《Architecture of a Database System》

（中文版）

Joseph M. Hellerstein, Michael Stonebraker and James Hamilton

翻译：林子雨

厦门大学数据库实验室

http://dblab.xmu.edu.cn

中文版网址：http://dblab.xmu.edu.cn/node/459

厦门大学计算机科学系教师 林子雨 翻译作品

http://www.cs.xmu.edu.cn/linziyu

2013 年 9 月

前言

本文翻译自经典英文论文《Architecture of a Database System》，原文作者是 Joseph M. Hellerstein, Michael Stonebraker 和 James Hamilton。该论文可以作为中国各大高校数据库实验室研究生的入门读物，帮助学生快速了解数据库的内部运行机制。

本文一共包括 6 章，分别是：第 1 章概述，第 2 章进程模型，第 3 章并行体系结构：进程和内存协调，第 4 章关系查询处理器，第 5 章存储管理，第 6 章事务：并发控制和恢复，第 7 章共享组件，第 8 章结束语。

本文翻译由厦门大学数据库实验室林子雨老师团队合力完成，其中，林子雨老师负责统稿校对，刘颖杰同学负责翻译第 1 章和第 2 章，罗道文同学负责翻译第 3 章和第 4 章，谢荣东同学负责翻译第 5 章、第 6 章、第 7 章和第 8 章，蔡珉星同学负责对林子雨老师的校对结果进行二次校对。

如果对本文翻译内容有任何疑问，欢迎联系林子雨老师。

林子雨的E-mail是：ziyulin@xmu.edu.cn。

林子雨的个人主页是：http://www.cs.xmu.edu.cn/linziyu。

厦门大学数据库实验室网站是：http://dblab.xmu.edu.cn。

本文中文版的网址是：http://dblab.xmu.edu.cn/node/459。

林子雨于厦门大学海韵园

2013 年 9 月

摘 要

数据库管理系统（DBMS）广泛存在于现代计算机系统中，并且是其重要的组成部分。它是学术界以及工业界数十年研究和发展的成果。在计算机发展史上，数据库属于最早开发的多用户服务系统之一，因此，它的研究也催生了许多为保证系统可拓展性以及稳定性的系统开发技术，这些技术如今被应用于许多其他的领域。虽然许多数据库的相关算法和概念广泛见于教科书中，但关于如何让一个数据库工作的系统设计问题却鲜有资料介绍。本文从体系架构角度探讨数据库设计的一些准则，包括处理模型、并行架构、存储系统设计、事务处理系统、查询处理及优化结构以及具有代表性的共享组件和应用。当业界有多种设计方式可供选择时，我们以当前成功的商业开源软件作为参考标准。

第 4 章 关系查询处理器

厦门大学计算机科学系教师 林子雨 编著

个人主页：http://www.cs.xmu.edu.cn/linziyu

中文版网址：http://dblab.xmu.edu.cn/node/459

2013 年 9 月

第 4 章 关系查询处理器

前面的章节强调了一个 DBMS 的宏观架构设计问题。现在，在接下来的章节中我们开始以更细的角度讨论设计问题，依次研究每一个主要的 DBMS 组件。接着第 1.1 节的讨论，我们开始从系统的顶部查询处理器开始，然后在随后的几个章节，我们向下介绍存储管理、事务处理以及实用工具。

一个关系查询处理器以一个 SQL 语句作为输入，然后进行验证，优化成为一个程序数据流执行计划，并且在获得准入许可以后可以代表一个客户程序执行数据流程序。然后，客户程序获取（“拉”）结果元组，通常一次一个元组或一小批元组。关系查询处理器的主要组件已经在图 1-1 介绍过了。在本章内容中，我们关注查询处理器和“存储管理器的访问方法的一些非事务处理方面”。一般而言，关系查询处理可以被看作是一个单用户、单线程任务。正如后面第五章描述的那样，并发控制是由系统较低层透明控制的。这个规则唯一的例外就是，当 DBMS 操作缓冲池页面的时候，DBMS 必须明确“固定”(pin)和“不固定”(unpin)缓冲池页面，这样就可以使它们在简短并且关键的操作执行时驻留在内存中，我们将在第 4.4.5 节讨论这点。

在本章中，我们将重点放在常见的 SQL 命令：数据操作语言（DML: Data Manipulation Language）语句包括 SELECT、INSERT、UPDATE 和 DELETE。像 CREATE TABLE 和 CREATE INDEX 这样的数据定义语言语句（DDL: Data Definition Language）通常是不被查询优化器处理的。这些语句通常是由静态 DBMS 逻辑通过调用存储引擎和目录管理器（在第 6.1 节描述）来实现的。一些 DBMS 产品也已经开始优化 DDL 语句的一个小子集，我们期待这个趋势将会持续。

4.1 查询解析和授权

给定一个 SQL 语句，SQL 解析器主要任务是： (1) 检查这个查询是否被正确地定义；(2) 解决名字和引用；(3) 将这个查询转化为优化器使用的内部形式；(4) 核实这个用户是否被授权执行这个查询。一些 DBMS 将一些或者全部安全检查延后到查询执行时才去做，但是，即使是在这些系统中，解析器仍然负责为查询执行时的安全检查收集所需要的数据。

给定一个 SQL 查询，解析器首先考虑的是在 FROM 子句中每个表的引用。解析器把每个表名规范化为“服务器. 数据库. 模式. 表名”，这被称为“四部分名称”。不支持跨越多个服务器执行查询的系统，只需要把表名规范化为“数据库. 模式. 表名”，而对于每个 DBMS 只支持一个数据库的系统来说，只需要把表名规范化为“模式. 表名”。这种规范化是必须的，因为，用户必须依赖上下文的默认值，这就使得在查询具体化过程中允许单个部分的名称被使用。某些系统支持一个表可以拥有多个名字，称为表的别名，而这些同样需要被完全的表名所代替。

在规范化表名之后，查询处理器开始调用目录管理器，检查表是否被注册到系统目录中。在这一步中，处理器可能也将表的元数据缓存到内部的查询数据结构。基于表格的信息，处理器接着使用目录来保证属性引用是正确的。属性的数据类型被用来消除那些存在于重载函数表达式、比较操作符和常量表达式中的逻辑含义模糊性。例如，考虑表达式（EMP.salary*1.15）<75000。乘法函数和比较操作符的代码，以及假定的数据类型和字符串“1.15”及“75000”的内部格式，都将取决于 EMP.salary 属性的数据类型。这数据类型可能为一个整数，一个浮点数或者一个“money”的值。

其他的标准 SQL 语法检查也被使用，包括元组变量的一致性使用，通过集合运算符（UNION/INTERSECT/EXCEPT）相结合的多个表之间的兼容性，在聚合查询中 SELECT 列表中的属性的使用，以及子查询的嵌套等等。

如果查询被成功解析，下一阶段就是授权检查，以确保用户对查询中引用到的这些表、用户自定义的函数以及其他对象具有适当的权限（SELECT/DELET/INSERT/UPDATE）。一些系统在语句解析阶段执行授权检查。然而这并不总是可行的。例如，支持行级安全检查的系统，直到执行时间才能进行完全的安全检查，因为，安全检查可能依赖于数据值。尽管理论上授权可以在编译时间内静态验证，但是，推迟授权验证到查询计划执行时间是有好处的。将安全检查推迟到执行时间的查询计划，可以在用户之间共享，而且，当安全条件变化时，不需要重新对查询进行编译。因此，部分安全检查通常被推迟到查询计划执行的时候。

在编译期间对约束常量表达式进行约束检查，也是可能的。例如，一个 UPDATE 命令可能有一个形如 SET EMP.salary = -1 的子句。如果约束条件指定为正值，查询甚至没必要执行。但是，把这个检查工作推迟到执行时间，也是相当普遍的。

如果一个查询已经被解析并且通过了验证，那么这个查询的内部形式就会被传递到查询的重写模块进行更深入的处理。

4.2 查询重写

查询重写模块，或重写器，负责简化和标准化查询，而无需改变查询语义。它只能依靠查询目录中的元数据，而不能访问表中的数据。尽管我们说是重写查询，但大多数重写实际操作的是查询的内部表示，而不是原始 SQL 语句文本。查询重写模块通常输出一个查询的内部表示，这种输出形式和它接受作为输入的内部格式相同。

许多商业系统的重写器是一个逻辑组件，它的实际执行要么发生在查询解析的后期阶段，要么发生在查询优化的前期阶段。例如，在 DB2 中，重写器是一个单独的组件，然而在 SQL 服务器中，查询重写是作为一个查询优化器的前期阶段完成的。尽管如此，单独考虑重写器是很有用的，即使在所有的系统中并不存在显示的架构界线。

重写器的主要职责是：

- **视图重写**：处理视图是重写器主要的传统角色。对于在 FROM 子句中出现的每一个视图引用，重写器都会从目录管理器中检索出视图定义。然后重写查询，用这个视图所引用的表和谓词来替换这个视图，以及将任何对这个视图的引用替换为对这个视图中的表的列引用。这个过程是递归的，直到这个查询表达式里只有表、没有视图。这种视图重写技术，率先是 INGRES[85]为基于集合的 QUEL 语言提出的，在 SQL 上需要一些额外手段去正确处理重复消除、嵌套查询、空值和一些其他棘手的细节。
- **常量运算表达式**：查询重写可以简化常量运算表达式，例如，R.x<10+2+R.y 被重写为 R.X<12+R.Y。
- **谓词逻辑重写**：逻辑重写是应用在基于 WHERE 子句中的谓词和常量的。简单的布尔逻辑往往是用来改进“表达式”和“基于索引的访问方法的能力”这二者之间的匹配程度。例如，一个诸如 NOT Emp.Salary>100000 的谓词，可能被重写为 Emp.Salary<=100000。通过简单的满足性测试，这些逻辑重写甚至可能导致短路查询执行。例如，表达式 Emp.salary<75000 AND Emp.salsary >1000000，可以被 FALSE 替换。这就允许系统返回一个空的查询值，而无需访问数据库。不可满足的查询可能看起来令人难以置信，但是回想一下，谓词可以被“隐藏”在视图定义中，而且不被外部查询的作者知道。例如，上述的查询，可能由于在一个称为“高管”的视图上查询“工资收入较低的员工”而导致的。在 Microsoft SQL Server 并行安装中，不可满足的谓词也形成了“分区消除”的基础：当一个关系通过区间谓词被水平跨

磁盘进行分区时，如果它的区间分区谓词和查询谓词的合取是不可能满足的，那么查询就没必要在一个卷上运行。另一个重要的逻辑重写使用谓词传递性来引入新的谓词。例如，R.x<10 AND R.x=S.y ，暗示着额外添加了一个谓词“AND S.y<10”。增添这些传递谓词增加了优化器选择方案的能力，这些方案在执行的早期阶段就可以过滤数据，尤其是通过使用基于索引的访问方法。

- **语义优化**：在许多情况下，模式的完整性约束是储存在目录中的，可以被用来帮助重写一些查询。这种优化的一个重要例子就是冗余连接消除。这种情形发生在一个外键约束把一个表的某一列（例如，Emp.deptno）绑定到另一个表（Dept）的时候。给定一个这样的外键约束，我们知道，对于每一个 Emp，只有一个 Dept 与之相对应，而且当缺少与 Emp 相对应的 Dept 元组（父母）时，Emp 元组是不可能存在。考虑一个连接两个表但没有使用 Dept 列的查询：

  SELECT Emp.name，Emp.salary

  FROM Emp,Dept

  WHERE Emp.deptno=Dept.dno

  这种查询可以被重写，从而删除 Dept 表（假定 Emp.deptno 被限制为非空），因此，就可以删除这个连接操作。同样，这种看起来令人难以置信的情形通常很自然地发生在视图上。例如，一个用户可能提交一个员工属性的查询，这个属性来自于连接两个表的视图 EMPDEPT。像 Siebel 这样的数据库应用程序使用非常宽的表，它们的底层数据库不支持足够宽度的表，它们就使用多个表和一个基于这些表的视图。如果缺少冗余连接消除机制，这个以视图的方式实现宽表，将会表现出很差的性能。当表的约束条件与查询谓词不兼容时，语义优化器也可以完全避免查询执行。

- **子查询的平面化和其他启发式重写**：在当代的商业数据库管理系统中，查询优化器是最复杂的组件之一。为了把这种复杂性控制在一定程度以内，大多数优化器都独立地优化单个 SELECT-FROM-WHERE 查询块，并不跨块优化。因此，许多系统把查询重写为一种更适合优化器的形式，而不是去想办法使优化器变得更加复杂。这种转变有时候称为查询规范化。规范化的一种类型就是把语义等价查询重写为规范化的形式，尽量确保语义等价查询被优化后可以产生相同的查询计划。另一个重要的启发式方法就是平面化嵌套查询，这样就可以最大程度地为查询优化器单块优化提供机会。由于重复语义、子查询、空值和相关性等问题，在 SQL 中的一些情况下，这将会是非常棘手的。在早期的时候，子查询平面化是一个纯粹的启发式重写，但是，现在一些产品

已经将重写决策建立在代价分析基础之上。其他跨块查询的重写也是可能的。例如，谓词传递性允许谓词在子查询之间被复制[52]。平面化相关的子查询，对于在并行架构上实现良好的性能是尤其重要的：相关子查询会导致“嵌套循环”式的、查询块之间的比较，这将会序列化执行子查询，尽管有可用的并行资源。

4.3 查询优化器

查询优化器的工作就是将一个内部的查询表示转化为一个高效地查询执行计划（如图4-1 所示）。一个查询计划可以被认为是一个数据流图，在这个数据流图中，表数据会像在管道中传输一样，从一个查询操作符（operator）传递到另一个查询操作符。在许多系统里，查询首先被分解为 SELECT-FROM-WHERE 查询块。每个单独的查询块的优化都是使用一些技术来完成的，这些技术与作家 Selinger et al 在 System R 优化器[79]的论文中描述的技术类似。在完成的时候，一些简单的操作符通常被添加到每个查询块的顶部，作为后处理来计算 GROUP BY、ORDER BY、HAVING 和 DISTINCT 子句。然后，不同的块就用一种简单的方式拼合在一起。

生成的查询计划可以表示成多种方式。原始的 System R 的原型系统，将查询计划编译成机器码，而早期的 INGRES 原型系统则生成一种可解释的查询计划。在 19 世纪 80 年代，INGRES 的作者在他的综述论文[85]里，将“可解释的查询计划”视作一个“错误”，但是，摩尔定律和软件工程在一定程度上证明 INGRES 的这种观点是正确的。讽刺的是，在 System R 项目中，一些研究员将“编译成机器码”视作一个错误。

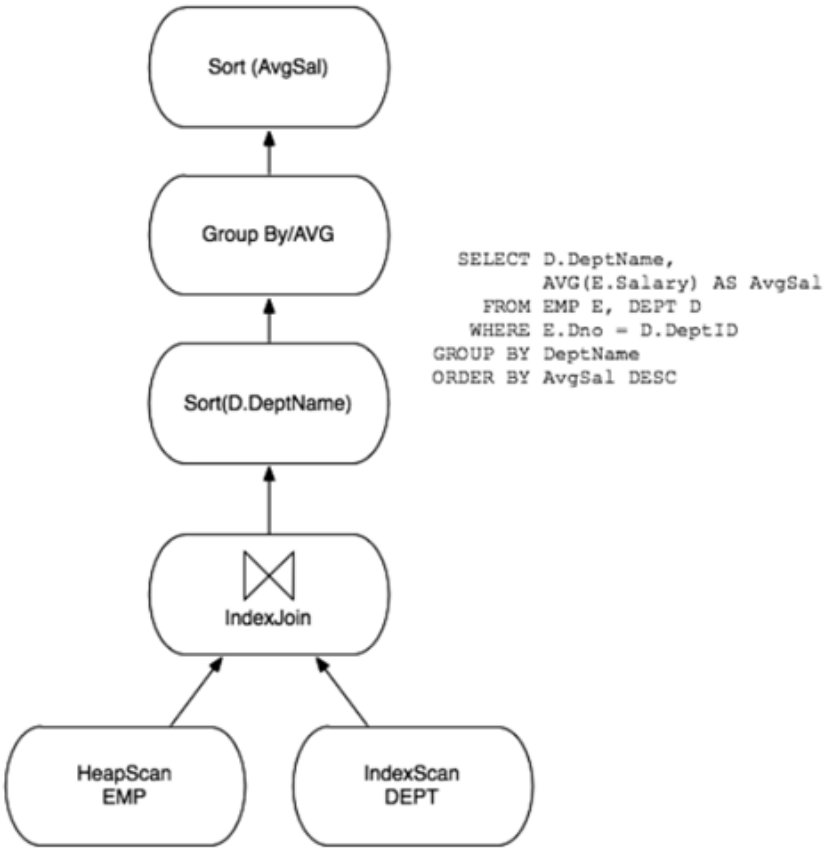

图 4-1 一个查询计划

当系统的代码库被制作成商业的数据库管理系统（SQL/DS）的时候，开发团队做出的第一个改变就是用一个解释器来替换机器代码执行器。

为了能够实现跨平台的可移植性，现在每一个主要的 DBMS 都将查询编译成某种可解释的数据结构，它们之间唯一的区别是中间形式的抽象级别。在某些系统中，查询计划是一个非常轻量级的对象，未必不可能是一个关系代数表达式（是由访问方法名称和连接算法等来表示的）。其他系统使用更低层次的“操作码”语言，相比于关系代数表达式，这种语言在思想上更像 Java 字节码。为了简单起见，在之后的讨论中，我们专注于类代数查询表示。

虽然 Selinger 的论文被广泛地认为是查询优化领域的“圣经”，但是，毕竟也只是初步的研究。所有系统在许多不同的角度上都显著地拓展了这篇论文的工作，主要的拓展有：

- **计划空间**：System R 优化器通过只专注于“left-deep”查询计划（一个连接操作的右手边的输入必须是一个基表），以及“推迟笛卡尔积”（保证在数据流中，求笛卡尔积的操作是出现在所有的连接之后），来限制查询计划空间。在当今的商业系统中，在一些情况下，“浓密的树”（具有嵌套式的右手边输入）和尽早计算笛卡

尔积是很有用的。因此，在大多数系统中，这两个选择在某些情况下是被考虑使用的。

- **选择性估算（selectivity estimation）**：Selinger 论文中的选择性估算技术是基于简单的表和索引基数，如果按照当今系统的标准，这种选择性估算技术是很初级的。如今，大多数的系统利用直方图和其他简易的统计数据来分析和概括属性值的分布。因为这涉及到访问每一个列值，所以代价比较昂贵。因此，一些系统使用抽样技术来得到大概的属性值分布，而无需付出完全扫描每一列值的代价。适合于基本表连接操作的选择性估算，可以通过把连接列上的直方图连接起来来实现。为了摆脱单列直方图的束缚，一些包含列之间的依赖性等问题的、更复杂方案[16,69]最近被提出来了。这些创新已经出现在商业产品上，但还有待取得更多的进步。这些机制被缓慢采用的一个原因是，在许多工业测试基准（benchmark）中存在一个长期的缺陷：如 TPC-D 和 TPC-H 基准数据发生器所生成的每一例，在数值的分布上具有统计独立性，因此，就不提倡采用那些用来处理“真实数据分布”的技术。这个测试基准的缺陷已经在 TPC-DS 测试基准[70]中得到解决。尽管被采用的步伐很慢，但是，改进的选择性估算所带来的好处还是被广泛认可的。Ioannidis 和 Christtodoulakis 注意到，在优化过程的早期的选择性估算错误，会被查询计划树成倍放大，以至于最后会得到一个非常糟糕的估算结果[45]。
- **搜索算法**：在一些商业系统中，特别是 Microsoft 和 Tandem，放弃了 Selinger 动态规划优化方法，转而支持一种基于级联式技术[25]的、目标导向的、自顶而下的搜索方案。在一些情况下，自顶向下搜索可以降低一个优化器需要考虑的计划的数量[82]，但是，同时产生了负面影响，即增加了优化器内存消耗。如果实践成功与否是衡量一个技术的质量的标准，那么，选择自顶向下搜索，还是选择动态规划优化方法，都是无关紧要的，因为，二者都被证实可以在先进的优化器上运行得很好，同时，不幸的是，它们的运行时间和内存需求，都与一次查询中所涉及表的数量成指数关系。针对涉及到太多表的查询，一些系统会采用启发式查询机制。尽管随机查询优化启发式方法的研究文献[5,18,44,84]是很有趣的，但是，在商业系统中使用的启发式方法是被申请了专利的，它们显然不同于那些研究文献中的随机查询优化启发式方法。一个教育实践就是去检查开源 MySQL 引擎的查询优化器，它在最后的检查是完全启发式的，而且大部分依赖于利用索引和键/外键约束。这让人想起了早期的、臭名昭著的 Oracle 版本。在一些系统中，在 FROM 子句中涉及到太多

表的查询只能在以下情况下才能运行，即用户必须明确指示优化器如何选择一个方案(通过嵌入在 SQL 中所谓的优化器“暗示”来实现)。

- **并行**：如今每一个主流的商业数据库管理系统都对并行处理有一定的支持。大多数也支持“查询内”并行：通过多处理器的使用来加速一个查询。查询优化器需要参与决定，如何在多个 CPU 之间以及在多个独立的计算机之间（在无共享或共享磁盘的情况下）调度操作符（被并行化的操作符）。Hong 和 Stonebraker[42]选择避开并行优化复杂性问题，使用两个阶段来实现：首先，传统的单系统优化器被调用来选择最好的单系统方案；然后，这个计划在多个处理器和机器上被调度。尽管不明确这些结果会在多大程度上影响当前的实践，但是，关于第二个优化阶段的研究文献[19,21]已经被发表。一些商业系统实现了上述描述的两阶段方法。其他一些系统则努力对集群网络拓扑结构以及集群机器之间的数据分布进行建模，然后，在一个阶段内产生一个最好的方案。虽然单阶段方法可以被证明在某些情况下产生更好的方案，但是，有一点仍然不明确，那就是，使用单阶段的方法产生的查询方案质量改进，相对于由此带来的额外的优化器复杂性而言，是否是值得的。因此，许多当前的系统，在实现时仍然倾向于采用两阶段方法。当前，这个领域看起来更像是艺术而非科学。Oracle OPS(如今称为 RAC)共享磁盘集群使用两阶段优化器。IBM DB2 并行版本（如今称为 DB2 数据库分区特色）一开始使用两阶段优化器来实现，但是，如今已经演变为单阶段来实现。
- **自动调优（tuning）**：各种各样正在进行的工业研究努力，尝试着改善 DBMS 来自动执行调优决策。这些技术中的一部分是基于收集查询负载，然后通过各种“what-if”分析来使用优化器来确定查询计划的代价，例如，如果其他索引已经存在时该怎么办。正如 Chaudhuri 和 Narasayya 描述的那样[12]，一个优化器需要在一定程度上被调整来高效地支持本次查询活动。Markl 等人的学习优化器（LEO）的工作[57]，也是这种类型。

4.3.1 一个查询编译和重新编译的标注

SQL 支持“预处理”查询的能力：将查询传递到解析器、重写器和优化器，把生成的查询执行计划存储起来，并且在后续的“执行”语句中使用。这可能同样适用于用程序变量代替查询常量的动态查询（例如，来自网页表单）。唯一的麻烦就是在选择性估算期间，由

对于那些由表单提供的变量，优化器将会采用“典型”的值。当无代表性的“典型”值被选择时，就会导致很差的的查询执行计划。查询预处理对于表单驱动以及在一些可预测数据的查询是很有用的。当应用程序被编写，查询会被预处理，当应用程序上线时，用户就没有必要经历解析、重写和优化，不会产生这些方面的代价。

尽管在编写一个程序时进行查询预处理是可以提高性能的，但是，这是一个非常有局限性的应用程序模型。许多应用程序开发者，以及像 Ruby on Rails 这样的工具箱，在程序执行期间动态构建 SQL 语句，因此，无法提供预编译。因为这种情况非常普遍，DBMSs 就在查询计划的缓存中存储这些动态查询执行计划。如果相同的（或者非常相似的）语句在随后被提交，那么就使用缓存里的版本。这项技术近似于预编译静态 SQL 的性能，不会受到应用程序模型的限制，因此，被大量使用。

随着数据库的演化，通常需要重新优化预编译计划。至少当一个索引被删除的时候，任何使用这个索引的执行计划都必须从存储计划缓存中删除，以保证在下次调用的时候，选择一个新的执行计划。

其他关于重新优化计划的设计决定是更加微妙的，显示了数据库厂商之间的设计理念区别。一些数据库厂商（例如 IBM）非常努力地在多次调用中获得可预测的性能，而不是每次调用查询执行计划都获得最优的性能。因此，在类似删去索引的情况下，供应商不会重新优化一个查询执行计划，除非该计划不再执行。其他供应商（例如 Microsoft）非常努力地使它们的系统实现自我调优，而且将会更加积极地重新优化执行计划。例如，如果一个表的基数发生了显著的变化，在 SQL 服务器中就会触发重新编译过程，因为，这种变化可能会影响到索引和连接顺序的最优使用。一个自我调优的系统是难以预测的，但是在动态环境中更加有效。

这种设计理念的区别主要来自于这些产品的历史客户群体基础的不同。IBM 传统上注重于有熟练技能的数据库管理员这样的高端用户和应用程序开发者。在这样的高预算的 IT 商店，从数据库中获得可预测性能是非常重要的。经过几个月的调优数据库设计和设置，数据库管理员不希望优化器不可预测地改变这些设置。相比之下，微软公司则战略性地进入了数据库的低端市场，他们的客户群体趋向于拥有较低的 IT 预算和专业知识，希望 DBMS 尽可能地“自我调优”。

随着时间的推移，这些公司的商业战略和客户基础逐渐融合，因此，这些公司就发生了直接的竞争，这使得他们的方法开始走向融合。微软拥有大规模的企业用户，这些用户想

要完全地控制和查询计划稳定性。IBM 有一些客户没有数据库管理员，所以，需要完全的自动控制。

4.4 查询执行器

查询执行器操作一个完全具体的查询计划。这通常是一个把很多操作连接在一起的数据流图，而这些操作封装了基本表（base table）的访问和各种查询执行算法。在一些系统中，这个数据流图已经被优化器编译成低级的操作码。在这种情况下，查询执行器基本上是一个运行时解释器。在其他系统中，查询执行器接收到一个数据流图的表示，然后递归调用基于图布局的操作程序。我们专注于后面这种情况，因为，操作码的方法在本质上就是把我们在这里描述的逻辑编译为一个程序。

大多数当代的查询执行器使用迭代器模型，该模型曾经使用在最早期的关系型系统中。迭代器大多数仅仅被描述为面向对象的形式。图 4-2 展示了一个迭代器简化的定义。每一个迭代器都规定了它的输入，即数据流图的边。

```
class iterator {
  iterator &inputs[];
  void init();
  tuple get_next();
  void close();
}
```

图 4-2 一个迭代器超类的伪代码定义

在一个查询计划中的所有操作——数据流图中的节点——都会被实现为迭代器类的一个子类。在一个典型的系统中，子类的集合可能包括文件扫描、索引扫描、排序、嵌套循环连接、合并连接、哈希连接、重复消除和分组聚类。迭代器模型的一个重要特性就是，迭代器任何一个子类可以作为其他子类的输入来使用。因此，在数据流图，每一个迭代器的逻辑是独立于它的子类和父母类的，不需要使用专门的代码对这些迭代器进行组合。

Graefe 在他的查询执行综述论文[24]中提供了更多关于迭代器的细节。建议感兴趣的读者去研究开源 PostgreSQL 代码库。PostgreSQL 使用了适度复杂的迭代器实现，而这些迭代器适用于大多数标准查询执行算法。

4.4.1 迭代器讨论

迭代器的一个重要性能就是，它们连接了数据流和控制流。get_next()调用是一个标准过程调用，该调用通过调用堆栈给调用者返回一个元组引用。因此，当一个控制返回的时候，一个元组就返回给数据流图的父类。这意味着，只需要一个单一的 DBMS 线程来执行一个完整的查询图，迭代器之间的队列和速率匹配是不需要的。这就使关系型查询执行器可以顺利地实现功能，并且易于调试，同时，也与其他环境中的数据流架构形成了鲜明的对比，例如，网络就需要在并发的生产者和消费者之间设计各种各样的队列和反馈协议。

单线程迭代器架构对于单系统（非集群）查询执行同样是高效的。在大多数数据库应用程序中，判别是否高效的性能指标是查询完成的时间，但是，也可能采用其他优化目标。例如，最大化 DBMS 的吞吐量是另一个合理可行的目标。在很多交互式应用程序的数据库系统中，会把到达第一行的时间作为性能指标。在一个单一处理器的环境中，当所有资源被完全利用的时候，把一个给定查询计划的完成时间作为优化目标是可以实现的。在一个迭代器模型中，因为其中一个迭代器总是处于活跃状态，所以，资源利用率是最大化的。

正如我们之前提到的，大多数当代的 DBMS 支持并行查询执行。幸运的是，基本上可以不用对迭代器模型和查询执行架构做任何修改就可以实现这种支持。并行性和网络通信可以被封装在 Graefe 描述的特殊交换迭代器中[23]。这些也实现了网络式的数据“推”操作，并且实现方式对于 DBMS 迭代器而言是不可见的，这些迭代器保留了“拉”式的 get_next() API。一些系统也在它们的查询执行模型中显示地确定“推”操作。

4.4.2 数据在哪里？

为了讨论方便，我们对迭代器的讨论已经避开了任何关于正在使用的数据的内存分配问题。我们既没有详细说明元组是如何在内存中储存的，也没有说明数据是如何在迭代器之间传递的。实际上，每一个迭代器都预分配了固定数量的元组描述符（tuple descriptor），每个输入对应一个元组描述符，输出对应一个描述符。一个元组描述符通常是一个关于“列引用”的数组，这个引用数组的每一个列引用，包括对内存中某个元组的引用和那个元组的列偏移。基本的迭代器超类的逻辑程序，永远不会动态地分配内存。这就提出了一个问题，被引用的真实元组到底被存储在内存中什么地方了。

对于这个问题，有两种可能的答案。第一种是，那个元组驻留在缓冲池的页面中，我们称这些为缓冲池元组。如果一个迭代器构造了一个引用缓冲池元组的元组描述符，那么它就必须增加那个元组所在的缓冲池页面的引脚数（pin count），即在那个页面上的元组的活跃（active）引用数量。当元组描述符被销毁的时候，迭代器就减少引脚数。第二种可能就是，一个迭代器实现可能为在内存堆中的元组分配空间。我们称这个为内存元组。一个迭代器可能通过复制缓冲池的列来构造一个内存元组，或者通过求查询中的表达式来构造一个内存元组（例如，像“EMP.sal*0.1”这样的表达式）。

一个通用的方法就是总是将缓冲池中的数据复制到内存元组中。这个设计使用内存元组作为唯一的查询时使用的元组结构，从而简化了执行器代码。这个设计也可以避免一些缺陷，这些缺陷来自于在缓冲池中分开执行 pin 和 unpin 调用的时间间隔很长（比如由于包含多行代码）。一种常见的错误是完全忘记对页面执行 unpin 操作（产生缓冲区泄漏）。不幸的是，正如第 4.2 节描述的那样，专门内存元组会成为一个主要性能问题，因为，在一个高性能系统中，内存副本通常是一个瓶颈。

另一方面，在某些方面构造内存元组是很有意义的。只要一个迭代器直接引用一个缓冲池元组，这个缓冲池元组所在的页面必须在缓冲池中固定不动。这就消耗一个页的缓冲池内存，而且束缚了缓冲替换策略作用的发挥。如果一个元组将长期被引用，那么将该元组从缓冲池复制出来是很有好处的。

该讨论的要点就是，同时支持缓冲池元组和内存元组的元组描述符是最有效方法。

4.4.3 数据修改语句

到目前为止，我们只讨论查询，即只读 SQL 语句。另一种数据操作语言是为修改数据而存在的，包括 INSERT、DELETE 和 UPDATE 语句。这些语句的执行计划通常看起来像简单的直线查询计划，即把单个访问方法作为源头，一个数据修改操作符作为数据流管道的尾部。

然而，在一些情况下，这些计划同时查询和修改同一个数据。这种针对同一个表的读和写混合操作（可能多次），需要格外小心。一个简单的例子就是声名狼藉的“万圣节问题”，因为，它是在由 System R 小组在 10 月 31 日发现的。万圣节问题是由像“给每个工资低于 20K 美元的人增加工资 10%”这样的语句的特殊执行策略而产生的。这个查询的朴素执行计划，会把 Emp.salary 域上的索引扫描迭代器以管道的方式输入给一个更新迭代器（如图

4-3 的左边所示，图 4-3 中左边的计划是容易发生万圣节问题的，右边计划是安全的，因为在执行任何更新之前，它会首先确定所有需要进行更新的元组）。这个管道提供了很好的 I/O 局部性，因为它只在元组被从 B+-树中获取到以后才对元组进行修改。然而，这个管道也可能导致索引扫描在修改结束之后，“重修发现”一个之前修改过的元组在 B+树上向右移动，导致每一个员工多次加薪。在我们的例子中，所有低薪员工将会收到重复的加薪，直到他们的收入超过 20K 美元。这不是这个语句的真实意图。

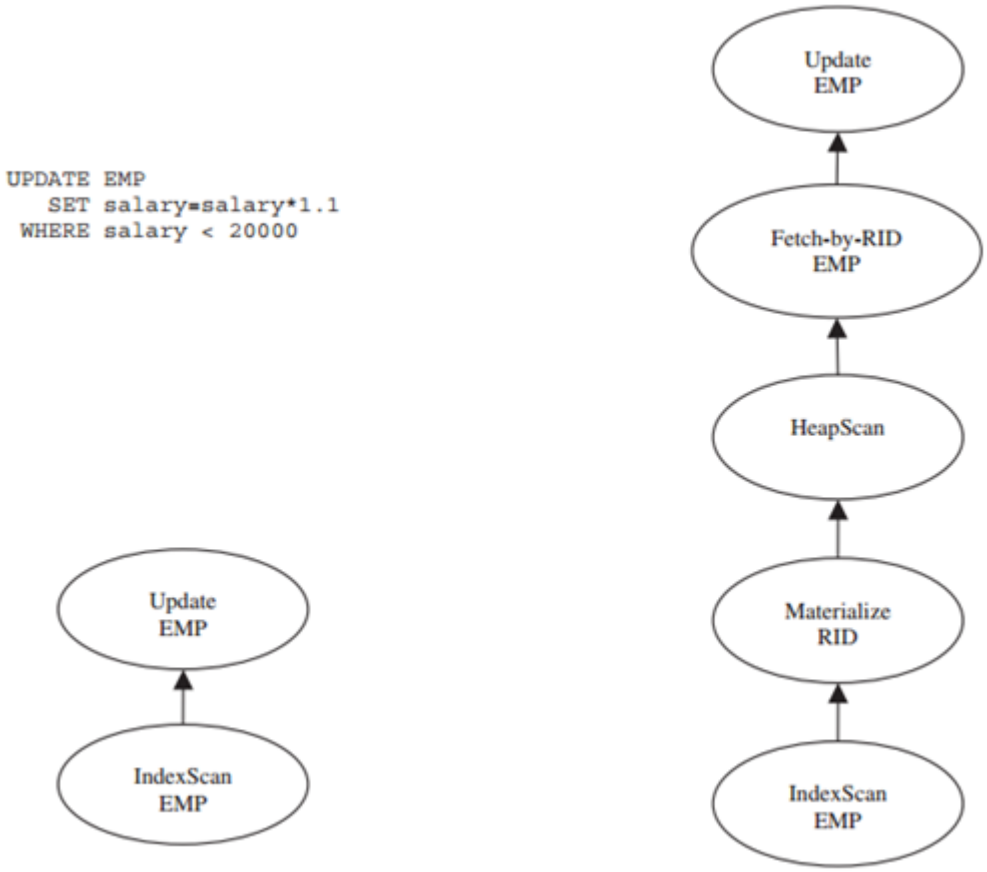

图 4-3 通过 IndexScan 更新一个表的两种查询计划

SQL 语义禁止这种行为：一个单一的 SQL 语句是不被允许“看到”自己的更新的。一定要小心保证遵守这个“可视”规则。一个简单、安全的实现，可以使得查询优化器所选择的计划能够避免对更新列进行索引。这在某些情况下是很低效的。另一种技术是使用批量的“先读后写”方案。这就需要在数据流图中（图 4.3 的右边），在索引扫描和数据修改操作符之间插入记录 ID 物化（materialization）操作符和数据抓取操作符。物化操作符接收所有需要修改的元组的 ID，并将它们存储在临时文件中。接着，物化操作符扫描临时文件，通过 RID（记录 ID）获取每一个物理元组 ID，并将结果元组提供给数据修改操作符。如果优化器选择一个索引，在大多数情况下，这会意味着只有一小部分元组发生改变。因此，这种

技术的明显低效率是可以被接受的，因为，临时表很可能完全保留在缓冲池中。管道化的更新机制也是可能的，但是，需要来自存储引擎的多版本支持[74]。

4.5 访问方法

访问方法是用来对系统支持的基于磁盘的数据结构的访问进行管理的，通常包括无序的文件（“堆”）和各种各样的索引。所有的商业系统都实现了堆和 B+树索引。Oracle 和 PostgreSQL 同时都支持“平等查找”（equality search）的哈希索引。一些系统开始引入对类似 R-树这样的多维索引的初步支持。PostgreSQL 支持一种叫做 Generalized Search Tree[39]的可扩展性索引，当前使用它来实现多维数据的 R-树索引以及针对文本数据的 RD-树索引[40]。第 8 个版本的 IBM UDB 引入了多维分簇索引[66]，通过多个维度上的区间来访问数据。以读操作为主的数据仓库负载通常采用专用的、索引的位图变种，正如我们在第 4.6 节描述的那样。

访问方法提供的基本 API 是一种迭代器 API。Int()例程会被扩展，从而可以接受一种列操作符常量形式的“搜索参数”（或者在 System R 术语中被称为 SARG）。一个 NULL SARG 被看成一个扫描表中所有元组的请求。当再也没有满足搜索参数的元组时，在访问方法层调用 get_text()将返回 NULL 值。

这里有两个原因将 SARG 传递给访问方法层。第一个原因是很清晰的，像 B+-树这样的索引访问方法需要 SARG 来高效地运行。第二个原因是更细微的性能问题，但是，适用于堆扫描和索引扫描。假定 SARG 是由调用访问方法层的例程来检查的。那么，每次访问方法从 get_next()返回时，它必须：（a）返回一个指向“驻留在缓冲池的某个帧中的元组”的句柄，并且固定住那个帧中的页面，从而避免发生页的替换；或者（b）复制一份元组。如果调用者发现 SARG 不满足，它就要负责：（a）减少那个页面上的 pin 数量，或者（b）删除复制的元组。然后必须重新调用 get_next()，以尝试在页面中的下一个元组。这个逻辑程序在函数“调用/返回”对（pair）上，消耗了较多数量的 CPU 周期，将会导致在缓冲池中“固定”（pin）了不必要的页面（产生了缓冲池帧的不必要的竞争），或者不必要地创建和销毁元组副本——当流式地通过数百万的元组的时候，这会是一个巨大的 CPU 开销。需要注意的是，一个典型的堆扫描将会访问给定页面中所有的元组，导致每个页面中这种交互的多次迭代。相比之下，如果所有这种逻辑在访问方法层就完成了，那么重复的“调用/返回对”和“销毁/不销毁或者复制/删除”，可以通过下面的方式来加以避免，即在测试 SARG

时一次测试一页，并且对于满足该 SARG 的元组，只从 get_next()调用返回。SARG 在存储引擎和关系型引擎中保着清晰的边界，同时保持着极好的性能。因此，许多系统支持非常丰富的 SARG 而且广泛使用它们。在主题层面上，这是关于在集合中的多个项目之间分摊工作的标准 DBMS 智慧的很好的一个实例，但是，在这种情况下，它是为了获得更好的 CPU 性能，而不是磁盘性能。

所有的 DBMS 需要某个方法来指向基本表的行，这样索引条目就可以恰当地引用行。在许多 DBMS 中，这是通过使用直接的行 ID（RID）来实现的，这些行 ID 是基本表的行在磁盘中的物理地址。这样做的好处就是速度快，但是，也带来了负面影响，即造成基本表的行在移动时开销较大，因为，所有指向这行的二级索引都需要更新。查找和更新这些行的代价都是很高的。当一个更新改变了行的大小，导致当前页面空间无法容纳更新后的行时，行就需要移动。当一个 B+-树分裂时，许多行也是需要移动的。DB2 使用一个前向指针来避免第一个问题。这就需要第二次 I/O 来找到移动的页面，但是，避免了更新二级索引。DB2 仅仅通过“不支持 B+-树作为基本表元组的主要存储”来避免第二个问题。Microsoft SQL Server 和 Oracle 支持 B+-树最为主要存储，就必须能够高效解决行的移动问题。采取的方法就是避免在二级索引中使用一个物理行地址，而是使用行主键（如果表没有一个主键，就提供额外的系统位来保证行的唯一性）而不是物理 RID。当使用二级索引来访问基本表行的时候，这会牺牲一些性能，但是，却可以避免一些由行的移动而导致的问题。Oracle 通过使用一个物理指针和主键来避免在某些情况下这种方法带来的性能损失。如果行没有移动，那么使用物理指针就可以很快地找到该行。但是，如果行移动了，那么将使用缓慢主键技术。Oracle 通过允许行跨越页面来避免在堆文件中移动行。因此，当一行被更新为更长的值，以至于不再适合存储在原来页面的时候，Oracle 并没有迫使去移动该行，而是采取如下措施：存储在原来页面的部分仍继续存储在原来页面，剩下的部分可以存储在下一个页面。

与所有其他的迭代器相比，访问方法和并发性、事务恢复逻辑等都有很深的交互，正如在第四章描述的那样。

4.6 数据仓库

数据仓库是服务于决策支持的、包含大量历史数据的数据库，OLTP 系统中的数据更新会被周期性地加载到数据仓库中。数据仓库已经发展到需要专业查询处理的支持，在下一章

内容中，我们将会总结一些它们需要具备的关键特性。数据仓库作为相关话题放在这里讨论，主要有两个原因：

- 数据仓库是 DBMS 技术的一项非常重要的应用。有些人声称数据仓库占了所有数据库管理活动的 1/3[26,63]。
- 本章内容到目前为止所讨论的传统的查询优化和执行引擎，在数据仓库上无法获得较好地性能。因此，为了获得更好的性能，有必要对它们进行扩展和修改。

关系型数据库管理系统最早构建于 20 世纪 70 年代到 20 世纪 80 年代之间，用来满足业务数据处理应用需求，因为，这是那个时代最主要的需求。在 20 世纪 90 年代早期，出现了数据仓库和业务分析的市场，并且从那时起增长很快。

在 20 世纪 90 年代，联机事务处理（OLTP）已经取代了批量业务数据处理成为数据库使用的主导形式。此外，大部分 OLTP 系统有很多计算机操作员提交事务，要么来自和终端客户的电话交流，要么把纸上的数据进行输入。自动柜员机已经广泛流行，允许客户不用操作员干预就能直接进行交互。此类交易的响应时间对于生产力来说是至关重要的。这种响应时间需求只会变得更加迫切，并且在多样化的今天，互联网正在以终端客户的自助服务来取代操作员。同时，企业在零售领域想要捕捉所有的历史销售业务，并存储它们一到两年。买家可以通过这样的历史销售数据来找出什么是热销的，什么不是热销的。这些信息可以用来影响采购模式。同样，这些数据可以用来决定哪些商品用来促销，哪些商品寄回厂家。在使用数据仓库时，大家的共识就是，在销售领域采用历史数据仓库，可以进行更好的库存管理、货架管理和商店布局管理，由此带来的收益，会超过数据仓库本身的投入。

很明显，一个数据仓库应部署在独立于 OLTP 系统之外的硬件里。使用该方法，漫长的（通常是不可预测的）商务智能查询，就不会破坏 OLTP 的响应时间。同时，数据的本质是非常不同的，数据仓库处理历史数据，OLTP 处理“现在”的数据。最后，往往会发现历史数据模式经常和当前数据模式不匹配，这就需要进行数据模式的转换。由于这些原因，人们构建了工作流系统，从 OLTP 系统不断“刮取”（scrape）数据，装载到数据仓库中。这样的系统被命名为“提取、转换和加载”（ETL）系统。受欢迎的 ETL 产品包括来自 IBM 的 Data Stage 和来自 Informatica 的 PowerCenter。在过去的十年里，ETL 供应商已经将产品扩展到数据清洗工具、重复数据删除工具和其它以质量为中心的服务。接下来，我们必须讨论几个在数据仓库环境中必须解决的问题。

4.6.1位图索引

B+树为记录的快速插入、删除和更新进行了优化。相比之下，一个数据仓库执行原始的负载，然后数据就几个月或几年都是静态的。而且，数据仓库经常包含只有少数量值的列，例如，考虑一下存储用户的性别，性别只有两种值（男或女），而且在位图上，性别可以用一个位记录来描述。相比之下，一个 B+树的每一个记录将需要（值，记录指针）对，而且通常情况下，每一个记录需要消耗 40 位。

位图在合取谓词过滤器上也很有优势，例如，Customer.sex=”F”,和 Customer.state=”California”。在这种情况下，结果集是由相交的位图决定的。有许多更加精致的位图算法技巧，可以被使用来提高常见的分析查询的性能。对于位图处理的讨论，有兴趣的读者可以参考文献[65]。

在当前的产品中，位图索引为 Oracle 中的 B+树索引提供了很好的补充，而 DB2 提供了一个更为有限的版本。Sybase IQ 大量使用位图索引。当然，位图的缺点就是它们昂贵的更新代价，所以，它们只限制在仓库环境中使用，因为，数据仓库一般是只读的，数据写入后就不会发生更新。

4.6.2 快速下载

通常情况下，数据库在半夜加载白天的数据交易。零售场所只在白天开放是一个显而易见的策略。实行夜间批量加载的第二个原因是，为了避免用户交易过程中出现更新。考虑一下，一个业务分析师希望制定一些即席查询的方案，也许会查询飓风对顾客购买模式的影响。这个查询的结果可能要求后续的查询，如调查大风暴中的购买模式。这两个查询的结果应该是兼容的，也就是说，应该基于相同的数据集来计算得到答案。如果数据同时被加载，就会导致包含最近历史的查询出现问题。

因此，数据仓库的快速批量加载至关重要。虽然人们可以用 SQL 插入语句来编写数据仓库加载程序，但是，从来没有人在实践中使用过这种战术。相反，人们一般利用批量加载机（loader）把大量记录存储到数据仓库中，它没有 SQL 层的开销，并充分利用了面向 B+-树的特殊批量加载方法的优点。批量加载比 SQL 插入的速度大约快一个数量级，并且，所有主要的数据库厂商都提供了一个高性能的批量加载机。

随着电子商务和 24 小时营业的全球化，这种批量加载的战术意义不大。但是，在“实时”仓库的道路上还存在两个问题。首先，插入操作（无论是批量加载还是事务中的插入操作），必须设置写锁，如第 6.3 节中讨论的那样。这就会和查询所获得的读锁发生冲突，可能导致仓库“冻结”。其次，设置如上所述的查询之间提供一致的答案也是有问题的。

避免就地更新，提供历史查询，可以避免这两个问题。如果保持更新之前和之后的值，那么，适当的时间戳可以提供在最近的时间的查询。运行相同历史时间的集合查询，将会提供一致的答案。此外，同样的历史查询，可以在没有设置只读锁的时候运行。如在第 5.2.1 节中讨论的那样，一些数据厂商（特别是 Oracle 公司）提供了多版本（MVCC）隔离级别。由于实时仓库变得越来越流行，其它厂商大概会效仿。

4.6.3 物化视图

数据仓库通常是巨大的，并且，连接多个大表的查询似乎有种要运行到“永远”的倾向。为了提高热门查询的性能，大多数厂商提供物化视图。不像本章前面讨论过的纯粹的逻辑视图，物化视图是可以查询的实际表，而不是建立在真正基本数据表基础之上的逻辑视图表达式。一个物化视图的查询，可以避免在运行时执行视图表达式中的连接操作。物化视图必须保持最新，因为更新一直在进行。

物化视图的应用主要有三个方面：（a）选择要物化的视图；（b）保持视图的更新；（c）考虑在即席查询中使用物化视图。其中，（a）是自动数据库调优的一个先进方面，我们在第 4.3 节中已经提到；（c）在各种产品上实施的程度不同，这个问题在理论上具有挑战性，甚至对于简单的单块查询[51]而言也是如此，对于包含聚集和子查询的通用 SQL 而言更是如此；至于（b），大多数厂商提供多个刷新技术，比如，每次当物化视图所基于的源头表的数据发生更新时，物化视图都要执行更新，再比如，定期丢弃一个物化视图然后重新创建新的物化视图。这些策略提供了一个在运行时的开销和物化视图数据一致性之间的权衡。

4.6.4 OLAP 和 Ad-hoc 查询支持

一些仓库工作负载有可预见的查询。例如，在每个月的月底，一份总结报告的运行可能提供在零售链每个销售地区的部门的销售总额。很明显地，可以通过适当地构建物化视图来支持可预测的查询。更普遍的是，由于大多数业务分析查询需要聚合（aggregation）操作，

因此，人们可以为每家商店的销售总额按照部门来进行聚合计算，为计算结果生成物化视图。然后，如果上面的区域查询被选定，我们就可以对每个区域每个商店进行“上卷”操作来满足这个查询的要求。这种聚合结果往往被称为数据立方体（data cube），是物化视图中一个有趣的类。20 世纪 90 年代早期的产品，如 Essbase，提供了定制工具，采用优先立方体格式存储数据，并且提供了基于数据立方体的用户界面来浏览数据。这个功能后来被称为联机分析处理（OLAP）。随着时间的推移，数据立方体支持已经被添加到全功能的关系数据库系统中，通常被称为关系型 OLAP（ROLAP）。很多提供 ROLAP 的 DBMS 已经发展到可以在内部实现一些早期的 OLAP 类型的存储机制，这种机制有时会简称为 HOLAP（混合 OLAP）机制。

显然，数据立方体为可预见的有限的查询类提供了很高的性能。然而，它们对于即时查询而言一般没有什么帮助。

4.6.5 雪花模式查询的优化

许多数据仓库遵循一个特定的模式设计方法。一般而言，它们会存储一系列事实表，在零售业环境中，事实表通常就是简单的记录，例如“客户 X 在时间 T 从商店 Z 购买了商品 Y”。一个核心的事实表记录了每个事实的信息，例如购买价格、账户、销售汇率信息等等。同时，在事实表中还有维度集合的外键。维度包括客户、产品、商店、时间等等。这种形式的方案经常被称为“星型方案”，因为，它在中心有一个事实表，这个事实表周围是很多维表，每一个维表与事实表之间都存在 1-N 的主外键关联。画成实体关系图的形式，这样的模式就是星型的。

许多维度是自然地具有层级结构的。例如，如果商店可以被聚合到地区中，那么商店的“维表”就增加一个外键指向地区维表。对于涉及到时间（月/日/年）和管理级别等属性而言，通常也存在类似的层级结构。在这些情形中，会出现一个多层次星型模式，或者称为雪花模式。

基本上所有数据仓库查询需要对雪花模式的一个或多个维度，在这些维表的一些属性上进行过滤，然后将过滤结果与中央的事实表进行连接操作，接着根据事实表或维表的一些属性进行分组，最后计算一个 SQL 聚合结果。

随着时间的推移，供应商在他们的优化器中拥有特殊的查询类，因为它是如此的流行，为如此长时间执行的命令选择一个好的计划是很重要的。

4.6.6 数据仓库：结论

正如我们看到的那样，数据仓库需要提供不同于 OLTP 环境的能力。除了 B+-树，数据仓库还需要位图索引。数据仓库需要将重点放在基于雪花模式的聚合查询，而不是一个通用的优化器。数据仓库需要物化的视图而不是常规的视图。数据仓库需要快速的批量加载而不是快速的事务更新等等。在文献[11]中有内容更加丰富的、关于数据仓库操作实践的阐述。

主要的关系型数据库厂商从面向 OLTP 的架构开始，随着时间的推移，供应商已经增加了面向数据仓库的功能。另外，已经有各种各样的小型供应商在这个领域提供了 DBMS 解决方案。这包括 Teradata 和 Netezza，他们提供了在他们的 DBMS 上运行的、无共享的专有硬件。而且，这个领域还有 Greenplum（PostSQL 的并行性）、DATAllegro 和 EnterpriseDB，以上这些都运行在传统的硬件上。

最后，一些人（包括本文的其中一个作者）认为，相对于传统的行存储引擎（即存储单元是表的行）而言，列存储在数据仓库空间中拥有巨大的优势。当表是“很宽的”（很多的列），而且仅仅趋向于访问一些列时，单独存储每一列是尤其高效的。列存储也可以带来简单而有效的磁盘压缩，因为，列中所有的数据来自于同一种类型，具有更高的数据压缩率。列存储面临的挑战是，表格中每一行的位置需要与所有列保持一致，或者需要额外的机制来把这些列重新连接起来得到一个完整的记录。这对于 OLTP 是一个大问题，但是，对于像数据仓库或者系统日志库这样的追加数据库而言，就不是一个主要问题。像 Sybase、Vertica、Sand、Vhayu 和 KX 等数据库厂商，都提供了列存储。该架构讨论的更多细节可以在[36,89,90]中找到。

4.7 数据库扩展性

传统上，人们都认为关系型数据库存储的数据类型是有限的，主要集中在企业和行政机构记录保存中使用的“事实和数字”。然而，如今，关系型数据库支持主流的编程语言描述的多种数据类型。这是通过用多种方式使核心的关系型数据库管理系统进行扩展来实现的。在这节，我们简要地总结一下广泛使用的扩展方式，重点阐述一些在实现这种扩展性中产生的架构问题。这些特性不同程度地出现在如今大多数商业数据库管理系统中，也包括开源的 PostgreSQL 数据库关系系统。

4.7.1 抽象数据类型

原则上，关系模型对可放置在模式列上的标量数据类型是一无所知的。但是，最初的数据库系统只支持一组静态的字母数字列类型，而且，这种限制与关系模型本身是相关的。一个关系型数据管理系统在运行时可以被扩展支持新的抽象数据类型，正如在早期 IngresADT 中阐述的，在后来的 Postgres 系统中表现更为明显。

为了实现这点，DBMS 类型系统——以及解析器——必须从系统目录中驱动，系统目录保存了系统已知的类型列表，以及指向操作类型的“方法”（代码）的指针。在这种方法中，DBMS 不解释数据类型，它仅仅在表达式计算时恰当地调用它们的方法；因此叫作“抽象数据类型”。作为一个典型的例子，DBMS 可以注册一个 2 维空间“矩形”的类型，以及像矩形相交或合并的操作方法。这也意味着系统必须为用户自定义的代码提供一个运行时引擎，而且安全地执行那个代码，没有一丝导致数据库服务器崩溃或毁坏数据的风险。如今所有主要的数据库管理系统，都允许用户采用现代 SQL 的“存储过程”子语言来定义函数。除了 MySQL，大多数数据库管理系统至少支持一些其他语言，通常是 C 和 Java。在 Windows 平台上，Microsoft SQL Server 和 IBM DB2 支持代码编译到 Microsoft .Net Common Language Runtime，它可以用多种语言进行编写，较为普遍的是 Visual Basic,C++和 C#。PostgreSQL 本身就支持 C、Perl、Python 和 Tcl，而且允许在运行时向系统添加对新语言的支持——流行的第三方 Ruby 插件和开源 R 统计包。

为了使抽象数据类型在 DBMS 上高效地运行，查询优化器必须在选择和连接谓词上解释“昂贵”的用户自定义代码，在一些情况下推迟“选择”操作直到“连接”操作完成[13,37]。为了使抽象数据类型更加高效，在它们上定义索引是很有用的。至少，B+-树需要扩展到可以为抽象数据类型上的表达式进行索引，而不只是对列进行索引（有时候称为“函数索引”），必须对优化器进行扩展，从而可以使用这些建立在表达式上的索引。对于不是线性命令（<，>，=）的谓词，B+-树是不够的，系统需要支持一种可扩展的索引机制。文献中记载的两种方法是：原始 Postgres 可扩展性访问方法接口[88]和 GiST[39]。

4.7.2 结构化类型和 XML

ADTs（抽象数据类型）被设计成完全兼容关系模型——它们不以任何方式改变基本的关系代数，只改变属性值表达式。然而，在过去几年，出现了许多积极改变数据库来支持关

系型结构类型的建议：例如，嵌套集合类型，像数组、集合、树和嵌套元组以及/或者关系。如今，与这些提议最相关的就是通过 XPath 和 XQuery 等语言提供对 XML 的支持。大约有三种方法来处理 XML 这种结构化数据类型。第一种方法就是建立一个自定义数据库，该数据库可以操作结构化类型的数据。历史上，在传统关系型 DBMS 内部容纳这种结构化数据类型的方法，已经取代了上面所说的这种方法，而且这种趋势在 XML 出现之后继续延续着。第二种方法就是将复杂的数据类型看成一个 ADT。例如，可以定义一个包含 XML 类型的列的关系表，每一行存储一个 XML 文档。这就意味着，搜索 XML 的表达式——XPath 匹配树模式——需要以一种对于查询优化器而言很难懂的方式来执行。对于 DBMS 的第三种方法就是将嵌套结构规范化为一个关系集合，用外键来连接子类和它们的父类。这种技术有时候称为“分割”XML 文档，在一个关系框架内部将数据的所有结构暴露给 DBMS，但是，增加了存储开销，而且在查询时需要“连接”操作重新连接数据。大多数 DBMS 供应商提供 ADT 和存储分割选项，而且允许数据库设计者在它们之间做出选择。对于 XML 这种情形，在“分割”方案中提供去除同一级别的 XML 嵌套元素之间的的顺序信息的功能是很普遍的，它可以通过允许重排序和其他关系型优化来提高查询性能。

一个相关的问题是扩展关系型模型来处理嵌套表和元组以及数组。例如，这在 Oracle 安装上是广泛使用的。设计的取舍与处理 XML 的权衡在很多方面是相似的。

4.7.3 全文检索

传统上，关系型数据库在处理丰富的文本数据以及伴随使用到的关键词搜索方面是出了名的差的。原理上，在数据库上建立自由文本模型是一个简单的存储文档问题，可以定义一个“倒排文件”关系，关系中的每个元组采用（word,documentID,position）这种形式，并且在 word 列上建立一个 B+-树索引。这大致是在任何文本搜索引擎中所发生的事情，此外，还会建模一些单词的语言规范以及利用一些额外元组属性来辅助排序搜索结果。

除了这个模型，大多数文本索引引擎针对这种模式实现了许多性能优化，这在典型的 DBMS 上是没有实现的。这些优化措施包括对模式进行非规范化，例如，（word，list<documentID，position>），从而使得每个 word 在出现列表中只出现一次。这就可以支持列表的增量压缩，对于文档中的单词具备扭曲（Zipfian）分布的情形而言，增量压缩是很重要的。而且，文本数据库趋向于以数据库仓库类型来被使用，规避了任何 DBMS 事务逻

辑。一般而言，大家普遍认为，在一个 DBMS 中简单采用文本搜索引擎，要比采用定制的文本索引引擎大致慢一个数量级。

然而，如今大多数 DBMS 要么包含文本索引的子系统，要么附带一个单独的引擎来做这项工作。文本索引功能通常可以适用于全文文档和元组中较短的文本属性的搜索。在大多数情况下，全文索引一般采用异步更新，而不是采用事务性的更新维护。在这一点上，PostgreSQL 的做法有点不同寻常，它为事务更新情形提供了全文索引。在一些系统中，全文索引是存储在 DBMS 之外的，因此需要独立的工具来备份和还原。在关系型数据库中处理全文搜索，一个重要挑战就是在关系型语义（结果是一个无序的、完整的集合）与排序文档搜索（结果是有序的、不完整的）之间建立语义桥梁。例如，当两个关系都有一个关键字搜索谓词，如何对一个查询的、来自两个关系的结果进行排序，是不明确的。这在当前实践中仍然是临时确定的。给定一个查询输出语义，关系型查询优化器的另一个挑战就是分析文本索引的选择性和代价，以及判断一个查询的合适的代价模型（这个查询的结果集在用户界面上是有序和分页的，可能不能完全地检索）。根据各种报道，许多主流的 DBMS 正在积极地解决上述的最后一个挑战。

4.7.4 额外的可扩展性问题

除了数据库可扩展性的三个驱动使用场景，我们这里讨论引擎中的两个核心组件，它们经常被扩展后服务于各种各样的用途。

有许多关于可扩展查询优化器的建议，包括支持 IBM DB2 优化器[54,68]的设计和支持 Tandem 与 Microsoft 优化器[25]的设计。所有这些方案提供规则驱动的子系统，该子系统产生和修改查询计划，以及允许新的优化规则独立地注册。当新的功能被添加到查询执行器或者当关于查询重写或计划优化方面的新想法产生时，这些技术对于更容易地扩展优化器而言是很有用的。这些通用的架构在上述描述的许多具体的可扩展类型的功能方面是很重要的。

另一个在早期系统中出现的可扩展性的形式是数据库把远程数据源“打包”（wrap）到自身模式（schema）内的能力，从而使得这些远程数据似乎就是本地的表，而且可以在查询处理过程中访问它们。在这方面的一个挑战就是需要优化器去处理那些“不支持扫描、但是会响应把值赋给变量的请求”的数据源；这就需要对那些“可以把索引 SARG 匹配到查询谓词”的查询优化器进行扩展[33]。执行器的另一个挑战就是高效地处理远程数据源，该数

据源在产生输出方面可能缓慢，也可能很快；使查询执行器执行异步的磁盘 I/O 是一个很大的设计挑战，它会使得访问时间可变性[22,92]增加了一个以上数量级。

4.8 标准实践

基本上所有的关系数据库查询引擎的底层架构，看起来都与 System R 的原型很相似[3]。过去这些年，查询处理的研究和发展都把重点放在了在这个架构范围内的创新，从而可以增加更多种类的查询和模式。不同系统之间的设计区别，主要体现在优化器搜索策略（自顶向下，还是自底向上）和查询执行器控制流模型（尤其是对无共享和共享磁盘的并行性，应该采用迭代器和交换操作模型，还是采用异步的生产者/消费者模式)。在一个更细粒度层面上，在优化器、执行器、访问方面等方面，都有大量的不同机制组合方案，从而可以使得在不同的工作负载下都能获得好的性能，这些负载类型包括 OLTP、数据仓库和 OLAP。这个商业产品的秘方决定了它们在特殊情况下可以表现得有多好；几乎所有的商业系统都认为自己可以在各种不同类型的工作负载下做得很好，但是，实际上可能会在某种特殊的工作负载上面看起来比较慢。

在开放源码领域，PostgreSQL 有一个具有合理复杂性的查询处理器，它拥有一个传统的基于代价的优化器、一个执行算法集合和大量的还没有在商业产品中发现的扩展功能。MySQL 的查询处理器更简单，采用了基于索引的嵌套循环连接。MySQL 的查询优化器着重于对查询进行分析，从而确保常用操作是轻量级和高效的，尤其是主外键连接，“外连接到连接”的重写，以及只要求结果集的前几行的查询。仔细阅读 MySQL 手册和查询处理的代码、并把它与更多传统的设计进行比较，是有借鉴意义的（要知道，MySQL 在实践方面具有很高的市场普及率），同时也要了解 MySQL 执行哪些任务比较出色。

4.9 讨论和附加材料

因为查询优化和执行形成了比较清晰地模块，在过去这些年，这个领域里已经研究出了大量的算法、技术、技巧，并且关系型查询处理的研究到今天还在持续。值得高兴的是，大部分已被用于实践的想法（很多还没有），都可以在已经发表的研究文献中找到。如果想从事查询优化研究，一个比较好的研究起始点是阅读 Chaudhuri 的简短综述文献[10]。对于查询处理研究而言，Graefe 提供了一个很全面的综述论文[24]。

除了传统的查询处理，近几年有大量工作开始把丰富的统计方法融合到处理大型数据集中。一个很自然的扩展就是使用抽样或汇总统计为聚合查询提供数值估算[20]，可能是以一种持续改进的、在线的方式[38]。然而，尽管有相当成熟的研究成果，但是，市场在采纳这些研究成果方面仍然步伐缓慢。Oracle 和 DB2 都提供了简单的基表采样技术，但是，没有提供统计上包括多个表的、鲁棒性较好的查询估算。他们没有把重点放在这些特性上，相反，大部分数据库厂商把主要精力放在丰富他们的 OLAP 功能上面，这就约束了那些可以被很快回答、并且可以为用户提供百分之百准确性的查询的数量。

另一个重要但更根本的扩展，就是把数据挖掘技术引入 DBMS 领域。流行的技术包括统计聚类、分类、回归和关联规则[14]，除了文献中研究的这些技术的独立实现以外，把这些技术和关系查询的体系架构进行整合，也是一个很大的挑战[77]。

最后，值得注意的是，最近，数据并行使得整个计算机研究领域变得异常活跃，比较有代表性的是，Google 的 Map-Reduce、Microsoft 的 Dryad、被 Yahoo 所支持的开源 Hadoop 代码。这些系统很像无共享、并行的关系查询执行器，使用应用程序开发者自己编写的自定义的查询操作符。它们包含了简单而明智的设计方法，用来管理参与节点的失败，在大规模集群中，这种节点失败是一种常见的情况。或许这一领域最有趣的方面就是，被创造性地应用于很多种类的计算领域数据密集问题，包括文本和图像处理、统计方法等。我们将会很有趣地看到，是否其他来自于数据库引擎的方法会被这些框架的使用者借鉴，例如，在 Yahoo，早期的工作曾经对 Hadoop 进行扩展，使它拥有声明性查询和优化器。在这些框架上的创新，反过来，也可以被融合到数据库引擎中。

附录 1:译者介绍

林子雨(1978－),男,博士,厦门大学计算机科学系助理教授,主要研究领域为数据库,数据仓库,数据挖掘.
主讲课程：《大数据技术基础》
办公地点：厦门大学海韵园科研 2 号楼
E-mail: ziyulin@xmu.edu.cn
个人网页：http://www.cs.xmu.edu.cn/linziyu