到一个程序的数组、核心表、CICS 表甚至 VSAM 文件中，这是一个很好的想法。如果把它们加载到一个数组中，一定要将这个数组定义为允许二分查询来实现快速搜索（Cobol 编程中的 Search All）。

对于这些类型的表，可以得到它们的数据缓存，这样做很好，特别是对于那些在处理中可能需要多次读和 / 或引用这些数据的程序尤其适用。

103. 避开通用型 SQL 语句

有时应用会设计为执行非常通用的 SQL 语句，很多调用程序都可以使用这些语句。这些通用 SQL 语句通常会带来性能问题，因为它们往往试图收集所需的全部数据以涵盖所有可能的调用。这通常会得到一般性能，而且对于大多数查询可能会完成很多本来并不需要的处理。这些常用模块设计很容易，但是往往无法得到很好的性能。

应该只执行一个模块真正需要的特定 SQL 语句，这样才能得到最好的性能。要记住，即使只增加一个额外的列，也有可能导致过多的额外处理，以至于带来性能问题。

104. 避免不必要的排序

开发人员应当全面了解 SQL 处理中哪些情况可能会导致排序。不过，要想确定地知道 DB2 是否会执行排序，需要对查询执行 Explain。如果所执行的查询需要有序数据，则应该通过物理设计支持这一点，即定义索引来支持排序。如果有这样一个索引能够支持排序需求，DB2 通常就能避免排序。

开发人员经常会剪切和粘贴其他程序中的查询，可能会在查询中遗留一些并不需要的语句。Group By、Order By、Distinct、Union、Intersect、Except 和 Join 处理都有可能导致排序。

还要记住，随着数据规模的增大，排序开销也会增加，所以在 DB2 Explain 中看到出现排序时，要知道排序的规模，这很重要。如果一个查询中出现两次排序，但是数据量很小，那么这往往不会带来性能问题。不过，如果能够消除排序，就应该尽可能将它消除。DBA 应当适当地指定 DSNZPARM SORTPOOL 的大小。

最高效的排序就是从来都不执行排序！

105. 了解表达式和列函数

结合列函数（如 AVG 和 SUM）和数学表达式时，要考虑到这样一个事实：表达式

```
AVG(SALARY) * 1.1
```

性能将优于

```
AVG(SAL9ARY * 1.1)
```

这两个表达式会返回相同的结果，但是第一个可以使用可用的索引来计算平均值，而第二个则由于 SALARY 列上的表达式而无法使用索引。

106. 结合使用谓词时要注意

用 AND 和 OR 结合使用谓词时，可能会影响谓词在哪个阶段计算。任何两个谓词（简单或复合）用 OR 连接时，会导致这两个谓词都在最高阶段谓词所在的阶段计算。例如，如果用 OR 将一个 Stage 1 谓词与一个 Stage 2 谓词连接，这两个谓词都将是不可索引的 Stage 2 谓词。如果一个 Stage 1 谓词与一个 Stage 1 不可索引谓词连接，那么这两个谓词都将是 Stage 1 不可索引谓词。例如：

```
SELECT EMPNO, LASTNAME
 FROM EMP
 WHERE LASTNAME = 'ANDREWS'
    OR YEAR(HIREDATE ) = 1990
```

如果 LASTNAME 上有一个索引，DB2 不会选择这个索引来完成处理，因为第二个谓词 YEAR(HIREDATE) 是一个 Stage 2 谓词。由于谓词用 OR 连接，所以第一个谓词也将是 Stage 2 谓词。

107. 为搜索查询增加冗余谓词

搜索屏幕通常很麻烦，因为有很多可能性，而且需要滚动。这

些搜索和滚动查询通常称为分页查询。由于会用“或”（OR）连接多个条件，DB2 不会选择单个索引匹配，而通常选择多索引处理或非匹配索引扫描。下面的例子中会增加一个冗余谓词，通过改善谓词间的可索引性来帮助优化工具，使 DB2 可以选择使用单匹配索引处理。这要结合调优技巧 #85，另外要编写布尔项谓词。例如：

```
SELECT WORKDEPT, EMPNO, LASTNAME
FROM EMP
WHERE WORKDEPT > ?
   OR     (WORKDEPT = ?
             AND LASTNAME > ?)
ORDER BY WORKDEPT, LASTNAME
```

增加冗余谓词后重写为：

```
SELECT WORKDEPT, EMPNO, LASTNAME
FROM EMP
WHERE WORKDEPT >= ?
   AND   (WORKDEPT > ?
            OR
          (WORKDEPT = ? AND
           LASTNAME > ? )
         )
ORDER BY EMPNO, LASTNAME;
```

V10 中优化有了一些变化。在 V10 中，根据多个“或”条件完成分页搜索时，DB2 会根据各个谓词单独地执行索引探查，直到达到所请求的行数。通过单独地处理，DB2 可以使用单个匹配索引处理。

重写后的 SQL 语句仍提供一个候选解决方案。

108. 充分利用改进的动态缓存（V10）

SAP 和 PeopleSoft 应用中运行 Java 和 ERP 工作负载的需求越来越大，使用范围也越来越广。这说明会有更多的动态 SQL。DB2 能够缓存这些动态 SQL 语句的优化路径，从而在多次准备和执行相同语句时可以避免重优化的开销。在 DB2 V10 中，基址寄存器

(bar) 以下的虚拟存储显著减少，使基址寄存器以上的当前存储能利用 64 位存储。这就允许更多动态缓存，同时减少虚拟存储的监视和管理。

还有一个动态 SQL 改进：能够在 SQL 语句中写入硬编码值，而单独执行时可以看作是相同的优化 SQL。要在之前的版本中做到这一点，开发人员需要使用参数标记（参见本章前面的调优技巧 #28）。DB2 V10 为此提供了一个新特性，即使在硬编码值不同的情况下，也可识别出接收到的动态 SQL 语句与之前缓存的版本相同。在 V10 之前，以下各个语句会看作是不同的语句，会分别优化和缓存。因此，对于每一个不同的语句，DB2 都必须重新准备，这就会带来大量 CPU 和性能开销：

```
SELECT LASTNAME
FROM EMP
WHERE EMPNO = '000010'
```


```
SELECT LASTNAME
FROM EMP
WHERE EMPNO = '000010'
```


使用参数标记可以得到更高的动态语句缓存命中率。不过，使用直接量通常能提供一个更好的访问路径。另外还有很多应用会生成带直接量的动态 SQL，而不是使用参数标记。如今在 V10 中，如果不同的动态语句只是直接量值有所不同，那么这些动态语句可以共享已准备的缓存语句。为了充分利用这一点，执行 Prepare 语句的程序必须指定 Attributes 子句（新增）：

```
'CONCENTRATE STATEMENTS WITH LITERALS'
```

还可以将 JCC 驱动程序改为 EnableReplacement = Yes 来启用这个特性。对于 ODBC，可以在初始化文件中设置 LiteralReplacement 。

在动态语句缓存中，直接量会替换为一个 & 字符，其表现类似于参数标记。以上语句在缓存中会替换为以下形式：

```
SELECT LASTNAME
FROM EMP
WHERE EMPNO = &
```

说明： 如果类似这样的SQL语句有一个参数标记，就不是一个匹配。DB2将包含参数标记和包含&字符的SQL语句看作是两个不同的语句。

109. 尝试当前提交来避免锁（V10）

前端联机事务环境往往涉及多个只读的查询进程，它们通常存在等待时间和锁定问题。在以前的DB2版本中，有一些选择可以提供帮助，如未提交读（Uncommitted Read）和跳过锁定数据（Skipped Locked Data）（参见本章前面的调优技巧#37和#94）。当前提交（Currently Committed）特性可以提供更大的灵活性：

- 跳过锁定数据。可以向任何SQL Select、Update或Delete语句增加这个选项，告诉DB2忽略所有锁，并跳过当前被其他进程锁定的数据页或行。
- 未提交读。可以向任何只读SQL语句增加这个选项，告诉DB2忽略所有锁，但返回其他进程正在更新或删除的未提交的数据。
- 当前提交。这个新功能只在V10中作为绑定参数（CONCURRENTACCESSRESOLUTION）或在准备语句（USE CONCURRENTLY COMMITTED）中得到支持。它允许访问发生锁定之前最后提交的数据。

说明： 访问当前提交数据只适用于删除和插入活动。读进程仍然采用更新锁定活动的正常锁定进程。

当前提交处理只适用于通用表空间。

对于插入活动，访问当前提交数据只适用于隔离级别CS或

RS 的查询。

对于删除活动，访问当前提交数据只适用于隔离级别 CS 和 Currentdata(No) 的查询。

有一个分区、表或表空间排他锁时，当前提交数据不可访问。

当锁持有者正在执行一个批量删除进程时，当前提交数据不可访问。

110. 尝试使用系统时态表获取历史数据（V10）

许多应用除了需要维护当前数据外，还要维护某些特定表的某种形式的历史数据。这很重要，编写查询来获取一个特定的时间点很困难（例如，一家保险公司可能需要准确地查看某个特定日期一个客户的业务覆盖情况）。对于开发人员来说，要随着数据变化为客户维护数据的这些变化情况也是一项很艰巨的任务，而且除了提供历史数据，还要保证最新的数据可用。当前数据变化时，需要更新和维护历史数据。那么设计呢？所有当前数据和历史数据都在相同的表中维护吗？还是应当在单独的表中维护？数据变化时维护这些数据的效率如何？

DB2 V10 新增了时态数据支持，提供了所有这些功能来作为其核心设计的一部分。利用这个特性，有可能大大减少应用开发代码、测试和运行时相关的开销，还能使应用代码更容易理解。

时态数据：

- 是需要在任何给定时间点记录的数据。
- 是可能需要在过去、现在或将来某个特定时间收集和查看的数据。
- 需要支持历史和审计查询。
- 需要提供业务时间或系统时间支持。

下面是使用系统时间的一个简单例子：

```
CREATE TABLE EMP_TEMPORAL_CUR
 (EMPNO       CHAR(6)                 NOT NULL,
  LASTNAME    VARCHAR(25)   NOT NULL,
  ......
  ADDRESS     VARCHAR(25)   NOT NULL
  .......
  FROM_TS      TIMESTAMP(12)      NOT NULL
                             GENERATED ALWAYS AS ROW BEGIN,
  TO_TS        TIMESTAMP(12)     NOT NULL
                              GENERATED ALWAYS AS ROW END,
 EMP_TS       TIMESTAMP(12)     NOT NULL
                              GENERATED ALWAYS
                             AS TRANSACTION START ID,
 PERIOD       SYSTEM_TIME (FROM_TS, TO_TS)   ) ;

CREATE TABLE EMP_TEMPORAL_HIST
 (EMPNO           CHAR(6)       NOT NULL,
 LASTNAME        VARCHAR(25)    NOT NULL,
 ......
 ADDRESS         VARCHAR(25)    NOT NULL
 .......
 FROM_TS         TIMESTAMP(12) NOT NULL,
 TO_TS           TIMESTAMP(12) NOT NULL
 EMP_TS          TIMESTAMP(12) NOT NULL )

ALTER TABLE EMPLOYEE
ADD VERSIONING USE HISTORY TABLE EMPLOYEE_HIST
```

说明： 执行Alter语句之后，EMP_TEMPORAL_CUR和EMP_TEMPORAL_HIST表会自动链接。列EMP_TS对于DB2是内部的，必须包含在系统时间表定义中。这一列不使用任何逻辑。

历史表与当前表必须有相同的列数（除了EMP_TS列以外）。这些列还必须有相同的列名和定义。

基表EMP_TEMPORAL_CUR中数据更新时，DB2会自动在EMP_TEMPORAL_HIST表中插入一行。历史表上可以执行任何SQL（包括DELETE和UPDATE）。对EMP_TEMPORAL_CUR表上的一行执行DELETE时，删除的这一行将插入到历史表中。

现在要查询一个特定的时间点，可以使用 SQL 逻辑来确定 As of 条件。SYSTEM_TIME 是一个 DB2 关键字。执行这个 SQL 时，DB2 可能会在当前表或历史表中找到所请求的数据（即使 SQL 中只提到当前表）。例如：

```
SELECT *
FROM EMP_TEMPRAL_CUR
FOR SYSTEM_TIME
  AS OF '2009-06-01-12.00.00.000000'
WHERE EMPNO = '000010'
```

111. 尝试使用业务时态表获取历史数据（V10）

另一类时态表使用 Business_Time，这一类时态表不包含第二个历史表。这对于跟踪一段时间的业务事件很有用。这个表中的每一行都包含一个起始日期和结束日期（不是一个时间戳），还可以设置为将来的日期。表中还定义了一个 Period Business_Time 列，其中包含这些起始和结束日期，这类似于 Period System_Time。

下面这个简单的例子使用了 Business_Time：

```
CREATE TABLE EMP_TEMPORAL_BUS
 (EMPNO                CHAR(6)       NOT NULL,
 LASTNAME             VARCHAR(25)   NOT NULL,
 ......
 ADDRESS              VARCHAR(25)   NOT NULL,
 .......
 FROM_DT              DATE          NOT NULL,
 TO_DT                DATE          NOT NULL
 PERIOD BUSINESS_TIME (FROM_DT, TO_DT)   )   ;

ALTER TABLE EMP_TEMPORAL_BUS
ADD UNIQUE(EMPNO, BUSINESS_TIME WITHOUT OVERLAPS)   ;

CREATE UNIQUE INDEX EMPX1 ON EMP_TEMPORAL_BUS
 (EMPNO, BUSINESS_TIME WITHOUT OVERLAPS) ;
```

还可以在这个时态表中结合主键定义一个对时间段唯一的属性。可以看到，这里可以有一个约束：即不允许时间重叠。为此，

可以在CREATE TABLE、ALTER TABLE或CREATE INDEX语句中定义这个约束。对于这个索引，这会按升序将Business_Time的FROM和TO日期列增加到EMPNO的指定索引键。

要查询一个特定的时间点，可以使用SQL逻辑确定As of条件。Business_Time是一个DB2关键字。例如：

```
SELECT *
FROM EMP_TEMPORAL_BUS
FOR BUSINESS_TIME
   AS OF '2009-06-01'
WHERE EMPNO = '000010'

The 'AS OF' logic is the same as:

      '2009-06-01'  >=  FROM_DT and
      '2009-06-01'  <    TO_DT
```

说明： SYSTEM和BUSINESS时态表可以结合到一个表中。这称为一个双时态表。为此，这个表可以同时有FROM和TO时间戳列以及FROM和TO日期列。这种双时态表不仅能得到系统时间的审计功能（跟踪何时做出改动），另外还能得到业务时间提供的As of功能（某些业务条件的有效日期）。

说明： 要完成插入、更新和删除维护，同时保证当前和历史表都是最新的，可以使用时态功能来实现，IBM测试表明，与使用触发器和/或存储过程执行同样的逻辑相比，时态方法的路径长度和CPU时间开销总是更胜一筹。

112. 了解分级函数 (V10)

可能需要很频繁地对数据分级。Rank和Dense_Rank函数会根据SQL语句中指定的分级规范创建一个顺序分级。这些函数从V9开始引入。在IBM手册中，它们称为联机分析处理的OLAP规范。例如，

以下查询按部门分级，根据其平均工资从最高级到最低级排列，：

```
SELECT EMPNO, SALARY,
       RANK() OVER  (ORDER BY SALARY DESC) AS RANK,
      DENSE_ RANK() OVER  (ORDER BY SALARY DESC) AS DENSE_ RANK
FROM EMP
GROUP BY  WORKDEPT
ORDER BY RANK
```

Rank 和 Dense_Rank 之间的区别在于如何处理关联。如果前两个部门的平均工资都是 $50 000.00，那么下一个部门排名是第 2 还是第 3？ Rank 会把下一个部门排为第 3，而 Dense_Rank 会把下一个部门排为第 2：

```
EMPNO          SALARY          RANK          DENSE_RANK
-------------  --------------  ---------  --------------------
000010         52750.00          1                    1
000011         52750.00          1                    1
000110         46500.00          3                    2
000020         41250.00          4                    3
........       ........         ..                   ..
```

这些函数中为了完成分级需要做一些排序，这通常会存在一些开销，不过与在程序代码中编写逻辑相比，使用这些函数还是简单和容易得多。

在 V10 中，可以编写逻辑计算滑动求和和滑动平均值。还可以编写其他聚集函数，如 STDDEV、CORRELATION，等等。如果增加‘Partition By’子句，则会告诉 DB2 在 WORKDEPT 上有中断时要重置所有滑动求和或平均值。

例如：

```
SELECT  WORKDEPT, EMPNO, SALARY,
SUM(SALARY)
OVER (PARTITION BY  WORKDEPT ORDER BY SALARY ASC
      ROWS UNBOUNDED PRECEDING)  AS SUM_SAL
FROM EMP
ORDER BY WORKDEPT, SALARY
```

```
 WORKDEPT          EMPNO        SALARY          SUM_SAL
------------  -------------  ----------  --------------------
```

A00	000120	29250.00	29250.00
A00	000110	46500.00	75750.00
A00	000010	52750.00	128500.00
A00	000011	52750.00	181250.00
B01	.....	..	..
.....	.....	..	..

113. 充分利用扩展指示符（V10）

对于每次执行时可能改变的部分列，通常需要为这些列插入或更新表数据。这里编程的难点在于要涵盖各种可能性，即所有列都有可能改变，也可能不变。例如，以一个顾客的地址为例。如果地址有改变，可能整个地址都会改变，或者只是其中一些部分会改变。

目前编程有哪些做法呢？

- 编写动态 SQL，在执行时特定于所输入或接收的数据编写一个更新或插入语句。
- 为涉及的列的所有可能组合编写特定的静态 SQL 语句。
- 更新所涉及的行的所有列。这要求已经得到所有当前值，然后将某些列重新更新为相同的值。

为了解决这个问题，即支持插入和更新时改变一个动态的列子集，V10 引入了非 0 和 −1 的指示符变量值。这些新的指示符值会告诉 DB2 某个特定的列没有值，并指示 DB2 如何处理缺少的值（针对更新或插入语句有不同含义）。

当前值 0 和 −1 与以往一样，仍有相同的含义：0 表示使用关联宿主变量中的值，−1 表示将关联列设置为 null。

新值 −5 和 −7 的含义取决于是在更新语句还是插入语句中使用。在一个插入语句中，它们都表示相同的意思：即使用默认值。对于更新语句，含义则有所不同，−7 指示 DB2 不做任何更新，−5 指示使用默认值更新。

要使用这些扩展的null指示符，需要为包提供新参数

EXTENDEDINDICATOR(YES)，另外要为动态 Prepare 语句指定 WITH EXTENDED INDICATORS。

下面的例子会更新地址列，有时可能需要更新所有列，有时只需要更新一部分。适当地将 null 指示符设置为 -5 或 -7 会告诉 DB2 该如何处理。在这个例子中，只有 ADDRESS1 和 ADDRESS2 列需要更新。这样利用一个 Update 就能覆盖所有可能的列，同时还能告诉 DB2 哪些列要更新，而哪些列要跳过。

```
MOVE 0 TO HV-ADDRESS1-NI.
MOVE 0 TO HV-ADDRESS2-NI.
MOVE -7 TO HV-CITY-NI = -7, HV-STATE-NI, HV-ZIPCODE-NI.

UPDATE EMP
SET ADDRESS1 = :HV-ADDRESS1 :HV-ADDRESS1-NI,
    ADDRESS2 = :HV-ADDRESS2 :HV-ADDRESS2-NI,
    CITY     = :HV-CITY     :HV-CITY-NI,
    STATE    = :HV-STATE    :HV-STATE-NI,
    ZIPCODE   = :HV-ZIPCODE :HV-ZIPCODE-NI
WHERE EMPNO = ?
```

114. 得到更大的时间戳精度（V10）

在以前的 DB2 版本中，时间戳列的存储表示总是 10 字节，精度为包含 6 位毫秒。现在小数部分的位数可以多达 12 位。精度属性范围是 1 到 12，默认值仍然是 6。例如：

```
CREATE TABLE TESTBL
 (ID          INTEGER ,
  COLA_TMS    TIMESTAMP,          -- default
  COLB_TMS    TIMESTAMP(0),       -- minimum
  COLC_TMS    TIMESTAMP(6),       -- same as default
  COLD_TMS    TIMESTAMP(9),       -- precision 9
  COLD_TMS   TIMESTAMP(12)      -- precision 12 picoseconds
)
```

以前时间戳的存储长度都是 10 字节。在 V10 中，现在可以定义为 7 字节（无精度）到 13 字节（精度为 12）。在唯一键中包含

一个时间戳时，更大的精度会很有用，这样可以减少出现重复的可能性。

115. 尝试 Index Includes（V10）

使用 Index Only（仅索引）处理执行查询会非常高效。过去，通常会在索引中增加额外的列来高效地处理查询，而不必访问数据页来得到列或完成检查逻辑（见本章前面的调优技巧 #77）。

DB2 V10 增加了一个很好的索引改进，称为索引包含（index includes）。这个特性允许在索引中包含额外的列，尽管它们实际上不是键值的一部分。用 INCLUDE 选项指定的列不用于强制唯一性。这样一来，只需要一个索引就可以同时提供唯一性和仅索引访问。这还允许在索引上包含作为唯一约束一部分的列：

```
CREATE UNIQUE INDEX XEMP1 ON EMP
  (EMPNO ASC)
  INCLUDE ( WORKDEPT, SALARY)
  FREESPACE ...
  BUFFERPOOL ...
```

下面的查询就能从这个特性获益：

```
SELECT WORKDEPTNO, AVG(SALARY)
  FROM EMP
  WHERE EMPNO BETWEEN ? and ?
  GROUP BY WORKDEPT
  ...
```

116. 使用 With Return to Client（V10）

通常情况下，存储过程可能会调用其他存储过程，而这些存储过程又会进一步调用另外的存储过程。在 V10 之前，如果原调用程序希望从调用序列中的某个存储过程得到返回的结果集，只有当这个存储过程是序列中第一个调用的存储过程时，才能访问这个存储过程返回的数据。

如果一个调用程序希望从调用的一个存储过程得到返回数据，而这个存储过程并不是序列中的第一个存储过程，可以通过一个全局临时表返回。在 V10 中，现在可以使用 With Return to Client 在游标定义中编写存储过程，这会把数据通过一个结果集返回给初始调用程序。测试表明，这种新选项的性能要优于全局临时表处理。

需要说明，这个选项只允许序列中的第一个调用程序访问数据，其他中间程序都不能访问。

下面的例子显示了嵌套存储过程如何将结果集返回给调用序列中的（第一个）起始程序：

```
DECLARE C1 CURSOR WITH RETURN TO CLIENT
  FOR SELECT     ........
      FROM       .........
      WHERE    .........
;

OPEN C1;
```

第 2 章
DB2 SQL 提示

每个RDBMS都有一些可以写在SQL语句中的提示(hint)，帮助优化工具在优化时选择不同的访问路径。例如，你可能希望DB2选择一个不同的索引、一个不同的表流顺序，或者一个不同的联接方法。如果SQL语句的访问路径表现不好，而且所有其他方法（包括下面的方法）都已经尝试过，就可以试着使用这些提示：

- 用不同的方式重写查询
- 写入硬编码值，而不是使用宿主变量
- 重新组织表和索引文件
- 对涉及的表使用 runstats
- 对涉及的特定列使用频率值 runstats
- 重新设计适当的索引
- 重写谓词，不过仍保证相同的逻辑

DB2优化工具很擅长为SQL语句选择正确的访问路径，而且各个DB2版本中优化工具越来越好；不过，它还是不能做到尽善尽美。因此，有时需要为SQL语句增加SQL编码提示。如果DB2选择了一个表现不好的访问路径，通常应该首先查看谓词中相关列上的统计信息，然后分析谓词本身，查看运行时是否需要REOPT ONCE。不过还有一些语句能提供帮助，可以把它们作为最后一道防线，这一章将简要讨论。

1. 在 SQL 语句的最后使用 Optimize for 1 Row 语句

所有 SQL 语句都不应该加 Optimize for 1 Row 语句。只是当一个查询性能不佳需要调优时才可以使用这个语句。如果对一个 SQL 语句运行了 Explain，尽管 Explain 输出看起来不错，但是性能还是很成问题，那么这个语句可能会以不同的方式优化，让它有更好的表现。

这个语句告诉优化工具 SQL 的目的可能只是获取整个结果集中的一个很小的子集，并指定获取第一行有更高的优先级。如果编写了这个语句，优化工具可能选择以下做法：

- 消除排序（特别是联接排序）。优化工具会尝试选择一个能避免排序的访问路径。如果指定另一个值而不是 1 Row，DB2 会根据成本选择访问路径（不一定避免排序）。在 V10 以前的版本中，即使增加了这个语句，优化工具仍可能选择完成一个排序。V10 会选择不需要排序的成本最低的访问路径。
- 消除列表预取。请求一个特定表的数据时，优化工具通常使用预取（Prefetch）来尽量减少 DB2 获取所有这些数据所需的 Getpage 请求数。不过，为了尽量减少 Getpage 数，DB2 必须先执行一个 RID 排序。因此可以有两种选择，一个选择是尽管 Getpage 请求较多但没有 RID 排序，另一个选择是 Getpage 请求较少但有一个 RID 排序，有时前者的性能会更好。

补充说明：

- 这个语句不会阻止 DB2 获取所有满足条件的行，也不会阻止应用程序或查询返回结果集中所有符合的数据行。
- 这个语句并不总能改善性能。如果在 SQL 语句中增加这个语句，它可能会增加获取所有符合行的总耗用时间。不过，对于目前性能不好的查询，还是值得一试的。
- 这个语句只对某些查询有效（DB2 可以对这些查询递增地建立结果集）。如果一个查询需要对最终结果集完成某种排序，

这样的查询就不会由此获益。例如，大多数使用 Distinct、Group By、Order By 或 Union 的查询即使加上这个语句可能也看不出任何改变。

- 很多 OLTP 应用在每个查询上都写上了 Optimize for 1 Row，因为在大多数查询中，这会消除列表预取（List Prefetch）。优化工具通常执行列表预取来尽量减少 Getpage 数。但是大多数 OLTP 查询并不会获取太多数据，所以它们执行的 Getpage 本来就很少。通过消除列表预取，DB2 可以避免创建排序工作文件、在其中加载少量数据以及排序。去除所有这些开销会让 OLTP 响应时间稍快一点。

2. 为 SQL 查询增加 A.PKEY = A.PKEY 谓词，这里 PKEY 等于表的主键列

所有 SQL 语句都不应该加 A.PKEY = A.PKEY 语句。只是当一个查询性能不佳需要调优时才可以使用这个语句。同样地，这也不能保证查询总能更好地执行，只是可能让优化工具选择一个不同的访问路径，让它有更好的表现。

对于涉及多表联接的查询，处理表的顺序对性能会有极大的影响。有些情况下，DB2 选择的表顺序可能并不是最优的。你希望 DB2 选择正确的表（对这个表应用谓词之后返回的行数最少）。为了确保在一个多表联接中最先处理某个特定的表，可以写上 A.PKEY = A.PKEY。下面给出一个例子：

```
SELECT E.EMPNO, E.LASTNAME
FROM EMP    E,
     DEPT D
WHERE D.DEPTNO IN ('A00', 'C11', 'D21')
   AND E.SALARY > 50000.00
   AND E.JOB IN ('MANAGER', 'PROGRAMMER')
   AND D.MGRNO = E.EMPNO
```

如果 DB2 选择 DEPT 表作为起始表，要想看看如果 DB2 选择从 EMP 表开始会发生什么，可以写以下代码：

```
SELECT E.EMPNO, E.LASTNAME
   FROM EMP    E,
        DEPT D
    WHERE D.DEPTNO IN ('A00', 'C11', 'D21')
      AND E.SALARY > 50000.00
      AND E.JOB IN ('MANAGER', 'PROGRAMMER')
      AND D.MGRNO = E.EMPNO
      AND E.EMPNO = E.EMPNO
```

补充说明：

- 这个语句不一定总能改变表的顺序，不过很多情况下确实可以。
- 这个语句通常会改变表之间的联接类型。改变联接类型可能会对索引选择、列表预取、排序等带来影响。如果优化工具选择一个合并扫描或复合联接，则增加这个语句通常会把它变成一个嵌套循环联接。另外，要记住这并不表示改变所选择的优化就一定有更好的性能，只是可能让优化工具选择一个不同的访问路径，可能会让它表现得更好。
- 如果在查询中增加了这个语句，并有效地改善了优化后性能，那么一定要完善相关的文档，以免别人删除这个语句。

3. 更换索引选择

有时优化工具在确定访问路径时可能有多个索引可以选择。它能很好地选择那些提供最佳性能的访问路径。不过有些情况下，如果性能不太好，则可能希望优化工具选择另一个索引来看是否运行得更好，可以考虑以下做法：

```
SELECT EMPNO, LASTNAME, HIREDATE
FROM EMP
WHERE EMPNO BETWEEN :WS-FIRST-EMP AND :WS_LAST_EMPNO
  AND  LASTNAME LIKE 'H%'
  AND WORKDEPT IN(:WS-DEPT1, :WS-DEPT2, :WS-DEPT3)
```

每个列上分别有一个索引：

```
Index1 on Empno
Index2 on Lastname
Index3 on Workdept
```

优化工具可能选择：

- 使用 EMPNO 上的索引来得到这些行，再将它们与其他谓词比较。
- 使用 LASTNAME 上的索引来得到这些行，再将它们与其他谓词比较。
- 使用 WORKDEPT 上的索引来得到这些行，再将它们与其他谓词比较。
- 使用多个谓词组合的索引访问。

假设优化工具选择使用 EMPNO 上的索引，不过这个谓词得到的行比另外两个谓词得到的行多。为了让优化工具选择其他索引，可以对 EMPNO 应用一个 SQL 标量函数，使 EMPNO 谓词成为一个不可索引谓词。例如：

```
SELECT EMPNO, LASTNAME, HIREDATE
 FROM EMP
WHERE RTRIM(EMPNO) BETWEEN :WS-FIRST-EMPNO
                   AND :WS_LAST_EMPNO
 AND  LASTNAME LIKE 'H%'
 AND WORKDEPT IN(:WS-DEPT1, :WS-DEPT2,  :WS-DEPT3)
```

这样编写代码时，优化工具将不再选择 EMPNO 索引，现在会强制它选择另外某个索引，这样执行速度可能会快得多。不过 RTRIM 函数会让谓词成为一个 Stage 2 谓词。

以下连接也会使一个谓词成为不可索引谓词，不过仍保证它是 Stage 1 谓词：

```
WHERE EMPNO BETWEEN :WS-FIRST-EMPNO CONCAT ''
                AND  :WS-LAST-EMPNO CONCAT ''
```

如果希望优化工具使用某个索引，可以对这个索引的前导列写一个 ORDER BY，通常这会让优化工具使用这个索引。

假设希望优化工具使用某个索引，如果在这个索引中再增加列，通常会对优化工具有影响，因为它可能会完成 index-only（仅索引）处理。现在 V10 允许在一个唯一索引中包含额外的列而无需删除和重建这个唯一索引。由于能够在索引中包含额外的列，当索引基于某个约束（如主键）时，这也很有帮助。

有时优化工具会查看一个谓词对某一列所做的处理，这就会自动将它变成一个不可索引谓词。如果这个列是一个字符串，那么任何字符标量函数都适用。如果这个列是一个数值列，那么可以应用任何数值标量函数，或者对它增加 +0 。例如，下面会创建一个不可索引谓词：

```
WHERE DEC(SALARY,7,2) > 50000.00
```

以下也会创建一个不可索引谓词：

```
WHERE SALARY + 0 > 50000.00
```

如果这是一个 Date 列，则可以编写以下代码使它成为一个不可索引列：

```
WHERE DATE(HIREDATE) > '2005-01-01'
```

说明： 最好首先注意谓词列上的统计信息，尽量明确 DB2 选择索引的原因。优化工具选择访问路径时会基于相关谓词计算的筛选率。有可能 DB2 未能得到一个特定谓词的太多信息，也可能根据运行时的值计算得到了不正确的筛选率。如果时间允许，将谓词重写为不可索引谓词之前应当尽可能明确这一点。

4. 改变表处理顺序

在内联接处理中，表按什么顺序来写并不代表优化工具会选择这个顺序处理这些表。有时 DB2 会过低地估计各个表上所写谓词获取的行数。通常希望 DB2 选择返回行数最少的那个表作为起始表，这样当它联接其他表时可以相应地减少 I/O。

假设表 A、表 B 和表 C 各自在列 C1、C2、C3 和 C4 上有一个索引。如果 DB2 过低地估计了从表 A 选择的行数，并且不正确地选择它作为联接的第一个表，则可以采取以下补救措施：

- 可以增加更多谓词，以减少表 A 的估计大小。
- 可以将表 A 上的联接谓词变成一个 Stage 2 谓词，这样就不会使用联接列上的任何索引。这有一个很好的例子，如果希望首先访问某个表，则可以对这个表的联接谓词增加一个函数。这会鼓励优化工具先访问这个表，以避免对第二个表应用 Stage 2 谓词。可以用以下查询来说明：

```
SELECT *
FROM T1 A, T2 B, T3 C
WHERE (A.C1 = B.C1 OR 0=1)
  AND A.C2 = C.C2
  AND A.C2 BETWEEN :HV1 AND :HV2
  AND A.C3 BETWEEN :HV3 AND :HV4
  AND A.C4 < :HV5
  AND B.C2 BETWEEN :HV6 AND :HV7
  AND B.C3 < :HV8
  AND C.C2 < :HV9;
```

这里将表 A 和 B 之间的联接谓词变为一个不可索引谓词（不能用于单索引访问），这样做的结果是将拒绝使用列 C1 上的索引。这样一来，可能使 DB2 先访问表 A 或表 B，或者有可能使 DB2 改变表 A 或表 B 的访问类型，相应地影响其他表的联接序列。另一种方法是增加 OR 0=1，使现有的谓词成为不可索引谓词。

有时，如果希望先访问某个表，可以在这个表的索引列上写一个 ORDER By，这就会把这个表变成起始表。优化工具通常希望使用这个索引来避免排序，相应地将首先访问这个表。

5. 使用分布式动态 SQL

如今，工作室往往会处理大量动态分布式 SQL。由于有了动态语句缓存，现在查找性能不佳的动态查询比从前要容易得多。在

这个领域中，要掌握为日常执行的动态查询完成的优化（参见第1章“SQL优化技巧宝典100+”中的调优技巧#28）。对于执行SQL‘Prepare’/‘Execute’语句的程序，考虑以下做法：

- 大多数情况下，参数标记(?)会优于直接量。使用直接量时，需要与直接量完全匹配才能重用语句缓存中的优化结果。直接量应当用于值分布很不均匀的列，而且对于这种不均匀的分布要有正确的统计。

如果两个SQL语句只是直接量值不同，其他都相同，DB2就能识别出它们是相同的，这会大大减少重用动态SQL语句所需的工作。这样一来，动态SQL应用就能更多地从动态语句缓存获益，还可以减少CPU成本，并能更好地使用为缓存分配的存储空间。V10还增加了一个改进来提供帮助，称为“动态语句缓存直接量替换”。可以采用多种方法启用这个特性：

- 改变客户代码中的PREPARE语句，在其中包含新增的ATTRIBUTES子句，指定CONCENTRATE STATEMENTS WITH LITERALS。
- 改变JCC驱动程序，包含“enableliteralReplacement=‘YES’”关键字（在数据源或连接属性中指定）。
- 修改ODBC初始化文件，在z/OS中设置LITERAL REPLACEMENT，这会使得通过ODBC提交到DB2的所有SQL启用直接量替换。

通过使用参数标记，可以提供一个更高的动态语句缓存命中率，不过使用直接量通常能提供更好的访问路径。因此，是否使用这个方法要根据动态SQL调用的性能特性做出权衡。使用CONCENTRATE STATEMENTS WITH LITERALS有时可能会得到一个降级的“仅执行”（execution-only）性能。不过，包含直接量的小SQL语句会得到最大的性能提升。例如：

```
SET :HV1 = 'CONCENTRATE STATEMENTS WITH LITERALS'
```

```
PREPARE STMT
ATTRIBUTES :HV1
FROM :WS-SQL-TEXT
```

- 要了解缓存中的优化语句，可以执行以下代码：

```
EXPLAIN STMTCACHE ALL
```

这会将所有优化及其运行时信息放在一个名为DSN_STATEMENT_CACHE_TABLE的表中。可以查询这个表，查找运行时间最长的查询并查看相应的文本。可能需要运行IFCID 318 trace来掌握耗用时间和CPU统计信息。这个表中的关键列包括CACHED_TS、STAT_ELAP、STAT_CPU和STMT_TEXT。

第 3 章
SQL 标准和原则

每个 IT 工作室要开发涉及 DB2 的应用时，都应当有一组要求开发人员遵守的 SQL 标准和原则。本章可以作为起点，供开发人员和项目经理在开发中充分利用。一旦有了标准和原则，就一定要遵循。要对所有程序完成代码走查，以确保不违反这些标准和原则。

下面要介绍的标准和原则有多种用途：

- 与性能有关
- 缓解异常终止和 / 或生产异常事件报告
- 减少 I/O 和 CPU 成本
- 增加生产力
- 提高客户满意度
- 提高可读性和可理解性

以下标准和原则主要分为两个不同领域：一个面向 COBOL SQL 开发人员，另一个面向所有 SQL 开发人员（不论他们在什么语言中嵌入 SQL 代码）。

面向 COBOL 开发人员

1. 执行每一条 SQL 语句之后都必须检查 SQLCODE。Declare cursor 语句只是一个声明，不会从 DB2 得到返回码，不过所有其他 SQL 都会得到一个返回码。从 DB2 数据库系统得到的返回码会自动加载到 SQLCA 通信域。

2. 每个程序都必须包含所处理的各个表的 SQLCA 和 DCLGEN。要用与列定义匹配的宿主变量预定义 DCLGEN。它们

可以用来选择数据，插入和更新数据，以及作为 Where 子句中的宿主变量。

如果没有使用 DCLGEN 字段，那么在代码中声明变量的所有程序都必须确保所声明的变量与 DB2 中的定义完全匹配。如果不匹配，DB2 就有可能不选择索引来进行处理。例如，如果列 1 定义为 Integer，那么 COBOL 中的宿主变量应当定义为 S9(9) comp。

3. 每个程序都必须有一个一致的 DB2 异常例程。对于批量程序，最容易的做法是调用一个程序，由它处理 SQLCA 字段的显示，并调用 DSNTIAR DB2 例程来显示进一步的 DB2 消息。对于联机程序，有时可以把 SQLCA 和 DSNTIAR 信息写到一个文件或表中，以便查找出现的错误。SQLCA 包含调用的大量相关信息，这对于排错至关重要。应当写出所掌握的全部信息，这很重要。要确保至少显示 SQLSTATE 以及 SQLCODE。

4. 绝对不要在程序中写 Select *，应当只选择真正需要的列。如果一个程序需要所有列，也要逐个写出。这会避免向表中增加一个新列时出现异常。在程序中选择越少的列，处理就越高效（参见第 1 章中的调优技巧 #3 和调优技巧 #29）。更多的列会对性能产生影响（因为排序规模会更大），另外对仅索引处理以及联接类型也会有影响。DB2 查看哪种联接类型最优时，会分析从各个表选择的列数作为依据。

5. 确保定义为 Nullable 的列包含一个 null 指示符宿主变量，作为 Select、Insert 或 Update 语句的一部分。这对于 Select 语句最为重要，因为 DB2 向程序返回一个 null 列而且没有指定 null 指示符时就会返回一个非法的 -305 SQLCODE。null 指示符必须在工作存储空间中定义为 Pic S9(4) Comp。

最好为所有可能为 null（Nullable）的列写上 VALUE、COALESCE 或 IFNULL SQL 标量函数，因为这样一来程序不会从 DB2 接收 null 指示符，就能避免程序没有处理 null 指示符时出现

的 - 305 SQL 错误，而且程序不必在工作存储空间中定义 null 指示符。

例如，对于 Select COALESCE(PK_ID, 0)，如果存在这样一个 PK_ID 值，就会返回这个值，或者如果为 null，则返回一个 0。另外这里也可以使用 VALUE 和 COALESCE 函数。所有这 3 种做法都会返回相同的结果。指定的默认值必须与列定义匹配。例如，由于 PK_ID 是数值，所以默认值也必须是数值，在这里就是 0。

6. 对于包含 MIN、MAX、AVG 和 SUM 等聚集函数之一的 SQL 语句，都应当在 select 部分包含一个 Null 指示符宿主变量。如果 DB2 没有找到可以处理这些函数的数据，它会返回一个 null 指示符，COBOL 程序必须定义一个 null 指示符。如果程序没有编写一个 null 指示符，就会返回一个非法的 -305 SQLCODE。更可取的做法是编写 VALUE、COALESCE 或 IFNULL 函数来缓解 null 指示符逻辑问题。例如：

```
SELECT IFNULL(AVG(SALARY), 0)
FROM EMP
WHERE WORKDEPT = 'XYZ'
```

如果找到相应的数据行，就会返回平均值，或者如果没有满足条件来计算平均值的数据行，就会返回一个 0。

7. 尽量减少执行期间打开和关闭游标的次数。如果大多数情况下打开游标只获取一行，就应该编写一个简单的 Select 语句，而只在返回一个 -811 SQLCODE（指示重复行）时才执行游标处理。

不要把处理分解到多个游标，除非看起来性能很成问题。如果要完成一个 7 个表的联接，就应该把这 7 个表都写在一个游标里，让 DB2 来完成具体工作。如果把它们分解到多个不同的游标中，进程执行的时间通常会更长，这是因为 DB2 向进程发送 SQL 语句需要额外的时间。所以只在所有其他调优方法都已经尝试过而且无效时才可以考虑分解联接。一般来讲执行 7 表联接会更高效。

8. CASE 表达式一定要包含一个 ELSE 子句。如果 CASE 中的

所有条件都不满足，DB2 就会（通过一个 null 指示符）向程序返回一个 null。如果程序没有处理从 CASE 表达式返回的 null，就会返回一个 -305 SQLCODE，这通常会导致程序异常终止。

9. 一定要显示程序中执行的 Select、Insert、Update、Delete 和打开游标次数。COBOL 中定义计数器并在处理中完成计数器递增存在一定的开销，不过这个开销相对于程序的总体运行时间来说微乎其微。显示这些计数可以在出现问题时提供很有价值的信息，帮助开发人员确定应当查看哪个程序。要确保每次异常终止和处理结束时都要显示计数。

10. 如果一个 SQL 语句得到一个非法的 SQL 返回码，并使程序进入异常错误例程，一定要显示这个 SQL 语句中宿主变量的值。出现一个程序错误或者甚至异常终止时不知道所处理的值是什么，这会让人很郁闷，相信每个开发人员对此都深有感触。

11. 注意 SQL 语句中可能出现的 SQL 警告。大多数程序都会忽略警告，实际上很多情况下这些警告对于检测潜在的问题很有帮助。SQLCA 中有两种警告消息指示：一种是正的 SQLCODE 值（非 +100）；另一种是 SQLCA 的 SQLWARN0 字段中有一个 W。出现以上任何一种情况时，DB2 就会发出一个警告，指出前一个调用发生了某种问题，尽管你可能已经接收到返回的数据，但这可能不是你原来期望的数据。SQLWARN0 为 W 时，DB2 还会在一个或多个其他 SQLWARNn 字段中提供有关问题的更多有帮助的信息。另外还要检查每个 SQL 语句返回的警告。例如：

```
Evaluate SQLCODE
  When 0
       If SQLWARN0 = 'W'
            Display '***  Warning error ***'
            Display 'Sqlstate = ' Sqlstate
       End-If
  When Other
       ...
End-Evaluate
```

12. 充分利用 SQLCA 的 SQLERRD (3)。SQLERRD 数组的第 3 个元素是 SQLCA 中最有用的字段之一。在一次成功的插入、更新或删除之后，这个字段会填充为所插入、更新或删除的行数。没有 Where 逻辑或者受删除级联影响而完成批量删除时并不填充这个字段。

13. 充分利用游标处理中的获取行集（参见第 1 章中的调优技巧 #46）。对于大的游标应当严格强制这种做法，因为这样可以大大节省运行时间。

14. 在 COBOL 代码中完成所有计算，然后把值移入一个宿主变量，再在 SQL 语句中引用这个宿主变量。要尽可能把计算移到 SQL 语句之外。

15. 在 SQL 语句中硬编码所有已知的值。例如，如果一个程序总是处理一个表的终止行，就可以使用 SQL 语句 Where Status_Code = 'T'。如果在编目表中提供了不同 Status_Code 值的频率值统计信息，这会很有帮助（参见第 1 章中的调优技巧 #10）。

面向所有 SQL 开发人员

1. 在 Select、Where、Group By 或 Order By 子句中引用时，所有 SQL 联接语句都应当用关联 ID（Correlation ID）指示各个表中的列。关联 ID 不应该是字母表中的单个字母，而应当使用能描述含义的 ID，这样其他人就能了解各个列来自哪个表。这会让联接逻辑更清晰，也更可读。

2. 不要对 Where 子句中的列应用任何 SQL 标量函数。对于表索引中包含的列这一点尤其重要。例如，如果写 Where Integer(CLM_ID)，这会自动禁止使用 CLM_ID 上的索引。再举一个例子，下面的代码：

```
WHERE YEAR(HIREDATE) = 2003
```

应当写为：

```
WHERE HIREDATE BETWEEN '2003-01-01' and '2003-12-31'
```

这样才能使它成为一个可索引的谓词。

3. 用 DB2 Explain 工具检查查询。要让 DBA 为你的 ID 创建一个 Plan_Table，或者使用为你的 DB2 子系统定义的 Plan_Table。例如：

```
Delete from Plan_Table
;
Explain Plan Set Queryno = 11 for
 SELECT EMPNO, LASTNAME,
        FIRSTNME, WORKDEPT
 FROM EMP
 WHERE DEPTNO = ?

  ;

 Select * from Plan_Table
 Order by Queryno, Planno, Qblockno, Mixopseq
;
```

4. 当心查询中的 Order By 和 Group By 语句。这些语句都可能导致一个排序，这就需要耗用资源。应当只在必要时才写这些语句。排序中涉及的列和行越少，排序运行得就越快，所以要确保只写必要的列。

5. 编写 SQL UNION 语句时，应当从 UNION ALL 开始。如果只写 UNION，这会执行一个排序来消除重复，导致耗用更多的资源。很多情况下实际上并没有重复，所以应该选择 UNION ALL 来避免发生排序。要尽可能避免使用 UNION。有时可以使用外联接、case 语句等重写这个逻辑。

6. 当心 DISTINCT。这也会导致一个排序，需要更多的运行时间。应当只在绝对必要时才写 DISTINCT。很多情况下，可以重写这个语句来得到同样的结果，而不需要 DISTINCT，从而能更高效地运行（见第 1 章中的调优技巧 #4 ）。

7. Select 语句中使用 CASE 表达式时要当心。这个表达式在执

行期间可能会有大量开销。如果查询可能返回很多数据行，则可以将这个逻辑转移为源代码，在返回各行之后处理，这会很有帮助。如果你的源代码是编译代码，这样做尤其有好处。

8. 不要使用 Select Count(*) 来检查存在性。只有在需要总行数时才使用这个语句。最好用 FETCH FIRST 1 ROW ONLY 编写一个 Select 语句，然后检查 SQLCODE = 0 还是 +100。

9. 如果使用 V9，一定要查看性能监控和调优指南，对于 V10 则要参考管理性能指南，来了解如何编写（或者不编写）谓词，使它们成为可索引谓词和 / 或 Stage 1 与 Stage 2 谓词（参见第 1 章中的调优技巧 #14）。IBM Data Studio Visual Explain 工具还会指出所有 Stage 2 谓词。

10. 当心 <>（不等于）谓词。这些谓词是不可索引的，不过它们是 Stage 1 谓词。

11. 要了解内联接和外联接。很多情况下，SQL 会写为 Table1 外联接到 Table2，然后内联接到 Table3。最后写的内联接可能会取消外联接中的例外情况。很多情况下，这 3 个表都可以写为内联接，这样运行得更高效。外联接并不是效率不高，而是如果它们引入额外的例外数据行，后续的一个内联接将消除这些额外的行，所以外联接中对这些例外数据行的处理就是多余的。

另外，要确保如果编写外联接，程序一定要处理未满足联接而从表返回 null 的情况。应当使用 VALUE、COALESCE 或 IFNULL 函数来避免 DB2 向程序发回一个 null 指示符。

12. 一般来讲，应当避免使用“非”（NOT）逻辑。尽量保证谓词是正面而不是负面的。例如，下面的谓词：

```
WHERE NOT HIREDATE > :WS-DATE
```

可以写为：

```
Where HIREDATE <= :WS-DATE
```

13. 编写谓词时，尽量不要在逻辑中处理列，以保证它是一个

可索引的谓词。例如：

```
WHERE SALARY * 1.10 > 100000.00
```

是一个不可索引谓词，应当写为：

```
WHERE SALARY??> 100000.00 / 1.1
```

14. 对一个日期使用日期标示时段时（加减年 / 月 / 日），按什么顺序编写和执行非常重要。例如，增加时段时，顺序应当是年在最前面，然后是月，再是日：

```
SELECT CURRENT DATE + 2 YEARS + 3 MONTHS + 1 DAY
```

减去时段时，顺序则正好相反：日在前，然后是月，再是年：

```
SELECT CURRENT DATE - 1 DAYS - 3 MONTHS - 2 YEARS
```

这很重要，因为如果用不同的顺序写，结果可能是不正确的！由于日期按月份调整，结果可能会不同。例如，从 3 月 31 日减去一个月会得到 2 月 28 或 29 日。

15. 如果需要知道一个月的最后一天，可以使用 Last_Day SQL 函数来得到。例如：

```
SELECT LAST_DAY(CURRENT DATE)
INTO :HV1           --  Where HV1 is some Host Variable
FROM SYSIBM.SYSDUMMY1
```

16. 要得到与以上技巧 #15 同样的结果，更高效的做法是使用 Set 语句。例如：

```
SET :HV1 = LAST_DAY(CURRENT DATE)
```

说明：尽可能采用设置宿主变量的方法而不是使用 SYSIBM.SYSDUMMY1，特别是运行时语句可能执行数百次或上千次时，设置宿主变量会高效得多。

---

17. 充分利用 SQL 中的日期函数，而不是自己编写代码来提供所需的信息：

Year/Month/Day 只返回日期值中相应的部分（年 / 月 / 日）。

DAYOFWEEK/DAYOFWEEK_ISO 返回一个数 (1–7)，取决于一周是从星期天还是星期一开始。DAYOFWEEK_ISO 指出星期一

作为一周的第一天。

DAYOFMONTH/DAYOFYEAR 返回月中的天数（1–31）或一年中的天数（1–366）。

LAST_DAY 返回一个特定日期当月的最后一天。如果指定日期为 10/15/2005，那么返回的日期就是 10/31/2005。

NEXT_DAY 返回一个时间戳，表示大于指定日期的第一个工作日。这个函数需要指定工作日。例如：

```
NEXTDAY('01/31/2005', 'MON')
```

会返回‘01/31/2005’之后下一个星期一的日期。

DAYS 用来得到两个日期之间所差的天数。例如：

```
SELECT DAYS(HIREDATE) - DAYS(BIRTHDATE)
```

会返回相差的天数。

WEEK 返回一个数 (1–54)，表示一年中的周数。WEEK 1 就是包含当年第一天的第一周。

WEEK_ISO 返回一个数 (1–53)，表示一年中的周数。WEEK 1 是当年包含一个星期四的第一周，这等价于包含 1 月 4 日的第一周。

CHAR 用来得到以指定格式（USA,ISO 或 JIS, EUR）返回的一个日期列。

两个日期相减会返回一个十进制数，包含这两个日期之间相差的年、月和日：

```
SELECT DATE('2010-01-01') - DATE('2007-10-15')
FROM SYSIBM.SYSDUMMY1
```

这会返回 20217，表示 2 年 2 个月 17 天。如果只想得到相差的年数，可以写为：

```
SELECT YEAR(DATE('2010-01-01') - DATE('2007-10-15') )
```

18. Not Between 是不可索引的。例如，以下谓词：

```
WHERE SALARY NOT BETWEEN 50000.00 and 100000.00
```

是一个不可索引谓词，应该写为：

```
WHERE SALARY < 50000.00
```

```
OR SALARY > 100000.00
```

19. 当心 Like 谓词。如果 Like 语句是一个 Begins With 谓词，那么这个谓词是可索引谓词。如果 Like 语句是一个 Contains 或 Ends With 谓词，则是不可索引谓词。例如：

```
WHERE LASTNAME LIKE 'A%'      - Begins with logic
WHERE LASTNAME LIKE '%A%'     - Contains logic
WHERE LASTNAME LIKE '%A'      - End with logic
```

20. Select 中只写必要的列。多余的列可能导致优化工具选择一个不同的访问路径，这可能不是最佳的选择。多余的列会使排序开销更大，而且会增加传输成本。甚至一个多余的列（有时）也可能导致优化工具选择一个不同的访问路径。基本说来，结果集越大，DB2 需要获取和传输的数据就越多。

21. 如果查询和 / 或游标只是在结果集中返回多行，则应该在查询末尾加上 For Fetch Only。这会告诉 DB2 不打算更新所获取的数据行。这样一来，DB2 就会避免锁定页，可能阻塞返回的数据行。For Read Only 也有同样的作用。

22. 先编写最受限的谓词。这并不是说 DB2 会按这个顺序执行查询。DB2 总是先选择 Stage 1 可索引谓词（不论它们放在哪里），不过，对于同一个阶段的谓词，使用正确的顺序很重要。

23. 重写 > Any 和 > All 子查询。例如，要把下面的查询：

```
SELECT EMPNO, LASTNAME
From Emp
Where Salary > Any
             (Select Salary
              from Emp
              Where Workdept = 'C11')
```

重写为：

```
Select Empno, lastname
From Emp
Where Salary >
              (Select Min(Salary)
               from Emp
               Where Workdept = 'C11')
```

第 4 章
SQL 程序走查

要让应用顺利进入生产阶段，程序走查是最重要的任务之一。正如本书最前面提到的，尽管世界各地的工作室有大量 SQL 开发人员，但他们中很多人根本不知道或者不考虑某些需要审查并指出的编码问题。通过审查代码，完成检查并询问以下问题，开发人员可以确保应用有更好的表现，从而大大减少生产阶段报告的缺陷和异常。

管理层总是说没有时间做代码走查。不过只需根据本章列出的各个项目做一个快速的 15 分钟检查，就能够消除很多潜在的问题。很多工作室的管理层居然不了解不好的程序设计或糟糕的 SQL 代码可能带来的严重后果，在我看来这实在让人很难理解；由于代码质量差，原本只需运行几分钟的程序现在要运行数小时，或者本该运行几秒而实际上需要运行几分钟。完全可以把走查作为一种培训，来提高开发人员的工作能力。而且走查不仅与性能有关。通过审查程序代码，可以减少程序提升到生产阶段之后可能产生的问题。开发人员可能因为大量生产阶段问题而颇受诟病，不过我认为管理层在这方面也是有责任的，他们本该起码设置一个 15 分钟的走查过程来尽量减少可能在生产阶段出现的问题。

有些环境使用了代码生成器（如 Hibernate），而且会执行大量动态 SQL，这使得走查变得更为困难。开发人员最好用某种方式掌握所执行的 SQL，以执行 DB2 Explain。

我说过，不论管理层找什么借口推托，总能找出时间来完成一个快速的程序走查。下面是走查应包含的内容：

1. 每次审查时，所有程序员要提供以下内容：
   - 最新的编译代码副本。
   - 最新的 DB2 Explain 副本。
   - 最新的执行版本副本。从中可以了解 CPU 时间、具体执行时间、控制计数（打开游标数、插入数、删除数、随机选择数、获取的行数、存储过程调用数，等等）。
2. 分析 DB2 Explain 中的各个 SQL 语句：
   - 检查所有表空间扫描。
   - 检查所有非匹配索引扫描（匹配列为 0 的索引）。
   - 检查发生的所有排序。是否需要排序？有没有其他办法编写查询来消除排序？排序有时对性能有帮助，可以避免随机 I/O。在消除排序之前要明确哪个排序会对性能有负面影响。
   - 检查联接方法以及联接方法的相关排序。
3. 分析代码中的各个 SQL 语句：
   - 内联接是不是应该写为外联接？了解各个联接中所有表之间的关系。确保开发人员理解这些关系，并正确编写联接。
   - 某些外联接是否应当写为内联接？
   - 检查所有 Union SQL 语句。是否需要写为 Union，或者 SQL 语句是否可以写为 Union All？
   - 检查各个查询中的所有列。其中有没有可为 null 的列？是否已编写相应的 SQL 来处理某些行不存在具体值时返回的 null 值？这会消除所有 SQL 异常终止（返回码 -305）。
   - 各个 SQL 语句中选择的所有列是否都需要？
   - 对于每个 AVG、MIN、MAX 或 SUM 语句，确保已经编写相应的 SQL 来处理 null 值。这会消除所有 SQL 异常终止（返回码 -305）。

- 检查所有 Order By 语句。真的需要这些 Order By 语句吗？有没有一个索引支持排序？有时如果与一个索引匹配，Order By 可能非常高效。
- 检查所有 Distinct 语句。是否可能出现重复？能不能用 Group By 编写这个语句，或者能不能重写为一个关联或非关联子查询？在 V9 中，Distinct 的做法与 Group By 语句类似，因为 DB2 会检查适当的索引以尽量避免排序。
- 检查所有 CASE 表达式，确保有一个 Else，以涵盖所有未满足条件的情况。这会消除所有 SQL 异常终止（返回码 -305）。
- 检查所有谓词，确保未对任何列应用标量函数。保证所有数学计算都在操作符的另一边完成，而不是应用在列本身。
- 查找所有“非”（Not）逻辑。很多情况下，可以采用一种正面（而非负面）的方式来编辑谓词，这样会更为高效。几乎所有包含非逻辑的谓词都是不可索引谓词。
- 为了保证可读性和可理解性，确保所有联接和关联子查询中都有关联 ID。要保证不仅联接谓词中写有关联 ID，还要在 Select、Order By、Group By 等中写有关联 ID。不要把关联 ID 写为 A、B、C 和 D 等单字母值；应该使用有含义的词，以使 SQL 语句更可读。
- 每个谓词另写一行。
- 多表联接中的各个表都另写一行。
- 要设置适当的缩进，使 SQL 更简洁、更可读。如果查询变得太过复杂，可以利用一些 SQL 格式化网站提供帮助，比如 Instant SQL Formatter、SQL Format 和 Free SQL Formatter，就是一些不错的 SQL 格式化网站。另外 Data Studio 也提供了一个查询格式化工具。

4. 分析总体程序逻辑：

- 有没有其他方法可以更快地得到最终结果？很多情况下，开发人员可能不知道还有其他方法可以处理数据，以建立报告或结果集。程序走查可以提供一个绝好的机会来审查其他方法。

在处理大量数据的批量程序中：

- 可以尽可能减少 I/O 吗？
- 可以减少 CPU 使用吗？
- 可以减少传递到数据库管理器的 SQL 请求吗？
- 可以充分利用（或构建）总结表、全局临时表或物化查询表来帮助进程尽可能减少 I/O 吗？

5. 检查所遵循的编程标准。编写编程标准有很多原因。其中最常见的原因是保证可读性、效率和尽可能减少生产阶段事件报告。如果让所有开人员都按他们习惯的方式编写代码，会出现很多问题。要确保所有开发人员都了解这些编程标准，更重要的是，在代码提升到生产阶段之前，要确保有人完成某种质量保证审查，保证确实遵循了这些标准。

第 5 章
检查存在性

程序中很多查询可能会执行 SQL Select Count(*) 语句，不过其目的只是为了查看是否存在包含某些数据的数据行。很多情况下，有一行或是有一百万行并不重要，关键是是否存在这样的数据行。在这种情况下，使用 Select Count(*) 来完成这种检查是一种成本最昂贵的做法，因为这样一来会计算所有行。应当巧妙地编写查询，确保命中第一行时就停止（当然除非确实需要得到总行数）。还可以采用另外一些方法编写代码来检查存在性，这些方法也可能效率不高，本章会列出这样一些方法，并指出如何将它们重写为更高效的代码。

例 1

下面是一个常见的 SQL 语句例子，其目的是根据宿主变量中的数据查看是否有数据行满足 WHERE 逻辑中的条件。在编程中，有多少行满足条件通常并不重要，关键是是否有满足条件的数据行。这不是一个检查存在性的好方法，因为这里会查看所有数据并计算总行数：

```
MOVE ZEROS TO HOST-VARIABLE1.
EXEC SQL
     SELECT COUNT(*)
     INTO :HOST-VARIABLE1
     FROM    TABLE1
     WHERE   COLUMN1     =   :HOST-VARIABLE-X
     AND   COLUMN2     =   :HOST-VARIABLE-Y
 END-EXEC
```

```
IF    HOST-VARIABLE1    >   ZERO
   SET  ROWS-FOUND   TO TRUE
END-IF
```

说明： HOST-VARIABLE1 包含满足条件的总行数。

执行存在性检查时，关联 EXISTS 子查询通常优于非关联 IN 子查询。要编写存在性检查，可能最好的办法是使用 Fetch First 1 Row Only 子句。

检查存在性的一种较早的办法是使用 SYSIBM.SYSDUMMY1 表，其中只包含一行一列。这是一个 dummy 表，它是 DB2 的一个系统表，在 SQL 中有很多不同的用途。在以下子查询中编写联接语句 D1.IBMREQD = D1.IBMREQD（使它成为一个关联子查询），可以确保这个 Exists 子查询在命中满足条件的第一行之后就停止。这可以免去数百甚至上千个 DB2 Getpage 请求，从而大大减少运行时间。这种方法的效果可能没有 FETCH FIRST 1 ROW ONLY 那么明显，不过确实很重要，因为在可以使用 Fetch First 1 Row 之前这是检查存在性的一种高效方法：

```
MOVE ZEROS TO HOST-VARIABLE1

SELECT  1
INTO    :HOST-VARIABLE1
FROM SYSIBM.SYSDUMMY1 D1
WHERE EXISTS
       (SELECT 1
        FROM    TABLE1
        WHERE   COLUMN1    =   :HOST-VARIABLE-X
             AND   COLUMN2    =   :HOST-VARIABLE-Y
             AND   D1.IBMREQD  =  D1.IBMREQD)

EVALUATE SQLCODE .....

IF    HOST-VARIABLE1    >   ZERO
   SET  ROWS-FOUND   TO TRUE
END-IF
```

对于 DB2 V7，带 FETCH FIRST 1 ROW ONLY 子句的 SQL 与关联子查询同样高效，如下所示，这是一种更直接的编码风格。前面已经提到，使用 SYSDUMMY1 检查存在性是一种比较老的方法，更新的、更可取的方法是使用 Fetch First 1 Row Only：

```
MOVE ZEROS TO HOST-VARIABLE1
SELECT  1
INTO    :HOST-VARIABLE1
FROM    TABLE1
WHERE   COLUMN1     =   :HOST-VARIABLE-X
  AND   COLUMN2     =   :HOST-VARIABLE-Y
FETCH FIRST 1 ROW ONLY

EVALUATE SQLCODE .....

IF SQLCODE = 0
    SET  ROWS-FOUND   TO TRUE
END-IF
```

例 2

从下面的例子可以看到较早的代码中采用的另一种方法，如果根据宿主变量中的值判断满足逻辑，则会在 HOST-VARIABLE1 中放入值 1。这种方法没有使用关联子查询或‘FETCH FIRST 1 ROW ONLY’那么高效：

```
MOVE ZEROS TO HOST-VARIABLE1.
 SELECT 1
 INTO :HOST-VARIABLE1
 FROM SYSIBM.SYSDUMMY1
 WHERE EXISTS
   (SELECT 1
    FROM    TABLE1
    WHERE   COLUMN1     =   :HOST-VARIABLE-X
      AND   COLUMN2     =   :HOST-VARIABLE-Y)
```

有些代码可能会用一个非关联子查询检查存在性。这种做法也不算高效，因为在指出存在性之前，子查询会检查所有满足条件的行。

同样，编写这个语句的更高效的做法如下，这是因为根据逻辑

找到第一行并在 HOST-VARIABLE1 中放入值 1 时逻辑就会停止：

```
MOVE ZEROS TO HOST-VARIABLE1.

SELECT 1
INTO :HOST-VARIABLE1
FROM    TABLE1
WHERE   COLUMN1     =   :HOST-VARIABLE-X
  AND   COLUMN2     =   :HOST-VARIABLE-Y
FETCH FIRST 1 ROW ONLY
```

第 6 章
Runstats

出现 SQL 性能问题的主要原因通常与缺乏足够的统计信息有关，即对所涉及的表没有充分的统计信息。从 DB2 V8 开始引入了一个很好的特性：对索引和非索引列都能够运行基数和频率值统计。这会为优化工具提供这些列上编写的谓词的更多有关信息，从而能更准确地预测筛选率。在 V8 之前，这些统计只能在索引列上运行，而且只能在前导列或前导列组合上运行。

对于多表联接，优化工具通常会对哪个表是前导（组合）表做出不正确的选择。DB2 应当选择筛选量最大的表，但是如果谓词列的统计信息不充分，DB2 就可能选择错误的起始表，这通常会导致额外的 I/O。

不充分的统计信息还可能导致优化工具对索引和 / 或表之间的联接方法做出错误的选择。对于单表访问，优化工具可能强制选择一个不正确的索引，或者不选择索引。

为了保证主动，IT 工作室应该采取下面的做法：

1. 所有表的每一列都应该有基数统计，或者至少在 SQL 代码中的 Where 逻辑中使用的那些列上要有基数统计。

2. 如果表中包含的列数据分布不均匀，就应该对这些列运行频率值统计。例如，假设 EMP 表中的 SEX 列包含 2 个值（M 和 F）。如果包含其中一个值的行的百分比较高（或者比较低），就应该在这一列上运行频率值统计。如果没有频率值统计，DB2 会假设为均匀分布（50% M，50% F）。

3. 如果表中包含的某些列有一个默认值，而且包含这个默认值

的行很多，就应该在这一列运行频率值统计。例如，如果一些数值列默认值为0，而且包含0的行占很大比例，那么DB2应该在这些列上运行频率值统计来了解这一点。

4. 有些SQL查询包含一些谓词，而且相应的谓词列有频率值统计，那么DB2必须知道这些SQL查询正在处理什么值。要让DB2知道这一点，唯一的方法就是硬编码写入所处理的值，使用REOPT参数绑定程序，或者将SQL语句写为动态SQL语句。

Java程序包含参数标记时，通常会执行SQL Prepare语句。不过对于数据分布不均匀的特殊列，应该在接收值并移至变量之后才完成SQL Prepare（参见第1章中的调优技巧#28）。

要知道在表中各列上运行了哪些频率值统计，可以参考SYSIBM.SYSCOLDIST编目表。IBM Data Studio工具也会显示这些统计信息。

Runstat原则：如果有以下情况，就应该运行Runstat工具。

- 加载一个表时
- 创建一个索引时
- 重新组织一个表空间时
- 表空间中有大量更新、删除或插入时
- 将表空间恢复到之前某个时间点之后

5. 现在z/OS V9还允许直方图统计，不过实际上面向Linux®，UNIX®和Windows®平台的DB2早已经提供了这些统计信息。这些特殊的统计信息会对一个特定列的数据完成汇总，将数据分解到不同区间（称为分位数（quantiles））。频率值统计只针对列的一个值子集（即只对这个子集收集频率值统计），与频率值统计不同，直方图统计会覆盖所有值（不过会分解到不同区间）。利用这些统计，能显著改善DB2优化。

例如，列DEPTNO的直方图统计信息可能如下所示：

```
Values 'A00' to 'F22'1% of the data
```

```
Values 'F23' to 'H22'5% of the data
etc...
```

这对于区间类型的谓词（如>、>=、<、<=、Between 和 Like）尤其有帮助。对于优化工具来说，与查找一个特定值的谓词不同，区间谓词总是很难计算一个真实的筛选率。如果一个列经常在区间谓词中使用，但没有特定的值作为异常数据（outlier），此时直方图统计就很有帮助，可以更好地预测筛选率。如果值区间中存在数据“热点”，直方图统计也会很有帮助。另外对其他情况也是有意义的。

第 7 章
查询初始调优步骤

开发人员通常会面对这样一个问题：程序或 SQL 语句本该运行得更快，但实际上并非如此。很多情况下，他们不确定该从哪里修正这个问题。下面给出一个步骤列表，可以按这些步骤来查找问题，改善性能：

1. 检查谓词。不论程序中有一个查询还是多个查询，要检查每一个查询中的每一个谓词，确保这些谓词是可索引的 Stage 1 谓词，而且尽可能简单直接。

2. 如果查询中有一个 Distinct 或 Group By，要确认这是否必要，然后查看 Explain，分析这是否会导致排序。如果这个 Distinct 或 Group By 是必要的，还可以采用另外一种方法重写这个查询来处理重复问题，同时不会带来排序（参见第 1 章中的调优技巧 #4）。

3. 执行 Explain。在 Explain 输出中检查以下内容：

   - 是否出现表空间扫描？如果是，参考第 1 章调优技巧 #59 中的清单明确原因。
   - 是否出现排序？如果查询包含 Union、Distinct、Group By 或 Order By，是否确实需要？
   - 如果涉及联接，表按什么顺序处理？ DB2 应当选择筛选量最多的表作为起始表。如果没有选择筛选量最多的表，则要检查谓词列，确保这些列上有足够的统计信息来帮助优化工具（参见第 1 章中的调优技巧 #12，另外可以参考第

6 章）。要确定哪个表筛选最多，必须知道运行时的值。可以执行 Select count(*) 语句来确定。

- 所有关联子查询都应当使用一个索引，而且应当尽可能指定 indexonly = yes（参见第 1 章中的调优技巧 #17 ）。关联子查询是指包含与外表某一列联接的子查询。
- 所有嵌套循环联接操作应当使用一个有匹配列的索引来处理所联接的表。如果起始（组合）表显示有一个表空间扫描，这可能不算什么问题，因为它只扫描一次。但是对于联接表，表空间扫描可能导致表被扫描很多次。

4. 查看各个表以及各个谓词列的 DB2 统计信息。确保统计信息是最新的，另外要确保数据分布不均匀的列上有频率值统计信息。可以在以下 DB2 编目表中查看到统计信息：SYSTABLES、SYSCOLUMNS、SYSCOLDIST 和 SYSINDEXES。使用 IBM Data Studio 工具也可以很容易地查看统计信息。

对于分区表，统计信息存在于 SYSTABSTATS、SYSCOLSTATS、SYSCOLDISTSTATS 和 SYSINDEXSTATS 中的每个分区。

5. 对于有频率值统计信息的列，DB2 要知道这些值，不过通常在绑定或准备时并不知道这些宿主变量的值（参见第 1 章中的调优技巧 #10）。

- 对于完成静态绑定的程序，需要在代码中为谓词硬编码写这个值，或者使用一个 REOPT 参数重新绑定程序（参见第 1 章中的调优技巧 #51 ）。
- 对于执行 SQL Prepare 语句的程序，可能需要对分布不均匀的列硬编码写入这个值。

6. 检查所选择的索引。对于所有这些谓词，DB2 是否可以使用其他可能的索引？处理索引时，是否与某些列匹配？可以利用

IBM Data Studio 工具来了解这些信息。

7. 是否有子查询谓词？可以把 IN 子查询改为 EXISTS 子查询，或者反之，查看这样是否有帮助。如果 SQL 语句中有多个子查询，就要先编写最受限的子查询（参见第 1 章中的调优技巧 #16）。

8. 检查程序中执行的 SQL 语句次数。是否有方法可以减少 OPEN 游标、SELECT 等的数量？（参见第 1 章中的调优技巧 #11。）

9. 在查询末尾尝试使用 Optimize for 1 Row（参见第 2 章提示 #1）。这不会改变逻辑，但是有可能改变选择的优化路径。要记住，增加这个语句不一定能改善查询的性能，不过这会让你有机会尝试用另外一种方法来完成优化。

10. 尝试 A.PKEY = A.PKEY，其中 A 表是你认为应该选择的起始（组合）表。（参见第 2 章提示 #2 ）。还要记住，这样不一定能改善查询的性能，不过这会让你有机会尝试用另外一种方法来完成优化。

11. 是否过高估计了查询中谓词的筛选率？如果 DB2 认为一个谓词会返回 24% 的数据，但是实际上只返回了 4% 的数据，这意味着 DB2 需要这一列更进一步的统计信息。这可能是查询整体优化的问题。通过使用 IBM Data Studio 工具可以很容易了解这一点。

12. 检查每个谓词。列上是否有函数或数学运算？它们是不是 Stage 2 谓词？能不能重写得更高效？

13. 有没有区间谓词？直方图统计有没有帮助？

14. 能不能用不同的方式重写查询和谓词，而且仍得到同样的结果？通常查询重写后会采用不同的方式优化，可能让优化工具选择一个不同的路径，有可能更为高效。

15. 如果能用 IBM Data Studio 工具提供查询解释和 SQL 调优，

要知道这个工具中有一个区域能提供它认为对查询有帮助的推荐数据统计信息。开发人员可以检查这些统计信息，这也是一个很好的选择。IBM 就曾在 IDUG 会议上指出，对于客户提出的很大一部分查询问题，就是利用这个工具得到推荐统计信息来解决的。另外，Runstat 工具的“免费”版本就有这个推荐功能。

附录 A
谓词重写示例

本附录给出了一些谓词重写例子，通过重写原来性能不佳的谓词，可以得到更高效、更简单、更可读和更可理解的谓词。对于 DB2 V8，这些谓词是 Stage 2 谓词，在将来的版本中可能会有所改变。有时可能不确定该如何重写谓词，希望下面这些例子能助你一臂之力。

说明： 要记住，编写 Between 谓词时，所检查的值会包含在内。对于 Not Between 谓词，则不包含指定值。另外，编写 Between 和 Not Between 谓词时，要先写低值，这在逻辑上很重要。

原谓词	谓词重写
`Where Year(Hiredate) = 1980`	`Where Hiredate between '1980-01-01' and '1980-12-31'`
`Where '2000-01-01' between Date_Col1 and Date_Col2`	`Where Date_Col1 <= '2000-01-01' and Date_Col2 >= '2000-01-01'`
`Where '2000-01-01' not between Date_Col1 and Date_Col2`	`Where Date_Col1 > '2000-01-01' Or Date_Col2 < '2000-01-01'`
`Where digits(Edlevel) = :hv3`	`Where Edlevel = smallint(:hv3)`
`Where Hiredate not between '1985-01-01' and '1985-12-31'`	`Where Hiredate < '1985-01-01' Or Hiredate > '1985-01-01'`
`Where Empno LIKE '0000' CONCAT '%'`	`Where Empno LIKE '0000%'`
`Where Hiredate + 7 days > :hv-date`	`Where Hiredate > date(:hv-date) - 7 days`
`Where Salary * 1.1 > 60000.00`	`Where Salary > 60000.00 / 1.1`

（续）

原谓词	谓词重写
Where Hiredate + 1 month> :hv-date	Where Hiredate > date(:hv-date) - 1 month
Where Empno = (Select max(Empno) From Emp Where …)	If Empno is defined as 'Not Null' Where Empno = (Select coalesce(max(Empno), 0) From Emp Where …_
Where substr(Deptno,1,1) = 'A' Where(d.Deptno > :hv- deptno and d.Loc = :hv-loc) or(d.Deptno > :hv-deptno and d.Admrdept = :hv-admrd-ept)	Where Deptno like 'A%' Where d.Deptno > :hv-deptno and (d.Loc = :hv-loc or d.Admrdept = :hv-admrdept)
Where Bonus = :hv1 + :hv2	Compute hv3 = hv1 + hv2 Select … From … Where Bonus = :hv3 Or Where Bonus = Dec(:hv1 + :hv2,9,2)
Select d.Deptno, d.Deptname From Department d left join Employee e on d.Deptno = e.Workdept Where e.Workdept is null	Select d.Deptno, d.Deptname From Department d Where not exists (Select 1 From Employee e Where d.Deptno = e.workdept)
If date columns are defined with a default of '9999-12-31' in a table. Where Date_Cola <> '9999-12-31'	Where Date_Cola < '9999-12-31'

谓词重写：传递闭包

在 V9 中，传递闭包（Transitive closure）以 <COLA op value>

类型的谓词出现。不过，与其他谓词类似，IN 谓词或子查询谓词对于传递闭包不可用（参见第 1 章中的调优技巧 #86）。在下面的几个例子中，传递闭包不可用，需要增加额外的谓词。这些例子针对 IN 谓词类型，不过对 LIKE 谓词也同样适用。

无传递闭包	有传递闭包
Select d.Deptno, d.Deptname, e.Lastname Frpm Department d, Employee e Where d.Deptno = e.Deptno and d.Deptno Like 'A%'	Select d.Deptno, d.Deptname, e.Lastname From Department d, Employee e Where d.Deptno = e.Deptno and d.Deptno Like 'A%' and e.Deptno Like 'A%'
Select d.Deptno, d.Deptname, e.Lastname, From Department d, Employee e Where d.Deptno = e.Deptno and d.Deptno in (?, ?, ?) ** V10 will now provide transitive closure for IN predicates.	Select d.Deptno, d.Deptname, e.Lastname, From Department d, Employee e Where d.Deptno = e.Deptno and d.Deptno in (?, ?, ?) and e.Deptno in (?, ?, ?)
Select d.Deptno, d.Deptname From Department d Where Exists (Select 1 From Employee e Where e.Workdept = d.Deptno and e.Workdept in (?, ?, ?)) ** V10 will now provide transitive closure for IN predicates.	Select d.Deptno, d.Deptname From Department d Where Exists (Select 1 From Employee e Where e.Workdept = d.Deptno and e.Workdept (?, ?, ?)) And d.Deptno in (?, ?, ?)

附录 B
DB2 SQL 术语

每个 DB2 应用都需要 3 个部分来完成操作：系统、数据库和应用本身。系统（system）是指 DB2 子系统安装，数据库（database）表示维护应用数据的数据库对象，应用（application）是为用户处理数据的宿主语言和 SQL 代码。为了提供优良的 DB2 性能，系统程序员、数据库分析人员和开发人员必须能够对这些部分进行监视和调优。

基数

基数（cardinality）通常是指对于某一列，表中所有数据行中该列不同值的个数。例如，假设 EMP 表有 42 行，共中列 EMPNO 有 42 个不同的值。那么就称 EMPNO 为高基数列。列 SEX 有 2 个值，则称它为低基数列。

聚簇索引

聚簇索引（clustering index）是确定表空间中如何对行物理排序（聚簇）的索引。如果一个分区表的聚簇索引不是一个分区索引，那么行会在各个数据分区中按聚簇顺序排序，而不是分散在分区中。在 z/OS V8 的 DB2 Universal Database ™以前，分区索引必须是聚簇索引。

DB2 应用性能

应用性能（application performance）由两部分组成：应用宿主语言代码和嵌入在应用程序中的 SQL 代码。很多语言（包括 SQL）学起来很容易，不过要想真正掌握却不那么简单。要想全面了解 SQL 的很多不完备性对性能的影响，这可能很困难。这本书

主要强调 DB2 优化方面 SQL 语言的性能低效问题。

DB2 数据库对象

DB2 数据库对象（DB2 database object）包括数据库应用中的表空间、表和索引。要确保得到高效的数据库对象，数据库设计是一个非常重要的要素。这通常意味着每个应用都要有一个充分规范化的逻辑数据模型。数据初始加载之后，要定期地对这些数据库对象运行统计，这很重要，这样 DB2 才能更多地了解数据结构中的数据分布。

DB2 子系统

性能和调优包括管理内存（缓冲区池、EDM 池、RID 池、排序池，等等）、管理 DASD 存储空间、适当地使用数据库日志、管理操作系统资源、配置锁定系统参数、管理 DB2 数据共享以及很多其他任务。一般认为子系统（subsystem）是一个拥有这些资源的环境。

筛选率

对于每个查询中的每一个谓词，DB2 要确定它的筛选率（filter factor），即 DB2 所估计的受这个谓词影响的行的百分比。例如，员工表有 42 行，其中 EMPNO 列是主键。考虑以下代码：

```
WHERE EMPNO = :HV-EMPNO
```

DB2 会估计筛选率为 1/42, 即 0 .02381% 的数据会受影响。受影响的总行数是 42 * 0.02381 = 1。这个谓词的筛选率就是 0.02381。

再来看一个例子，对于下面这个谓词：

```
WHERE WORKDEPT = :HV-DEPTNO
```

DB2 估计筛选率为 1/8 或 0.125%，因为根据基数统计，它知道总共 42 行中列 DEPTNO 有 8 个不同的值。受影响的总行数是 42 * 0.125 = 5.25。

频率值统计

除了基数统计，还有一种 Runstat 统计称为频率值（frequency value）统计。这种统计会按特定的值列出表中受影响的行的百分比。例如，如果在 EMPLOYEE 表的 WORKDEPT 列上运行频率值

统计，DB2 会知道 8 个不同值的有关信息：所以，对于以下谓词：

```
WHERE WORKDEPT = :HV-DEPTNO
```

DB2 认为筛选率是 0.125。对于下面的谓词：

```
WHERE WORKDEPT = 'D11'
```

DB2 知道准确的筛选率是 0.26。

频率值统计对于优化工具很有帮助，因为它能确切地知道受影响的具体行数，不过只有当绑定或准备时知道具体值时这种做法才有效。为了知道具体的值，必须硬编码，或者在绑定过程中使用 REOPT 参数，这样会在运行时（将值放入宿主变量中时）再做一次优化。

不准确、过期或缺少统计

这些是导致应用性能低下的首要原因。可以通过 DB2 Runstats 工具积累和加载统计信息。性能低下的另一个原因可能是对存放文件的表空间文件管理不当。这些文件有物理结构，需要适当地确定大小来完成初始加载和以后的扩展。如果分解为分区（extent）或分段（fragmentation），或者更糟糕的，如果空间耗尽，就可能导致性能问题。

索引后备

如果程序处理中反复地检查一个索引，DB2 就会查看最近处理的当前索引叶页面，来看在该页面上是否能找到索引键。如果上一个页面中不存在，DB2 就会从根页面向下执行索引处理。如果找到，DB2 会使用同一个叶页面上的值。这是一个很重要的原因，正是因此很有必要以有序的顺序执行数据处理。确保索引后备（index lookaside）处理的关键一点是，要用参数 RELEASE (DEALLOCATE) 绑定程序。这个绑定参数对远程连接无效，但是对批量处理很有用。

可索引谓词

可索引谓词是指这样一类谓词：对于这种谓词，优化工具可以匹配谓词列上的索引项（如果存在）。例如，下面的谓词是不可索引的：

```
Where Hiredate <> '1985-01-01'
or Where Year(Hiredate) = 1985
or Where hiredate - 1 month < :hv-date
```

这说明，即使列 HIREDATE 上有一个索引，也不会发生匹配索引处理。也有可能选择这个索引完成处理，但是这会是一个索引扫描，没有匹配列。

内联接

内联接（inner join）是一个联接操作的结果，只包含所联接表的匹配行。

联接方法

对于 OS/390® 和 z/OS DB2 系统，DB2 可以选择 3 种不同的联接类型，来处理联接中各表之间的 I/O：

- **嵌套循环联接** 要完成一个嵌套循环联接，会在外表中识别一个符合行，然后在内表中扫描匹配。符合行（qualifying row）是指对于表中的列能够使谓词匹配的行。内表扫描完成时，会在外表中找出另一个符合行。然后再次扫描内表来查找匹配，如此继续。反复扫描内表最好利用一个索引完成，否则内表会扫描很多次，导致运行时间很长。
- **合并扫描联接** 利用一个合并扫描联接，所联接的表需要按联接谓词排序。这意味着，必须按指定联接条件的列的顺序来访问各个表。这个顺序可以是一个排序或索引访问的结果。确保外表和内表都有正确的顺序之后，可以顺序地读取各个表，并匹配联接列。在合并扫描联接过程中，外表和内表都不会多次读取。排序会在选择了所有符合行之后进行。
- **混合联接** 混合联接会结合数据和指针来访问和合并所联接的表中的行。

一般来讲，如果只有很少的行符合起始组合表的条件，那么从执行成本角度来讲更适合采用嵌套循环联接。随着符合行的增多，合并联接会是更好的选择。

列表预取

列表预取（List Prefetch）过程中，DB2 从所处理的索引得到 RIDS，并在访问表空间文件之前按页面顺序对这些 RIDS 排序。这会确保顺序访问整个表空间文件，另外可能会完成多页 I/O 操作。优化工具选择复合联接进程、多索引访问或通过一个非聚簇索引完成处理时，会经常看到这种列表预取。

外联接

外联接（outer join）是一个联接操作的结果，包含两个表的匹配行以及被联接表的未匹配行。这些未匹配的行称为例外（exception），外联接会保留这些行（即使在被联接表中并未找到匹配）。从被联接表返回的列都作为 null 返回。

谓词

谓词（Predicate）是 SQL 查询的搜索和筛选条件中使用的语句。查询的 Where、Having 和 On 子句部分会出现谓词。例如，以下语句在 SQL 中就认为是谓词：

```
Where Lastname like 'A%'
Workdept in ('A00', 'B11')
```

重优化

重优化（Reoptimization）过程是指在运行时重新考虑之前已经在绑定或准备时优化过的 SQL 语句访问路径。在重优化期间，会在选择访问路径时考虑宿主变量、参数标记和特殊寄存器的具体值。

Stage 1 谓词

Stage 1 谓词在获取数据行时计算（即 sargable，这是一个比较早的 RDMS 术语，表示可搜索的参数）。Stage 1 谓词会使性能有很大提高，因为传递到第 2 阶段（Stage 2）的行会更少，从而使 I/O 最少，并减少 CPU 时间。

Stage 2 谓词

Stage 2 谓词在数据获取之后计算（非 sargable），与 Stage 1

谓词相比，成本会更昂贵。第 2 阶段（Stage 2）会处理排序问题，完成优化工具的工作，并负责应用和检查更复杂的谓词。字符连接、大多数标量函数、数据转换和算术表达式也可能作为 Stage 2 谓词。

Visual Explain

这是一个很棒的工具，所有 DB2 SQL 开发人员都可以利用这个工具详细了解 DB2 优化工具选择的访问路径。这也是显示 DB2 Explain 信息的另一种方式，可以图形化列出访问路径。所提供的信息对于解决程序或查询遇到的多种性能问题极有意义。使用 Plan_Table 和 DSN_Statemnt_Table，它能提供常规 DB2 Explain 的所有详细信息，还可以提供统计信息、谓词类型，等等。要提供所有其他细节，还需要很多其他 Explain 表。Visual Explain GUI 输出是 IBM Data Studio 工具的一部分。

IBM Data Studio Visual Explain 工具会提供以下信息：

- 处理表的顺序
- 是否已选择索引
- 发生的排序
- 发生排序的原因
- 表之间的联接类型
- Stage 1 和 Stage 2 谓词
- 匹配索引谓词与索引扫描谓词
- 每个谓词生成的筛选率
- 生成的总体筛选率
- 所有基数统计
- 所有频率值统计
- 相关表上的所有索引
- 表和列统计
- 所有物化的工作文件表