HZ BOOKS
华章科技

数据库
技术丛书

数据库查询优化器的艺术

原理解析与SQL性能优化

李海翔 著

The Art of Database Query Optimizer
Principle and SQL Performance Optimization

- Oracle公司MySQL全球开发团队资深专家撰写，拥有10余年数据库内核研究经验，数据库领域泰斗王珊教授亲自作序推荐
- PostgreSQL中国社区和中国用户会发起人，以及来自Oracle、新浪、网易、华为等企业的数位资深数据库专家联袂推荐
- 从原理角度深度解读和展示数据库查询优化器的技术细节和全貌；从源码实现角度全方位深入分析MySQL和PostgreSQL两大主流开源数据库查询优化器的实现原理；从工程实践的角度对比了两大数据库的查询优化器的功能异同和实现异同

机械工业出版社
China Machine Press

数据库技术丛书

数据库查询优化器的艺术：原理解析与 SQL 性能优化

李海翔 著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

数据库查询优化器的艺术：原理解析与 SQL 性能优化 / 李海翔著 . —北京：机械工业出版社，2014.1
（数据库技术丛书）

ISBN 978-7-111-44746-7

Ⅰ. 数… Ⅱ. 李… Ⅲ. 关系数据库系统—系统最优化 Ⅳ. TP311. 138

中国版本图书馆 CIP 数据核字（2013）第 269607 号

本书是数据库查询优化领域的里程碑之作，由 Oracle 公司 MySQL 全球开发团队、资深专家撰写，作者有 10 余年数据库内核和查询优化器研究经验。数据库领域泰斗王珊教授亲自作序推荐，PostgreSQL 中国社区和中国用户会发起人以及来自 Oracle、新浪、网易、华为等企业的数位资深数据库专家联袂推荐。从原理角度深度解读和展示数据库查询优化器的技术细节和全貌；从源码实现角度全方位深入分析 MySQL 和 PostgreSQL 两大主流开源数据库查询优化器的实现原理；从工程实践的角度对比了两大数据库的查询优化器的功能异同和实现异同。它是所有数据开发工程师、内核工程师、DBA 以及其他数据库相关工作人员值得反复研读的一本书。

全书一共 19 章，分为四个部分：第一篇（第 1 ～ 4 章）对数据库查询优化技术的范围、逻辑查询优化、物理查询优化，以及查询优化器与其他模块的关系做了非常细致、深入的讲解；第二篇（第 5 ～ 10 章）首先从源码角度对 PostgreSQL 查询优化器的架构、层次、设计思想、相关数据结构和实现原理进行了深入、系统的分析，然后从功能角度对 PostgreSQL 的逻辑查询优化、物理查询优化、查询优化器的关键算法，以及 PostgreSQL 查询优化器与其他模块的关系做了深入的讲解；第三篇（第 11 ～ 16 章）首先从源码角度对 MySQL 查询优化器的架构、层次、设计思想、相关数据结构和实现原理进行了深入、系统的分析，然后从功能角度对 MySQL 的逻辑查询优化、物理查询优化、查询优化器的关键算法，以及 MySQL 查询优化器与其他模块的关系做了深入的讲解；第四篇（第 17 ～ 19 章）对 PostgreSQL 与 MySQL 的逻辑查询优化技术、物理查询优化技术、设计思想和编码规范等各方面进行了深度的比较。

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：孙海亮
印刷
2014 年 1 月第 1 版第 1 次印刷
186mm × 240mm • 33.25 印张
标准书号：ISBN 978-7-111-44746-7
定 价：89.00 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 88379649 68995259 读者信箱：hzjsj@hzbook.com

推荐序一

随着数据库系统应用的广泛和深入，特别是近年来实际应用中数据库数据量不断增长，形成了所谓的海量数据，进一步的，现在人人都在讲的大数据时代已经到来。数据库系统的性能提升在传统数据库应用中同样受到现实的挑战。

对于一个选定的数据库管理系统（DBMS）产品，数据库系统性能的表现可以有很大差异，它和 DBMS 参数的选择、数据库模式的设计、应用系统的设计、软硬件环境的配置等多个因素密切相关。这就对数据库管理员（即 DBA）、应用系统分析和设计人员提出了要求，要求他们能够根据实际的应用环境、应用需求结合 DBMS 查询优化技术找到提升系统性能的方法或解决方案。

数据库查询优化技术一直是 DBMS 实现技术中的精华，也是难点和重点。数据库领域广大的研究工作者和实际开发者几十年来一直对查询优化技术孜孜不倦地探索着，更快、更好、更有效，这种精神使得查询优化技术不断推陈出新，形成了较为完善的优化技术体系，包括基于语法的、基于语义的、基于规则的、基于代价的等多角度、多方面的优化技术。

对于广大的工程第一线的技术人员、对于 DBMS 开发设计人员、对于数据管理领域的学生来说，**仅了解查询优化技术的原理是不够的，还必须从工程实践的角度入手，把 DBMS 查询优化基本原理和实现代码结合起来，才能真正掌握 DBMS 查询优化技术，才能运用掌握的技术解决实际系统中性能提升的问题。**

本书以 PostgreSQL 和 MySQL 两大开源 DBMS 查询优化器的实现为蓝本，介绍和分析了它们的实现原理、实现过程，帮助读者深入理解和掌握数据库查询优化的核心技术，在 DBMS 调优过程中不仅掌握怎么调优而且掌握为什么可以调优。

本书作者努力把数据库查询优化原理、查询优化引擎源码实现和 SQL 优化工程实践相结合，这是本书的特色，期望本书能成为读者的良师益友。

王珊

中国人民大学信息学院教授 / 博士生导师

教育部数据工程与知识工程重点实验室学术委员会副主任

推荐序二

PostgreSQL、MySQL、MongoDB、HBase 等都是非常优秀的开源数据库，大数据的到来，使得数据管理正走向一个新的复兴时期，PostgreSQL 在这次复兴中是一支重要力量。我一直在国内从事 PostgreSQL 社区的培育和推广工作，在这个过程中，我的一个比较深的体会是，目前国内非常缺乏与之相关的书籍和资料，这给 PostgreSQL 的推广和普及工作带来了非常大的阻力，所以社区非常希望有更多的出版社推动 PostgreSQL 相关书籍的出版。机械工业出版社华章公司非常具有前瞻性，对 PostgreSQL 非常关注和支持，他们策划的《 PostgreSQL 数据库内核分析》一书在业界引起了很大的反响，我身边的朋友人手一本，为整个社区带来了非常大的帮助。海翔的这本书，是他们努力下的又一个丰硕成果。

数据库除了性能，还是性能，性能无疑是数据库工作者最为关注的重要方面之一。DBA 需要深入理解数据库优化的原则，只有这样才能有效调整和驾驭数据库的性能；数据库内核开发人员更需要理解其中的具体原理和实现，这样才能开发出高性能的产品。这本书独辟蹊径，从内核中选出查询优化这个最为复杂和引人入胜的部分为主题，同时也涵盖了两大开源主流数据库 PostgreSQL 和 MySQL 的内部实现，辅以理论和具体实现相结合的手法，使得读者可以在充分理解原理的基础上，深入掌握在具体工作中用到的技术，这不能不说是本书的一个精巧的地方。不仅是 PostgreSQL 的相关工作者，MySQL、Oracle 的数据库工作者，也可以通过本书，从原理和代码层面真正理解数据库性能优化，从而使自己成为一名性能优化“大师”。本书会给大家真正深入理解查询优化带来莫大的帮助。

李元佳

PostgreSQL 中国用户会发起人

前　言

为什么写这本书

数据库引擎是一项包罗万象的技术大全，纷繁复杂，不能轻易穷尽。

数据库查询优化器是数据库引擎的重中之重。掌握了查询优化器，就等于掌握了数据库的精髓，得查询优化器者则得数据库天下。

在数据库查询优化技术领域，尚没有一本将理论与实践结合的书籍，**能帮助查询优化技术的爱好者掌握其全貌、掌握其细节、掌握其精髓、掌握其奥妙**。

如果能有一本书可以包括数据库查询优化技术的完整面貌，相信一定会使探索者少些崎岖摸索，这自然是一件好事。于是笔者心怀忐忑，虽自知力不能及，但还是决定奋笔一试，如工匠砌砖、蝼蚁啃土，一点一滴，将自己多年总结的经验汇集于此，终于小成。

本书的主要内容

本书主要讨论以下问题：

- “查询优化”究竟包括什么样的优化技术？为什么能做那些优化？
- “查询优化器”是怎么实现的？“查询优化器”是怎么做优化工作的？
- PostgreSQL 和 MySQL 的查询优化器各自实现了什么样的查询优化技术？
- PostgreSQL 和 MySQL 的查询优化功能有什么异同？

第一个问题在第一篇中进行了讨论。第二、第三个问题在第二、第三篇中通过对 PostgreSQL 和 MySQL 查询优化器的源码分析进行了回答。第四个问题在第四篇中通过对 PostgreSQL 和 MySQL 查询优化的功能对比，对两者的差异进行了回答。

从全书来看，本书涉及查询优化器的原理、代码实现、工程实践 3 个部分。

原理部分，概括总结了查询优化器的两大优化策略，一是逻辑优化，二是物理优化。在每一个优化策略中，都结合原理、运用示例，努力展示了查询优化器的全貌。这部分内容将解答：“查询

优化”究竟包括什么样的优化技术？为什么能做那些优化？

代码实现部分，通过深入数据库内核的源码，对查询优化器的实现进行深度的、全方位的探索，力求为程序员和数据库爱好者展示数据库引擎查询优化器的真实面目。行文选取了开源数据库 PostgreSQL 和 MySQL 的查询优化器，抽丝剥茧，进行了详细、全面的分析。这部分内容将解答：“查询优化器”是怎么实现的？“查询优化器”是怎么做优化工作的？

工程实践部分，通过原理分析、代码分析和示例，进行了以下 3 个对比：

- 对比 PostgreSQL 和 MySQL 的查询优化器的实现异同。
- 对比 PostgreSQL 和 MySQL 的查询优化器的功能异同。
- 对比查询优化器和原理之间的异同。

工程实践部分还通过大量详细的示例比较，总结了 PostgreSQL、MySQL 支持的优化点和不支持的优化点，为 DBA 调优提供了许多新颖的、好的思路。揭秘内核，帮助读者掌握其实现方式，可使读者在面对 SQL 调优时不再是雾里看花，而是心中有数。系统地掌握这些，可以超越依靠日积月累的学习方式，能够快速进阶。

工程实践部分通过 3 个对比，完成了以下 3 个结合：

- 通过代码分析，首次将关系代数、代价估算等查询优化原理与代码实现、工程实践结合。
- 通过对 PostgreSQL 和 MySQL 的对比，首次揭秘了两大开源数据库的查询优化器的异同，使得数据库爱好者能够以本书为基础，更好地结合查询优化技术，进而掌握各种类型的数据库 SQL 调优。
- 通过原理、代码、分析、示例，完成了将数据库爱好者对查询优化器强烈渴望掌握的愿望与轻松掌握的现实的结合，使得查询优化器不再那么神秘莫测。

本书的主要特色

本书的主要特色如下：

- **全面揭秘数据库查询优化技术**。把数据库的各种查询优化技术以工程化的视角进行剖析，集原理、源码分析、示例实践为一体。
- **重点分析了 PostgreSQL 和 MySQL 的查询优化器的实现**。以两大开源数据库的查询优化器的实现为蓝本，分析了它们的实现过程、实现原理，帮助读者深入理解掌握数据库的核心技术，使他们在 SQL 调优过程中不仅掌握怎么调优而且知道为什么可以调优。
- **详细对比了 PostgreSQL 和 MySQL 的查询优化技术的支持能力**。通过大量示例，分析对比了两大开源数据库的查询优化功能的强弱，帮助读者掌握两个数据库查询优化功能相同和不同之处，为应用系统选型提供衡量查询优化功能的依据。

本书面向的主要读者

如果您是一名数据库内核的工作者，本书对数据库查询优化器的内核代码分析，一定会帮您全面掌握查询优化这一技术。再结合理论部分，将使您理解得更加深刻。

本书不仅讲述了 PostgreSQL 查询优化器的实现，还讲述了 MySQL 查询优化器的实现，这对于从事这两个数据库内核开发的人员来说尤其有帮助。如果您能把结合本书和源码（务必要读代码）两者相互印证进行掌握，将像弄潮儿涛头立，查询优化技术尽在您的掌中。

本书从理论角度出发构造了大量示例，通过源码和功能的对比分析，更是进一步揭示了这两个数据库查询优化器的功能异同。对于从事查询优化器性能提升的工作人员帮助更大——因为原理部分讲了为什么，代码分析部分讲了怎么做，示例与对比部分讲了有什么不同，可做的事情仅剩下“没有实现的优化怎么做”了。路已铺垫，启示已明，发挥就靠您自己了。

如果您是一名数据库实践者、DBA，本书一定会帮到您。查询优化器能做什么样的优化？为什么查询优化器能做某种类型的优化？对于这两个问题，原理部分能帮您解惑。如果您是一名 DBA，您将更多地从代码分析、工程实践部分获益。源码级的内幕揭秘，无疑能帮您更好地做好 SQL 语句的日常优化工作；代码、功能、实践的比较，无疑能帮助您从初学到理解，进而过渡到得心应手地应用。

生活不是一成不变的，工作中您也许会在 PostgreSQL 与 MySQL 间切换，掌握原理又能帮您以不变应万变。本书所涉及的查询优化器的知识，适用于 Oracle 的 DBA、DB2 的 DBA、SQL Server 的 DBA、SYSBASE 的 DBA。也许从此以后，您会成为一名万能的 DBA！

如果您是一名数据库研究者、开发者、教学者、学生、爱好者，那么本书对您来说是一本不可多得的参考资料。查询优化器作为数据库中最重要、最难学、最精彩的部分，探索起来道路漫长，艰苦而无趣，也许本书能有幸成为您的同行伙伴，一路走来，不离不弃。探索数据库内核，犹如面对一座大山，陡峭艰险，无路可寻。看似非常艰难的事情，当您一书在手，仿佛为您提供了一柄手杖，步行于山涧沟壑时，且健且速。

特别是对于学生，如果能凭着这本薄书摸索进入数据库内核的世界，当是笔者最大的心愿。学生是未来的栋梁，是将技术发扬光大的希望。

多年来，作者一直翘首以盼，希望能有这么一类书，与己成为好友。现在，有了这么一本薄书，唯愿读者开卷有益。

如何阅读本书

从模块的角度看，本书可分为 4 部分：原理、PostgreSQL 实现、MySQL 实现、三者对比总结。

所以建议读者以模块为单位分别阅读。如果是读 PostgreSQL 实现、MySQL 实现的章节，则应结合代码，前后联系沟通为好，不要掌握局部忽视整体。

从内容的分布上看，原理是综述部分，既讲述了原理对查询优化的指导，也对 PostgreSQL 实现、MySQL 实现进行了总结，所以要从实践个例上升到理论的高度才能看清全貌。把原理和 PostgreSQL 实现结合起来，反复互为印证，会对掌握 PostgreSQL 有帮助；把原理和 MySQL 实现结合起来，反复互为印证，会对掌握 MySQL 有帮助；把 4 部分完全结合起来一起掌握，不仅可以帮助 DBA 和爱好者理解 PostgreSQL、MySQL，还可以帮助他们理解其他数据库。因为本书中的两套源码的比较加上原理的解析，有助于帮助理解其他数据库查询优化器的原理；示例则有助于在其他数据库上执行以检查数据库对查询优化的支持程度。

另外，为便于读者更好地掌握数据库查询优化技术及本书内容，本书的附录从方法学的角度对阅读本书提供了 4 个方面的建议。

为什么特别推荐学生阅读本书

对于爱好数据库技术的学生来说，选择数据处理技术，是一条明智之路。数据越来越多，数据源越来越多，大数据的时代不仅已经来临了，而且会一直发展、不断发展。数据库技术是处理大数据的核心技术，数据库技术会随着时代的需求，一直前进。而查询优化，在数据库领域，不仅有着重要的理论地位，而且在实践中也异常重要且用途广泛，SQL 调优占据着数据库性能调优的半壁江山。

现今，就业竞争异常激烈，一份好的工作往往会有数以千计的人同时去争取。这对于诸多刚刚毕业、没有什么工作经验的学生来说，无疑是非常严峻的考验，尤其对于想找到理想工作的学生更是难上加难。因为现在用人单位提供的工作岗位对应聘者的素质和技能的要求越来越高，这无疑是刚刚毕业的学生最大的短板。

对于用人单位来说，想招到一名合格的员工也不容易，招到一名技能纯熟、素质高的员工更难。

在这样的大背景下，如果学生在求学期间能够打下良好的专业基础，找工作时定会增加很多机会；如果学生能够在自己的专业领域深入探索，对数据结构、操作系统、数据库等重要课程熟练掌握，使自己具备良好的专业素质，必然能使自己在找工作时具备更强的竞争力；如果学生能在某些课程上把理论和实践结合起来，把理论和源码结合起来，熟练掌握重量级软件的内核代码，找到心仪的工作将更为容易。用人单位喜欢有理论、工作初始就能上手做事的人才。而学生们想成为这样的人才，就需要在学校里打下坚实的基础。

本书在数据库领域就能起到这样的作用。引领学生把数据库理论和数据库实践结合起来，把数据库理论和数据库引擎源码结合起来，有效帮助读者梳理数据库技术中最重要的部分——查询优化技术的全貌，这必定对掌握这部分技术有很大的帮助。

所以，笔者特别推荐学生阅读本书。建议在本科高年级阶段或研究生阶段，**大家能与本书为伴，边读源码（PostgreSQL 和 MySQL 的源码）边读本书，互为印证**。笔者相信通过短短数月的努力，大家得到的将是足够好的技能、足够高的起点、足够强的竞争力！

学生是未来、是希望，学生强，社会亦将受益。学生若能通过本书使自己变强，将是笔者最大之幸。

勘误及支持

由于笔者的水平有限，书中难免会有笔误、差错、格式错误或遗漏等问题，希望广大读者能把发现的错误告诉笔者，我将不胜感谢。本书的进步和完善，有您的帮助和爱护，定能再上一层楼。您可以发送电子邮件到：database_XX@163.com。由于时间有限，也许笔者不能一一答复所有的电子邮件，但是会定期整理并汇总信息到笔者的博客（http://bolg.163.com/li_hx）上。

书中分析、探索的数据库源码的下载地址如下：

PostgreSQL，V9.2.3，源自 http://www.postgresql.org/ftp/source/v9.2.3/。

MySQL，V5.6.10，源自 http://dev.mysql.com/downloads/mysql/。

致谢

在我的生命中，家人是最重要的。伏案疾书不舍昼夜，心中一直牵挂的是父母、姐妹、妻子和各地的亲人们。他们是我写作的动力源泉，他们是我得以坚持完稿的坚强后盾。感谢父母给予了我生命并育我成人，感谢妻子朝夕相伴给予我鼓励和慰藉，感谢我远方的亲人们让我心意暖暖。

感谢中国人民大学信息学院王珊教授多年的教导，并高屋建瓴地为本书提出修订意见，她严谨的工作作风影响了本书的风格。成书之际，王老师乐为作序，我心怀感激，念念在心。

感谢为开源社区无私奉献的人们，没有他们就没有本书。本书的内容源于对开源数据库内核代码的分析，故笔者认为本书也应该回归社区为众人服务。坚持写下自己的所得，与他人共享，是一件快事，也是一件幸事。

感谢编辑杨福川先生和孙海亮先生为本书付出的努力和耗费的心血，书名源于杨先生，书稿样式由孙先生设定。写了一本书，交了两个好朋友。谢谢他们。

感谢每一位读者，我们一起进步，你们将是本书继续完善的新动力。

我深知本书尚不能达到书名标识的高度，而今迈步从头越。

目　录

推荐序一
推荐序二
前　言

第一篇　查询优化技术

第 1 章　数据管理系统的查询优化 ····2
1.1　数据库调优 ····3
1.2　查询优化技术 ····5
1.2.1　查询重用 ····5
1.2.2　查询重写规则 ····6
1.2.3　查询算法优化 ····6
1.2.4　并行查询优化 ····8
1.2.5　分布式查询优化 ····9
1.2.6　其他优化 ····9
1.3　本章小结 ····9

第 2 章　逻辑查询优化 ····10
2.1　查询优化技术的理论基础 ····10
2.1.1　关系代数 ····11
2.1.2　关系代数等价变换规则对优化的指导意义 ····13
2.2　查询重写规则 ····17
2.2.1　子查询的优化 ····18
2.2.2　视图重写 ····28
2.2.3　等价谓词重写 ····29
2.2.4　条件化简 ····32
2.2.5　外连接消除 ····33
2.2.6　嵌套连接消除 ····37
2.2.7　连接消除 ····38
2.2.8　语义优化 ····40
2.2.9　针对非 SPJ 的优化 ····41
2.3　启发式规则在逻辑优化阶段的应用 ····42
2.4　本章小结 ····43

第 3 章　物理查询优化 ····44
3.1　查询代价估算 ····44
3.1.1　代价模型 ····44
3.1.2　选择率计算的常用方法 ····45
3.2　单表扫描算法 ····45
3.2.1　常用的单表扫描算法 ····45
3.2.2　单表扫描代价计算 ····47
3.3　索引 ····47
3.3.1　如何利用索引 ····47
3.3.2　索引列的位置对使用索引的影响 ····50
3.3.3　联合索引对索引使用的影响 ····56
3.3.4　多个索引对索引使用的影响 ····57

3.4 两表连接算法 ····· 59
3.4.1 基本的两表连接算法 ····· 59
3.4.2 进一步认识两表连接算法 ·· 61
3.4.3 连接操作代价计算 ····· 61
3.5 多表连接算法 ····· 62
3.5.1 多表连接顺序 ····· 62
3.5.2 常用的多表连接算法 ····· 63
3.5.3 多表连接算法的比较 ····· 68
3.6 本章小结 ····· 68

第 4 章 查询优化器与其他模块的关系 ····· 70

4.1 查询优化器整体介绍 ····· 70
4.2 查询优化器与其他模块的关系 ··· 73
4.3 本章小结 ····· 74

第二篇 PostgreSQL 查询优化器原理解析

第 5 章 PostgreSQL 查询优化器概述 ····· 76

5.1 PostgreSQL 查询执行过程 ····· 76
5.2 PostgreSQL 查询优化器的架构和设计思想 ····· 78
5.2.1 PostgreSQL 查询优化器架构 ····· 79
5.2.2 PostgreSQL 查询优化器的层次 ····· 81
5.2.3 PostgreSQL 查询优化器设计思想 ····· 81
5.3 主要概念 ····· 81
5.4 代码层次结构 ····· 85
5.5 本章小结 ····· 86

第 6 章 PostgreSQL 查询优化器相关数据结构 ····· 88

6.1 主要数据结构 ····· 88
6.1.1 基本数据结构 ····· 88
6.1.2 查询树 ····· 91
6.1.3 各种对象的结构 ····· 95
6.1.4 连接操作相关的结构 ····· 99
6.1.5 查询执行计划相关的结构 ····· 104
6.2 各个结构之间的关系 ····· 108
6.3 各个阶段间和主要结构体间的关系 ····· 109
6.4 本章小结 ····· 110

第 7 章 PostgreSQL 查询优化器实现原理解析 ····· 111

7.1 查询优化整体流程 ····· 111
7.2 查询优化器实现原理解析 ····· 115
7.2.1 planner——主入口函数 ··· 115
7.2.2 standard_planner——标准的查询优化器函数 ····· 116
7.2.3 subquery_planner——生成（子）查询执行计划函数 ··· 117
7.2.4 grouping_planner——生成查询执行计划并对非 SPJ 优化 ····· 139
7.2.5 build_minmax_path——聚集函数 MIN/MAX 的优化函数 ····· 141

7.2.6 query_planner——生成最优的查询路径函数 …… 142
7.2.7 make_one_rel——构造多表连接路径并选出最优路径函数 …… 148
7.2.8 make_rel_from_joinlist——生成多表连接路径函数 … 153
7.2.9 optimize_minmax_aggregates——聚集操作MIN/MAX 优化函数 …… 163
7.2.10 create_plan——创建查询执行计划函数 …… 164
7.2.11 非 SPJ 处理——grouping_planner 的各个子模块 …· 166
7.2.12 其他重要的函数与操作 ·· 170
7.3 代价估算实现原理解析 …… 174
7.3.1 查询代价估算 …… 174
7.3.2 单表扫描方式的代价估算 …… 174
7.3.3 两表连接的代价估算 …… 178
7.3.4 其他代价估算函数 …… 184
7.3.5 选择率的计算 …… 185
7.4 从目录结构和文件功能角度看查询优化器 …… 186
7.4.1 查询优化子模块与主要文件的关系 …… 187
7.4.2 查询优化器代码结构 …… 187
7.5 本章小结 …… 190

第 8 章 从功能的角度看 PostgreSQL 查询优化 …· 192

8.1 优化器之逻辑查询优化 …… 192
8.1.1 视图重写 …… 193
8.1.2 子查询优化 …… 197
8.1.3 等价谓词重写 …… 209
8.1.4 条件化简 …… 209
8.1.5 外连接消除 …… 210
8.1.6 嵌套连接消除 …… 217
8.1.7 连接的消除 …… 218
8.1.8 语义优化 …… 221
8.1.9 选择操作下推 …… 226
8.1.10 非 SPJ 优化 …… 226
8.2 优化器之物理查询优化 …… 229
8.2.1 PostgreSQL 的物理优化主要完成的工作 …… 229
8.2.2 启发式规则在物理查询优化阶段的使用 …… 230
8.2.3 两表连接 …… 230
8.2.4 代价估算 …… 230
8.2.5 PostgreSQL 的索引与查询优化 …… 231
8.3 其他 …… 237
8.3.1 grouping_planner 函数主干再分析 …… 238
8.3.2 用户指定的连接语义与 PostgreSQL 实现两表连接的函数及算法的关系 …… 240
8.3.3 集合操作优化 …… 242
8.4 本章小结 …… 245

第 9 章 PostgreSQL 查询优化的关键算法 …… 246

9.1 动态规划算法 …… 246
9.1.1 动态规划算法的处理流程 …… 247

9.1.2 紧密树处理流程 ……248
9.2 遗传算法 ……248
9.2.1 PostgreSQL 遗传算法的处理流程 ……248
9.2.2 主要的数据结构 ……250
9.2.3 主要的函数和变量 ……251
9.2.4 应用遗传算法实现表连接的语义 ……253
9.2.5 应用遗传算法计算适应度 ……254
9.2.6 进一步理解 PostgreSQL 的遗传算法 ……255
9.3 动态规划算法与遗传算法对比 …256
9.4 本章小结 ……257

第 10 章 PostgreSQL 查询优化器与其他部分的关系 …259

10.1 查询优化器与语法分析器 ……259
10.2 查询优化器与执行器 ……260
10.3 查询优化器与缓冲区管理模块 ……261
10.4 查询优化器与对象访问模块 …262
10.5 查询优化器与统计模块 ……262
10.6 查询优化器与索引模块 ……263
10.7 本章小结 ……263

第三篇 MySQL 查询优化器原理解析

第 11 章 MySQL 查询优化器概述 ……266

11.1 MySQL 查询执行过程 ……266
11.2 MySQL 查询优化器的架构和设计思想 ……267
11.2.1 MySQL 查询优化器架构 ……268
11.2.2 MySQL 查询优化器的层次 ……269
11.2.3 MySQL 查询优化器设计思想 ……269
11.3 主要概念 ……270
11.3.1 常量表 ……270
11.3.2 表数据的访问方式 ……270
11.4 代码层次结构 ……272
11.5 本章小结 ……274

第 12 章 MySQL 查询优化器相关数据结构 ……275

12.1 主要的类和数据结构 ……275
12.1.1 查询树 ……275
12.1.2 基本对象 ……276
12.1.3 连接对象与执行计划 ……278
12.1.4 代价估算类 ……281
12.2 各个阶段主要结构体间的关系 ……282
12.3 本章小结 ……282

第 13 章 MySQL 查询优化器的原理解析 ……283

13.1 查询优化器整体流程 ……283
13.2 优化器的代码详解 ……285
13.2.1 JOIN.prepare——优化前的准备工作 ……286

13.2.2 JOIN.optimize——优化器主入口方法 ……299
13.2.3 make_join_statistics——计算最优的查询优化执行计划 ……315
13.2.4 choose_table_order——求解多表连接最优连接路径 ……324
13.2.5 make_join_statistics函数的其他子函数 ……339
13.2.6 make_join_select——对条件求值、下推连接条件到表中 ……348
13.2.7 test_if_skip_sort_order——排序操作的优化 ……350
13.2.8 make_join_readinfo——为连接的每个表构造信息 ……351
13.2.9 JOIN.exec——执行查询执行计划的函数 ……353
13.3 代价估算 ……354
13.3.1 查询代价估算模型 ……354
13.3.2 查询代价估算过程 ……355
13.3.3 其他的代价估算 ……358
13.3.4 对存储引擎的调用接口 ……362
13.3.5 统计信息 ……364
13.4 本章小结 ……365

第 14 章 从功能的角度看MySQL 查询优化 ……366

14.1 优化器之逻辑查询优化 ……366
14.1.1 视图重写 ……367
14.1.2 子查询优化 ……371
14.1.3 等价谓词重写 ……387
14.1.4 条件化简 ……388
14.1.5 外连接消除 ……389
14.1.6 嵌套连接消除 ……396
14.1.7 连接的消除 ……398
14.1.8 语义优化 ……400
14.1.9 非 SPJ 优化 ……406
14.2 优化器之物理查询优化 ……412
14.2.1 MySQL 的物理优化主要完成的工作 ……412
14.2.2 启发式规则在物理查询优化阶段的使用 ……413
14.2.3 MySQL 的索引与查询优化 ……413
14.2.4 用户指定的连接语义与 MySQL 实现两表连接的算法 ……417
14.3 本章小结 ……418

第 15 章 MySQL 查询优化的关键算法 ……419

15.1 深入理解 MySQL 的多表连接算法 ……419
15.2 本章小结 ……424

第 16 章 MySQL 查询优化器与其他部分的关系 ……425

16.1 查询优化器与语法分析器 ……425
16.2 查询优化器与执行器 ……426
16.3 查询优化器与缓冲区管理模块 ……426

16.4 查询优化器与索引模块 ……426
16.5 本章小结 ……427

第四篇 PostgreSQL 查询优化器 VS MySQL 查询优化器

第 17 章 PostgreSQL 和 MySQL 的逻辑查询优化技术 ……430

17.1 查询重写 ……430
17.1.1 子查询优化 ……430
17.1.2 视图重写 ……443
17.1.3 等价谓词重写 ……446
17.1.4 条件化简 ……447
17.1.5 外连接消除 ……448
17.1.6 嵌套连接消除 ……449
17.1.7 连接消除 ……451
17.1.8 语义优化 ……452
17.2 非 SPJ 的优化 ……452
17.3 本章小结 ……456

第 18 章 PostgreSQL 和 MySQL 的物理查询优化技术 ……457

18.1 查询代价估算模型比较 ……457
18.2 单表扫描算法 ……458
18.3 索引 ……458
18.4 两表连接算法 ……466
18.5 多表连接算法 ……467
18.6 本章小结 ……467

第 19 章 PostgreSQL 和 MySQL 的其他异同 ……468

19.1 启发式规则的使用比较 ……468
19.2 综合比较 ……469
19.2.1 基本概念的比较 ……469
19.2.2 数据结构的比较 ……469
19.2.3 设计思想的比较 ……469
19.2.4 编码规范的比较 ……470
19.3 本章小结 ……471

附录 A 如何掌握数据库内核 ……472

附录 B 如何阅读本书 ……496

附录 C 如何阅读查询执行计划 ……498

附录 D 如何跟踪查询执行计划 ……508

第一篇

查询优化技术

本篇介绍了数据库的查询优化技术，从数据库的理论出发界定查询优化技术的范围，讨论了包括逻辑查询优化、物理查询优化两个方面的查询优化技术。全篇立足于数据库的基本理论——关系代数，在第 1 章首先界定了本书讨论的查询优化的技术范围；在第 2 章运用关系代数理论和关系法则，对逻辑查询优化进行全面而深入的探讨，对各种逻辑查询优化技术进行全面介绍，指出关系代数对于查询优化技术的指导意义，通过示例巩固对查询优化技术的理解和认识；在第 3 章，通过对代价估算模型、索引和表扫描、表连接算法的介绍，对各种物理查询优化技术进行全面描绘，构造了清晰、完整的物理查询优化技术图谱；在第 4 章，对实现查询优化技术的数据库查询优化器的相关模块进行了介绍，意在使读者了解查询优化技术的相关上下文内容。

第 1 章
数据管理系统的查询优化

数据库管理系统（DataBase Management System，DBMS，简称数据库）是位于用户与操作系统之间的一层数据管理软件，主要功能包括：数据定义、数据操纵、数据库的运行管理、数据库的建立和维护[⊖]等。

数据操纵是数据库管理系统中一种最基本的操作，这种操作包括查询、插入、删除和修改等，其中，查询操作称为查询处理。

查询的执行，就是查询处理的过程，即数据库按用户指定的 SQL 语句中的语义，执行语义所限定的操作。但 SQL 语句的执行效率对数据库的效率影响较大。为了提高查询语句的执行效率，对查询语句进行优化是必不可少的。

对查询语句进行优化的技术就是查询优化技术，运用查询技术实现数据操纵功能的过程是确定给定查询的高效执行计划的过程。所谓执行计划就是查询树，它由一系列内部的操作符组成，这些操作符按一定的运算关系构成查询的一个执行方案。查询优化的追求目标，就是在数据库查询优化引擎生成一个执行策略的过程中，尽量使查询的总开销（总开销通常包括 IO、CPU、网络传输等）达到最小。

数据库查询优化技术主要包括查询重用技术、查询重写规则、查询算法优化技术、并行查询优化技术、分布式查询优化技术及其他方面（如框架结构）的优化技术，这 6 项技术构成了一个“**广义的数据库查询优化**”的概念。

本书重点探讨“查询重写规则”和“查询算法优化”，这是多数书籍在提及“数据库查询优化”时所限定的范围，这两项技术在本书中称为“**狭义的数据库查询优化**”。从优化的内容角度看，查询优化又分为**代数优化**和**非代数优化**，或称为**逻辑优化**和**物理优化**。逻辑优化主要依据关系代数的等价变换做一些逻辑变换，物理优化主要根据数据读取、表连接方式、表连接顺序、排序等技术对查询进行优化。“查询重写规则”属于逻辑优化方式，运用了关系代数和启发式规则；“查询算法优化”属于物理优化方式，运用了基于代价估算的多表连接算法求解最小花费的技术。

⊖《数据库系统概论》，王珊，萨师暄。

查询优化技术中“查询重写规则”的理论基础是关系代数，但本书的重点不是讨论关系代数理论，而是着眼于关系代数和查询优化相结合的部分，指出理论与实践的对应关系，分析理论是如何指导数据库引擎执行查询优化的，进而使读者明白数据库查询优化器的实现原理，掌握 SQL 语句的优化方法。这些主要包括以下内容：

- **查询优化引擎能做什么样的查询优化操作**。这第 1 章的重点内容，将全面指明查询优化技术的范围，以期建立查询优化技术的全局概念。
- **查询优化引擎为什么能进行查询优化**。这是第 2 章和第 3 章的重点内容，将从理论的角度出发进行介绍。其中第 2 章介绍逻辑优化包括的具体技术、为什么可做逻辑优化，以及怎么做逻辑优化；第 3 章介绍物理优化的具体技术、为什么可做物理优化，以及怎么做物理优化。

对于并行查询优化、分布式并行查询优化、其他优化等只做简单介绍，之所以如此，是为了保持全文的完整性，提醒读者从概念上要明确“数据库的查询优化技术”的范围。

本章先从数据库层面的优化进行概述，这是全局级别的优化，着眼点在整个数据库管理系统，借以区别本书重点——查询优化技术。接着，详细介绍查询优化技术，这是局部的、SQL 语句级别的优化，也是本书着重探讨的内容。

1.1　数据库调优

数据库调优可以使数据库应用运行得更快，其目标是使数据库有更高的吞吐量和更短的响应时间。被调优的对象是整个数据库管理系统总体。

在数据库层面进行调优，有很多的资源、数据库配置参数需要考虑。数据库调优的方式通常有如下几种：

- **人工调优**。主要依赖于人，效率低下；要求操作者完全理解常识所依赖的原理，还需要对应用、数据库管理系统、操作系统以及硬件有广泛而深刻的理解。
- **基于案例的调优**。总结典型应用案例情况中数据库参数的推荐配置值、数据逻辑层设计等情况，从而为用户的调优工作提供一定的参考和借鉴。但这种方式忽略了系统的动态性和不同系统间存在的差异。
- **自调优**。为数据库系统建立一个模型，根据“影响数据库系统性能效率的因素”，数据库系统自动进行参数的配置。一些商业数据库实现了部分自调优技术，典型的如 Oracle 提供了如下一些技术或工具。
  - Redo Logfile Sizing Advisor：为避免因频繁出现的检查点而导致过多的磁盘 IO，系统可推荐重做日志文件的最佳大小。
  - Automatic Checkpoint Tuning：为取得良好的恢复速度，同时降低对正常吞吐量的影响，系统可以自动优检查点。
  - Automatic Shared Memory Tuning：为高效地利用可用内存并提高性能，系统可自动地配置与 System Global Area（SGA）内存相关的参数（如缓冲区缓存、共享池等）。

- Transaction Rollback and Recovery Monitoring：为掌握事务状况，系统可估计回滚一个事务要花多少时间，监视被恢复的事务的进度，并估计事务恢复的平均速度。
- SQL Tuning Advisor：为提高 SQL 语句的执行效率，系统可自动调优高成本的 SQL 语句，给出建立索引的建议、SQL 重写的建议等。
- SQL Analyzer：对 SQL 语句的不同查询执行计划进行性能分析和比较。
- SQL Access Advisor：对物理设计给出改进建议。
- SQL Plan Management：使 SQL 语句能够根据环境的变化选择稳定、高效的查询执行计划。
- Segment Advisor：根据对象内的空间碎片化程度，给出是否应该对对象执行新的在线收缩操作的建议；提供关于段的历史增长趋势的报告，为容量规划提供有效的信息。
- Undo Advisor：帮助管理员在 flashback 和非 flashback 特性中调整撤销表空间大小时做出正确的判断；为管理员恰当地设置 UNDO_RETENTION 提供建议，避免快照过于陈旧。

通常，数据库调优主要分为 5 阶段，如表 1-1 所示。涉及的技术主要有以下几种：

- **应用情况的估算**。应用的使用方式（把业务逻辑转换为数据库的读写分布逻辑，以读多写少或读写均衡等来区分 OLAP 和 OLTP；应用对数据库的并发情况、并发是否可以池化等）、数据量、对数据库的压力、峰值压力等做一个预估。
- **系统选型策略**。确定什么样的数据库可以适用应用需求，并确定使用开源的数据库还是商业的数据库，使用集群还是单机的系统，同时对操作系统、中间件、硬件、网络等进行选型。
- **数据模型的设计**。主要根据业务逻辑，从几个角度考虑表的逻辑结构，内容如下。
  - **E-R 模型设计**：遵循 E-R 模型设计原理。偶尔的、适当程度的非规范化可以改善系统查询性能。
  - **数据逻辑分布策略**：目的是减少数据请求中不必要的数据量，只返回用户需要的数据。可用的技术如分区、用 E-R 模型分表等（如互联网企业典型的用法，根据业务的不同，进行分库、分表等操作）。
  - **数据物理存储策略**：目的是减少 IO 操作，如启用压缩技术、把索引和表数据的存储分开，不同的表数据分布在不同的表空间上，不同的表空间分布在不同的物理存储上（尤其是读写量大的表空间分布在不同的物理存储上），日志、索引和数据分布在不同的物理存储上等。
  - **索引**：在查询频繁的对象上建立恰当的索引，使索引的正效应大于负效应（索引的维护存在消耗）。
- **SQL 设计**。编写正确的、查询效率高的 SQL 语句，依据的主要是“查询重写规则”。编写语句的过程中要注意，要有意识地保障 SQL 能利用到索引。
- **数据库功能的启用**。数据库为提高性能提供了一些功能，可合理使用，具体如下。

- ❍ **查询重用**：根据实际情况进行配置，可缓存查询执行计划、查询结果等。
- ❍ **数据库参数的设置**：可设置合适的参数，如数据缓冲区等。

❑ **模型系统预运行**。在备用系统上模拟实际运行环境，加大压力进行系统测试，提前发现问题。

❑ **系统监控与分析**。在工业环境下，加强对系统的运行监控和日常的分析工作，具体如下。

- ❍ **应用系统表现**：收集用户对应用系统的使用意见、系统存在问题等，因为这些可能是用户在第一时间发现的。
- ❍ **OS 环境监控**：实时监控 CPU、内存、IO 等，并对比实时情况与历史正常情况。
- ❍ **数据库内部状况监控**：一些数据库提供系统表、视图、工具等手段，向用户提供数据库运行过程中内部状况的信息，如锁的情况，这些都需要实时监控，并对比实时情况与历史正常情况。
- ❍ **日志分析**：在数据库的日志、操作系统的日志中找出异常事件，定位问题。

表 1-1　数据库调优 5 个阶段

阶　段　号	阶段名称	使用的技术
第一阶段	需求分析期	应用情况的估算、系统选型策略
第二阶段	项目设计期	数据模型的设计
第三阶段	开发期	SQL 设计、数据库功能的启用
第四阶段	测试与试运行	数据库功能的启用、模型系统预运行、系统监控与分析
第五阶段	上线和维护	系统监控与分析

1.2　查询优化技术

查询优化技术是 SQL 层面的优化，属于局部优化，有别于“数据库调优”式的全局优化。上面我们说过，查询优化技术主要包括查询重用技术、查询重写规则技术、查询算法优化技术、并行查询的优化技术、分布式查询优化技术、其他优化技术 6 个方面的技术。下面我们就从这 6 个方面介绍查询优化技术。

1.2.1　查询重用

查询重用是指尽可能利用先前的执行结果，以达到节约查询计算全过程的时间并减少资源消耗的目的。

目前查询重用技术主要集中在两个方面：

❑ **查询结果的重用**。在缓存区中分配一块缓冲块，存放该 SQL 语句文本和最后的结果集，当遇到同样的 SQL 输入时，可直接把结果返回。查询结果的重用技术节约了查询计划生成时间和查询执行过程的时间，减少了查询执行全过程的资源消耗。

❑ **查询计划的重用**。缓存一条查询语句的执行计划及其相应语法树结构。查询计划的

重用技术减少了查询计划生成的时间和资源消耗。

查询重用技术有利有弊：弊端，如结果集很大会消耗很大的内存资源，同样的 SQL 不同用户获取的结果集可能不完全相同；益处，节约了 CPU 和 IO 消耗。在使用的过程中，趋利避害，应根据实际情况选用。

1.2.2 查询重写规则

查询重写是查询语句的一种**等价转换**，即对于任何相关模式的任意状态都会产生相同的结果（相同的关系替代两个表达式中相应的关系，所得到的结果是相同的）。查询重写有两个目标：

- ❑ 将查询转换为等价的、效率更高的形式，例如将效率低的谓词转换为效率高的谓词、消除重复条件等。
- ❑ 尽量将查询重写为等价、简单且不受表顺序限制的形式，为物理查询优化阶段提供更多的选择，如视图的重写、子查询的合并转换等。

查询重写的依据，是关系代数。**关系代数的等价变换规则对查询重写提供了理论上的支持**。查询重写后，查询优化器可能生成多个连接路径，可以从候选者中择优。

对查询优化技术进行分类，可有以下 4 个角度：

- ❑ **语法级**。查询语言层的优化，基于语法进行优化。
- ❑ **代数级**。查询使用形式逻辑进行优化，运用关系代数的原理进行优化。
- ❑ **语义级**。根据完整性约束，对查询语句进行语义理解，推知一些可优化的操作。
- ❑ **物理级**。物理优化技术，基于代价估算模型，比较得出各种执行方式中代价最小的。

查询重写是基于语法级、代数级、语义级的优化，可以统一归属到逻辑优化的范畴：基于代价估算模型是物理层面的优化，是从连接路径中选择代价最小的路径的过程。

查询重写技术优化思路主要包括：

- ❑ 将过程性查询转换为描述性的查询，如视图重写。
- ❑ 将复杂的查询（如嵌套子查询、外连接、嵌套连接）尽可能转换为多表连接查询。
- ❑ 将效率低的谓词转换为等价的效率高的谓词（如等价谓词重写）。
- ❑ 利用等式和不等式的性质，简化 WHERE、HAVING 和 ON 条件。

如何改进现有查询重写规则的效率，如何发现更多更有效的重写规则，是查询优化的研究内容之一。常见的查询重写技术类型，每一类都有自己的规则，这些规则没有确定的、统一的规律，**但重写的核心一定是“等价转换”，只有等价才能转换，这是需要特别强调的**。

1.2.3 查询算法优化

查询优化即求解给定查询语句的高效执行计划（有的书籍称为执行方案）的过程。

查询计划，也称为查询树，它由一系列内部的操作符组成，这些操作符按一定的运算关系构成查询的一个执行方案。简单说，就是先将表 A 和表 B 连接得到中间结果，然后再

和另外的表 C 连接得到新的中间方式，直至所有表连接完毕（连接操作就是操作符，这个示例有两个连接操作符。A 连接 B 连接 C、C 连接 B 连接 A 就是两种不同的执行方案，是两个不同的执行计划，查询优化要选出最高效的一个执行方案）。

查询计划，从形式上看是一颗二叉树，树叶是每个单表对象，两个树叶的父结点是一个连接操作符（如左外连接操作符，A left-out join B）连接后的中间结果（另外还有一些其他结点如排序操作等也可以作为中间结果），这个结果是一个临时“关系”，这样直至根结点。

所以从一个查询计划看，涉及的主要“关系结点”包括：

- **单表结点**。考虑单表的数据获取方式，是直接通过 IO 获得数据，还是通过索引获取数据，或者是通过索引定位数据的位置后再经过 IO 到数据块中获取数据。这是一个从**物理存储到内存解析成逻辑字段**的过程，即符合冯・诺依曼体系结构的要求（外存数据读入内存才能被处理）。
- **两表结点**。考虑两表以何种方式连接、代价有多大、连接路径有哪些等。表示的是内存中的元组怎么进行元组间的连接。此时，**元组通常已经存在于内存中**，直接使用即可。这是一个完成用户语义的逻辑操作，但是只是局部操作，只涉及两个具体的关系。完成用户全部语义（用户连接的语义），需要配合多表的连接顺序的操作。不同的连接算法导致的连接效率不同，如数据多时可使用 Hash 连接，外表数据量小且内表数据量大时可使用嵌套连接，数据如果有序可使用归并连接或先排序后使用归并连接等。
- **多表中间结点**。考虑多表连接顺序如何构成代价最少的“执行计划”。决定是 AB 先连接还是 BC 先连接，这是一个比较花费大小的运算。如果判断的连接方式太多，也会导致效率问题。多个关系采用不同次序进行连接，花费的 CPU 资源、内存资源差异可能较大。许多数据库采用左深树、右深树、紧密树 3 种方式或其中一部分对多表进行连接，得到多种连接路径。

查询优化目的就是生成最好的查询计划。生成最好的查询计划的策略通常有两个：

- **基于规则优化**。根据经验或一些已经探知或被证明有效的方式，定义为“规则”（如根据关系代数得知的规则、根据经验得知的规则等），用这些规则化简查询计划生成过程中符合可被化简的操作，使用启发式规则排除一些明显不好的存取路径，这就是基于规则的优化。
- **基于代价优化**。根据一个代价评估模型，在生成查询计划的过程中，计算每条存取路径（存取路径主要包括上述 3 个“关系结点”）的花费，然后选择代价最小的作为子路径，这样直至所有表连接完毕得到一个完整的路径。主流数据库都采用了基于代价策略进行优化的技术。

基于规则优化具有操作简单且能快速确定连接方式的优点，但这种方法只是排除了一部分不好的可能，所以得到的结果未必是最好的；基于代价优化是对各种可能的情况进行量化比较，从而可以得到花费最小的情况，但如果组合情况比较多则花费的判断时间就会很多；查询优化器的实现，多是两种优化策略组合使用，如 MySQL 和 PostgreSQL 就采取

了基于规则和代价估算的查询优化策略。

多表连接的优化算法中，使用最广泛的算法有如下几种：

- ❑ **SYSTEM-R 算法**。近乎穷举的搜索算法（一种空间搜索算法，其变形算法与其本质相同）。
- ❑ **启发式搜索算法**。基于规则（基于“启发式规则”抛弃不好的存取路径挑选好的）。
- ❑ **贪婪算法**。根据某种优化方式，以当前情况为基础做出最优选择，认为每次搜索过的局部存取路径是最优的，然后继续探索与其他表的连接路径。
- ❑ **动态规划算法**。将待求解的问题分解为若干个子问题（阶段），按顺序求解子问题，前一子问题的解，为后一子问题的求解提供了有用的信息。在求解任一子问题时，列出各种可能的局部解，通过决策保留那些可能达到最优的局部解，丢弃其他局部解。依次解决各子问题，最后一个子问题就是初始问题的解。
- ❑ **遗传算法**。一种启发式的优化算法，抛弃了传统的搜索方式，模拟自然界生物进化过程，基于自然群体遗传演化机制，采用人工进化的方式对目标空间进行随机化搜索。

我们知道，查询的基本操作是选择、投影和连接。选择和投影的优化规则适用于 SPJ（Select-Project-Join）和非 SPJ（SPJ+GROUPBY 等操作）；连接包括两表连接和多表连接，多表连接是其中最难的，因为多个表连接时可以有多种不同的连接次序，所以查询的执行计划的数目会随着该查询包含的表个数呈指数级增长（最大组合次数是 *n* 个关系全排列），当表个数很多时，将导致搜索空间极度膨胀，仅搜索花费最小的查询计划就需要耗费巨大的时间和资源，这是查询优化器实现时需要考虑的问题。

1.2.4 并行查询优化

传统单机数据库系统中，给定一个查询（Query），查询优化算法只需**找到查询的一个具有最小执行花费的执行计划**，这样的计划必定具有最快的响应时间。

在并行数据库系统中，查询优化的目标是**寻找具有最小响应时间的查询执行计划**，这需要把查询工作分解为一些可以并行运行的子工作。一些商业数据库提供了并行查询的功能，用以优化查询执行操作。

一个查询能否并行执行，取决于以下因素：

- ❑ **系统中的可用资源**（如内存、高速缓存中的数据量等）。
- ❑ **CPU 的数目**。
- ❑ **运算中的特定代数运算符**。如 A、B、C、D 4 个表进行连接，每个表的单表扫描可以并行进行；在生成 4 个表连接的查询计划过程中，可选择 A 和 B 连接的同时 C 和 D 进行连接，这样连接操作能并行运行。不同商业数据库，对查询并行的实现也不尽相同。在同一个 SQL 内，查询并行可以分为以下两种：
  - ❍ **操作内并行**。将同一操作如单表扫描操作、两表连接操作、排序操作等分解成多个独立的子操作，由不同的 CPU 同时执行。
  - ❍ **操作间并行**。一条 SQL 查询语句可以分解成多个子操作，由多个 CPU 执行。

1.2.5　分布式查询优化

在分布式数据库系统中，查询策略优化（主要是数据传输策略，A、B 两结点的数据进行连接，是 A 结点数据传输到 B 结点或从 B 到 A 或先各自进行过滤然后再传输等）和局部处理优化（传统的单结点数据库的查询优化技术）是查询优化的重点。

在查询优化策略中，数据的通信开销是优化算法考虑的主要因素。分布式查询优化**以减少传输的次数和数据量作为查询优化的目标**。所以，分布式数据库系统中的代价估算模型，除了考虑 CPU 代价和 IO 代价外，还要考虑通过网络在结点间传输数据的代价。这是分布式并行查询优化技术与传统单结点数据库系统最大的不同之处。

在分布式数据库系统中，代价估算模型如下：

总代价 = IO 代价 + CPU 代价 + **通信代价**

1.2.6　其他优化

数据库的查询性能，还取决于其他一些因素，如数据库集群系统中的 SD（Share Disk）集群和 SN（Share Nothing）集群，不同的架构查询优化技术也不同。SD 集群采用的是共享存储方式，在数据的读写时可能产生读写冲突，所以单表扫描会受到影响；SN 集群采用的是非共享式存储方式，所以在考虑了通信代价后单结点的优化方式依然适用。这些都不作为本书的重点，所以就不过多介绍了。

1.3　本章小结

本章辨析了数据库调优和查询优化技术的区别，从整体上介绍了查询优化涉及的主要技术，但本章内容不涉及原理的证明，只是从利于工程实践的角度出发，力图构建查询优化技术的整体概念，描绘查询优化技术的范围，从而帮助读者全面了解查询优化的技术。

第 2 章

逻辑查询优化

查询优化器在逻辑优化阶段主要解决的问题是：如何找出 SQL 语句等价的变换形式，使得 SQL 执行更高效。

一条 SQL 查询语句结构复杂，包含多种类型的子句，优化操作依赖于表的一些属性信息（如索引和约束等）。可用于优化的思路包括：

- ❑ 子句局部优化。每种类型子句都可能存在优化方式，这是子句局部的优化，如等价谓词重写、WHERE 和 HAVING 条件化简中的大部分情况，都属于这种子句范围内的局部优化。
- ❑ 子句间关联优化。子句与子句之间关联的语义存在优化的可能，如外连接消除、连接消除、子查询优化、视图重写等都属于子句间的关联优化，因为它们的优化都需要借助其他子句、表定义或列属性等信息进行。
- ❑ 局部与整体的优化。需要协同考虑局部表达式和整体的关系，如 OR 重写并集规则需要考虑 UNION 操作（UNION 是变换后的整体的形式）的花费和 OR 操作（OR 是局部表达式）的花费。
- ❑ 形式变化优化。多个子句存在嵌套，可以通过形式的变化完成优化，如嵌套连接消除。
- ❑ 语义优化。根据完整性约束、SQL 表达的含义等信息对语句进行语义优化。
- ❑ 其他优化。根据一些规则对非 SPJ 做的其他优化、根据硬件环境进行的并行查询优化等。

各种逻辑优化技术依据关系代数和启发式规则进行。

2.1 查询优化技术的理论基础

查询优化技术的理论基础是关系代数。本节就关系代数的最基本内容进行介绍，目的是引出关系代数对查询优化的指导意义。

2.1.1　关系代数

1970 年，E.F. Codd 在题为《 A Relational Model of Data for Shared Data Banks 》的论文中提出了数据的关系模型的概念。Codd 提议将关系代数作为数据库查询语言的基础。关系数据库基于关系代数。关系数据库的对外接口是 SQL 语句，所以 SQL 语句中的 DML、DQL 基于关系代数实现了关系的运算。

作为数据库查询语言的基础，关系模型由关系数据结构、关系操作集合和关系完整性约束三部分组成。以下是几个和关系模型有关的概念：

- 关系模型的数据结构就是我们在关系数据库中提到的二维结构，是一个横纵结合的二维表。在关系模型中，现实世界的实体以及实体间的各种联系均用关系来表示。
- 关系是一种对象。
- 关系的另外一个常用词是表，在本书中，关系和表混用，因为它们基本表达同一含义，只是关系更偏向于理论，表更偏向于实际数据库中可进行增、删、改、查等操作的表对象。
- 关系的元数据，即表的结构，通常称为列或属性。数据库实现中，有的用 field 表示，有的用 Item 表示。
- 关系的数据，即表的行数据，通常称为元组（tuple），也称为记录（record）。一个表可有多行元组。
- 对关系进行的操作就是关系运算。关系运算是将一定的运算符作用于一定的关系对象上，得到预期的运算结果（预期就是用户语义，用户语义通过运算符表达基本语义，通过对不同对象上的各种运算进行组合表达其对关系操作的真实语义）。运算对象、运算符、运算结果是运算的三大要素，所以关系运算就是关系运算符作用在关系上、得到的结果形式也是关系形式的操作。

关系代数的运算符包括以下 4 类[⊖]：

- **传统集合运算符**。并（UNION）、交（INTERSECTION）、差（DIFFERENCE）、积（EXTENDED CARTESIAN PRODUCT）。
- **专门的关系运算符**。选择（SELECT）、投影（PROJECT）、连接（JOIN）、除（DIVIDE）。
- **辅助运算符**。用来辅助专门的关系运算符进行操作的，包括算术比较符和逻辑运算符。
- Codd 定义了 8 个基本关系运算符后，许多人提出了新的代数操作符——**关系扩展运算符**，如半连接（SEMIJOIN）、半差（SEMIDIFFERENCE）、扩展（EXTEND）、合计（COMPOSITION）、传递闭包（TCLOSE）等，这些操作符增强了关系代数的表达能力，但不常用。

关系代数基本关系运算如表 2-1 所示，各种连接运算的语义如表 2-2 所示。

⊖ Codd 的代数的 6 个原始运算是选择、投影、笛卡儿积（又称叉积或交叉连接）、并集、差集和重命名。

表 2-1 基本关系运算与对应的 SQL 表

运算符	SQL 操作的语义	SQL 示例（R(x,y) <op> S(y,z)）
选择	单个关系中筛选元组	SELECT * FROM R WHERE **condiction**
投影	单个关系中筛选列	SELECT **col_1, col_2+2** FROM R
连接	多个关系中根据列间的逻辑运算筛选元组（通常有自然连接和等值连接）	SELECT r.col_1, s.col_2 FROM R,S [WHERE **condiction**]
除	多个关系中根据条件筛选元组（用 NOT EXISTS 的子查询实现除）	SELECT DISTINCT r1.x FROM R r1 WHERE **NOT EXISTS** (SELECT S.y FROM S WHERE **NOT EXISTS** (SELECT * FROM R r2 WHERE r2.x=r1.x AND r2.y=S.y))
并	多个关系合并元组（用 UNION 实现并）	SELECT * FROM R **UNION** SELECT * FROM S
交	多个关系中根据条件筛选元组（用两次 NOT IN 实现差 (R ∩ S=R-(R-S)））	SELECT FROM R WHERE kr **NOT IN** (SELECT kr FROM R WHERE kr **NOT IN** (SELECT ks FROM S))
差	多个关系中根据条件筛选元组（用 NOT IN 子查询实现差）	SELECT * FROM R WHERE kr **NOT IN** (SELECT ks FROM S)
积	无连接条件	SELECT R.*, S.* FROM R, S

表 2-2 各种连接运算的语义表

类 型		*R* <op> *S* 中的 op 因类型不同的语义	备 注
自然连接⊖		*R* 和 *S* 中有**公共属性**，结果包括公共属性名字上相等的所有元组的组合，在结果中把重复的列去掉	需要去掉重复列，是**同时从行和列的角度进行运算**
θ- 连接		*R* 和 *S* 中**没有公共属性**，结果包括在 *R* 和 *S* 中满足操作符 θ 的所有组合，操作符 θ 通常包括＜、≤、=、＞、≥	从关系 *R* 与 *S* 的广义笛卡儿积中选取 A、B **属性值相等**的那些元组。是**从行的角度进行运算**
等值连接		操作符是 = 的 θ- 连接	
半连接		结果包括在 *S* 中**公共属性**名字上相等的元组的所有的 *R* 中的元组（即结果中包括 *R* 的部分元组，而 *R* 中的部分元组的公共属性的值在 *S* 中同样存在）	SQL 中没有自己的连接操作符，使用 EXISTS、IN 关键字做子句的子查询常被查询优化器转换为半连接
反连接		反连接的结果是在 *S* 中**没有在公共属性**名字上相等的元组的 *R* 中的那些元组	为半连接的补集，反连接有时称为反半连接。在 SQL 中没有自己的连接操作符，使用了 NOT EXISTS 则被查询优化器转换为反半连接
外连接	左外连接	结果包含 *R* 中所有元组，对每个元组，若在 *S* 中有在公共属性名字上相等的元组，则正常连接；若在 *S* 中没有在公共属性名字上相等的元组，则依旧保留此元组，并将对应的其他列设为 NULL	

⊖ 在数据库中对应了主外键的语义。

（续）

类　型		R <op> S 中的 op 因类型不同的语义	备　注
外连接	右外连接	结果包含 S 中所有元组，对每个元组，若在 R 中有在公共属性名字上相等的元组，则正常连接；若在 R 中没有在公共属性名字上相等的元组，则依旧保留此元组，并将对应的其他列设为 NULL	
	全外连接	结果包含 R 与 S 中所有元组，对每个元组，若在另一关系中有在公共属性名字上相等的元组，则正常连接：若在另一关系中没有在公共属性名字上相等的元组，则依旧保留此元组，并将对应的其他列设为 NULL	

2.1.2　关系代数等价变换规则对优化的指导意义

关系代数表达式的等价：也就是说用相同的关系代替两个表达式中相应的关系，所得到的结果是相同的。两个关系表达式 El 和 E2 是等价的，记为 E1 ≡ E2。

查询语句可以表示为一棵二叉树[⊖]，其中：

- ❑ 叶子是关系。
- ❑ 内部结点是运算符（或称算子、操作符，如 LEFT OUT JOIN），表示左右子树的运算方式。
- ❑ 子树是子表达式或 SQL 片段。
- ❑ 根结点是最后运算的操作符。
- ❑ 根结点运算之后，得到的是 SQL 查询优化后的结果。
- ❑ 这样一棵树就是一个查询的路径。
- ❑ 多个关系连接，连接顺序不同，可以得出多个类似的二叉树。
- ❑ 查询优化就是找出代价最小的二叉树，即最优的查询路径。每条路径的生成，包括了单表扫描、两表连接、多表连接顺序、多表连接搜索空间等技术。
- ❑ 基于代价估算的查询优化就是通过计算和比较，找出花费最少的最优二叉树。

上述的最后两项，主要依据本章查询重写规则和第 3 章物理查询优化中涉及的技术，对查询优化引擎做关系代数和启发式规则的逻辑查询优化、基于代价估算模型择优的物理查询优化，从而帮助数据库查询优化器实现查询优化。

1. 从运算符的角度考虑优化

不同运算符根据其特点，可以对查询语句做不同的优化，优化可以减少中间生成物的大小和数量，节约 IO、内存等，从而提高了执行速度。**但优化的前提是：优化前和优化后的语义必须等价**。不同运算符的优化规则和可优化的原因如表 2-3 所示。

⊖ “查询语句表示为一棵二叉树”的过程，首先是语法分析得到一棵查询树的过程；其次伴有语义分析等工作；再次是根据关系代数进行了数据库的逻辑查询优化；最后是根据代价估算算法进行物理查询优化。优化后的结果，被送到执行器执行。

表 2-3 运算符主导的优化

运算符	子类型	根据特点可得到的优化规则	可优化的原因
选择	基本选择性质	对同一个表的同样选择条件，作一次即可	幂等性：多次应用同一个选择有同样效果 交换性：应用选择的次序在最终结果中没有影响 选择可有效减少在它的操作数中的元组数的运算（元组个数减少）
	分解有复杂条件的选择	合取，合并多个选择为更少的需要求值的选择，多个等式则可以合并⊖	合取的选择等价于针对这些单独条件的一系列选择
		析取，分解它们使得其成员选择可以被移动或单独优化⊜	析取的选择等价于选择的并集
	选择和叉积	尽可能先做选择	关系有 N 和 M 行，先做积运算将包含 $N \times M$ 行。先做选择运算减少 N 和 M，则可避免不满足条件的元组参与积的运算，节约时间同时减少结果的大小
		尽可能下推选择	如果积不跟随着选择运算，可以尝试使用其他规则从表达式树更高层下推选择
	选择和集合运算	选择下推到的集合运算中，如表 2-4 中的 3 种情况	选择在差集、交集和并集算子上满足分配律
	选择和投影	在投影之前进行选择	如果选择条件中引用的字段是投影中的字段的子集，则选择与投影满足交换性
投影	基本投影性质	尽可能先做投影	投影是幂等性的；投影可以减少元组大小
	投影和集合运算	投影下推到的集合运算中，如表 2-5 中的情况	投影在差集、交集和并集算子上满足分配律

表 2-4 选择下推到集合的运算

初 始 式	优化后的等价表达式		
	等价表达式一	等价表达式二	等价表达式三
$\sigma_A(R \backslash S)$	$\sigma_A(R) \backslash \sigma_A(S)$	$\sigma_A(R) \backslash S$	
$\sigma_A(R \cup S)$	$\sigma_A(R) \cup \sigma_A(S)$		
$\sigma_A(R \cap S)$	$\sigma_A(R) \cap \sigma_A(S)$	$\sigma_A(R) \cap S$	$R \cap \sigma_A(S)$

表 2-5 投影下推到集合的运算

初始式	优化后的等价表达式
$\Pi_{A_1,A_2,\cdots,A_n}(R \backslash S)$	$\Pi_{A_1,A_2,\cdots,A_n}(R) \backslash \Pi_{A_1,A_2,\cdots,A_n}(S)$
$\Pi_{A_1,A_2,\cdots,A_n}(R \cup S)$	$\Pi_{A_1,A_2,\cdots,A_n}(R) \cup \Pi_{A_1,A_2,\cdots,A_n}(S)$
$\Pi_{A_1,A_2,\cdots,A_n}(R \cap S)$	$\Pi_{A_1,A_2,\cdots,A_n}(R) \cap \Pi_{A_1,A_2,\cdots,A_n}(S)$

⊖ 如 WHERE A.a=B.b AND B.b=C.c 可以合并为 ={A.a, B.b, C.c} 而不是两个等式 ={A.a, B.b } 和 ={B.b, C.c}。

⊜ 如 WHERE A.a=3 OR A.b>8，如果 A.a、A.b 列上分别有索引，也许 SELECT * FROM A WHERE A.a=3 UNION SELECT * FROM A WHERE A.b>8 可以分别利用各自的索引提高查询效率。

对于表 2-4 和表 2-5，我们以“$\sigma_A(R \times S)$”为例，表明它们可做优化的共同意义。

初始式是 $\sigma_A(R \times S)$，条件 A 可分解为“$B \wedge C \wedge D$”，条件 B 只与关系 R 有关，条件 C 只与关系 S 有关，则初始式可以变形为：$\sigma_{B \wedge C \wedge D}(R \times S)$，表示为查询树的结构如图 2-1 所示。

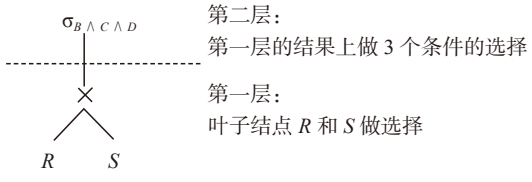


图 2-1　查询树结构的初始样式

图 2-2 所示为第一层做完选择后，每个叶子结点对应的元组数“可能”比图 2-1 中的 R 和 S 少，如果 B 条件和 C 条件至少有一个能够大量减少 R 或 S 的元组，则中间结果大量减少，优化的效果会更好（B 条件和 C 条件对元组过滤程度依赖于“选择率”）。如果 R 和 S 上有 B 条件、C 条件可以使用的索引，则利用索引可加快元组的获取，优化的效果会更好。

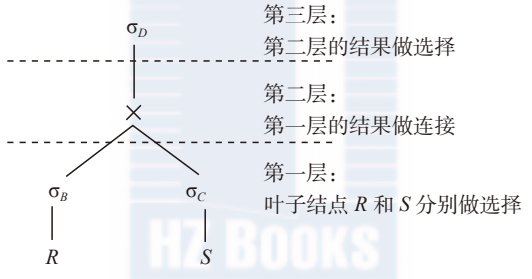


图 2-2　查询树结构等价的变形式

如果索引是唯一索引或主键（主键不允许重复，不允许为 NULL 值，多数数据库为主键实现了一个唯一索引）之类，条件表达式是等值表达式（= 运算非范围运算），定位元组的速度更快（可直接利用索引而不用扫描数据）、满足条件的元组更少，所以优化的效果会更佳。

图 2-1 中在连接后进行选择操作，一是中间结果的元组数量大，二是需要对中间结果的每条元组都进行 B、C、D 3 个条件的判断，增加了条件判断的成本，效率很低。

经过等价变换优化带来的好处，再加上避免了原始方式引入的坏处，使得查询效率明显获得提升。

2. 从运算规则的角度考虑优化

前面我们从运算符的角度出发考虑了优化。因为运算符中考虑的子类型（见表 2-3 中的“子类型”列）实则是部分考虑了运算符间的关系、运算符和操作数间的关系，其本质是运算规则在起作用。所以前节考虑过关系代数运算规则对优化的作用，但不完整，这里补充余下的对优化有作用的主要关系代数运算规则。下面的运算规则是**查询重写技术作等价转换的基础**，如表 2-6 所示。

表 2-6 运算规则主导的优化

名 称	公 式	对优化的意义
连接㊀、笛卡儿积交换律㊁	$E_1 \times E_2 \equiv E_2 \times E_1$ $E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$ $E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$ F F	做连接、做积运算，可以交换前后位置，其结果不变。如两表连接算法中有嵌套连接算法，对外表和内表有要求，外表尽可能小则有利于做“基于块的嵌套循环连接”，所以通过交换律可以把元组少的表作为外表
连接、笛卡儿积的结合律㊂	$(E_1 \times E_2) \times E_3 \equiv E_1 \times (E_2 \times E_3)$ $(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$ $(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$ F_1 F_2 F_1 F_2	做连接、笛卡儿积运算，如果新的结合有利于减少中间关系的大小，则可以优先处理
投影的串接定律	$\Pi_{A_1,A_2,\cdots,A_n}(\Pi B1,B2,\cdots,B_m(E)) \equiv \Pi_{A_1,A_2,\cdots,A_n}(E)$ $A_i(i=1,2,\cdots,n)$, $B_j(j=1,2,\cdots,m)$ 是列名，且 $\{A_1,A_2,\cdots,A_n\}$ 是 $\{_{B_1},_{B_2},\cdots,_{B_m}\}$ 的子集	在同一个关系上，只需做一次投影运算，且一次投影时选择多列同时完成。所以许多数据库优化引擎为同一个关系收集齐本关系上的所有列（目标列和 WHERE、GROUPBY 等子句的本关系的列）
选择的串接定律	$\sigma_{F_1}(\sigma_{F_2}(E)) \equiv \sigma_{F_1 \wedge F_2}(E)$	选择条件可以合并，使得可一次就检查全部条件，不必多次过滤元组，所以可以把同层的合取条件收集在一次，统一进行判断
选择与投影的交换律	$\sigma_F(\Pi_{A_1,A_2,\cdots,A_n}(E)) \equiv \Pi_{A_1,A_2,\cdots,A_n}(\sigma_F(E))$ 选择条件 F 只涉及属性 $A_1,\cdots,A_n$	先投影后选择可以改为先选择后投影，这对于以行为存储格式的主流数据库而言，很有优化意义。存储方式总是在先获得元组后才能解析得到其中的列
	$\Pi_{A_1,A_2,\cdots,A_n}(\sigma_F(E)) \equiv \Pi_{A_1,A_2,\cdots,A_n}(\sigma_F(\Pi_{A_1,A_2,\cdots,A_n,B_1,B_2,\cdots,B_m}(E)))$ 选择条件 F 中有不属于 $A_1,\cdots,A_n$ 的属性 $B_1,\cdots,B_m$	先选择后投影可以改为带有选择条件中列的投影后再选择，最后再完成最外层的投影，这样，使得内层的选择和投影可以同时进行
选择与笛卡儿积的分配律	$\sigma_F(E_1 \times E_2) \equiv \sigma_F(E_1) \times E_2$ F 中涉及的属性都是 E_1 中的属性	条件下推到相关的关系上，先做选择后做积运算，这样可以减小中间结果的大小
	$\sigma_F(E_1 \times E_2) \equiv \sigma_{F_1}(E_1) \times \sigma_{F_2}(E_2)$ 如果 $F=F_1 \wedge F_2$，F_1 只涉及 E_1 中的属性，F_2 只涉及 E_2 中的属性	
	$\sigma_F(E_1 \times E_2) \equiv \sigma_{F_2}(\sigma_{F_1}(E_1) \times E_2)$ F_1 只涉及 E_1 中的属性，F_2 涉及 E_1 和 E_2 两者的属性	
选择与并的分配律	$\sigma_F(E_1 \cup E_2) \equiv \sigma_F(E_1) \cup \sigma_F(E_2)$ E_1、E_2 有相同的列名	条件下推到相关的关系上，先选择后做并运算，可以减小每个关系输出结果的大小
选择与差运算的分配律	$\sigma_F(E_1-E_2) \equiv \sigma_F(E_1)-\sigma F(E_2)$ E_1 与 E_2 有相同的列名	条件下推到相关的关系上，先选择后做差运算，可以减小每个关系输出结果的大小

㊀ $\bowtie$，表示连接；$\times$，表示积。

㊁ 并和交操作，也满足交换律，但优化的意义不大。

㊂ 并和交操作，也满足结合律，但优化的意义不大。

（续）

名　称	公　式	对优化的意义
投影与笛卡儿积的分配律	$\Pi_{A_1,A_2,\cdots,A_n,B_1,B_2,\cdots,B_m}(E_1 \times E_2) \equiv \Pi_{A_1,A_2,\cdots,A_n}(E_1) \times \Pi_{B_1,B_2,\cdots,B_m}(E_2)$ A_1，…，A_n 是 E_1 的属性，B_1，…，B_m 是 E_2 的属性	先投影后做积，可减少做积前每个元组的长度，使得做积后得到的新元组的长度也变短
投影与并的分配律	$\Pi_{A_1,A_2,\cdots,A_n}(E_1 \cup E_2) \equiv \Pi_{A_1,A_2,\cdots,A_n}(E_1) \cup \Pi_{A_1,A_2,\cdots,A_n}(E_2)$ E_1 和 E_2 有相同的列名	先投影后做并，可减少做并前每个元组的长度

2.2　查询重写规则

传统的联机事务处理（On-line Transaction Processing，OLTP）使用基于选择（SELECT）、投影（PROJECT）、连接（JOIN）3 种基本操作相结合的查询，这种查询称为 SPJ 查询。

数据库在查询优化的过程中，会对这 3 种基本操作进行优化。优化的方式如下：

- **选择操作**。对应的是**限制条件**（格式类似 field <op> consant，field 表示列对象，op 是操作符，如 =、> 等），优化方式是选择操作下推，目的是尽量减少连接操作前的元组数，使得中间临时关系**尽量少**（元组数少，连接得到的元组数就少），这样可减少 IO 和 CPU 的消耗，节约内存空间。
- **投影操作**。对应的 SELECT 查询的目的列对象，优化方式是投影操作下推，目的是尽量减少连接操作前的列数，使得中间临时关系**尽量小（特别注意差别：选择操作是使元组的个数“尽量少”，投影操作是使一条元组“尽量小”）**，这样虽然不能减少 IO（多数数据库存储方式是行存储，元组是读取的最基本单位，所以要想操作列则必须读取一行数据），但可以各减少连接后的中间关系的元组大小，节约内存空间。
- **连接操作**。对应的是**连接条件**（格式类似 field_1 <op> field_2，field_1 和 field_2 表示不同表上的列对象，op 是操作符，如 =、> 等），表示两个表连接的条件。这里涉及以下两个子问题。
  - **多表连接中每个表被连接的顺序决定着效率**。如果一个查询语句只有一个表，则这样的语句很简单；但如果有多个表，则会涉及表之间以什么样的顺序连接效率最高效（如 A、B、C 三表连接，如果 ABC、ACB、BCA 等连接后的结果集一样，则计算哪种连接次序的效率最高，是需要考虑的问题）。
  - **多表连接每个表被连接的顺序由用户语义决定**。查询语句多表连接有着不同的语义（如是笛卡儿集、内连接，还是外连接中的左外连接等），这决定着表之间的前后连接次序是不能随意更换的，否则，结果集中数据是不同的。因此，表的前后连接次序是不能随意交换的。

另外，根据 SQL 语句的形式特点，还可以做如下区分：

- **针对 SPJ 的查询优化**。基于选择、投影、连接 3 种基本操作相结合的查询。

- **针对非 SPJ 的查询优化**。在 SPJ 的基础上存在 GROUPBY 操作的查询，这是一种较为复杂的查询。

所以，针对 SPJ 和非 SPJ 的查询优化，其实是对以上多种操作的优化。“选择”和“投影”操作，可以在关系代数规则的指导下进行优化。表连接，需要多表连接的相关算法完成优化。其他操作的优化多是基于索引和代价估算完成的。

2.2.1 子查询的优化

子查询是查询语句中经常出现的一种类型，是比较耗时的操作。优化子查询对查询效率的提升有着直接的影响，所以子查询优化技术，是数据库查询优化引擎的重要研究内容。

从子查询出现在 SQL 语句的位置看，它可以出现在目标列、FROM 子句、WHERE 子句、JOIN/ON 子句、GROUPBY 子句、HAVING 子句、ORDERBY 子句等位置。子查询出现在不同位置对优化的影响如下：

- **目标列位置**。子查询如果位于目标列，则只能是标量子查询，否则数据库可能返回类似“错误：子查询必须只能返回一个字段”的提示。
- **FROM 子句位置**。相关子查询出现在 FROM 子句中，数据库可能返回类似“在 FROM 子句中的子查询无法参考相同查询级别中的关系”的提示，所以相关子查询不能出现在 FROM 子句中；非相关子查询出现在 FROM 子句中，可上拉子查询到父层，在多表连接时统一考虑连接代价后择优。
- **WHERE 子句位置**。出现在 WHERE 子句中的子查询是一个条件表达式的一部分，而表达式可以分解为操作符和操作数；根据参与运算的数据类型的不同，操作符也不尽相同，如 INT 型有 >、<、=、<> 等操作，这对子查询均有一定的要求（如 INT 型的等值操作，要求子查询必须是标量子查询）。另外，子查询出现在 WHERE 子句中的格式也有用谓词指定的一些操作，如 IN、BETWEEN、EXISTS 等。
- **JOIN/ON 子句位置**。JOIN/ON 子句可以拆分为两部分，一是 JOIN 块类似于 FROM 子句，二是 ON 子句块类似于 WHERE 子句，这两部分都可以出现子查询。子查询的处理方式同 FROM 子句和 WHERE 子句。
- **GROUPBY 子句位置**。目标列必须和 GROUPBY 关联[⊖]。可将子查询写在 GROUPBY 位置处，但子查询用在 GROUPBY 处没有实用意义。
- **ORDERBY 子句位置**。可将子查询写在 ORDERBY 位置处。但 ORDERBY 操作是作用在整条 SQL 语句上的，子查询用在 ORDERBY 处没有实用意义。

1. 子查询的分类

根据子查询中涉及的关系对象与外层关系对象间的关系，子查询可以分为以下两类：

⊖ SQL 规范要求，如果有 GROUPBY 子句存在，所有出现在 SELECT 目标列的列（除了用于聚集函数的列外）都必须出现在 GROUPBY 子句中。

❑ **相关子查询**。子查询的执行依赖于外层父查询的一些属性值。子查询因依赖于父查询的参数，当父查询的参数改变时，子查询需要根据新参数值重新执行（查询优化器对相关子查询进行优化有一定意义），如：

```
SELECT * FROM t1 WHERE col_1 = ANY
    (SELECT col_1 FROM t2 WHERE t2.col_2 = t1.col_2);/* 子查询语句中存在父查询的 t1 表的 col_2 列 */
```

❑ **非相关子查询**。子查询的执行不依赖于外层父查询的任何属性值，这样的子查询具有独立性，可独自求解，形成一个子查询计划先于外层的查询求解，如：

```
SELECT * FROM t1 WHERE col_1 = ANY
    (SELECT col_1 FROM t2 WHERE t2.col_2 = 10);// 子查询语句中 (t2) 不存在父查询 (t1) 的属性
```

从特定谓词看，子查询可分为以下 3 类：

❑ **[NOT] IN/ALL/ANY/SOME 子查询**。语义相近，表示“[取反] 存在 / 所有 / 任何 / 任何”，左面是操作数，右面是子查询，是最常见的子查询类型之一。

❑ **[NOT] EXISTS 子查询**。半连接语义，表示“[取反] 存在”，没有左操作数，右面是子查询，也是最常见的子查询类型之一。

❑ **其他子查询**。除了上述两种外的所有子查询。

从语句的构成复杂程度看，子查询可分为以下 3 类：

❑ **SPJ 子查询**。由选择、连接、投影操作组成的查询。

❑ **GROUPBY 子查询**。SPJ 子查询加上分组、聚集操作组成的查询。

❑ **其他子查询**。GROUPBY 子查询中加上其他子句如 Top-N、LIMIT/OFFSET、集合、排序等操作。后两种子查询有时合称非 SPJ 子查询。

从结果集的角度看，子查询分为以下 4 类：

❑ **标量子查询**。子查询返回的结果集类型是一个单一值（return a scalar，a single value）。

❑ **列子查询**。子查询返回的结果集类型是一条单一元组（return a single row）。

❑ **行子查询**。子查询返回的结果集类型是一个单一列（return a single column）。

❑ **表子查询**。子查询返回的结果集类型是一个表（多行多列）（return a table，one or more rows of one or more columns）。

2. 子查询的优化思路

通过上面的介绍，我们知道了都有哪些类型的子查询及每类子查询的特点，下面就讲一下子查询的优化思路。要明白子查询是如何优化的，就要先明白为什么要做子查询优化。

（1）做子查询优化的原因

为什么要做子查询优化呢？

在数据库实现早期，查询优化器对子查询一般采用嵌套执行的方式，即对父查询中的每一行，都执行一次子查询，这样子查询会执行很多次。这种执行方式效率很低。

而对子查询进行优化，可能带来几个数量级的查询效率的提高。子查询转变成为连接操作之后，会得到如下好处：

- ❑ 子查询不用执行很多次。
- ❑ 优化器可以根据统计信息来选择不同的连接方法和不同的连接顺序。
- ❑ 子查询中的连接条件、过滤条件分别变成了父查询的连接条件、过滤条件，优化器可以对这些条件进行下推，以提高执行效率。

（2）子查询优化技术

子查询优化技术的思路如下：

- ❑ **子查询合并**（Subquery Coalescing）。在某些条件下（语义等价：两个查询块产生同样的结果集），**多个子查询能够合并成一个子查询**（合并后还是子查询，以后可以通过其他技术消除子查询）。这样可以把多次表扫描、多次连接减少为单次表扫描和单次连接，如：

```
SELECT * FROM t1 WHERE a1<10 AND (
    EXISTS (SELECT a2 FROM t2 WHERE t2.a2<5 AND t2.b2=1) OR
    EXISTS (SELECT a2 FROM t2 WHERE t2.a2<5 AND t2.b2=2)
);
```

可优化为：

```
SELECT * FROM t1 WHERE a1<10 AND (
     EXISTS (SELECT a2 FROM t2 WHERE t2.a2<5 AND (t2.b2=1 OR t2.b2=2) /* 两个 EXISTS 子句合并为一个，条件也进行了合并 */
);
```

- ❑ **子查询展开**（Subquery Unnesting）。又称子查询反嵌套，又称为子查询上拉。把一些子查询置于外层的父查询中，**作为连接关系**与外层父查询并列，其实质是**把某些子查询重写为等价的多表连接操作**（展开后，子查询不存在了，外层查询变成了多表连接）。带来的好处是，有关的访问路径、连接方法和连接顺序可能被有效使用，使得查询语句的层次尽可能地减少。常见的 IN/ANY/SOME/ALL/EXISTS 依据情况转换为半连接（SEMI JOIN）、普通类型的子查询消除等情况属于此类，如：

```
SELECT * FROM t1, (SELECT * FROM t2 WHERE t2.a2 >10) v_t2
WHERE t1.a1<10 AND v_t2.a2<20;
```

可优化为：

```
SELECT * FROM t1, t2 WHERE t1.a1<10 AND t2.a2<20 AND t2.a2 >10; /* 子查询变为了 t1、t2 表的连接操作，相当于把 t2 表从子查询中上拉了一层 */
```

- ❑ **聚集子查询消除**（Aggregate Subquery Elimination）。聚集函数上推，将子查询转变为一个新的不包含聚集函数的子查询，并与父查询的部分或者全部表做左外连接。通常，一些系统支持的是标量聚集子查询[⊖]消除，如：

⊖ 标量聚集子查询：在子查询中，存在聚集函数（如 MAX()、COUNT() 等），没有指定 GROUPBY 子句。

```
SELECT * FROM t1 WHERE t1.a1>(SELECT avg(t2.a2) FROM t2);
```

❏ **其他**。利用窗口函数消除子查询的技术（Remove Subquery using Window functions，RSW）、子查询推进（Push Subquery）等技术可用于子查询的优化，这里不展开讨论。

（3）子查询展开

子查询展开是一种最为常用的子查询优化技术，子查询展开有以下两种形式：

❏ 如果子查询中出现了聚集、GROUPBY、DISTINCT 子句，则子查询只能单独求解，不可以上拉到上层。

❏ 如果子查询只是一个简单格式（SPJ 格式）的查询语句，则可以上拉到上层，这样往往能提高查询效率。子查询上拉讨论的就是这种格式，这也是子查询展开技术处理的范围。

把子查询上拉到上层查询，前提是上拉（展开）后的结果不能带来多余的元组，所以子查询展开需要遵循如下规则：

❏ 如果上层查询的结果没有重复（即 SELECT 子句中包含主码），则可以展开其子查询，并且展开后的查询的 SELECT 子句前应加上 DISTINCT 标志。

❏ 如果上层查询的 SELECT 语句中有 DISTINCT 标志，则可以直接进行子查询展开。

❏ 如果内层查询结果没有重复元组，则可以展开。

子查询展开的具体步骤如下：

1）将子查询和上层查询的 FROM 子句连接为同一个 FROM 子句，并且修改相应的运行参数。

2）将子查询的谓词符号进行相应修改（如：IN 修改为 =ANY）。

3）将子查询的 WHERE 条件作为一个整体与上层查询的 WHERE 条件合并，并用 AND 条件连接词连接，从而保证新生成的谓词与原谓词的上下文意思相同，且成为一个整体。

3. 最常见的子查询类型的优化

子查询的格式有多种，常见的子查询格式有 IN 类型、ALL/ANY/SOME 类型、EXISTS 类型。下面我们就基于这 3 种常见类型对子查询类型的优化进行介绍。

（1）IN 类型

IN 类型有 3 种不同的格式，具体如下。

格式一：

```
outer_expr [NOT] IN (SELECT inner_expr FROM ... WHERE subquery_where)
```

格式二：

```
outer_expr =ANY (SELECT inner_expr FROM ... WHERE subquery_where)
```

格式三：

```
(oe_1, ..., oe_N) [NOT] IN (SELECT ie_1, ..., ie_N FROM ... WHERE subquery_where)
```

对于 IN 类型子查询的优化（可以转换的形式和转换需要的必备条件分为几种情况），如

表 2-7 所示。

表 2-7 IN 类型子查询优化的情况表

算子 IN 的左操作数 \ 算子 IN 的右操作数		inner_expr	
		结果为 NULL 值	结果为非 NULL 值
outer_expr	结果为 NULL 值	情况三	无意义
	结果为非 NULL 值	情况二	情况一

情况一：outer_expr 和 inner_expr 均为非 NULL 值。

优化后的表达式（外部条件 outer_expr 下推到子查询中）如下[⊖]：

```
EXISTS (SELECT 1 FROM ... WHERE subquery_where AND outer_expr=inner_expr)
```

即：

```
EXISTS (SELECT 1 FROM ... WHERE subquery_where
                            AND oe_1 = ie_1
                            AND ...
                            AND oe_N = ie_N)
```

子查询优化需要满足的以下两个条件（全部满足）：

- outer_expr 和 inner_expr 不能为 NULL。
- 不需要从结果为 FALSE 的子查询中区分 NULL。

情况二：outer_expr 是非 NULL 值（情况一的两个转换条件中至少有一个不满足时）。

优化后的表达式（外部条件 outer_expr 下推到子查询中，另外内部条件 inner_expr 为 NULL）如下：

```
EXISTS (SELECT 1 FROM ... WHERE subquery_where AND
        (outer_expr=inner_expr OR inner_expr IS NULL))
```

假设 outer_expr 是非 NULL 值，但是如果 outer_expr=inner_expr 表达式不产生数据，则 outer_expr IN (SELECT ...) 的计算结果如果是 NULL 值或 FALSE 值时，转换条件如下：

- 为 NULL 值。SELECT 语句查询得到任意的行数据，inner_expr 是 NULL（outer_expr IN (SELECT ...)==NULL）。
- 为 FALSE 值。SELECT 语句产生非 NULL 值或不产生数据（outer_expr IN (SELECT ...)==FALSE）。

情况三：outer_expr 为 NULL 值。

原先的表达式等价于如下形式：

```
NULL IN (SELECT inner_expr FROM ... WHERE subquery_where)
```

假设 outer_expr 是 NULL 值，NULL IN (SELECT inner_expr ...) 的计算结果如果是

⊖ 注意，IN 优化为 EXISTS，是一种表意的方式，不是 IN 谓词被转化为 EXISTS 谓词，其表达的是 IN 谓词被优化为“半连接”操作，这适用于情况一、情况二和情况三。

NULL 值或 FALSE 值时，转换条件如下：

- ❏ 为 NULL 值。SELECT 语句产生任意行数据。
- ❏ 为 FALSE 值。SELECT 语句不产生数据。

对于上面的等价形式，还有两点需要说明：

- ❏ 谓词 IN 等价于 = ANY，例如下面的两条 SQL 语句是等价的：

```
SELECT col_1 FROM t1 WHERE col_1 = ANY (SELECT col_1 FROM t2);
SELECT col_1 FROM t1 WHERE col_1 IN      (SELECT col_1 FROM t2);
```

- ❏ 带有谓词 IN 的子查询，如果满足上述 3 种情况，可以做等价变换，把外层的条件下推到子查询中，变形为一个 EXISTS 类型的逻辑表达式判断；而子查询为 EXISTS 类型则可以被半连接算法实现优化。

下面我们看几个具体的示例（以 PostgreSQL 为例说明子查询优化的情况）。

示例 1　初始数据如下所示：

```
test=# SELECT * FROM x;
x_num | x_name
----------+----------
       1 | X_1
       2 | X_2
         | X_3
(3 rows)
test=# SELECT * FROM y;
y_num | y_name
----------+----------
       1 | Y_1
         | Y_2
       3 | Y_3
(3 rows)
```

执行如下子查询命令：

```
test=# SELECT * FROM x WHERE x_num IN(SELECT y_num FROM y); //IN 子查询
 x_num | x_name
----------+----------
       1 | X_1
(1 row)
```

执行计划具体如下：

```
test=# EXPLAIN SELECT * FROM xWHERE x_num IN(SELECT y_num FROM y);
                              QUERY PLAN
-----------------------------------------------------------------
 Hash IN Join  (cost=1.07..2.14 rows=3 width=18)
    Hash Cond: (x.x_num = y.y_num)
    ->Seq Scan on x  (cost=0.00..1.03 rows=3 width=18)
    ->Hash  (cost=1.03..1.03 rows=3 width=4)
          ->Seq Scan on y  (cost=0.00..1.03 rows=3 width=4)
(5 rows)
```

示例说明：

- ❑ Hash IN Join 表示执行的是两表 Hash 连接（已经使用两表连接替代了子查询），IN 表示原子查询是“IN 子查询”。
- ❑ 原始查询 SQL 中没有 (x.x_num = y.y_num)，只是一个 IN 子查询，而查询计划中出现 (x.x_num = y.y_num)，表明此 IN 子查询被优化，优化后执行的是符合连接条件 (x.x_num = y.y_num) 的一个两表连接的查询。

示例 2 表 t_1、表 t_2 定义和数据如下所示：

```
CREATE TABLE t_1 (t_1_id INT UNIQUE, t_1_col_1 INT, t_1_col_2 VARCHAR(10));
CREATE TABLE t_2 (t_2_id INT UNIQUE, t_2_col_1 INT, t_2_col_2 VARCHAR(10));
INSERT INTO t_1 VALUES (1, 11, 't_1_1');
INSERT INTO t_1 VALUES (2, 12, NULL);
INSERT INTO t_1 VALUES (3, NULL, 't_1_3');
INSERT INTO t_1 VALUES (4, 14, 't_1_4');
INSERT INTO t_1 VALUES (5, 15, NULL);
INSERT INTO t_1 VALUES (7, NULL, NULL);
INSERT INTO t_2 VALUES (1, 11, 't_2_1');
INSERT INTO t_2 VALUES (2, NULL, 't_2_2');
INSERT INTO t_2 VALUES (3, 13, NULL);
INSERT INTO t_2 VALUES (4, 14, 't_2_4');
INSERT INTO t_2 VALUES (6, 16, 't_2_6');
INSERT INTO t_2 VALUES (7, NULL, NULL);
```

语句一 简单的 IN 子查询如下：

```
SELECT t_1.* FROM t_1 WHERE t_1_id IN (SELECT t_2_id FROM t_2);
```

语句二 简单 IN 子查询但子查询结果不影响父查询，如下所示：

```
SELECT t_1.* FROM t_1 WHERE 10 IN (SELECT t_2_id FROM t_2);
```

语句三 父查询中的易失函数影响子查询的优化，如下所示：

```
SELECT t_1.* FROM t_1 WHERE t_1_id + div((random()*10)::numeric,2) IN (SELECT t_2_idFROM t_2);
```

对于语句一，简单的 IN 子查询的查询执行计划如下：

```
test=# EXPLAIN SELECT t_1.* FROM t_1 WHERE t_1_id IN (SELECT t_2_id FROM t_2);
                              QUERY PLAN
-----------------------------------------------------------------
 Hash Semi Join  (cost=45.33..94.58 rows=1570 width=22)
    Hash Cond: (t_1.t_1_id = t_2.t_2_id)
    ->Seq Scan on t_1  (cost=0.00..25.70 rows=1570 width=22)
    ->Hash  (cost=25.70..25.70 rows=1570 width=4)
          ->Seq Scan on t_2  (cost=0.00..25.70 rows=1570 width=4)
(5 rows)
```

查询优化器对查询做了优化，把子查询转换使用 t_1.t_1_id = t_2.t_2_id 作为连接条件的为 Hash 半连接（Hash Semi Join）。如果不做优化，则 t_1 表有多少条记录，就需要扫描

t_2 表多少次，而且每次都得对 t_2 表做全表顺序扫描，这样会花费较多时间；而优化后，对 t_2 表只做了一次全表顺序扫描，然后采用 Hash Semi Join 算法，对 t_1 和 t_2 做连接操作即可，节约了很多次对 t_2 表的全表扫描，达到了优化的目的。

对于语句二，简单却不影响父查询的 IN 子查询的查询执行计划如下：

```
test=# EXPLAIN SELECT t_1.* FROM t_1 WHERE 10 IN (SELECT t_2_id FROM t_2);
                              QUERY PLAN
-----------------------------------------------------------------------------
 Result  (cost=29.63..55.33 rows=1570 width=22)
   One-Time Filter: (hashed SubPlan 1)
   ->Seq Scan on t_1  (cost=29.63..55.33 rows=1570 width=22)
   SubPlan 1
      ->Seq Scan on t_2  (cost=0.00..25.70 rows=1570 width=4)
(5 rows)
```

查询优化器不能对查询做优化。查询计划是对 t_2 做一个顺序扫描，结果作为过滤器为 t_1 扫描服务。子查询与上层查询（t_1 表所在的查询）没有关系，而 IN 子查询的左操作符是常量，所以对子查询直接求值，没有办法做上拉优化。

对于语句三，带有易失函数 random() 的 IN 子查询的查询执行计划如下：

```
test=# EXPLAIN SELECT t_1.* FROM t_1 WHERE t_1_id + div((random()*10)::numeric,2) IN (SELECT t_2_idFROM t_2);
                              QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on t_1  (cost=29.63..86.73 rows=785 width=22)
   Filter: (hashed SubPlan 1)
   SubPlan 1
      ->Seq Scan on t_2  (cost=0.00..25.70 rows=1570 width=4)
(4 rows)
```

查询中出现了易失函数 random()，子查询结果不确定，查询优化器就不能对子查询做优化。

（2）ALL/ANY/SOME 类型

ALL/ANY/SOME 类型的子查询格式，具体如下：

```
outer_expr operator ANY (subquery)
outer_expr operator SOME (subquery)
outer_expr operator ALL (subquery)
```

ALL 表示对于子查询的全部元组进行 operator 指定的操作；ANY 表示对于子查询的任何元组进行 operator 指定的操作；SOME 表示对于子查询的某些元组进行 operator 指定的操作。另外，使用 ALL/ANY/SOME 类型的子查询，还需要注意：

- operator 为操作符，通常可以是 <、=<、>、>= 中的任何一个。具体是否支持某个操作符，取决于表达式值的类型[⊖]。

⊖ 不同的数据库实现，可能为不同的数据类型实现了不完全相同的操作，所以，所支持的操作符不一定完全相同。

- = ANY 与 IN 含义相同，可以采用 IN 子查询优化方法。
- SOME 与 ANY 含义相同。
- NOT IN 与 <>ALL 含义相同。
- NOT IN 与 <>ANY 含义不相同。
- <>ANY 表示不等于任何值。

对于 ALL/ANY/SOME 类子查询的优化，如果子查询中没有 GROUPBY 子句，也没有聚集函数，则下面的表达式还可以使用聚集函数 MAX/MIN 做类似下面的等价转换：

- val>ALL (SELECT...) 等价变化为 val>MAX (SELECT...)
- val<ALL (SELECT...) 等价变化为 val <MIN (SELECT...)
- val>ANY (SELECT...) 等价变化为 val>MIN (SELECT...)
- val<ANY (SELECT...) 等价变化为 val <MAX (SELECT...)
- val>=ALL (SELECT...) 等价变化为 val >=MAX (SELECT...)
- val<=ALL (SELECT...) 等价变化为 val <=MIN (SELECT...)
- val>=ANY (SELECT...) 等价变化为 val>=MIN (SELECT...)
- val<=ANY (SELECT...) 等价变化为 val <=MAX (SELECT...)

具体变换的形式如下：

```
val>ANY (SELECT item ...) 等价变化为 val>SELECT MIN(item)...
```

（3）EXISTS 类型

EXISTS 类型的子查询格式如下：

```
[NOT] EXISTS (subquery)
```

这里有以下几点需要注意：

- EXISTS 对于子查询而言，其结果值是布尔值；如果 subquery 有返回值，则整个 EXISTS (subquery) 的值为 TRUE，否则为 FALSE。
- EXISTS (subquery) 不关心 subquery 返回的内容，这使得带有 EXISTS (subquery) 的子查询存在优化的可能。
- EXISTS (subquery) 自身有着“半连接”的语义，所以，一些数据库实现代码中（如 PostgreSQL），用半连接完成 EXISTS (subquery) 求值。
- NOT EXISTS (subquery) 通常会被标识为“反半连接”处理。
- 一些诸如 IN (subquery) 的子查询可以等价转换为 EXISTS (subquery) 格式，所以可以看到 IN (subquery) 的子查询可被优化为半连接实现的表连接。

下面通过一个示例来进行说明。

示例 3 沿用示例 2 的表和数据。

语句四 EXISTS 类型的普通相关子查询，子查询条件和父查询有关联：

```
SELECT t_1.* FROM t_1 WHERE EXISTS (SELECT t_2_id FROM t_2 WHERE t_1_id=t_2_id);
```

语句五　EXISTS 类型的普通相关子查询，子查询条件和子查询没有关系：

```
SELECT t_1.* FROM t_1 WHERE EXISTS (SELECT t_2_id FROM t_2 WHERE t_1_id=10);
```

语句六　EXISTS 类型的普通非相关子查询：

```
SELECT t_1.* FROM t_1 WHERE EXISTS (SELECT t_2_id FROM t_2 WHERE t_2_id=10);
```

语句七　EXISTS 类型的普通非相关子查询，子查询简单没有表存在：

```
SELECT t_1.* FROM t_1 WHERE EXISTS (SELECT 10);
```

对于语句四，EXISTS 类型的普通相关子查询的查询执行计划如下：

```
test=# EXPLAIN SELECT t_1.* FROM t_1 WHERE EXISTS (SELECT t_2_id FROM t_2 WHERE t_1_id=t_2_id);
                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Semi Join  (cost=45.33..94.58 rows=1570 width=22)
    Hash Cond: (t_1.t_1_id = t_2.t_2_id)
    ->Seq Scan on t_1  (cost=0.00..25.70 rows=1570 width=22)
    ->Hash  (cost=25.70..25.70 rows=1570 width=4)
        ->Seq Scan on t_2  (cost=0.00..25.70 rows=1570 width=4)
(5 rows))
```

查询优化器对查询做了优化，通过子查询上拉技术，把子查询转换为使用 t_1.t_1_id = t_2.t_2_id 作为连接条件实现 Hash 半连接（Hash Semi Join）操作。如果不做优化，则 t_1 表有多少条记录，都需要扫描 t_2 表多少次，每次都得对 t_2 表做全表顺序扫描，这样会花费较多时间；而优化后，对 t_2 表只做了一次全表顺序扫描，然后采用 Hash Semi Join 算法，对 t_1 和 t_2 做连接操作即可，节约了很多次对 t_2 表的全表扫描，达到了优化的目的。

对于语句五，EXISTS 类型的普通相关子查询的查询执行计划如下：

```
test=# EXPLAIN SELECT t_1.* FROM t_1 WHERE EXISTS (SELECT t_2_id FROM t_2 WHERE t_1_id=10);
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Nested Loop Semi Join  (cost=0.00..33.99 rows=1 width=22)
    ->Index Scan using t_1_t_1_id_key on t_1  (cost=0.00..8.27 rows=1 width=22)
        Index Cond: (t_1_id = 10)
    ->Seq Scan on t_2  (cost=0.00..25.70 rows=1570 width=0)
(4 rows)
```

查询优化器能对查询做优化。通过子查询上拉技术，查询执行计划对子查询 t_2 做一个顺序扫描，然后与做顺序扫描的 t_1 表扫描的结果进行嵌套循环半连接（Nested Loop Semi Join）。

对于语句六，EXISTS 类型的普通非相关子查询的查询执行计划如下：

```
test=# EXPLAIN SELECT t_1.* FROM t_1 WHERE EXISTS (SELECT t_2_id FROM t_2 WHERE t_2_id=10);
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Result  (cost=8.27..33.97 rows=1570 width=22)
```

```
    One-Time Filter: $0
    InitPlan 1 (returns $0)
      ->Index Scan using t_2_t_2_id_key on t_2  (cost=0.00..8.27 rows=1 width=0)
            Index Cond: (t_2_id = 10)
    ->Seq Scan on t_1  (cost=0.00..25.70 rows=1570 width=22)
(6 rows)
```

子查询是非相关子查询，子查询只要执行一次即可推知 EXISTS 的结果是 TRUE 或 FALSE，所以不会执行多次，也没有必要进行优化，所以查询执行计划中存在 One-Time，表示只执行一次。

对于语句七，EXISTS 类型的非相关子查询的查询执行计划如下：

```
test=# EXPLAIN SELECT t_1.* FROM t_1 WHERE EXISTS (SELECT 10);
                              QUERY PLAN
-----------------------------------------------------------------------------------
 Result  (cost=0.01..25.71 rows=1570 width=22)
   One-Time Filter: $0
   InitPlan 1 (returns $0)
     ->Result  (cost=0.00..0.01 rows=1 width=0)
   ->Seq Scan on t_1  (cost=0.00..25.70 rows=1570 width=22)
(5 rows)
```

子查询比语句三更为简单，道理同对语句六的分析。查询执行计划中存在 One-Time，表示只执行一次。

2.2.2 视图重写

视图是数据库中基于表的一种对象，视图重写就是将对视图的引用重写为对基本表的引用。视图重写后的 SQL 多被作为子查询进行进一步优化。所有的视图都可以被子查询替换，但不是所有的子查询都可以用视图替换。这是因为，子查询的结果作为一个结果集，如果是单行单列（标量），则可以出现在查询语句的目标列；如果是多行多列，可以出现在 FROM、WHERE 等子句中。但即使是标量视图（视图等同于表对象），也不可以作为目标列单独出现在查询语句中。

从视图的构成形式看，类似于查询的 SPJ 与非 SPJ，视图可以分为简单视图和复杂视图。

- ❑ 用 SPJ 格式构造的视图，称为简单视图。
- ❑ 用非 SPJ 格式构造（带有 GROUPBY 等操作）的视图，称为复杂视图。

下面通过一个例子来具体看一下视图重写。SQL 语句如下：

```
CREATE TABLE t_a(a INT, b INT);
CREATE VIEW v_a AS SELECT * FROM t_a;
```

基于视图的查询命令如下：

```
SELECT col_a FROM v_a WHERE col_b>100;
```

经过视图重写后可变换为如下形式：

```
SELECT col_a FROM
(
    SELECT col_a, col_b FROM t_a
)
WHERE col_b>100;
```

未来经过优化，可以变换为如下等价形式：

```
SELECT col_a FROM t_a WHERE col_b>100;
```

简单视图能够被查询优化器较好地处理；但是复杂视图则不能被查询优化器很好地处理。一些商业数据库，如 Oracle，提供了一些视图的优化技术，如“复杂视图合并”、“物化视图查询重写”等。但从整体上看，复杂视图优化技术还有待继续提高。

2.2.3　等价谓词重写

数据库执行引擎对一些谓词处理的效率要高于其他谓词，基于这点，把逻辑表达式重写成等价的且效率更高的形式，能有效提高查询执行效率。这就是等价谓词重写。

常见的等价谓词重写规则如下。

1. LIKE 规则

LIKE 谓词是 SQL 标准支持的一种模式匹配比较操作，LIKE 规则是对 LIKE 谓词的等价重写，即改写 LIKE 谓词为其他等价的谓词，以更好地利用索引进行优化。如列名为 name 的 LIKE 操作示例如下：

```
name LIKE 'Abc%'
```

重写为：

```
name>='Abc' AND name <'Abd'
```

应用 LIKE 规则的好处是：转换前针对 LIKE 谓词只能进行全表扫描，如果 name 列上存在索引，则转换后可以进行索引范围扫描。

LIKE 其他形式还可以转换：LIKE 匹配的表达式中，若没有通配符（% 或 _），则与 = 等价。如：

```
name LIKE 'Abc'
```

重写为：

```
name='Abc'
```

2. BETWEEN-AND 规则

BETWEEN-AND 谓词是 SQL 标准支持的一种范围比较操作，BETWEEN-AND 规则是指 BETWEEN-AND 谓词的等价重写，即改写 BETWEEN-AND 谓词为其他等价的谓词，以更好地利用索引进行优化。BETWEEN-AND 谓词的等价重写类似于 LIKE 谓词的等价重写，如：

```
sno BETWEEN 10 AND 20
```

重写为：

```
sno>=10 AND sno <=20
```

应用 BETWEEN-AND 规则的好处是：如果 sno 上建立了索引，则可以用索引扫描代替原来 BETWEEN-AND 谓词限定的全表扫描，从而提高了查询的效率。

3. IN 转换 OR 规则

IN 是只 IN 操作符操作，不是 IN 子查询。IN 转换 OR 规则就是 IN 谓词的 OR 等价重写，即改写 IN 谓词为等价的 OR 谓词，以更好地利用索引进行优化。将 IN 谓词等价重写为若干个 OR 谓词，可能会提高执行效率。如：

```
age IN (8,12,21)
```

重写为：

```
age=8 OR age=12 OR age=21
```

应用 IN 转换 OR 规则后效率是否能够提高，需要看数据库对 IN 谓词是否只支持全表扫描。如果数据库对 IN 谓词只支持全表扫描且 OR 谓词中表的 age 列上存在索引，则转换后查询效率会提高。

4. IN 转换 ANY 规则

IN 转换 ANY 规则就是 IN 谓词的 ANY 等价重写，即改写 IN 谓词为等价的 ANY 谓词。因为 IN 可以转换为 OR，OR 可以转为 ANY，所以可以直接把 IN 转换为 ANY。将 IN 谓词等价重写为 ANY 谓词，可能会提高执行效率。如：

```
age IN (8,12,21)
```

重写为：

```
age ANY(8,12,21)
```

应用 IN 转换 ANY 规则后效率是否能够提高，依赖于数据库对于 ANY 操作的支持情况。如，PostgreSQL 没有显式支持 ANY 操作，但是在内部实现时把 IN 操作转换为了 ANY 操作，如下所示：

```
test=# \d t1;
   资料表 "public.t1"
 栏位 | 型别  | 修饰词
--------+----------+----------
 id1    | integer | 非空
 a1     | integer |
 b1     | integer |
索引:
    "t1_pkey" PRIMARY KEY, btree (id1)
```

```
test=# EXPLAIN SELECT * FROM t1 WHERE a1 IN (1,3,5);
                        QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..192.50 rows=3 width=12)
   Filter: (a1 = ANY ('{1,3,5}'::integer[]))
(2 行记录 )
```

5. OR 转换 ANY 规则

OR 转换 ANY 规则就是 OR 谓词的 ANY 等价重写，即改写 OR 谓词为等价的 ANY 谓词，以更好地利用 MIN/MAX 操作进行优化。如：

```
sal>1000 OR dno=3 AND (sal>1100 OR sal>base_sal+100) OR sal>base_sal+200 OR sal>base_sal*2
```

重写为：

```
dno=3 AND (sal>1100 OR sal>base_sal+100) OR sal>ANY (1000, base_sal+200, base_sal*2)
```

OR 转换 ANY 规则依赖于数据库对于 ANY 操作的支持情况。PostgreSQL V9.2.3 和 MySQLV5.6.10 目前都不支持本条规则。

6. ALL/ANY 转换集函数规则

ALL/ANY 转换集函数规则就是将 ALL/ANY 谓词改写为等价的聚集函数 MIN/MAX 谓词操作，以更好地利用 MIN/MAX 操作进行优化。如：

```
sno>ANY(10, 2*5+3, sqrt(9))
```

重写为：

```
sno>sqrt(9)
```

上面这个 ALL/ANY 转换集函数规则的示例，有以下两点需要注意：

- ❏ 本示例中存在 > 和 ANY，其意是在找出 (10, 2*5+3,sqrt(9)) 中的最小值，所以可以重写为 sno>sqrt(9)。
- ❏ 通常，聚集函数 MAX()、MIN() 等的执行效率比 ANY、ALL 谓词的执行效率高，因此在这种情况下对其进行重写可以起到比较好的效果。如果有索引存在，求解 MAX/MIN 的效率更高。

7. NOT 规则

NOT 谓词的等价重写。如下：

```
NOT (col_1 !=2) 重写为 col_1=2
NOT (col_1 !=col_2) 重写为 col_1=col_2
NOT (col_1 =col_2) 重写为 col_1!=col_2
NOT (col_1 <col_2) 重写为 col_1>=col_2
NOT (col_1 >col_2) 重写为 col_1<=col_2
```

NOT 规则重写的好处是：如果在 col_1 上建立了索引，则可以用索引扫描代替原来的

全表扫描，从而提高查询的效率。

8. OR 重写并集规则

OR 条件重写为并集操作，形如以下 SQL 示例：

```
SELECT *
FROM student
WHERE(sex='f' AND sno>15) OR age>18;
```

假设所有条件表达式的列上都有索引（即 sex 列和 age 列上都存在索引），数据库可能对于示例中的 WHERE 语句强迫查询优化器使用顺序存取，因为这个语句要检索的是 OR 操作的集合。为了能利用索引处理上面的查询，可以将语句改成如下形式：

```
SELECT *
FROM student
WHERE sex='f' and sno>15
UNION
SELECT *
FROM student
WHERE age>18;
```

改写后的形式，可以分别利用列 sex 和 age 上的索引，进行索引扫描，然后再提供执行 UNION 操作获得最终结果。

2.2.4 条件化简

WHERE、HAVING 和 ON 条件由许多表达式组成，而这些表达式在某些时候彼此之间存在一定的联系。利用等式和不等式的性质，可以将 WHERE、HAVING 和 ON 条件化简，但不同数据库的实现可能不完全相同。

将 WHERE、HAVING 和 ON 条件化简的方式通常包括如下几个：

- **把 HAVING 条件并入 WHERE 条件**。便于统一、集中化解条件子句，节约多次化解时间。但不是任何情况下 HAVING 条件都可以并入 WHERE 条件，只有在 SQL 语句中不存在 GROUPBY 条件或聚集函数的情况下，才能将 HAVING 条件与 WHERE 条件的进行合并。
- **去除表达式中冗余的括号**。这样可以减少语法分析时产生的 AND 和 OR 树的层次。如 ((a AND b) AND (c AND d)) 就可以化简为 a AND b AND c AND d。
- **常量传递**。对不同关系可以使得条件分离后有效实施“选择下推”，从而可以极大地减小中间关系的规模。如 col_1 = col_2 AND col_2 = 3 就可以化简为 col_1=3 AND col_2=3。操作符 =、<、>、<=、>=、<>、LIKE 中的任何一个，在 col_1 < 操作符 >col_2 条件中都会发生常量传递。
- **消除死码**。化简条件，将不必要的条件去除。如 WHERE (0 > 1 AND s1 = 5)，0 > 1 使得 AND 恒为假，则 WHERE 条件恒为假。此时就不必再对该 SQL 语句进行优化和执行了，加快了查询执行的速度。

- **表达式计算**。对可以求解的表达式进行计算，得出结果。如 WHERE col_1 = 1 + 2 变换为 WHERE col_1 = 3。
- **等式变换**。化简条件（如反转关系操作符的操作数的顺序），从而改变某些表的访问路径。如 −a = 3 可化简为 a = −3。这样的好处是如果 a 上有索引，则可以利用索引扫描来加快访问。
- **不等式变换**。化简条件，将不必要的重复条件去除。如 a > 10 AND b = 6 AND a > 2 可化简为 b = 6 AND a > 10。
- **布尔表达式变换**。在上面的内容中，涉及了一些布尔表达式参与的变换（如上一条中的示例是 AND 的表达式）。布尔表达式还有如下规则指导化简。
  - **谓词传递闭包**。一些比较操作符，如 <、> 等，具有传递性，可以起到化简表达式的作用。如由 a>b AND b>2 可以推导出 a>b AND b>2 AND a>2，a>2 是一个隐含条件，这样把 a>2 和 b>2 分别下推到对应的关系上，就可以减少参与比较操作 a>b 的元组了。
  - **任何一个布尔表达式都能被转换为一个等价的合取范式（CNF）**[⊖]。因为合取项只要有一个为假，整个表达式就为假，故代码中可以在发现一个合取项为假时，即停止其他合取项的判断，以加快判断速度。另外因为 AND 操作符是可交换的，所以优化器可以按照先易后难的顺序计算表达式，一旦发现一个合取项为假时，即停止其他合取项的判断，以加快判断速度。
  - **索引的利用**。如果一个合取项上存在索引，则先判断索引是否可用，如能利用索引快速得出合取项的值，则能加快判断速度。同理，OR 表达式中的子项也可以利用索引。

2.2.5　外连接消除

外连接消除是查询优化器的主要功能之一。下面从外连接消除的意义和条件两方面对外连接优化技术进行介绍。

1. 外连接消除的意义

外连接操作可分为左外连接、右外连接和全外连接。连接过程中，外连接的左右子树不能互换，并且外连接与其他连接交换连接顺序时，必须满足严格的条件以进行等价变换。这种性质限制了优化器在选择连接顺序时能够考虑的表与表交换连接位置的优化方式。

查询重写的一项技术就是把外连接转换为内连接，**转换的意义（对优化的意义）如下：**

- 查询优化器在处理外连接操作时所需执行的操作和时间多于内连接。
- 优化器在选择表连接顺序时，可以有更多更灵活的选择，从而可以选择更好的表连接顺序，加快查询执行的速度。

⊖ 合取范式格式为：C1 AND C2 AND⋯ AND Cn；其中，Ck（1<=k<=n）称为合取项，每个合取项是不包含 AND 的布尔表达式。

- 表的一些连接算法（如块嵌套连接和索引循环连接等）将规模小的或筛选条件最严格的表作为“外表”（放在连接顺序的最前面，是多层循环体的外循环层），可以减少不必要的 IO 开销，极大地加快算法执行的速度。

为什么外连接可以转换为内连接？为了弄明白其中的原因，我们先来看一下表 2-8（借助 PostgreSQL 对外连接的注释）。

表 2-8 PostgreSQL 外连接注释表

类型	PG 代码表示方式	R <op> S 的结果＝A＋B＋C			说明
		A	B	C	
θ- 连接	JOIN_INNER	pairs[⊖]	—	—	LHS 表示左部分的关系 R，RHS 表示右部分的关系 S
左外连接	JOIN_LEFT	pairs	unmatched LHS tuples	—	
全外连接	JOIN_FULL	pairs	unmatched LHS	unmatched RHS tuples	
右外连接	JOIN_RIGHT	pairs	—	unmatched RHS tuples	

下面我们分 3 种情况讨论表 2-8。

情况一：左外连接向内连接转换。

- 观察表 2-8，先看 θ - 连接和左外连接，相同之处在于都具有 A 部分，不同之处表现在 B 部分有差异。左外连接比 θ - 连接多了 B 部分。这表明左外连接的结果中，会比 θ- 连接多出“不满足连接条件的外表中的元组（unmatched LHS tuples）”。
- 假设一个左外连接执行之后，其结果等同于内连接，即 B 部分不存在，则这种情况下，左外连接就可以向内连接转换。这种转换是有条件的，条件是：右面关系中对应的条件，保证最后的结果集中不会出现 B 部分这样特殊的元组（这样的条件称为“reject-NULL 条件”）。
- 不是所有的左外连接都可以向内连接转换，需要满足一定的条件才可以是等价转换。
- 左外连接向内连接转换，语义满足左外连接，但实际结果因两个表中数据的有特点（满足 reject-NULL 条件）。形式上看是外连接，其实质成为一种褪化的内连接。

情况二：全外连接向左外连接转换。

- 观察表 2-8，先看全外连接和左外连接，相同之处在于都具有 A、B 部分，不同之处表现在 C 部分有差异。全外连接比左外连接多了 C 部分。
- 假设一个全外连接执行之后，其结果等同于左外连接，即 C 部分不存在，则这种情况下，全外连接就可以向左外连接转换。这种转换是有条件的，条件是：左面关系中对应的条件，保证最后的结果集中不会出现 C 部分这样特殊的元组。
- 不是所有的全外连接都可以向左外连接转换，需要满足一定条件才可以是等价转换。
- 全外连接向左连接转换，形式满足全外连接，但实际结果因两个表中数据的特点，其实质成为一种褪化的左外连接。
- 全外连接向右外连接转换，基本道理等同向左外连接转换，不赘述。

⊖ matching tuple pairs only（只匹配元组对）。

- ❏ 全外连接如果能同时向左外连接和右外连接转换，则意味着全外连接能转换为内连接。其条件是：左面和右面关系中对应的条件，均能满足 reject-NULL 条件。

情况三：右外连接向内连接转换。

- ❏ 在实际处理中，因右外连接对称等同左外连接，所以通常都是把右外连接转换为左外连接，然后再向内连接转换。
- ❏ 右外连接向左外连接转换，通常是发生在语法分析阶段（这是一种外表样式的转换，但不是全部），在进入查询优化阶段后，才对左外连接和全外连接进行等价转换（进入优化器后执行的这个转换，需要对连接的列和查询的列做判断，所以语法分析阶段没有得到列的详细信息是不能进行此优化的）。

下面我们通过一个例子，来帮助读者对上述知识加深理解。

示例 4　具体的示例代码如下：

```
SELECT * FROM t_1 LEFT JOIN t_2 ON t_2.a=t_1.a WHERE t_2.b = 5;
```

由上述示例代码可知，左外连接的内表（t_2）不能先于外表（t_1）被访问，所以查询计划优化阶段，只能将表的连接顺序定为 (t_1,t_2)，而不是 (t_2,t_1)。

如果 t_1 表的元组数很大，t_2.b 上建有唯一索引，但查询效率会很低。但是，如果能交换 t_1 和 t_2 的表连接顺序，即表的连接顺序为 (t_2,t_1)，则可以利用 t_2.b = 5 条件迅速降低运算的中间规模，提高查询的速度。

2. 外连接消除的条件

外连接可转换为内连接的条件：WHERE 子句中与内表相关的条件满足“空值拒绝”（reject-NULL 条件）。那么条件怎么才算是满足空值拒绝呢？一般认为满足下面任意一种情况时，即满足空值拒绝：

- ❏ 条件可以保证从结果中排除外连接右侧（右表）生成的值为 NULL 的行（即条件确保应用在右表带有空值的列对象上时，条件不满足，条件的结果值为 FLASE 或 UNKOWEN，这样右表就不会有值为 NULL 的行生成），所以能使该查询在语义上等效于内连接。
- ❏ 外连接的提供空值的一侧（可能是左侧的外表也可能是右侧的内表）为另一侧的每行只返回一行。如果该条件为真，则不存在提供空值的行，并且外连接等价于内连接。

示例 5　以 PostgreSQL 对外连接的优化为例，初始数据如下：

```
create table X (X_num int, X_name varchar(10));
create table Y (Y_num int, Y_name varchar(10));
insert into X values(1, 'X_1');
insert into X values(2, 'X_2');
insert into X values(NULL, 'X_3');
insert into Y values(1, 'Y_1');
insert into Y values(NULL, 'Y_2');
insert into Y values(3, 'Y_3');
```

例如，如下 3 种外连接查询语句，外连接处理方式不同。

语句一 表 X 和 Y 执行简单左外连接：

```
SELECT * FROM X LEFT JOIN Y on (X.X_num=Y.Y_num);
```

语句二 表 X 和 Y 执行简单左外连接，但带有 WHERE 条件且内表 Y 的连接条件列非空：

```
SELECT * FROM X LEFT JOIN Y ON (X.X_num=Y.Y_num) WHERE Y.Y_num IS NOT NULL;
```

语句三 表 X 和 Y 执行内连接，带有 WHERE 条件：

```
SELECT * FROM X, Y WHERE X.X_num=Y.Y_num;
```

通过查询计划看优化结果，具体如下：

对于语句一，左外连接不满足优化条件，没有被优化。查询执行计划如下：

```
test=# EXPLAIN SELECT * FROM X LEFT JOIN Y on (X.X_num=Y.Y_num);
                                QUERY PLAN
-------------------------------------------------------------------------
Merge Left Join  (cost=236.43..461.68 rows=14450 width=36)// 左外连接
    Merge Cond: (x.x_num = y.y_num)
    ->Sort  (cost=118.22..122.47 rows=1700 width=18)
        Sort Key: x.x_num
        ->Seq Scan on x  (cost=0.00..27.00 rows=1700 width=18)
    ->Sort  (cost=118.22..122.47 rows=1700 width=18)
        Sort Key: y.y_num
        ->Seq Scan on y  (cost=0.00..27.00 rows=1700 width=18)
(8 rows)
```

查询语句中只有连接条件 X.X_num=Y.Y_num，没有 WHERE 条件，所以不满足空值拒绝中的任何一个条件，所以查询优化器没有把左外连接优化为连接操作，执行的是归并左外连接操作（Merge Left Join）。

对于语句二，左外连接被优化为内连接，查询执行计划如下：

```
test=# EXPLAIN SELECT * FROM X LEFT JOIN Y ON (X.X_num=Y.Y_num) WHERE Y.Y_num ISNOT NULL;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
Merge Join  (cost=235.88..459.93 rows=14373 width=36)/* 因为存在 WHERE 子句中内表对应的列满足 “空值拒绝”,
使得左外连接可以被优化为内连接 */
    Merge Cond: (y.y_num = x.x_num)
    ->Sort  (cost=117.67..121.90 rows=1691 width=18)
        Sort Key: y.y_num
        ->Seq Scan on y  (cost=0.00..27.00 rows=1691 width=18)
            Filter: (y_num IS NOT NULL)
    ->Sort  (cost=118.22..122.47 rows=1700 width=18)
        Sort Key: x.x_num
        ->Seq Scan on x  (cost=0.00..27.00 rows=1700 width=18)
(9 rows)
```

查询语句除了连接条件 X.X_num=Y.Y_num 外，包含 WHERE 条件（Y.Y_num IS NOT NULL），满足空值拒绝中的第一个条件，所以查询优化器把左外连接优化为连接操作，执行的是归并连接操作（Merge Join）。

对于语句三，内连接等价于上一种的可被优化的左外连接，查询执行计划如下：

```
test=# explain SELECT * FROM X, Y WHERE X.X_num=Y.Y_num;
                                QUERY PLAN
-------------------------------------------------------------------------------------
Merge Join (cost=236.43..461.68 rows=14450 width=36)// 内连接
    Merge Cond: (x.x_num = y.y_num)
    ->Sort  (cost=118.22..122.47 rows=1700 width=18)
        Sort Key: x.x_num
        ->Seq Scan on x  (cost=0.00..27.00 rows=1700 width=18)
    ->Sort (cost=118.22..122.47 rows=1700 width=18)
        Sort Key: y.y_num
        ->Seq Scan on y  (cost=0.00..27.00 rows=1700 width=18)
(8 rows)
```

普通查询语句连接条件为 X.X_num=Y.Y_num，查询优化器把执行连接操作等价于上一种的可被优化的左外连接，查询执行计划除花费（cost）外完全相同。

最后，可以通过执行上两条 SQL 语句，查看查询结果，验证两者等价。具体结果如下：

```
test=# SELECT * FROM X, Y WHERE X.X_num=Y.Y_num;
 x_num | x_name | y_num | y_name
-------+--------+-------+--------
     1 | X_1    |     1 | Y_1
(1 row)
test=# SELECT * FROM X LEFT JOIN Y on (X.X_num=Y.Y_num) WHERE Y.Y_num IS NOT NULL;
 x_num | x_name | y_num | y_name
-------+--------+-------+--------
     1 | X_1    |     1 | Y_1
(1 row)
```

注意

对于外连接的查询优化，从格式上看用户书写的 SQL 可能是语句一的格式，但这种格式因其他原因，存在被改写为语句二的格式的可能。查询优化前通过对条件的分析，就是要确认是否可以为语句一的格式找到满足语句二的格式中的 WHERE Y.Y_num IS NOT NULL 类似条件，如果能找到这样的条件，则查询优化器可自动对第一种左外连接做优化，转换为语句二的格式。

2.2.6　嵌套连接消除

多表连接有时会存在嵌套的情况。对于一个无嵌套的多表连接，表之间的连接次序是

可以交换的，这样能灵活求解不同连接方式的花费，进而得到最小花费的连接方式。而嵌套连接则不能够利用交换表的位置而获得优化。

那么，什么是嵌套连接呢?

当执行连接操作的次序不是从左到右逐个进行时，就说明这样的连接表达式存在嵌套。看下面的例子：

```
SELECT *
FROM t1 LEFT JOIN (t2 LEFT JOIN t3 ON t2.b=t3.b) ON t1.a=t2.a
WHERE t1.a > 1
```

先 t2 与 t3 连接，得到中间结果 {t2 t3} 后再与 t1 连接，这种方式就是嵌套连接，括号不可以去掉，没有去掉括号的等价形式。如果连接顺序是 t1、t2、t3，则不存在嵌套。

另外，如下格式也是嵌套，这种格式用括号把连接次序做了区分。

```
SELECT * FROM A JOIN (B JOIN C ON B.b1=C.c1) ON A.a1=B.b1 WHERE A.a1 > 1;
```

上面的格式可以等价转换为（圆括号去掉，不影响原先语义）下面的格式：

```
SELECT * FROM A JOIN B JOIN C ON B.b1=C.c1 ON A.a1=B.b1 WHERE A.a1 > 1;
```

综上所述，我们得到以下两条结论：

- ❑ 如果连接表达式只包括内连接，括号可以去掉，这意味着表之间的次序可以交换，这是关系代数中连接的交换律的应用。
- ❑ 如果连接表达式包括外连接，括号不可以去掉，意味着表之间的次序只能按照原语义进行，至多能执行的就是外连接向内连接转换的优化，该部分的内容详见 2.2.5 节。

2.2.7 连接消除

根据不同分类角度，连接可以分成很多种：根据连接语义方式的不同分为内连接、外连接、半连接、反半连接等；根据连接对象的不同分为自连接（FROM 子句后同一个表出现多次，并连接）和非自连接；根据连接条件的有无分为笛卡儿积式的连接和带有限定条件的连接；根据连接条件形式的不同分为等值连接（如支持 =）和范围连接（如支持 >、<、<>）。

有的连接类型可能有特定的优化方式（如外连接可优化）。除了以上可能的连接类型外，在某些特殊的情况下（例如下文的情况一、情况二、情况三），可能存在一些连接，这些连接中的**连接对象可以被去掉（因为这样的连接对象存在只会带来连接计算的耗费，而对连接结果没有影响）**，所以这类连接存在优化的可能，其中的一些连接是可以消除掉的。

下面我们就分情况看看这些连接。

情况一：主外键关系的表进行的连接，可消除主键表，这不会影响对外键表的查询。

例如，创建表定义如下：

```
CREATE TABLE B (b1 INT, b2 VARCHAR(9), PRIMARY KEY(b1));
CREATE TABLE A (a1 INT, a2 VARCHAR(9), FOREIGN KEY(a1) REFERENCES B(b1) );
CREATE TABLE C (c1 INT, c2 VARCHAR(9));
```

对于关系 A、B、C，如果存在 A 参照 B（连接条件上是主外键关系，A 依赖于 B），且三者之间的连接条件是等值连接（A join B join C，连接条件是 A.a1=B.b1 AND B.b1=C.c1），结果集不包含关系 B 中的任何属性，且在 B 上没有任何限定条件，那么 A 的连接条件上添加连接属性不为空的限制后（因为 A.a1=B.b1 连接，而 B.b1 是主键，不为 NULL），可以去除关系 B，这样优化不会影响结果（即优化为 A join C，连接条件变为 A.a1=C.c1 AND A.a1 IS NOT NULL；因为等值连接 A.a1=C.c1，如果遇到 NULL=NULL 的情况，通常的处理都是 FALSE，所以。以上条件可以进一步简化为 A.a1=C.c1，但请注意 A 依赖于 B 的条件不变）。

如果关系 A、B、C 中 B 没有主键（主键保证了非空约束的存在），则当 B.b1=NULL 时，A.a1 和 C.c1 为非 NULL 时，A.a1=C.c1 的连接方式（即消除 B 的连接方式）和 A.a1=B.b1 AND B.b1=C.c1 的连接方式（即没有消除 B 的连接方式）相比，前一种连接方式比后一种产生了新的元组，所以不存在连接可以消除的可能。

再进一步，如果关系 A、B、C 中 B 有主键，但 A 不依赖于 B（无主外键关系），三者之间的连接条件是等值连接（A join B join C，连接条件是 A.a1=B.b1 AND B.b1=C.c1），因为 B 有主键，意味着 B 非 NULL 所以 A.a1 和 C.c1 都应该为非 NULL，才可能保证连接可以消除（连接条件变为 A.a1=C.c1 AND A.a1 IS NOT NULL AND C.c1 IS NOT NULL）。但实际上不是这样，B 和 A 没有依赖关系的时候，单纯靠 A、C 之间的条件，因为去掉 B 使得 A 中存在的值没有受到 B 的限制，所以不能保证 B 可被放心地去掉。在有主外键约束的情况下，A.a1 依赖 B.b1 使得 A.a1 的值范围受到约束（A.a1 的值在生成时已经参照了 B.b1），所以可以放心地去掉 B。

如果对于关系 A、B 存在 A 参照 B（连接条件上是主外键关系，A 依赖于 B），且二者之间的连接条件是等值连接（A join B，连接条件是 A.a1=B.b1），则经过连接消除，二表连接变为单表扫描（A.a1 IS NOT NULL），这样能有效提高 SQL 的执行效率。

情况二：唯一键作为连接条件，三表内连接可以去掉中间表（中间表的列只作为连接条件）。

举例说明，假设 3 个表都有唯一键存在，如下所示：

```
CREATE TABLE A (a1 INT UNIQUE, a2 VARCHAR(9), a3 INT);
CREATE TABLE B (b1 INT UNIQUE, b2 VARCHAR(9), c2 INT);
CREATE TABLE C (c1 INT UNIQUE, c2 VARCHAR(9), c3 INT);
```

B 的列在 WHERE 条件子句中只作为等值连接条件存在，则查询可以去掉对 B 的连接操作。

```
SELECT A.*, C.* FROM A JOIN B ON (a1=b1) JOIN CON (b1=c1);
```

相当于：

```
SELECT A.*, C.* FROM A JOIN C ON (a1= c1);
```

情况三：其他一些特殊形式，可以消除连接操作（可消除的表除了作为连接对象外，不出现在任何子句中，创建表的语句参见情况一）。示例如下：

```
SELECT MAX(a1) FROM A, B;/* 在这样格式中的 MIN、MAX 函数操作可以消除连接，去掉 B 表不影响结果；其他聚集函数不可以 */
SELECT DISTINCT a3 FROM A, B; /* 对连接结果中的 a3 列执行去重操作 */
SELECT a1 FROM A, B GROUP BY a1;/* 对连接结果中的 a1 列执行分组操作 */
```

2.2.8 语义优化

因为语义的原因，使得 SQL 可以被优化。这里包括以下两个基本概念：

- **语义转换**。因为完整性限制等的原因使得一个转换成立的情况称为语义转换。
- **语义优化**。因为语义转换形成的优化称为语义优化。

通常，任何的完整性限制条件等都可以用于语义优化。语义转换其实是根据完整性约束等信息对“某特定语义”进行推理，进而得到的一种查询效率不同但结果相同的查询。语义优化是从语义的角度对 SQL 进行优化，不是一种形式上的优化，所以其优化的范围，可能覆盖其他类型的优化范围。

语义优化[⊖]常见的方式如下：

- **连接消除**（Join Elimination）。对一些连接操作先不必评估代价，根据已知信息（主要依据完整性约束等，但不全是依据完整性约束）能推知结果或得到一个简化的操作。例如，利用 A、B 两个基表做自然连接，创建一个视图 V，如果在视图 V 上执行查询只涉及其中一个基表的信息，则对视图的查询完全可以转化为对某个基表的查询。
- **连接引入**（Join Introduction）。增加连接有助于原关系变小或原关系的选择率降低。
- **谓词引入**（Predicate Introduction）。根据完整性约束等信息引入新谓词，如引入基于索引的列，可能使得查询更快。如一个表上，有 c1<c2 的列约束，c2 列上存在一个索引，查询语句中的 WHERE 条件有 c1>200，则可以推知 c2>200，WHERE 条件变更为 c1>200 AND c2>200 AND c1<c2，由此可以利用 c2 列上的索引，对查询语句进行优化。如果 c2 列上的索引的选择率很低，则优化效果会更高。
- **检测空回答集**（Detecting the Empty Answer Set）。查询语句中的谓词与约束相悖，可以推知条件结果为 FALSE，也许最终的结果集能为空；如 CHECK 约束限定 score 列的范围是 60 到 100，而一个查询条件是 score<60，则能立刻推知条件不成立。
- **排序优化**（Order Optimizer）。ORDERBY 操作通常由索引或排序（sort）完成；如果能够利用索引，则排序操作可省略。另外，结合分组等操作，考虑 ORDERBY 操作的优化。
- **唯一性使用**（Exploiting Uniqueness）。利用唯一性、索引等特点，检查是否存在不必

⊖ 语义优化有别于其他类型的优化，这种优化方式从语义的角度考虑，被优化对象可能涉及其他优化技术中的内容，如视图重写、ORDERBY 优化等。这在一定程度上有重叠，但是思维方式不同。

要的 DISTINCT 操作，如在主键上执行 DISTINCT 操作，若有则可以把 DISTINCT 消除掉。

例如：

```
CREATE TABLE student (name VARCHAR(30) NOT NULL, age INT);
```

可被优化为如下形式：

```
SELECT * FROM student WHERE name IS NULL AND age>18;
```

在上面的例子中，name 列被定义为非空，这是完整性限制，但查询条件和语义相悖，则可以直接返回 name IS NULL=false，进而 false AND age>18=false，查询语句可以终止。

2.2.9　针对非 SPJ 的优化

如果查询中包含 GROUPBY 子句，那么这种查询就称为非 SPJ 查询。

现在，决策支持系统、数据仓库、OLAP 系统的应用日益广泛，SQL 语句中的 GROUPBY、聚集函数、WINDOWS 函数（分析函数）等成为 SQL 语言的重要特性，被应用广泛。

早期的关系数据库系如 System-R，对 GROUPBY 和聚集等操作一般都放在其所在的查询的最后进行处理，即在执行完所有的连接和选择操作之后再执行 GROUPBY 和聚集。这样的处理方式比较简单，而且编码容易，但执行效率会比较低。所以，现代的商业数据库和开源的数据库，通常都使用了一些非 SPJ 的优化技术。

1. GROUPBY 优化

对于 GROUPBY 的优化，可考虑**分组转换技术**，即对分组操作、聚集操作与连接操作的位置进行交换。常见的方式如下：

- **分组操作下移**。GROUPBY 操作可能较大幅度地减少关系元组的个数，如果能够对某个关系先进行分组操作，然后再进行表之间的连接，很可能提高连接效率。这种优化方式是把分组操作提前执行。下移的含义，是在查询树上让分组操作尽量靠近叶子结点，使得分组操作的结点低于一些选择操作。
- **分组操作上移**。如果连接操作能够过滤掉大部分元组，则先进行连接后进行 GROUPBY 操作，可能提高分组操作的效率。这种优化方式是把分组操作置后执行。上移的含义和下移正好相反。

对于带有 GROUPBY 等操作的非 SPJ 格式的 SQL 语句，在本节之前提及的技术都适用，只是结合了 GROUPBY 操作的语义进行分组操作。因为 GROUPBY 操作下移或上移均不能保证重写后的查询效率一定更好，所以，要在查询优化器中采用基于代价的方式来估算某几种路径的优劣。

另外，GROUPBY、ORDERBY 优化的另外一个思路是尽量利用索引，这部分内容将在 3.3 节详细讨论。

2. ORDERBY 优化

对于 ORDERBY 的优化，可有如下方面的考虑：

- **排序消除**（Order By Elimination，OBYE）。优化器在生成执行计划前，将语句中没有必要的排序操作消除（如利用索引），避免在执行计划中出现排序操作或由排序导致的操作（如在索引列上排序，可以利用索引消除排序操作）。
- **排序下推**（Sort push down）。把排序操作尽量下推到基表中，有序的基表进行连接后的结果符合排序的语义，这样能避免在最终的大的连接结果集上执行排序操作。

3. DISTICT 优化

对于 DISTICT 的优化，可有如下方面的考虑：

- **DISTINCT 消除**（Distinct Elimination）。如果表中存在主键、唯一约束、索引等，则可以消除查询语句中的 DISTINCT（这种优化方式，在语义优化中也涉及，本质上是语义优化研究的范畴）。
- **DISTINCT 推入**（Distinct Push Down）。生成含 DISTINCT 的反半连接查询执行计划时，先进行反半连接再进行 DISTICT 操作，也许先执行 DISTICT 操作再执行反半连接更优，这是利用连接语义上确保唯一功能特性进行 DISTINCT 的优化。
- **DISTINCT 迁移**（Distinct Placement）。对连接操作的结果执行 DISTINCT，可能把 DISTINCT 移到一个子查询中优先进行（有的书籍把这项技术称为“DISTINCT 配置”）。

2.3 启发式规则在逻辑优化阶段的应用

逻辑优化阶段使用的启发式规则通常包括如下两类。

- 一定能带来优化效果的，主要包括：
  - 优先做选择和投影（连接条件在查询树上下推）。
  - 子查询的消除。
  - 嵌套连接的消除。
  - 外连接的消除。
  - 连接的消除。
  - 使用等价谓词重写对条件化简。
  - 语义优化。
  - 剪掉冗余操作（一些剪枝优化技术）、最小化查询块。
- 变换未必会带来性能的提高，需根据代价选择，主要包括：
  - 分组的合并。
  - 借用索引优化分组、排序、DISTINCT 等操作。
  - 对视图的查询变为基于表的查询。

- 连接条件的下推。
- 分组的下推。
- 连接提取公共表达式。
- 谓词的上拉。
- 用连接取代集合操作。
- 用 UNIONALL 取代 OR 操作。

2.4　本章小结

本章从逻辑查询优化的基础关系代数讲起，首先简略概述了关系代数的基础知识，这部分不侧重关系代数原理的推导和证明，重点在于描绘关系代数的主要概念，分析关系代数的原理对查询优化的指导意义，为查询优化技术做铺垫；然后以 PostgresSQL 为示例，全面分析了查询优化可用的技术，这是本章的重点；最后对查询优化技术做了一个简单总结，从整体上提升对查询优化的理解和认识。

第 3 章

物理查询优化

查询优化器，在物理优化阶段，主要解决的问题如下：

- 从可选的单表扫描方式中，挑选什么样的单表扫描方式是最优的？
- 对于两个表连接时，如何连接是最优的？
- 对于多个表连接，连接顺序有多种组合，哪种连接顺序是最优的？
- 对于多个表连接，连接顺序有多种组合，是否要对每种组合都探索？如果不全部探索，怎么找到最优的一种组合？

在查询优化器实现的早期，使用的是逻辑优化技术，即使用关系代数规则和启发式规则对查询进行优化后，认为生成的执行计划就是最优的。

在引入了基于代价的查询优化方式后，对查询执行计划做了定量的分析，对每一个可能的执行方式进行评估，挑出代价最小的作为最优的计划。

目前，数据库的查询优化器通常融合了这两种方式。

3.1 查询代价估算

查询代价估算的重点是代价估算模型，这是物理查询优化的依据。除了代价模型外，选择率也是很重要的一个概念，对代价求解起着重要作用。

3.1.1 代价模型

查询代价估算基于 CPU 代价和 IO 代价，所以代价模型可以用以下计算公式表示：

$$\text{总代价} = \text{IO 代价} + \text{CPU 代价}$$

$$\text{COST} = P*a\_page\_cpu\_time + W * T$$

在上面的两个公式中：

- P 为计划运行时访问的页面数，a_page_cpu_time 是每个页面读取的时间花费，其乘积反映了 IO 花费。

- T 为访问的元组数，反映了 CPU 花费（存储层是以页面为单位，数据以页面的形式被读入内存，每个页面上可能有多条元组，访问元组需要解析元组结构，才能把元组上的字段读出，这消耗的是 CPU）。如果是索引扫描，则还会包括索引读取的花费。
- W 为权重因子，表明 IO 到 CPU 的相关性，又称选择率（selectivity）[⊖]。选择率用于表示在关系 R 中，满足条件“A <op> a”的元组数与 R 的所有元组数 N 的比值。

3.1.2　选择率计算的常用方法

选择率在代价估算模型中占有重要地位，其精确程度直接影响最优计划的选取。选择率计算的常用方法如下：

- **无参数方法**（Non-Parametric Method）。使用 ad hoc 数据结构或直方图维护属性值的分布，最常用的是直方图方法。
- **参数法**（Parametric Method）。使用具有一些自由统计参数（参数是预先估计出来的）的数学分布函数逼近真实分布。
- **曲线拟合法**（Curve Fitting）。为克服参数法的不灵活性，用一般多项式和标准最小方差来逼近属性值的分布。
- **抽样法**（sampling）。从数据库中抽取部分样本元组，针对这些样本进行查询，然后收集统计数据，只有足够的样本被测试之后，才能达到预期的精度。
- **综合法**。将以上几种方法结合起来，如抽样法和直方图法结合。

3.2　单表扫描算法

单表扫描需要从表上获取元组，直接关联到物理 IO 的读取，所以不同的单表扫描方式，有不同的代价。

3.2.1　常用的单表扫描算法

单表扫描是完成表连接的基础。对于单表数据的获取，通常有如下方式：

- **全表扫描表数据**。为获取表的全部元组，读取表对应的全部数据页。
- **局部扫描表数据**。为获取表的部分元组，读取指定位置对应的数据页。

对于全表扫描，通常采取顺序读取的算法。为了提高表扫描的效率，有很多算法和优化方式被提出来。**单表扫描和 IO 操作密切相关，所以很多算法在 IO 上倾注精力**。

⊖ 在 System R 算法中，每个可能的选择率固化在代价估计的代码中，在数据字典中存储代价公式和需要的统计信息。PostgresSQL 选择率的处理不同于 System R，每个选择率公式使用一个参数化的函数来表示，选择率计算通过将限制条件中的关系、属性、常量信息和索引标识符以及索引键数目等参数传递给相应的函数来计算。对于 MySQL，在利用索引对表进行分类得出常量表和各种依据索引进行数据的访问方式后，选择率的计算数据源自统计分析模块，这些数据一部分来自 MySQL，一部分来自存储引擎（查询优化器是在 MySQL 层面实现的，而存储引擎有多个，所以一部分统计数据源自各种不同的存储引擎）。

常用的单表扫描算法如下：

- ❑ **顺序扫描**（SeqScan）。从物理存储上按照存储顺序直接读取表的数据；当无索引可用，或访问表中的大部分数据，或表的数据量很小时，使用顺序扫描效果较好。
- ❑ **索引扫描**（IndexScan）。根据索引键读索引，找出物理元组的位置；根据从索引中找到的位置，从存储中读取数据页面；索引扫描可以将元组按排序的顺序返回；索引扫描时若选择率较低，则读数据花费的 IO 会显著减少；换句话说，如果选择率很高，不适宜使用索引扫描。
- ❑ **只读索引扫描**（IndexOnlyScan）。根据索引键读索引，索引中的数据能够满足条件判断，不需要读取数据页面；比索引扫描少了读取数据的 IO 花费。
- ❑ **行扫描**（RowIdScan）。用于直接定位表中的某一行。对于元组，通常为元组增加特殊的列，可以通过特殊的列计算出元组的物理位置，然后直接读取元组对应的页面，获取元组；在 PostgreSQL 中称为 Tid 扫描，此种方式是在元组头上增加名为 CTID 的列，用这列的值可以直接计算本条元组的物理存储位置。
- ❑ **并行表扫描**（ParallelTableScan）。对同一个表，并行地、通过顺序的方式获取表的数据，结果是得到一个完整的表数据。
- ❑ **并行索引扫描**（ParallelIndexScan）。对同一个表，并行地、通过索引的方式获取表的数据，将结果合并在一起。
- ❑ **组合多个索引扫描**。有的系统称为 MultipleIndexScan。对同一个元组的组合条件（AND 或者 OR 谓词组合的多个索引列上的多条件查询）进行多次索引扫描，然后在内存里组织一个位图，用位图描述索引扫描结果中符合索引条件的元组位置。组合多个索引（包括同一索引的多次使用）来处理单个索引扫描不能实现的情况。本质上不是单表的扫描方式，是构建在单表的多个索引扫描基础上的。

对于局部扫描，根据数据量的情况和元组获取条件，可能采用顺序读取或随机读取存储系统的方式。选择率在这种情况下会起一定作用。如果选择率的值很大，意味着采取顺序扫描方式可能比局部扫描的随机读的方式效率更高，这是因为顺序 IO 会减少磁盘头移动的等待时间，如果数据库文件在磁盘上没有碎片，这种扫描方式对性能的改善将更为明显。对于大表，顺序扫描会一次读取多个页，这将进一步降低顺序表扫描的开销。

对于局部扫描，通常采用索引，实现少量数据的读取优化。这是一种随机读取数据的方式。虽然顺序表扫描可能会比读取许多行的索引扫描花费的时间少，但如果顺序扫描被执行多次，且不能有效地利用缓存，则总体花费巨大。索引扫描访问的页面可能较少，而且这些页很可能会保存在数据缓冲区，访问的速度会更快。所以，对于重复的表访问（如嵌套循环连接的右表），采用索引扫描比较好。究竟采取哪种扫描方式，查询优化器在采用代价估算比较代价的大小后才决定。

有的系统对于随机读采取了优化措施，即把要读取的数据的物理位置排序，然后一批读入，保障了磁盘单向一次扫描即可获取一批数据，提高了 IO 效率。

在并行操作的时候，可能因不同隔离级别的要求，需要解决数据一致性的问题。如可

串行化的处理，需要在表级加锁或者表的所有元组上加锁，这是因为索引扫描只在满足条件的元组上加锁，所以索引扫描在多用户环境中可能会比顺序扫描效率高。查询优化器在此种情况下倾向于选择索引扫描，这是一条启发式优化规则。

3.2.2　单表扫描代价计算

因单表扫描需要把数据从存储系统中调入内存，所以单表扫描的代价需要考虑 IO 的花费。顺序扫描，主要是 IO 花费加上元组从页面中解析的花费；索引扫描和其他方式的扫描，由于元组数不是全部元组，需要考虑选择率的问题。单表扫描操作的代价估算公式如表 3-1 所示。

表 3-1　单表扫描算法代价表

扫描方式	代价估算公式
顺序扫描	N_page * a_tuple_IO_time + N_tuple * a_tuple_CPU_time
索引扫描	C_index + N_page_index * a_tuple_IO_time

说明如下：

- a_page_IO_time，一个页面的 IO 花费。
- N_page，数据页面数。
- N_page_index，索引页面数。
- a_tuple_CPU_time，一个元组从页面中解析的 CPU 花费。
- N_tuple，元组数。
- C_index，索引的 IO 花费，C_index = N_page_index × a_page_IO_time。
- N_tuple_index，索引作用下的可用元组数，N_tuple_index =N_tuple × 索引选择率。

3.3　索引

索引是建立在表上的，本质上是通过索引直接定位表的物理元组，加快数据获取的方式，所以索引优化的手段应该归属到物理查询优化阶段。

3.3.1　如何利用索引

索引是提高查询效率的有效手段。如果某个列上存在索引，并不意味着索引能够有效使用。通常查询优化器使用索引的原则如下：

- 索引列作为条件出现在 WHERE、HAVING、ON 子句中，这样有利于利用索引过滤元组。
- 索引列是被连接的表（内表）对象的列且存在于连接条件中。

除了上述的两种情况外，还有一些特殊情况可以使用索引，如排序操作、在索引列上求 MIN、MAX 值等。

示例如下。首先创建表：

```
test=# CREATE TABLE A (a1 INT UNIQUE, a2 VARCHAR(9), a3 INT);
注意：CREATE TABLE / UNIQUE 将要为表 "a" 创建隐含索引 "a_a1_key"
```

对表做查询，没有列对象作为过滤条件（如出现在 WHERE 子句中），只能顺序扫描，例如：

```
test=# EXPLAIN SELECT A.* FROM A;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on a  (cost=0.00..21.00 rows=1100 width=44)
(1 行记录 )
```

对表做查询，有列对象且索引列作为过滤条件，可做索引扫描，例如：

```
test=# EXPLAIN SELECT A.* FROM A WHERE A.a1=1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using a_a1_key on a  (cost=0.00..8.27 rows=1 width=44)
    Index Cond: (a1 = 1)
(2 行记录 )
```

对表做查询，有列对象作为过滤条件，但索引列被运算符“-”处理，查询优化器不能在执行前进行取反运算，不可利用索引扫描，只能做顺序扫描。例如：

```
test=# EXPLAIN SELECT A.* FROM A WHERE -A.a1=-2;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on a  (cost=0.00..26.50 rows=6 width=44)
    Filter: ((- a1) = (-2))
(2 行记录 )
```

对表做查询，有列对象作为过滤条件，且目标列没有超出索引列，可做只读索引扫描[⊖]，这种扫描方式比单纯的索引扫描的效率更高。例如：

```
test=# EXPLAIN SELECT A.a1 FROM A WHERE A.a1=1;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using a_a1_key on a  (cost=0.00..3.16 rows=1 width=4)
    Index Cond: (a1 = 1)
(2 行记录 )
```

对表做查询，有索引存在，但选择条件不包括索引列对象，只能使用顺序扫描。例如：

```
test=# EXPLAIN SELECT A.a1 FROM A WHERE A.a3=1;
                        QUERY PLAN
-----------------------------------------------------
 Seq Scan on a  (cost=0.00..23.75 rows=6 width=4)
    Filter: (a3 = 1)
(2 行记录 )
```

⊖ 只读索引扫描是 PostgreSQL 9.2.X 新增加的索引扫描方式，早期的版本不支持。

对表做查询，有索引存在，选择条件包括索引列对象，可使用索引扫描，对选择条件中不存在索引的列作为过滤器被使用。例如：

```
test=# EXPLAIN SELECT A.a1 FROM A WHERE A.a1=1 AND A.a3=1;
                                QUERY PLAN
---------------------------------------------------------------------
 Index Scan using a_a1_key on a (cost=0.00..8.27 rows=1 width=4)
     Index Cond: (a1 = 1)
Filter: (a3 = 1)/* 不存在索引的列作为过滤器被使用 */
(3 行记录 )
```

对表做查询，有索引存在，选择条件包括索引列对象，但索引列对象位于一个表达式中，参与了运算，不是“key= 常量”格式，则索引不可使用，只能是顺序扫描。例如：

```
test=# EXPLAIN SELECT A.* FROM A WHERE A.a1+A.a3=2; // 索引列在表达式中
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on a (cost=0.00..26.50 rows=6 width=44)
    Filter: ((a1 + a3) = 2)
(2 行记录 )
test=# EXPLAIN SELECT A.* FROM A WHERE A.a1=2-A.a3; // 索引列参与运算，但不可使用索引扫描
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on a  (cost=0.00..26.50 rows=6 width=44)
    Filter: (a1 = (2 - a3))
(2 行记录 )
test=# EXPLAIN SELECT A.a1 FROM A WHERE A.a1<1+2; // 索引列不在 1+2 的表达式中，被独立使用，可使用索引扫描
                                QUERY PLAN
---------------------------------------------------------------------
 Bitmap Heap Scan on a  (cost=7.09..21.68 rows=367 width=4)
    Recheck Cond: (a1 < 3)
    ->Bitmap Index Scan on a_a1_key  (cost=0.00..7.00 rows=367 width=0)
          Index Cond: (a1 < 3)
(4 行记录 )
```

对表做查询，有索引列对象作为过滤条件，操作符是范围操作符 > 或 <，可做索引扫描。例如：

```
test=# EXPLAIN SELECT A.a1 FROM A WHERE A.a1>1;
                                QUERY PLAN
---------------------------------------------------------------------
 Bitmap Heap Scan on a  (cost=7.09..21.68 rows=367 width=4)
    Recheck Cond: (a1 > 1)
    ->Bitmap Index Scan on a_a1_key  (cost=0.00..7.00 rows=367 width=0) /* 使用了位图索引扫描 */
          Index Cond: (a1 > 1)
(4 行记录 )
test=# EXPLAIN SELECT A.a1 FROM A WHERE A.a1<1;
                                QUERY PLAN
---------------------------------------------------------------------
```

```
 Bitmap Heap Scan on a  (cost=7.09..21.68 rows=367 width=4)
   Recheck Cond: (a1 < 1)
   ->Bitmap Index Scan on a_a1_key  (cost=0.00..7.00 rows=367 width=0) /* 使用了位图索引扫描 */
         Index Cond: (a1 < 1)
(4 行记录 )
```

对表做查询，有索引列对象作为过滤条件，操作符是范围操作符 <>，不可做索引扫描。例如：

```
test=# EXPLAIN SELECT A.a1 FROM A WHERE A.a1<>1;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on a  (cost=0.00..23.75 rows=1099 width=4)
    Filter: (a1 <> 1)
(2 行记录 )
```

对表做查询，有索引列对象作为过滤条件，操作符是范围操作符 BETWEEN-AND，可做索引扫描。例如：

```
test=# EXPLAIN SELECT A.a1 FROM A WHERE A.a1 BETWEEN 1 AND 2;
                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on a  (cost=4.31..13.79 rows=6 width=4)
   Recheck Cond: ((a1 >= 1) AND (a1 <= 2))
   ->Bitmap Index Scan on a_a1_key  (cost=0.00..4.31 rows=6 width=0)
         Index Cond: ((a1 >= 1) AND (a1 <= 2))
(4 行记录 )
```

所以，对于索引列，索引可用的条件如下：

- ❏ 在 WHERE、JOIN/ON、HAVING 的条件中出现“ key <op> 常量”格式的条件子句（索引列不能参与带有变量的表达式的运算）。
- ❏ 操作符不能是 <> 操作符（不等于操作符在任何类型的列上不能使用索引，可以认为这是一个优化规则，在这种情况下，顺序扫描的效果通常好于索引扫描）。
- ❏ 索引列的值选择率越低，索引越有效，通常认为选择率小于 0.1 则索引扫描效果会好一些。

3.3.2 索引列的位置对使用索引的影响

在查询语句中，索引列出现在不同的位置，对索引的使用有着不同的影响。

1. 对目标列、WHERE 等条件子句的影响

索引列出现在目标列，通常不可使用索引（但不是全部情况都不能使用索引）。例如：

```
test=# EXPLAIN SELECT A.a1 FROM A;
                        QUERY PLAN
-----------------------------------------------------------
Seq Scan on a  (cost=0.00..21.00 rows=1100 width=4)
(1 行记录 )
```

从查询执行计划看，使用了顺序扫描而不是索引扫描。由此可见，索引列出现在目标列，对查询语句的优化没有好的影响。

聚集函数 MIN/MAX 用在索引列上，出现在目标列，可使用索引。例如：

```
test=# EXPLAIN SELECT MAX(A.a1) FROM A ;
                                        QUERY PLAN
-------------------------------------------------------------------------
 Result  (cost=0.06..0.07 rows=1 width=0)
   InitPlan 1 (returns $0)
     ->Limit  (cost=0.00..0.06 rows=1 width=4)
          ->Index Only Scan Backward using a_a1_key on a  (cost=0.00..67.41 rows=1095 width=4)
               Index Cond: (a1 IS NOT NULL)
(5 行记录 )
```

从查询执行计划看，使用了只读索引扫描进行 MAX 聚集函数的求解，所得结果（InitPlan 1）作为了 Result 的输入，这表明了 MIN/MAX 聚集函数的优化可以利用索引进行。

索引列出现在 WHERE 子句中，可使用索引。例如：

```
test=# EXPLAIN SELECT A.* FROM A WHERE A.a1=1;
                              QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using a_a1_key on a  (cost=0.00..8.27 rows=1 width=44)
    Index Cond: (a1 = 1)
(2 行记录 )
```

索引列 a1 作为 WHERE 子句的内容，可以高效利用索引完成扫描。

索引列出现在 JOIN/ON 子句中，作为连接条件，不可使用索引。例如：

```
test=# CREATE TABLE B (b1 INT UNIQUE, b2 VARCHAR(9), c2 INT); // 先创建 B 表
注意：CREATE TABLE / UNIQUE 将要为表 b 创建隐含索引 b_b1_key
CREATE TABLE
test=# EXPLAIN SELECT A.*, B.* FROM A JOIN B ON (a1=b1);
                              QUERY PLAN
-------------------------------------------------------------------------
 Hash Join  (cost=34.75..70.88 rows=1100 width=88)
    Hash Cond: (a.a1 = b.b1)
    ->Seq Scan on a  (cost=0.00..21.00 rows=1100 width=44)
    ->Hash  (cost=21.00..21.00 rows=1100 width=44)
         ->Seq Scan on b  (cost=0.00..21.00 rows=1100 width=44)
(5 行记录 )
```

从查询执行计划看，尽管连接条件 a1=b1 中的 a1 列和 b1 列都是索引列，但在表 A 和表 B 上使用了顺序扫描，没有利用索引。所以，在过滤元组的条件中快速定位元组可以用索引，做连接条件的元组定位不一定用索引（代价估算决定那种扫描方式最优）。

索引列出现在 JOIN/ON 子句中，作为限制条件满足“key <op> 常量”格式可用索引。例如：

```
test=# EXPLAIN SELECT A.*, B.* FROM A JOIN B ON (a1=b1) AND A.a1=1;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop  (cost=0.00..16.55 rows=1 width=88)
    ->Index Scan using a_a1_key on a  (cost=0.00..8.27 rows=1 width=44)
          Index Cond: (a1 = 1)
    ->Index Scan using b_b1_key on b  (cost=0.00..8.27 rows=1 width=44)
          Index Cond: (b1 = 1)// 发生了常量传递使得 b1=1
(5 行记录 )
```

从查询执行计划看，连接条件 a1=b1 中的 a1 列和 b1 列都是索引列，这和本示例的前一个示例不同，不同的原因是在连接条件中，出现了 A.a1=1 条件，使得查询优化器可以根据“常量传递”优化技术推知 b1=1，所以在表 A 和表 B 上可以各自使用索引扫描。

索引列出现在 WHERE 子句中，可用索引。例如：

```
test=# EXPLAIN SELECT A.*, B.* FROM A JOIN B ON (a1=b1) WHERE A.a1=1;
                                QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop  (cost=0.00..16.55 rows=1 width=88)
    ->Index Scan using a_a1_key on a  (cost=0.00..8.27 rows=1 width=44)
          Index Cond: (a1 = 1)
    ->Index Scan using b_b1_key on b  (cost=0.00..8.27 rows=1 width=44)
          Index Cond: (b1 = 1)// 发生了常量传递使得 b1=1
(5 行记录 )
```

从查询执行计划看，连接条件 a1=b1 中的 a1 列和 b1 列都是索引列，而 WHERE 条件中出现了 A.a1=1 条件，使得查询优化器在合并了连接条件和 WHERE 条件后，可以根据“常量传递”优化技术推知 b1=1，所以可以在表 A 和表 B 上各自使用索引扫描。

索引列出现在 WHERE 子句中，但与子查询比较，格式上不满足“key <op> 常量”，不可用索引。例如：

```
test=# EXPLAIN SELECT E.e1 FROM E WHERE E.e1 IN (SELECT A.a1 FROM A) ;
                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Semi Join (cost=34.75..64.76 rows=550 width=4)
    Hash Cond: (e.e1 = a.a1)
    ->Seq Scan on e  (cost=0.00..21.00 rows=1100 width=4)
    ->Hash  (cost=21.00..21.00 rows=1100 width=4)
          ->Seq Scan on a  (cost=0.00..21.00 rows=1100 width=4)
(5 行记录 )
```

从查询执行计划看，尽管 WHERE 条件子句中的 e1 是索引列，但不满足索引被使用的条件“key <op> 常量”，e1 索引列之后是 IN 子查询，所以不能使用索引。

2. 对 GROUPBY 子句的影响

索引列出现在 GROUPBY 子句中，不触发索引扫描，查询执行计划如下：

```
test=# EXPLAIN SELECT A.a1 FROM A GROUP BY a1;
```

```
                              QUERY PLAN
-------------------------------------------------------------------
 HashAggregate  (cost=28.88..30.88 rows=200 width=4)
   ->Seq Scan on a  (cost=0.00..25.10 rows=1510 width=4)
(2 行记录 )
```

从查询执行计划看，尽管索引列 a1 出现在 GROUPBY 子句中，但扫描表 A 使用了顺序扫描方式，没有使用索引。

WHERE 子句出现索引列，且 GROUPBY 子句出现索引列，索引扫描被使用。例如：

```
test=# EXPLAIN SELECT A.a1 FROM A WHERE a1>2 GROUP BY a1;
                                  QUERY PLAN
-------------------------------------------------------------------------
 HashAggregate  (cost=22.60..26.27 rows=367 width=4)
   ->Bitmap Heap Scan on a  (cost=7.09..21.68 rows=367 width=4)
       Recheck Cond: (a1 > 2)
       ->Bitmap Index Scan on a_a1_key  (cost=0.00..7.00 rows=367 width=0)
             Index Cond: (a1 > 2)
(5 行记录 )
```

从查询执行计划看，索引列 a1 出现在 GROUPBY 子句和 WHERE 子句中，且 WHERE 子句中的索引列使用符合“key <op> 常量”的格式，扫描表 A 使用了索引扫描方式。

WHERE 子句中出现非索引列，且 GROUPBY 子句出现索引列，索引扫描不被使用，例如：

```
test=# EXPLAIN SELECT A.a1 FROM A WHERE a3>2 GROUP BY a1;
                              QUERY PLAN
---------------------------------------------------------------
 HashAggregate  (cost=24.67..28.34 rows=367 width=4)
   ->Seq Scan ona  (cost=0.00..23.75 rows=367 width=4)
         Filter: (a3 > 2)
(3 行记录 )
```

从查询执行计划看，WHERE 条件中出现非索引列，所以只能进行顺序扫描。HashAggregate 表明使用了 Hash 聚集操作完成 GROUPBY（HashAggregate 用于数据量较少情况下的聚集或分组操作）。

如果禁止 HashAggregate，查看同样 SQL 语句 GROUPBY 操作的执行计划，索引扫描不被使用，但要依据索引列做排序操作，例如：

```
test=# set enable_hashagg=off; // 禁止 HashAggregate 操作
SET
test=# EXPLAIN SELECT A.a1 FROM A WHERE a3>2 GROUP BY a1;
                              QUERY PLAN
-------------------------------------------------------------------
Group  (cost=39.38..41.22 rows=367 width=4)
   ->Sort  (cost=39.38..40.30 rows=367 width=4)
      Sort Key: a1// 索引列做了排序键
```

```
      ->Seq Scan on a  (cost=0.00..23.75 rows=367 width=4)
            Filter: (a3 > 2)
(5 行记录 )
```

从查询执行计划看，出现了 Sort Key: a1 的内容，表明依据索引列 a1 进行排序操作。另外，采用 Group 完成分组操作，而不是采用 HashAggregate 完成分组操作。

如果禁止 HashAggregate，查看非索引列在 GROUPBY 中的执行计划，索引扫描不被使用，但要依据非索引列做排序操作，排序与索引列无关，例如：

```
test=# EXPLAIN SELECT A.a3 FROM A WHERE a3>2 GROUP BY a3;
                              QUERY PLAN
-----------------------------------------------------------------------
 Group  (cost=39.38..41.22 rows=67 width=4)
  ->Sort  (cost=39.38..40.30 rows=367 width=4)
      Sort Key: a3
      ->Seq Scan on a  (cost=0.00..23.75 rows=367 width=4)
            Filter: (a3 > 2)
(5 行记录 )
test=# set enable_hashagg=on; // 允许 HashAggregate 操作
SET
```

从查询执行计划看，出现了 Sort Key: a3 的内容，表明依据非索引列 a1 进行排序操作，所以排序操作和索引列的使用没有必然联系。

3. 对 HAVING 子句的影响

索引列出现在 HAVING 子句中与出现在 WHERE 子句中类似，是否能够使用索引，要看具体情况。

WHERE 子句中出现非索引列，且 GROUPBY 和 HAVING 子句出现索引列，索引扫描被使用，例如：

```
test=# EXPLAIN SELECT A.a1 FROM A WHERE a3>2 GROUP BY a1 HAVING a1>2;
                                    QUERY PLAN
-----------------------------------------------------------------------
 HashAggregate  (cost=22.84..24.06 rows=122 width=4)
   ->Bitmap Heap Scan on a  (cost=7.03..22.54 rows=122 width=4)
       Recheck Cond: (a1 > 2)
       Filter: (a3 > 2)
       ->Bitmap Index Scan on a_a1_key  (cost=0.00..7.00 rows=367 width=0)
             Index Cond: (a1 > 2)
(6 行记录 )
```

从查询执行计划看，尽管 WHERE 子句使用了非索引列 a3，但 HAVING 子句使用了索引列 a1，且 a1>2 表达式符合“key <op> 常量”的格式，所以使用了位图索引扫描。

4. 对 ORDERBY 子句的影响

索引列出现在 ORDERBY 子句中，可使用索引。

ORDERBY 子句中出现索引列可使用索引扫描，例如：

```
test=# EXPLAIN SELECT A.* FROM A ORDER BY a1;
                                  QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using a_a1_key on a  (cost=0.00..64.75 rows=1100 width=44)
(1 行记录 )
```

从查询执行计划看，使用了索引扫描，这表明 ORDERBY 操作可以利用索引列完成排序优化。

ORDERBY 子句中出现非索引列不可使用索引扫描，例如：

```
test=# EXPLAIN SELECT A.* FROM A ORDER BY a3;
                               QUERY PLAN
-----------------------------------------------------------------
 Sort  (cost=76.57..79.32 rows=1100 width=44)
   Sort Key: a3
   ->Seq Scan on a  (cost=0.00..21.00 rows=1100 width=44)
(3 行记录 )
```

从查询执行计划看，使用了顺序扫描，这表明 ORDERBY 操作在非索引列的作用下，不可以利用索引列完成排序优化。

5. 对 DISTINCT 的影响

索引列出现在 DISTINCT 子句管辖的范围中，与索引没有关联。

DISTINCT 子句管辖范围内出现索引列，不可使用索引扫描，例如：

```
test=# EXPLAIN SELECT DISTINCT A.a1 FROM A;
                               QUERY PLAN
-----------------------------------------------------------------
 HashAggregate  (cost=23.75..34.75 rows=1100 width=4)
   ->Seq Scan on a  (cost=0.00..21.00 rows=1100 width=4)
(2 行记录 )
```

从查询执行计划看，使用了顺序扫描，这表明 DISTINCT 操作在索引列的作用下，不可以利用索引列完成去重优化。

DISTINCT 子句管辖范围内出现索引列，因 WHERE 子句内使用索引列，故其可使用索引扫描，例如：

```
test=# EXPLAIN SELECT DISTINCT A.a1 FROM A WHERE A.a1>1;
                                  QUERY PLAN
-------------------------------------------------------------------------
 HashAggregate  (cost=22.60..26.27 rows=367 width=4)
   ->Bitmap Heap Scan on a  (cost=7.09..21.68 rows=367 width=4)
       Recheck Cond: (a1 > 1)
       ->Bitmap Index Scan on a_a1_key  (cost=0.00..7.00 rows=367 width=0)
             Index Cond: (a1 > 1)
(5 行记录 )
```

从查询执行计划看，因为 WHERE 条件中出现了索引列，所以使用了索引扫描，但这和 DISTINCT 操作没有关系。

3.3.3 联合索引对索引使用的影响

上一节利用单一索引列（索引定义中只包括一个列）介绍了索引列出现在 SQL 查询语句中的不同位置，对索引的利用存在的不同影响。本节介绍联合索引（索引定义中至少包含两个索引列）对使用索引的影响。

示例如下。首先创建表：

```
test=# CREATE TABLE E (e1 INT, e2 VARCHAR(9), e3 INT, PRIMARY KEY(e1, e3));
注意：CREATE TABLE / PRIMARY KEY 将要为表 "e" 创建隐含索引 "e_pkey"
```

使用联合索引的全部索引键，可触发索引的使用，例如：

```
test=# EXPLAIN SELECT E.* FROM E WHERE E.e1=1 AND E.e3=2;
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using e_pkey on e  (cost=0.00..8.27 rows=1 width=44)
    Index Cond: ((e1 = 1) AND (e3 = 2))
(2 行记录 )
```

从查询执行计划看，索引扫描建立在 WHERE 条件的 e1 列和 e3 列上，而这两列正是主键索引的全部列，所以可以使用索引扫描。

使用联合索引的前缀部分索引键，如“key_part_1 <op> 常量”，可触发索引的使用，例如：

```
test=# EXPLAIN SELECT E.* FROM E WHERE E.e1=1;
                                QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on e  (cost=4.30..13.76 rows=6 width=44)
   Recheck Cond: (e1 = 1)
   ->Bitmap Index Scan on e_pkey  (cost=0.00..4.30 rows=6 width=0)
         Index Cond: (e1 = 1)
(4 行记录 )
```

从查询执行计划看，位图索引扫描建立在 WHERE 条件的 e1 列上，而这列正是主键索引的第一列，所以可以使用索引扫描。

使用部分索引键，但不是联合索引的前缀部分，如“key_part_2 <op> 常量”，不可触发索引的使用，例如：

```
test=# EXPLAIN SELECT E.* FROM E WHERE E.e3=1;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on e  (cost=0.00..23.75 rows=6 width=44)
    Filter: (e3 = 1)
(2 行记录 )
```

从查询执行计划看，查询优化器选取了顺序索引扫描，没有利用索引的第二个键值在e3上使用索引。

使用联合索引的全部索引键，但索引键不是AND操作，不可触发索引的使用，例如：

```
test=# EXPLAIN SELECT E.* FROM E WHERE E.e3=2 OR E.e1=1 ;
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on e  (cost=0.00..26.50 rows=11 width=44)
    Filter: ((e3 = 2) OR (e1 = 1))
(2 行记录 )
```

从查询执行计划看，尽管WHERE条件中出现了索引键e1列和e3列，但这两列是OR操作而非AND操作，致使查询优化器选取了顺序扫描。

3.3.4　多个索引对索引使用的影响

上一小节介绍了联合索引对使用索引的影响。本小节介绍多个索引对使用索引的影响。示例如下。首先创建表：

```
test=# CREATE TABLE F (f1 INT UNIQUE NOT NULL, f2 INT UNIQUE NOT NULL, f3 INT, PRIMARY KEY(f1, f3));
注意：CREATE TABLE / PRIMARY KEY 将要为表 f 创建隐含索引 f_pkey
注意：CREATE TABLE / UNIQUE 将要为表 f 创建隐含索引 f_f1_key
注意：CREATE TABLE / UNIQUE 将要为表 f 创建隐含索引 f_f2_key
CREATE TABLE
```

WHERE条件子句出现两个可利用的索引，优选最简单的索引。

```
test=# EXPLAIN SELECT * FROM F WHERE f1=2 AND f2=1;
                                QUERY PLAN
---------------------------------------------------------------------------
 Index Scan using f_f2_key on f  (cost=0.00..8.27 rows=1 width=12)
    Index Cond: (f2 = 1)
    Filter: (f1 = 2)
(3 行记录 )
```

从查询执行计划看，索引扫描选取了f2列上的索引，没有选取f1列上的索引，这是因为f1列上存在两个索引（一个独立索引、一个联合索引），这比f2列上的索引复杂，查询优化器优先选择了最为简单的索引（但不是所有数据库都这样，本质上，扫描方式的选取取决于代价估算模型对每种扫描试的评估）。

WHERE条件子句出现两个可利用的索引且索引键有重叠部分（f1列出现在两个索引中），优选最简单的索引。

```
test=# EXPLAIN SELECT * FROM F WHERE f1=1 AND (f1=2 AND f3=3);
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Result  (cost=0.00..8.27 rows=1 width=12)
   One-Time Filter: false
   ->Index Scan using f_f1_key on f  (cost=0.00..8.27 rows=1 width=12)
         Index Cond: (f1 = 1)
         Filter: (f3 = 3)
(5 行记录 )
```

从查询执行计划看，索引扫描建立在 WHERE 条件中的 f1=1 条件上，这表明查询优化器选取的是 f1 列上的独立索引而不是联合索引。另外，f1=1 和 f1=2 这两个条件冲突，查询优化器可以进一步优化但没有优化。

WHERE 条件子句出现两个可利用的索引，优选最简单的索引。

```
test=# EXPLAIN SELECT * FROM F WHERE (f1=2 AND f3=3) AND f2=1;
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using f_f2_key on f  (cost=0.00..8.27 rows=1 width=12)
    Index Cond: (f2 = 1)
    Filter: ((f1 = 2) AND (f3 = 3))
(3 行记录 )
```

从查询执行计划看，尽管联合索引的条件表达式（f1=2 AND f3=3）在单独索引的条件表达式（f2=1）的前面，索引扫描却建立在 WHERE 条件中的 f2=1 条件上，这表明查询优化器选取的是 f2 列上的独立索引而不是 f1 和 f3 构成的联合索引。

WHERE 条件子句出现两个可利用的索引，优选最简单的索引。

```
test=# EXPLAIN SELECT * FROM F WHERE f2>1 AND f2<100 AND f1=3;
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using f_f1_key on f  (cost=0.00..8.27 rows=1 width=12)
    Index Cond: (f1 = 3)
    Filter: ((f2 > 1) AND (f2 < 100))
(3 行记录 )
```

从查询执行计划看，索引扫描选取了 f1 列上的索引，虽然 f2 列上也存在索引，但 f2 列的条件构成范围扫描，通常范围扫描选择率比等值比较的选择率大，所以查询优化器选择了 f1 上的索引。

WHERE 条件子句出现两个可利用的索引，优选最简单的索引。

```
test=# EXPLAIN SELECT * FROM F WHERE f2>1 AND f2<100 AND f3=3;
                                QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on f  (cost=4.35..14.96 rows=1 width=12)
   Recheck Cond: ((f2 > 1) AND (f2 < 100))
   Filter: (f3 = 3)
   ->Bitmap Index Scan on f_f2_key  (cost=0.00..4.35 rows=10 width=0)
         Index Cond: ((f2 > 1) AND (f2 < 100))
(5 行记录 )
```

列 f2 和 f3 构成联合索引，但 f3 不是索引键的前缀部分，多数情况下不可以用做索引扫描。从查询执行计划上看，查询执行计划选取了范围扫描（f2>1 AND f2<100）确定的位图索引扫描的条件。

3.4　两表连接算法

关系代数的一项重要操作是连接运算，多个表连接是建立在两表之间连接的基础上的。研究两表连接的方式，对连接效率的提高有着直接的影响。

3.4.1　基本的两表连接算法

基本的两表连接算法主要有嵌套循环连接算法、归并连接算法、Hash 连接算法等。

1. 嵌套循环连接算法

两表做连接，采用的最基本算法是**嵌套循环连接算法**。算法的描述如下：

```
FOR EACH ROW r1 IN t1 {
    FOR EACH ROW r2 IN t2 {
        IF r1,r2 SATISFIES JOIN CONDITIONS
            JOIN r1,r2
    }
}
```

数据库引擎在实现该算法的时候，以元组为单位进行连接。元组是从一个内存页面获取来的，而内存页面是从存储系统通过 IO 操作获得的，每个 IO 申请以“块”为单位尽量读入多个页面。所以，如果考虑获取元组的方式，则可以改进嵌套循环连接算法，改进后的算法称为**基于块的嵌套循环连接算法**。算法描述如下：

```
FOR EACH CHUNK c1 OF t1 {
    IF c1 NOT IN MEMORY // 系统一次读入多个页面，所以不需要每次都从存储系统读入，消耗 IO
        READ CHUNK c1 INTO MEMORY
    FOR EACH ROW r1 IN CHUNK c1 {// 从页面中分析出元组，消耗 CPU
        FOR EACH CHUNK c2 OF t2 {
            IF c2 NOT IN MEMORY
                READ CHUNK c2 INTO MEMORY
            FOR EACH ROW r2 IN c2 {// 从页面中分析出元组，消耗 CPU
                IF r1,r2 SATISFIES JOIN CONDITIONS
                    JOIN r1,r2
            }
        }
    }
}
```

无论是嵌套循环连接还是基于块的嵌套循环连接，其本质都是在一个两层的循环中拿出各自的元组，逐一匹配是否满足连接条件。其他一些两表连接算法，多是在此基础上进行的改进。如基于索引做改进，在考虑了**聚簇和非聚簇索引**[⊖]的情况下，如果内表有索引可用，则可以加快连接操作的速度。另外，如果内层循环的最后一个块使用后作为下次循环

⊖ 聚簇索引是指表的一个或多个列作为索引的关键字，以关键字的具体值为依据，把所有具有相同值的元组连续存放在外存上。当从磁盘扫描读取的块进入内存时，相同值的其他元组在内存中的概率增大，能有效减少 IO。

的第一个块，则可以节约一次 IO。如果外层元组较少，内层的元组驻留内存多一些（如一些查询优化器采用物化技术固化内层的元组），则能有效提高连接的效率。

嵌套循环连接算法和基于块的嵌套循环连接算法适用于内连接、左外连接、半连接、反半连接等语义的处理。

2. 排序归并连接算法

排序归并连接算法又称归并排序连接算法，简称归并连接算法。这种算法的步骤是：为两个表创建可用内存缓冲区数为 M 的 M 个子表，将每个子表排好序；然后读入每个子表的第一块到 M 个块中，找出其中最小的先进行两个表的元组的匹配，找出次小的匹配……依此类推，完成其他子表的两表连接。

归并连接算法要求内外表都是有序的，所以对于内外表都要排序。如果连接列是索引列，可以利用索引进行排序。

归并连接算法适用于内连接、左外连接、右外连接、全外连接、半连接、反半连接等语义的处理。

3. Hash 连接算法

基于 Hash 的两表连接算法有多种，常见的有如下 3 种：

- 用连接列作为 Hash 的关键字，对内表进行 Hash 运算建立 Hash 表，然后对外表的每个元组的连接列用 Hash 函数求值，值映射到内表建立好的 Hash 表就可以连接了；否则，探索外表的下一个元组。这样的 Hash 连接算法称为**简单 Hash 连接（Simple Hash Join，SHJ）算法**。
- 如果把内表和外表划分成等大小的子表，然后对外表和内表的每个相同下标值的子表进行 SHJ 算法的操作，可以避免因内存小反复读入内外表的数据的问题。这样的改进算法称为**优美 Hash 连接（Grace Hash Join，GHJ）算法**。
- 结合了 SHJ 和 GHJ 算法的优点的**混合 Hash 连接（Hybrid Hash Join，HHJ）算法**。HHJ 算法是把第一个子表保存到内存不刷出，如果内存很大，则子表能容纳更大量的数据，效率接近于 SHJ。

Hash 类的算法都可能存在 Hash 冲突，如 GHJ 算法，当内存小或数据倾斜（不能均衡地分布到 Hash 桶，Hash 处理后集中在少量桶中）时，通过把一个表划分为多个子表的方式，仍然不能消除反复读入的内外表数据的问题（称为“分区溢出”）。

Hash 连接算法只适用于数据类型相同的等值连接。Hash 连接需要存储 Hash 元组到 Hash 桶，要求较大的内存。如果表中连接列值重复率很高不能均匀分布，相同值的元组映射到少数几个桶中，Hash 连接算法效率就不会高。Hash 算法要求内表不能太大，通常查询优化器申请一段内存存放 Hash 表，如果超出且不能继续动态申请，则需要写临时文件，这会导致 IO 的颠簸（PostgreSQL 存在此类问题）。

Hash 连接算法适用于内连接、左外连接、右外连接、全外连接、半连接、反半连接等语义的处理。

3.4.2　进一步认识两表连接算法

从内存的容量角度看，两表连接算法可以分为一趟算法、两趟算法，甚至多趟算法。所谓“趟”是指从存储系统获取全部数据的次数。一趟算法因内存空间能容纳下全部数据，所以读取一次即可。两趟算法的第一趟从存储系统获取两表的数据，如做排序等处理后，再写入外存的临时文件；第二趟重新读入临时文件进行进一步处理（有的算法对其中一个表的元组只读取一次即可，属于一趟，因此两趟算法变为一趟半，但依然称为两趟算法）。多趟算法的思想和两趟算法基本相同，用以处理更大量的数据。趟数是一种方式，不是算法思想的改进，是代码实现中为减少 IO 所做的改进工作。

结合连接算法和索引、趟数的关系，两表连接算法对于查询优化器的意义如表 3-2 所示。

表 3-2　连接算法和索引及趟数的关系表

方式 / 算法	用索引改进算法（以 R 连接 S 为例）	趟数
嵌套连接	支持用索引改进算法 对外表 R 的每一个元组，如果 r 的值可作为 S 连接列上的索引键值，用索引扫描 S 的元组，与 r 判断是否匹配	一趟
基于 Hash	不支持用索引改进算法 一趟读入 S 的数据，根据连接条件构造 Hash，把元组散列到桶中；对于 R，读入一部分到缓冲区，对读入的每个元组散列，如果有同样散列值的桶已经在读入 S 的过程中构造出来，则可进行连接；以此类推，逐步把 R 的其他部分读入缓存的同样方式处理	一趟
基于排序	支持用索引改进算法 第一趟，利用索引进行内外表的排序；第二趟，读入两个表排序的数据进行连接	两趟

3.4.3　连接操作代价计算

连接操作花费 CPU 资源。从理论的角度分析[⊖]，连接操作的代价估算原理如表 3-3 所示。

表 3-3　两表连接算法的代价表

算　法	代价估算公式
嵌套循环连接	基本的嵌套循环连接：C-outer + C-inner 内表使用索引改进嵌套循环连接：C-outer + C-inner-index
归并连接	基本的归并连接：C-outer + C-inner + C-outersort + C-innersort 内外表使用索引，只影响排序，C-outersort、C-innersort 可能变化
Hash 连接	C-createhash + (N-outer * N-inner * 选择率) * a_tuple_cpu_time

表 3-3 中涉及公式的参数如下：

- a_tuple_cpu_time，获取一个元组消耗的 CPU 时间。

⊖ 在编码实现中，理论上的算法通常会被改进，所以实际的计算方式会有所不同，但原理不变。

- N-outer，扫描获取的外表元组数。
- N-inner，扫描获取的内表元组数，N-inner = N-inner-all × 选择率，其中 N-inner-all 表示内表的所有元组数。
- C-outer，扫描外表的代价，C-outer = N-outer × a_tuple_cpu_time。
- C-inner，扫描内表的代价，C-inner = N-inner × a_tuple_cpu_time。
- C-inner-index，使用索引扫描内表的代价，通常 C-inner-index 会小于 C-inner。
- C-outersort，外表排序的代价。
- C-innersort，内表排序的代价。
- C-createhash，创建 Hash 的代价。

3.5 多表连接算法

多表连接算法实现的是在查询路径生成的过程中，根据代价估算，从各种可能的候选路径中找出最优的路径（最优路径是代价最小的路径）。

多表连接算法需要解决两个问题：

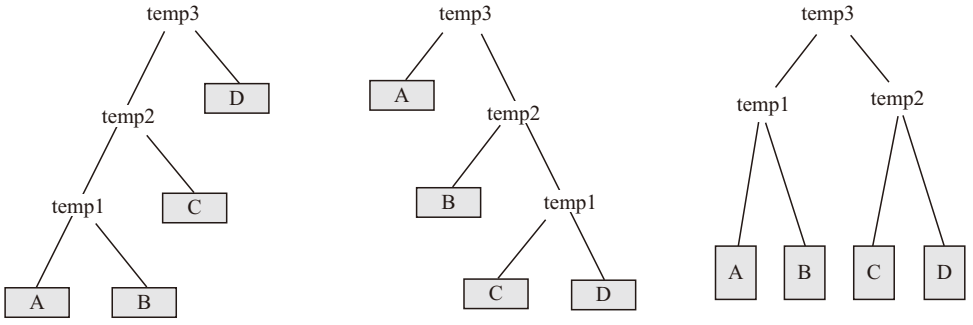
- **多表连接的顺序**：表的不同的连接顺序，会产生许多不同的连接路径；不同的连接路径有不同的效率。
- **多表连接的搜索空间**：因为多表连接的顺序不同，产生的连接组合会有多种，如果这个组合的数目巨大，连接次数会达到一个很高的数量级，最大可能的连接次数是 *N*！（*N* 的阶乘）。比如，*N*=5，连接次数是 120；*N*=10，连接次数是 3 628 800；*N*=20，连接次数是 2 432 902 008 176 640 000。所有的连接可能构成一个巨大的“搜索空间”。如何将搜索空间限制在一个可接受的时间范围内，并高效地生成查询执行计划将成为一个难点。

3.5.1 多表连接顺序

多表间的连接顺序表示了查询计划树的基本形态。一棵树就是一种查询路径，SQL 的语义可以由多棵这样的树表达，从中选择花费最少的树，就是最优查询计划形成的过程。

而一棵树包括左深连接树、右深连接树、紧密树（1990 年，Schneder 等在研究查询树模型时提出了左深树 1eft deep trees、右深树 right deep trees 和紧密树 bushy trees）3 种形态，如图 3-1 所示。左深树将从最下面的左子树 A 起，进行 AB 连接，连接后得到新的中间关系 temp1，再和 C 连接，生成新的中间关系 temp2，temp2 和 D 连接得到最终的连接路径 temp3（如图 3-1a 所示）。右深树的连接方式是从最右子树 D 开始，一直连接到 A 为止（如图 3-1b 所示）。而紧密树是 AB 连接生成 temp1、CD 连接 temp2，之后 temp1 和 temp2 连接得到 temp3 如图 3-1c 所示。不同的连接顺序，会生成不同大小的中间关系，这意味着 CPU 和 IO 消耗不同，所以 PostgreSQL 中会尝试多种连接方式存放到 path 上，以找出花费

最小的路径。



a）左深连接树　　b）右深连接树　　c）紧密树

图 3-1　三种树的形态

另外，即使是同一种树的生成方式，也有细节需要考虑。在图 3-1a 中，{A，B} 和 {B，A} 两种连接方式花费可能不同。比如最终连接结果是 {A，B，C}，但是需要验证是 {A，B，C}、{A，C，B}、{B，C，A}、{B，A，C}、{C，A，B}、{C，B，A} 中哪一个连接方式得到的结果，这就要求无论是哪种结果，都需要计算这 6 种连接方式中每一种的花费，找出最优的一种作为下次和其他表连接的依据。

人们针对以上树的形成、形成的树的花费代价最少的，提出了诸多算法。树的形成过程，主要有以下两种策略：

- **至顶向下**。从 SQL 表达式树的树根开始，向下进行，估计每个结点可能的执行方法，计算每种组合的代价，从中挑选最优的。
- **自底向上**。从 SQL 表达式树的树叶开始，向上进行，计算每个子表达式的所有实现方法的代价，从中挑选最优的，再和上层（靠近树根）的进行连接，周而复始直至树根。

在数据库实现中，多数数据库采取了第二种方式——自底向上，构造查询计划树，常用算法参见 3.5.2 节。

3.5.2　常用的多表连接算法

表与表进行连接，对多表连接进行搜索查找最优查询树，通常有多种算法，比如启发式、分枝界定计划枚举、爬山法、动态规划、System R 优化方法等。

1. 动态规划

20 世纪 40 年代，Richard Bellman 最早使用了动态规划这一概念，用以表述通过遍历寻找最优决策解问题。

“动态规划”（dynamic programming）中的 programming 来自“数学规划”（mathematical

programming，又称规划），与“计算机编程”（computer programming）中的 programming 没有关系。规划的含义是指**生成活动的优化策略**，规划意味着找到一个可行的活动计划。动态规划，是指决策依赖于当前状态，又随即引起状态的转移，一个**决策序列就是在变化的状态中产生出来的**，这就是“动态”的含义。

“动态规划”将待求解的问题分解为若干个子问题（子阶段），按顺序求解子问题，前一子问题的解为后一子问题的求解提供了有用的信息。在求解任一子问题时，列出各种可能的局部解，通过决策保留那些有可能达到最优的局部解，丢弃其他局部解。依次解决各子问题，最后一个子问题就是初始问题的解。

动态规划算法包括的主要概念有：

- **阶段**。把求解问题的过程分成若干个相互联系的阶段，以便于求解。在多数情况下，阶段变量是离散的。
- **状态**。表示每个阶段开始面临的自然状况或客观条件，它不以人们的主观意志为转移，也称为不可控因素。
- **无后效性**。状态应该具有的性质，如果给定某一阶段的状态，则在这一阶段以后过程的发展不受这阶段以前各段状态的影响。
- **决策**。一个阶段的状态确定后，从该状态演变到下一阶段某个状态的选择（行动）称为决策。
- **策略**。由每个阶段的决策组成的序列称为策略。对于每一个实际的多阶段决策过程，可供选取的策略有一定的范围限制，这个范围称为允许策略集合。允许策略集合中达到最优效果的策略称为最优策略。
- **最优化原理**。如果问题的最优解所包含的子问题的解也是最优的，就称该问题具有最优子结构，即满足最优化原理。最优化原理实际上是要求问题的最优策略的子策略也是最优的。

在数据库领域，动态规划算法主要解决的是多表连接的问题。动态规划算法是从底向上进行的，即从叶子（单个表）开始算作一层，然后由底层开始对每层的关系做两两连接（如果满足内连接则两两连接，不满足内连接则不可对全部表进行两两连接操作），构造出上层，逐次递推到树根。下面介绍具体步骤。

步骤 1　初始状态。

构造第一层关系，即叶子结点，每个叶子对应一个单表，为每一个待连接的关系计算最优路径（单表的最优路径就是单表的最佳访问方式，通过评估不同的单表的数据扫描方式花费，找出代价最小的作为每个单表的局部最优路径）。

步骤 2　归纳。

当层数从第 1 到 $n-1$，假设已经生成，则如何求解第 n 层的关系？方法为：将第 $n-1$ 层的关系（有多个关系）与第一层中的每个关系连接，生成新的关系（对新关系的大小进行估算），放于第 n 层，且每一个新关系，均求解其最优路径。

以上虽然分为两步，但实际上步骤 2 多次执行，每一次执行后生成的结果被下一次使用，即每层路径的生成都是**基于上层生成的最优路径**的，这满足最优化原理的要求。

动态规划算法与 System R 算法相比，增加了中间关系的大小估算。还有的改进算法，在生成第 n 层的时候，除了通过第 $n-1$ 层和第一层连接外，还可以通过第 $n-2$ 层和第 2 层连接，通过第 $n-3$ 层和第 3 层连接……

示例　看下面的查询语句：

```
SELECT *
FROM A, B, C, D
WHERE A.col=B.col AND A.col=C.col AND A.col=D.col
```

上面的查询语句生成最优查询计划的过程（按计算机中树的结构表述，自底向上层数依次是 1 到 4，1 是树叶层，4 是树顶层）如表 3-4 所示。

表 3-4　动态规划算法运行过程表

层数	说　明	可能得到的连接的中间结果
4	第四层可以通过第一、三层或第二层与第二层，进行每层间的每个表两两连接得到	{A,B,C,D}
3	第三层可以通过第一、二层两两连接得到。第三层得到的最终中间结果不多，但同一个中间结果可能由多种连接组合得到，如 {A,B,C} 可以由“{A,B} 和 {A,C}”或“{A,B} 和 {B,C}”连接得到，也可以由“{A,C} 和 {C,B}”或“{A,C} 和 {B,C}”等得到，方式多，但选取哪个，需要经过代价估算选取花费最小的	{A,B,C},{A,B,D}, {A,C,D},{B,C,D}
2	第二层可以通过树叶层中的表两两连接得到 {A,B},{B,A} 可能的连接代价是不同的，所以算作两个候选（有的书籍忽略了 {A,B} 和 {B,A} 的差别，似乎连接向来是从左至右的，所以如果树叶是“{A},{B},{C},{D}”次序，则结果只有 {A,B} 没有 {B,A}），这对于没有特别限定表的顺序的连接是有遗漏的	{A,B},{A,C},{A,D},{B,A}, {C,A},{D,A}, {B,C},{C,B},{B,D},{D,B}, {C,D},{D,C}
1	树叶，初始层。每个单表都计算单表扫描代价	{A},{B},{C},{D}

PostgreSQL 查询优化器求解多表连接时，采用了这种算法。

2. 启发式方法

启发式算法（heuristic algorithm）是相对于最优化算法提出的，是一个**基于直观或经验**构造的算法。

在数据库的查询优化器中，启发式一直贯穿于整个查询优化阶段，在逻辑查询优化阶段和物理查询优化阶段，都有一些启发规则可用。

启发式方法不能保证找到最好的查询计划。PostgreSQL、MySQL 和 Oracle 等数据库在实现查询优化器时，采用了启发式和其他方式相结合的方式。

注意

启发式应对的不是多个关系连接的探索问题，而是根据已知可优化（等价且等价变换后的花费更少）规则，对 SQL 语句做出语义等价转换（query transformation）的优化，或者基

于经验对某个物理操作进行改进（如物化操作，是一种保存临时结果用于多处计算的物理优化方式）。

常用的启发式规则分别是逻辑查询优化阶段可用的优化规则和物理查询优化阶段可用的优化规则。逻辑查询优化阶段可用的启发式优化规则参见第 2 章。

在物理查询优化阶段常用的启发式规则如下：

- ❑ 关系 R 在列 X 上建立索引，且对 R 的选择操作发生在列 X 上，则采用索引扫描方式。
- ❑ R 连接 S，其中一个关系上的连接列存在索引，则采用索引连接且此关系作为内表（放在内存循环中）。
- ❑ R 连接 S，其中一个关系上的连接列是排序的，则采用排序进行连接比 Hash 连接好。

3. 贪婪算法

贪婪（Greedy）算法，又称贪心算法。在对问题求解时，贪婪算法总是做出在当前看来是最好的选择，而这种选择是局部最优。局部最优不一定是整体最优，所以贪婪算法不从整体最优上加以考虑，省去了为找最优解要穷尽所有可能而必须耗费的大量时间（这点正是动态规划算法所做的事情），得到的是局部最优解。

贪婪算法为了解决问题需要寻找一个构成解的候选对象集合。其主要实现步骤如下：

1）初始，算法选出的候选对象的集合为空；

2）根据选择函数，从剩余候选对象中选出最有希望构成解的对象；

3）如果集合中加上该对象后不可行，那么该对象就被丢弃并不再考虑；

4）如果集合中加上该对象后可行，就加到集合里；

5）扩充集合，检查该集合是否构成解；

6）如果贪婪算法正确工作，那么找到的第一个解通常是最优的，可以终止算法；

7）继续执行 2，（每做一次贪婪选择就将所求问题简化为一个规模更小的子问题，最终可得到问题的一个可能的整体最优解）。

MySQL 查询优化器求解多表连接时采用了这种算法。

4. System R 算法

System R 算法对自底向上的动态规划算法进行了改进，主要的思想是把子树的查询计划的**最优查询计划**和**次优的查询计划**保留，用于上层的查询计划生成，以便使得查询计划总体上最优。

5. 遗传算法

遗传算法（Genetic Algorithm，GA）是美国学者 Holland 于 1975 年首先提出来的。它是一种启发式的优化算法，是基于自然群体遗传演化机制的高效探索算法。

遗传算法抛弃了传统的搜索方式，模拟自然界生物进化过程，采用人工进化的方式对目标空间进行随机化搜索。它将问题域中的可能解看作是群体的一个个体（染色体），并将

每一个个体编码成符号串形式，模拟达尔文的遗传选择和自然淘汰的生物进化过程，对群体反复进行基于遗传学的操作（选择、交叉、变异），根据预定的目标适应度函数对每个个体进行评价，依据“适者生存，优胜劣汰”的进化规则，不断得到更优的群体，同时以全局并行搜索方式来搜索优化群体中的最优个体，求得满足要求的最优解[⊖]。

遗传算法可以有效地利用已经有的信息处理来搜索那些有希望改善解质量的串，类似于自然进化，遗传算法通过作用于“染色体”上的“基因”，寻找好的“染色体”来求解问题（对算法所产生的每个“染色体”进行评价，并基于适应度值来改造“染色体”，使适用性好的“染色体”比适应性差的“染色体”有更多的“繁殖机会”）。

下面介绍遗传算法中的主要概念。

- **群体（population）**。一定数量的个体组成了群体，表示 GA 的遗传搜索空间。
- **个体（individual）**。多个个体组成群体，在多表连接中是每个基本关系或中间生成的临时关系。
- **染色体（chromosome）**。个体的特征代表，即个体的标志，由若干基因组成，是 GA 操作的基本对象，所以操作个体实则是操作染色体（个体几乎可以简单理解为“等同染色体”）。染色体用字符串表示。
- **基因（gene）**。基因是染色体的片段，多段基因组成染色体，基因变异导致基因不断被优化。
- **适应度（fitness）**。表示个体对环境的适应程度，通常由某一适应度函数表示。对应执行策略的执行代价。
- **选择（selection）**。GA 的基本操作之一，即根据个体的适应度，在群体中按照一定的概率选择可以作为父本的个体，选择依据是适应度大的个体被选中的概率高。选择操作体现了“适者生存，优胜劣汰”的进化规则。
- **交叉（crossover）**。GA 的基本操作之一，即将父本个体按照一定的概率随机地交换基因形成新的个体。
- **变异（mutate）**。GA 的基本操作之一，即按一定概率随机改变某个个体的基因值。

接下来介绍遗传算法涉及的关键问题。

- **串的编码方式**。本质是编码问题。一般把问题的各种参数用二进制形式进行编码，构成子串；然后把子串拼接构成“染色体”串。串长度及编码形式对算法收敛影响极大。
- **适应度函数的确定**。适应度函数（fitness function）又称对象函数（object function）或问题的“环境”，是问题求解品质的测量函数。一般可以把问题的模型函数作为对象函数，但有时需要另行构造。
- **遗传算法自身参数设定**。遗传算法自身参数有 3 个，即群体大小 n、交叉概率 P_c 和变异概率 P_m，具体如下：

⊖ 通过有组织地、随机地交换信息重新组合那些适应性好的串，在每一代中，利用上一代串结构中适应好的位和段来生成一个新的串的群体；偶尔要在串结构中尝试用新的位和段来替代原来的部分。

- 群体大小 n 太小时难以求出最优解，太大则增长收敛时间，一般 $n = 30 \sim 160$。
- 交叉概率 P_c 太小时难以向前搜索，太大则容易破坏高适应值的结构，一般取 P_c=0.25 ～ 0.75。
- 变异概率 P_m 太小时难以产生新的基因结构，太大使遗传算法成了单纯的随机搜索，一般取 $P_m = 0.01 \sim 0.2$。

遗传算法主要步骤如下：

1）随机初始化种群；

2）评估初始的种群，即为种群计算每个个体的适应值且对所有个体排序；

3）如果没有达到预定演化数（可以是一个确定的、与连接的表的个数无关的值，这样保证搜索空间一定不会因连接的表的个数增多导致搜索空间指数级增大），则继续下一步，否则结束算法；

4）选择父体，随机挑选父体 dad 和母体 mum；

5）杂交，父体和母体杂交得到新个体 child；

6）变异，在某些个别条件下对新个体变异（不是大概率变异，不是每次都需要变异）；

7）计算新个体的适应值，并把适应值排名插入到种群，种群中排名最后的则被淘汰；

8）继续步骤 3）。

6. 其他算法

还有其他的一些算法，都可以用于查询优化多表连接的生成，如爬山法、分支界定枚举法、随机算法、模拟退火算法或多种算法相结合等。

3.5.3 多表连接算法的比较

多表连接有着多种算法，它们的异同如表 3-5 所示。

表 3-5 多表连接常用算法比较表

算法名称	特点与适用范围	缺　点
启发式算法	适用于任何范围，与其他算法结合，能有效提高整体效率	不知道得到的解是否最优
贪婪算法	非穷举类型的算法。适合解决较多关系的搜索	得到局部最优解
爬山法	适合查询中包含较多关系的搜索，基于贪婪算法	随机性强，得到局部最优解
遗传算法	非穷举类型的算法。适合解决较多关系的搜索	得到局部最优解
动态规划算法	穷举类型的算法。适合查询中包含较少关系的搜索，可得到全局最优解	搜索空间随关系个数增长呈指数增长
System R 优化	基于自底向上的动态规划算法，为上层提供更多可能的备选路径，可得到全局最优解	搜索空间可能比动态规划算法更大一些

3.6 本章小结

本章从物理查询优化的角度，讲述了物理查询优化的依据、查询代价估算模型。整个

物理查询优化，无论是单表扫描、两表连接，还是多表连接，都要依据代价估算模型，对每一种局部查询路径进行估算。物理优化阶段代价估算无处不在。

对于单表扫描、两表连接、多表连接的算法进行了概要性的讲述，目的是点明其中的优化点，帮助读者从整体上认识物理优化所做的优化工作。关于多表连接涉及的算法，在后面的相关章节中将结合代码详细讲述其实现方式。对于这部分知识的学习，一定要注意理论联系实际，只有这样才能真正全部掌握。

索引技术单独作为一节，讲述了索引对于路径选取和花费的影响，这有助于读者对索引有进一步的认识。是否使用索引、如何使用索引，这些问题都值得注意，这对查询语句的性能有着较大影响。