Computer Science and Application 计算机科学与应用, 2013, 3, 159-164
http://dx.doi.org/10.12677/csa.2013.33028 Published Online June 2013 (http://www.hanspub.org/journal/csa.html)

The Corporate Mass Data Query Optimizer

Yong Ding

Yunnan Institute of Business and Technology, Kunming
Email: greenvc@gmail.com

Received: Jan. 19th, 2013; revised: Mar. 9th, 2013; accepted: Mar. 24th, 2013

Abstract: In an application system, data query and processing speed has become a standard to measure the success or failure of the system. Thus, in large database queries, you should pay attention to the efficiency of the data query, so as not to cause a serious waste of system resources. Based on the grammar-based query optimizer works, this paper reasonably establishes indexing, writes good specifications SQL statement, uses appropriate expressions or keywords, makes full use of the index, avoids full table scan and improves query efficiency.

Keywords: Optimize the Query; Index; SQL Statements

企业海量数据查询优化

丁　勇

云南工商学院，昆明
Email: greenvc@gmail.com

收稿日期：2013 年 1 月 19 日；修回日期：2013 年 3 月 9 日；录用日期：2013 年 3 月 24 日

摘　要：在一个应用系统中，对数据查询及处理速度已成为衡量该系统成败的标准，所以在对大型数据库查询时，应注意数据查询的工作效率，以免造成系统资源严重浪费。本文根据以语法为基础的查询优化器的工作原理，合理建立索引，书写规范良好的 SQL 语句，使用合适的表达式或关键字，充分利用索引，避免全表扫描，提高查询效率。

关键词：优化查询；索引；SQL 语句

1. 引言

数据库系统是管理信息系统的核心，基于数据库的联机事务处理(OLTP)和联机分析处理(OLAP)是各使用单位最为重要的计算机应用之一。从大多数系统的应用实例来看，查询、分析、统计是系统的最终应用，而查询、分析、统计操作所基于的 SELECT 语句在 SQL 语句中又是付出资源代价最大的语句。举个具体的例子，比如一个数据库表有上百万甚至上千万条记录，全表扫描一次往往需要数十分钟，甚至数小时。如果采用比全表扫描更好的查询策略，往往可以使查询时间降为几分钟甚至几秒钟，由此可见查询优化技术的重要性。

2. 数据库应用现状

随着网络与信息技术的突飞猛进，信息出现了爆炸式增长。地理信息系统及科考应用的工程数据库要求在 1 TB 以上；通信业的数据库要求 1 GB 以上；电子商务、产业信息化等企业级应用，尤其用到数据仓库技术的一类数据库，要求的支撑数据库也有几百、几千兆字节容量，数据库的并行访问需求大大提高，

系统要求支持多用户协同工作。面对海量数据，企业级应用的性能愈来愈依赖数据库的性能，企业海量数据库优化的重要性日益突现。

在数据量愈来愈大，并行访问的用户愈来愈多的情况下，经济有效地提高数据库系统的吞吐量和减少事务的响应时间成为数据库系统发展的关键问题。在企业级的实际解决方案中，在很大程度上应用合理与否决定数据库性能。要做到应用合理，语句是重点。是面向结果语言，用户只要学会抽象的语句，具体内部实现由厂商实现并改进。然而具体应用是否合理却是工程人员的事情，而且应用合理与否对性能影响明显。

3. 海量数据查询工作原理

一个好的查询计划往往可以使程序性能提高数十倍。查询计划是用户所提交的 SQL 语句的集合，查询规划是经过优化处理之后所产生的语句集合。DBMS(数据库管理系统)处理查询计划的过程是这样的：在做完查询语句的词法、语法检查之后，将语句提交给 DBMS 的查询优化器，优化器做完代数优化和存取路径的优化之后，由预编译模块对语句进行处理并生成查询规划，然后提交给系统处理执行，最后将执行结果返回给用户。在实际的数据库产品(如 MS SQLServer)的高版本中都是采用“基于语法的查询优化器”和“基于开销的查询优化器”。“基于语法的查询优化器”为获得对 SQL 查询的应答结果创建一个过程计划，但是它选择的特定计划取决于查询的确切语法及查询中的子句顺序。无论数据库中记录的数目或组合是否随时间变化而更改，基于语法的查询优化器每次都执行同样的计划。与基于开销的查询优化器不同，它不查看或维护数据库的统计记录。“基于开销的查询优化器”在备选计划中选择应答 SQL 查询的计划[1]。选择是基于对执行特殊计划的开销估算(I/O 操作数、CPU 秒数，等等)而做出的。它通过记录表或索引中记录的数目和构成的统计数字估算这些开销。与基于语法的查询优化器不同，它不依赖于查询的确切语法或查询中的子句顺序。虽然现在的数据库产品在查询优化方面已经做得越来越好，但由用户提交的 SQL 语句是系统优化的基础，很难设想一个原本糟糕的查询语句经过系统的优化之后会变得高效，因此用户所写语句的优劣至关重要[2]。“基于开销的查询优化器”的优化方法我们暂不讨论，下面重点说明“基于语法的查询优化器”的解决方案。

4. 优化策略

4.1. 合理使用索引

索引是数据库中重要的数据结构，它的根本目的就是为了提高查询效率。现在大多数的数据库产品都采用 IBM 最先提出的 ISAM 索引结构。索引的使用要恰到好处，其使用原则如下：

1) 在经常进行连接，但是没有指定为外键的列上建立索引。

2) 在频繁进行排序或分组(即进行 group by 或 order by 操作)的列上建立索引。

3) 在条件表达式中经常用到的不同值较多的列上建立索引，在不同值少的列上不要建立索引。比如在雇员表的“性别”列上只有“男”与“女”两个不同值，因此就没有必要建立索引。如果建立索引不但不会提高查询效率，反而会严重降低更新速度。

4) 如果待排序的列有多个，可以在这些列上建立复合索引(compound index)。

5) 不能用 null 作索引，任何包含 null 值的列都将不会被包含在索引中。也就是说如果某列存在空值，即使对该列建索引也不会提高性能。

6) 对查询型的表，建立多个索引会大大提高查询速度，对更新型的表，如果索引过多，会增大开销。

4.2. 避免或简化排序

应当简化或避免对大型表进行重复的排序。当能够利用索引自动以适当的次序产生输出时，优化器就避免了排序的步骤。以下是一些影响因素：

1) 索引中不包括一个或几个待排序的列；

2) group by 或 order by 子句中列的次序与索引的次序不一样；

3) 排序的列来自不同的表。

为了避免不必要的排序，就要正确地增建索引，合理地合并数据库表(尽管有时可能影响表的规范化，但相对于效率的提高是值得的)。如果排序不可避免，那么应当试图简化它，如缩小排序的列的范围等。

4.3. 消除对大型表行数据的顺序存取

在嵌套查询中，对表的顺序存取对查询效率可能产生致命的影响。比如采用顺序存取策略，一个嵌套3层的查询，如果每层都查询1000行，那么这个查询就要查询 10 亿行数据。避免这种情况的主要方法就是对连接的列进行索引。例如，两个表：学生表(学号、姓名、年龄……)和选课表(学号、课程号、成绩)。如果两个表要做连接，就要在“学号”这个连接字段上建立索引。还可以使用并集来避免顺序存取。尽管在所有的检查列上都有索引，但某些形式的 where 子句强迫优化器使用顺序存取。下面的查询将强迫对orders 表执行顺序操作：

```
SELECT*
FROM Orders
WHERE (customer_num = 104 AND order_num > 1001) OR order_num = 1008
```

虽然在 customer_num 和 order_num 上建有索引，但是在上面的语句中优化器还是使用顺序存取路径扫描整个表。因为这个语句要检索的是分离的行的集合，所以应该改为如下语句：

```
SELECT * FROM orders WHERE customer_num = 104 AND order_num > 1001
UNION
SELECT * FROM orders WHERE order_num = 1008
```

这样就能利用索引路径处理查询。

4.4. 避免相关子查询

一个列的标签同时在主查询和 where 子句中的查询中出现，那么很可能当主查询中的列值改变之后，子查询必须重新查询一次。查询嵌套层次越多，效率越低，因此应当尽量避免子查询。如果子查询不可避免，那么要在子查询中过滤掉尽可能多的行。

4.5. 避免困难的正规表达式

某些关键字的应用是正确的，技术上叫正规表达式，但有时搭配不当会非常耗费时间，特别是在大型数据表中体现的尤为突出，我们把这种正规表达式称为困难的正规表达式。

1) 支持通配符的 CHARINDEX 和 LIKE 关键字。例如：

```
SELECT * FROM table1 WHERE user_id LIKE '98___'
```

即使在 user_id 字段上建立了索引，在这种情况下也还是采用顺序扫描的方式。如果把语句改为 SELECT * FROM table1 WHERE user_id > '98000'在执行查询时就会利用索引来查询，显然会大大提高速度。

比如查找用户名包含有“c”的所有用户，可以用

```
SELECT * FROM table1 WHERE user_name LIKE '%c%'
```

下面是完成上面功能的另一种写法：

```
SELECT * FROM table1 WHERE CHARINDEX ('c',user_name) > 0
```

这种方法理论上比上一种方法多了一个判断语句，即>0，但这个判断过程是最快的，我想信 80%以上的运算都是花在查找字符串及其它的运算上。用这种方法也有好处，那就是对“%”、“|”等在不能直接用 LIKE 查找到的字符中可以直接在这 CHARINDEX 中运用，如下：

```
SELECT * FROM table1 WHERE CHARINDEX ('%',user_name) > 0
```

2) 少使用“*”。例如语句：

```
SELECT COUNT(*) FROM table1
```

这时用“*”和一个实际的列名得到的都是一个行数的结果，但是用“*”会统计所有列，显然要比用一个实际的列名效率慢。同样，尽管很多开发人员都习惯采用“SELECT * FROM TBL”的模式进行查询，但是为了提高系统的效率，如果你只需要其中某几个字段的值的话，最好把这几个字段直接写出来。

3) 尽量不要在 WHERE 子句中对字段使用函数或参与表达式计算，这样会导致无法使用索引进行全表扫描。

4) 不要使用 NOT。查询时可以在 WHERE 子句使用一些逻辑表达式，如大于、小于、等于以及不等于等等，也可以使用 and(与)、or(或)以及 not(非)。NOT 可用来对任何逻辑运算符号取反。下面是一个 NOT 子句的例子：

```
WHERE NOT (col = 'VALID')
```

NOT 运算符包含在另外一个逻辑运算符中，这就是不等于(<>)运算符。换句话说，即使不在查询 where 子句中显式地加入 NOT 词，NOT 仍在运算符中，见下例：

SELECT * FROM table1 WHERE user_id<>3000；

对这个查询，可以改写为不使用 NOT：

SELECT * FROM table1 WHERE user_id < 3000 OR user_id > 3000；

虽然这两种查询的结果一样，但是第二种查询方案会比第一种查询方案更快些。第二种查询允许对 user_id 列使用索引，而第一种查询则不能使用索引。

5) IN 和 EXISTS。EXISTS 要远比 IN 的效率高，里面关系到 full table scan 和 range scan。同时应尽可能使用 NOT EXISTS 来代替 NOT IN，尽管二者都使用了 NOT(不能使用索引而降低速度)，但是 NOT EXISTS 要比 NOT IN 查询效率更高。

6) 慎用游标。在某些必须使用游标的场合，可考虑将符合条件的数据行转入临时表中，再对临时表定义游标进行操作，这样可使性能得到明显提高。

7) 在海量查询时尽量少用格式转换。

8) IN、OR 子句常会使工作表的索引失效。如果不产生大量重复值，可以考虑把子句拆开，拆开的子句中应该包含索引。

4.6. 使用临时表加速查询

把表的一个子集进行排序并创建临时表，有时能加速查询。有助于避免多重排序操作，而且在其他方面还能简化优化器的工作。例如：

```
SELECT cust.name, rcvbles.balance
FROM cust, rcvbles
WHERE cust.customer_id = rcvlbes.customer_id
AND rcvblls.balance > 0
AND cust.postcode > '98000'
ORDER BY cust.name
```

如果这个查询要被执行多次而不止一次，可以把所有未付款的客户找出来放在一个临时文件中，并按客户的名字进行排序：

```
SELECT cust.name, rcvbles.balance
FROM cust, rcvbles
WHERE cust.customer_id = rcvlbes.customer_id
AND rcvblls.balance > 0
ORDER BY cust.name
INTO TEMP cust_with_balance
```

然后以下面的方式在临时表中查询：

```
SELECT * FROM cust_with_balance
WHERE postcode > '98000'
```

临时表中的行要比主表中的行少，而且物理顺序就是所要求的顺序，减少了磁盘 I/O，所以查询工作量可以得到大幅减少。

注意：临时表创建后不会反映主表的修改。在主表中数据频繁修改的情况下，注意不要丢失数据。

4.7. 用排序来取代非顺序存取

非顺序磁盘存取是最慢的操作，表现在磁盘存取臂的来回移动。SQL 语句隐藏了这一情况，使得我们在写应用程序时很容易写出要求存取大量非顺序页的查询。有些时候，用数据库的排序能力来替代非顺序的存取能改进查询。

5. 实例分析

下面我们举一个制造公司的例子来说明如何进行查询优化。制造公司数据库中包括 3 个表，模式如下所示：

1) part 表

零件号(part_num)	零件描述(part_desc)	其他列 (other column)
102, 032	Seageat 30 G disk	……
500, 049	Novel 10 M network card	……

……

2) vendor 表

厂商号(vendor_num)	厂商名(vendor_name)	其他列 (other column)
910, 257	Seageat Corp	……
523, 045	IBM Corp	……

……

3) parven 表

零件号(part_num)	厂商号(vendor_num)	零件数量 (part_amount)
102, 032	910, 257	3,450,000
234, 423	321, 001	4,000,000

……

下面的查询将在这些表上定期运行，并产生关于所有零件数量的报表：

```
SELECT part_desc，vendor_name，part_amount
FROM part，vendor，parven
WHERE part.part_num = parven.part_num
AND parven.vendor_num = vendor.vendor_num
ORDER BY part.part_num
```

如果不建立索引，上述查询代码的开销将十分巨大。为此，我们在零件号和厂商号上建立索引。索引的建立避免了在嵌套中反复扫描。关于表与索引的统计信息如下：

表 (table)	行尺寸 (row size)	行数量 (Row count)	每页行数量 (Rows/Pages)	数据页数量 (Data Pages)
Part	150	10,000	25	400
Vendor	150	1000	25	40
Parven	13	15,000	300	50

索引	键尺寸	每页键数量	页面数量
(Indexes)	(Key Size)	(Keys/Page)	(Leaf Pages)
Part	4	500	20
Vendor	4	500	2
Parven	8	250	60

看起来是个相对简单的 3 表连接，但是其查询开销是很大的。通过查看系统表可以看到，在 part_num 上和 vendor_num 上有簇索引，因此索引是按照物理顺序存放的。parven 表没有特定的存放次序。这些表的大小说明从缓冲页中非顺序存取的成功率很小。此语句的优化查询规划是：首先从 part 中顺序读取 400 页，然后再对 parven 表非顺序存取 1 万次，每次 2 页(一个索引页、一个数据页)，总计 2 万个磁盘页，最后对 vendor 表非顺序存取 1.5 万次，合 3 万个磁盘页。可以看出在这个索引好的连接上花费的磁盘存取为 5.04 万次。

实际上，我们可以通过使用临时表分 3 个步骤来提高查询效率：

1) 从 parven 表中按 vendor_num 的次序读数据：

```
SELECT part_num, vendor_num, price
FROM parven
ORDER BY vendor_num
INTO temp pv_by_vn
```

这个语句顺序读 parven(50 页)，写一个临时表(50 页)，并排序。假定排序的开销为 200 页，总共是 300 页。

2) 把临时表和 vendor 表连接，把结果输出到一个临时表，并按 part_num 排序：

```
SELECT pv_by_vn, *vendor.vendor_num
FROM pv_by_vn, vendor
WHERE pv_by_vn.vendor_num = vendor.vendor_num
ORDER BY pv_by_vn.part_num
INTO TMP pvvn_by_pn
DROP TABLE pv_by_vn
```

这个查询读取 pv_by_vn(50 页)，它通过索引存取 vendor 表 1.5 万次，但由于按 vendor_num 次序排列，实际上只是通过索引顺序地读 vendor 表(40 + 2 = 42 页)，输出的表每页约 95 行，共 160 页。写并存取这些页引发 5 × 160 = 800 次的读写，索引共读写 892 页。

3) 把输出和 part 连接得到最后的结果：

```
SELECT pvvn_by_pn.*, part.part_desc
FROM pvvn_by_pn, part
WHERE pvvn_by_pn.part_num = part.part_num
DROP TABLE pvvn_by_pn
```

这样，查询顺序地读 pvvn_by_pn(160 页)，通过索引读 part 表 1.5 万次，由于建有索引，所以实际上进行 1772 次磁盘读写，优化比例为 30:1。笔者在 Informix Dynamic Sever 上做同样的实验，发现在时间耗费上的优化比例为 5:1(如果增加数据量，比例可能会更大)。

6. 总结

20%的代码用去了 80%的时间，这是程序设计中的一个著名定律，在数据库应用程序中也同样如此。对于数据库应用程序来说，重点在于 SQL 的执行效率，所谓优化的重点环节即 WHERE 子句利用了索引，不可优化即发生了全表扫描或额外开销。经验显示，数据库性能的最大改进得益于逻辑的数据库设计、索引设计和查询设计方面。反过来说，最大的性能降低问题常常是由这些方面中的不足引起的。其实 SQL 优化的实质就是在结果正确的前提下，用优化器可以识别的语句，充份利用索引，减少表扫描的 I/O 次数，

尽量避免全表搜索的发生。其实 SQL 的查询性能优化是一个复杂的过程，上述这些只是在应用层面的一种体现，深入研究还会涉及数据库层的资源配置、网络层的流量控制以及操作系统层的总体设计。

参考文献 (References)

[1] 汪照东. Oracle 11g 数据库管理与优化宝典[M]. 北京: 电子工业出版社, 2008.

[2] 谭怀远. 让 Oracle 跑得更快 2——基于海量数据的数据库设计与优化[M]. 北京: 电子工业出版社, 2011.