**序言**

SQL 语言是一种强大而且灵活的语言，在使用 SQL 语言来执行某个关系查询的时候，用户可以写出很多不同的 SQL 语句来获取相同的结果。也就是说，语法 (syntactical) 不同的 SQL 语句，有可能在语义 (semantical) 上是完全相同的。但是尽管这些 SQL 语句最后都能返回同样的查询结果，它们在 DB2 中执行所需要的时间却有可能差别很大。这是为什么？

众所周知，**DB2 数据库具有强大的功能**，可以自动地把用户输入的 SQL 语句改写为多个语义相同的形式并从中选取一个耗时最少的语句来执行。但是 DB2 并不能够永远对所有的 SQL 语句都成功的改写来取得最优的执行方案。其中一个方面的原因就是数据库应用程序的开发人员在写 SQL 语句的时候有一些习惯性的“小问题”，而正是这些小问题带来了 SQL 语句运行时性能上的大问题。正如平时所说“条条大路通罗马”，但是并非所有通往罗马的路都是坦途，我们应该找到那条最有效的道路。

**编写高效 SQL 语句的一些最佳实践**

这里我们将介绍在编写 SQL 语句时可能影响 DB2 查询性能的一些常见问题，并给出相应的编写高效 SQL 语句的最佳实践（best-practices）。

**避免不恰当的使用“SELECT \*”**

像“SELECT \*”这样的写法在用户使用中可能很常见，它表示把满足查询条件的每一条记录（Row）的所有列都返回。但是有时候这种用法很可能导致数据库查询时候的性能问题。假定 Sale 是一个包括 25 个列（column）的表，那么下面这条查询语句就有可能在执行时性能较差，其中一部分原因就是在 SELECT 中使用了"\*".

SELECT \*FROM Sales WHERE YEAR(Date) > 2004 AND Amount > 1000

如果 SQL 语句使用了“SELECT \*”，DB2 就需要把表的所有列都从外部存储介质上（如磁带或者硬盘）复制到 DB2 的内存中来进行处理并且返回给用户，这显然会增加 I/O 和 CPU 的开销。而且如果这条 SQL 语句还包括了排序（Sort）操作（比如 ORDER BY），那么对全部这些列进行排序也可能会影响到性能。而且当表定义的列越多，每个列定义的数据类型（Data type）长度越长，这对性能的影响就可能越明显。除此之外，DB2 还有一种被称为“**Index-Only**”的数据访问方法，如果某个表上需要检索的所有列都能在某个合适的索引（Index）上找到，DB2 就会使用“Index-Only”这种数据访问方式。因为这种访问方式仅需要对索引进行检索而无需对表本身进行读取，所以是一种较快高速的访问方式。但是如果用户输入的 SQL 语句中使用了“SELECT \*”，就意味着需要访问表上的所有列。而通常情况下并不存在一个合适的索引是定义在这个表所有的列之上的（尤其是对于定义了许多列的表），这就使得 DB2 无法使用“Index-Only”这种较快的数据访问方式，而改用其他数据访问方式，这也有可能导致查询性能的问题。

所以除非真的需要读取表中的所有列，否则基于提高查询性能的考虑，在写 SQL 语句的时候应该尽量避免使用“SELECT \*”这样的情况。这是一条很简单却常常被用户忽略的最佳实践。

下面来看一个具体的示例。需要说明的是，本文示例中用到的表，除特别说明外，均为 TPC-D 标准中定义的表，这样有助于读者更好的理解 SQL 语句本身。对于 TPC-D 标准的介绍，见文章最后的参考资源。

在这个例子中，我们比较 2 个不同的 SQL 语句在性能上的差别。2 个 SQL 的谓词完全相同，并且这个谓词有符合的索引可以使用。

SQL 1：select \* from lineitem where l\_orderkey = ?

SQL 2：select l\_suppkey, l\_partkey from lineitem where l\_orderkey = ?

但是 SQL 1 使用了 select \*，所以 DB2 在读取索引之后，必须再去对表中进行一次 Fetch 操作，读取那些索引中不存在的列数据。而在实际业务需求中如果并不需要这些数据，这个 Fetch 操作就是多余的而且会带来性能问题。对比 SQL 2，它明确指出了在结果集中希望得到的列 l\_suppkey, l\_partkey，而这些列已经全部包含在索引中，所以数据库采用了 Index-Only 的扫描方式，仅仅读取了索引，不再需要对表本身的 fetch 操作，从而使得性能得到了大幅提升。

图 1. SQL 1 的访问路径图

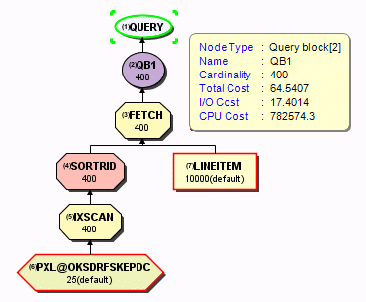
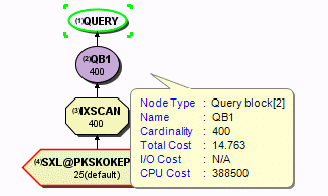


图 2. SQL 2 的访问路径图



**避免在本地谓词的列上使用复杂表达式**

所谓的**本地谓词**（Local predicate）是与连接谓词（Join predicate）相对应，它一般是指该谓词当中只包含一个表上的一个列。

在上一节看到的例子当中，YEAR(Date) > 2004 和 Amount > 1000 都是两个本地谓词。然而在前一个谓词 YEAR(Date) > 2004 中，它对 Date 这个列有一个函数 YEAR 的调用。在这种情况下，即使 Date 上存在一个索引，DB2 也无法使用这个索引来访问数据。如果能够在确保语义不变的前提下，适当改写这个谓词，避免在 Date 列上调用函数，那么情况可能会有所不同。例如，这个谓词可以改写为如下的样子：

Date > ‘ 2004-12-31 ’

这样的改写首先确保了语义上的一致性，更重要的是，DB2 对于这样的谓词是可以通过索引来访问数据，这样查询性能可能会比之前快很多。

这里再给出一个类似的例子。对于 INTEGER（Sale）/100 ＝ 900 这样的谓词，也可以将其改写为 Sale BETWEEN 90000 AND 90099 来提高查询性能。

通过上面两个例子，可以得出一个对应的最佳实践的理论公式。如果在本地谓词中出现如下的形式：

Function(Column\_A) = ‘ constant ’

那么尽可能的将其改写为如下的形式会有助于查询性能的提高：

**Column\_A = Inverse\_Function( ‘ constant ’ )**

**这里 Column\_A 是表上的某个列，constant 是常量，而 Function 与 Inverse\_Function 是两个互逆的函数。**

这是在写 SQL 语句时的另一个最佳实践：尽量避免在本地谓词中对于表的某个列使用复杂的表达式（函数调用或者数学运算等等）。

下面来看一个具体的示例：

SQL 3：select l\_quantity, l\_comment from lineitem

where l\_orderkey + 100 = 200

SQL 4：select l\_quantity, l\_comment from lineitem

where l\_orderkey = 100

图 3. SQL 3 的访问路径图

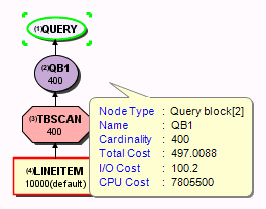
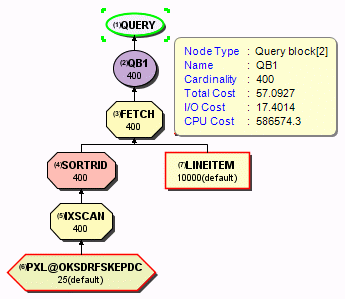


图 4. SQL 4 的访问路径图



这里两个 SQL 语句在语义上是完全相等的，只有谓词在写法上存在一些差异。SQL 3 的谓词包含了一个 计算表达式 l\_orderkey + 100 = 200，而 SQL 4 的谓词是与之等价的简单形式 l\_orderkey = 100。但是它们的查询访问路径可能会截然不同。对于 SQL 4，DB2 利用已有的索引，采用了较为高效的索引访问方式（Index-scan）；而 SQL 3 的谓词存在计算表达式，DB2 必须先计算出 l\_orderkey + 100 的值再进行匹配，这使得直接利用索引的索引访问方式无法采用。这两种不同的访问路径所带来的性能也是大不一样的，这一点对比图中两者的 Total Cost 就可以看出，谓词中含有计算表达式 l\_orderkey + 100 = 200 的 SQL 3 的 Total Cost 较高，性能不好。

**避免用复杂表达式构建连接谓词**

所谓**连接谓词**（Join predicate），一般是指该谓词引用到了不同表上的多个列。比如在如下的 SQL 语句中：

Select T1.C1 From T1, T2 Where T1.C1=T2.C2 And T1.C2=10

T1.C1=T2.C2 就是一个典型的连接谓词，这种写法也是常见的连接谓词的形式。对于这种常见的连接谓词，DB2 可以考虑采用几种不同的**表连接方式（Join Method），常见的连接方法有嵌套循环连接（Nested-Loop-Join, NLJ），归并排序连接（Merge-Scan-Join, MSJoin），哈希连接（Hash-Join）等**。DB2 优化器会根据实际情况选择从中选取一个性能最佳的来将 T1 和 T2 连接起来。

但是有的 SQL 语句可能会是如下这个样子：

Select T1.C1 From T1, T2 Where T1.C1 \* T1.C2 = T2.C2

在连接谓词 T1.C1 \* T1.C2 = T2.C2 中，“＝”左边不是一个列名，而是一个表达式，它涉及 T1 表上不同列之间的计算。对于这样一个用复杂表达式构建的连接谓词，DB2 只能用 Nested-Loop-Join 这种最基本的方式来建立 T1 和 T2 之间的连接，而不考虑用其他的连接方式，从而也就无法选择最优的连接方式。所以这种在连接谓词中使用复杂表达式的写法不是一个好的习惯，在写 SQL 语句时应该注意避免。

看下面这个示例：

SQL 5：

SELECT l\_comment, o\_comment FROM lineitem, order

WHERE l\_orderkey = o\_orderkey + 100

SQL 6：

SELECT l\_comment, o\_comment FROM lineitem, order

WHERE l\_orderkey = o\_orderkey

在这个例子中，SQL 5 中的连接谓词中包含了一个计算表达式 l\_orderkey = o\_orderkey + 100，而 SQL 6 中的连接谓词是简单的等式 l\_orderkey = o\_orderkey。这样不同的连接谓词对 DB2 选择连接方法时有重要的影响。在 SQL 5 中，连接谓词中包含计算表达式，DB2 只能选用最基本的 Nested-Loop-Join 连接方法（参见图 5）。对比 SQL 6，假设连接谓词是 l\_orderkey = o\_orderkey 这样简洁的形式，DB2 就会采用 Merge-Scan-Join 的连接方式（见图 6）。注意这里 SQL5 与 SQL6 在语义上是不等价的，在这里用这样的示例是为了说明连接谓词的写法会导致连接方式的改变。**如果想在满足业务逻辑需求的情况下，同时保证连接谓词的简洁，也可以考虑增加一个新的列**（例如 SQL 5 中，定义新的列 o\_orderkey2，其值等于 o\_orderkey + 100），直接构造连接谓词（l\_orderkey = o\_orderkey2），从而最大程度的提高 SQL 语句的性能。

图 5. SQL 5 的访问路径图

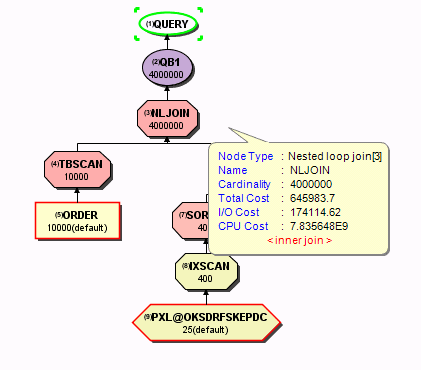
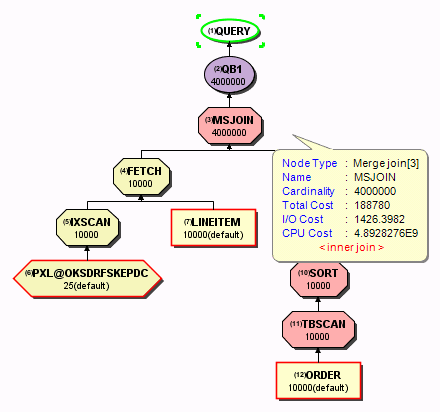


图 6. SQL 6 的访问路径图



**避免连接谓词中的数据类型不一致**

在用连接谓词连接不同的表的时候，还有一点需要注意。即使对于 T1.C1=T2.C1 这样典型的连接谓词，也应该确保 T1.C1 和 T2.C1 具有同样的数据类型。

在某些情况下，连接谓词中两个列的数据类型定义的不一致会导致 DB2 放弃使用某些表连接方式。比如 Hash-Join 这种表连接方式对连接谓词就有更多的限制条件，条件之一就是连接谓词中的两个列的数据类型必须完全一致，否则 Hash-Join 不能使用。例如，如果连接谓词中的 T1.C1 是 FLOAT 类型，而 T2.C1 是 REAL 类型，那么 DB2 不会使用 Hash-Join 来连接 T1 和 T2。此外，如果 T1.C1 的数据类型是 CHAR，GRAPHIC，DECIMAL 或者 DECFLOAT，那么 T2.C1 除了需要是相同的数据类型外，它所定义的数据类型的长度也需要和 T1.C1 一致，比如都被定义为 CHAR（5），否则也不能使用 Hash-Join 来连接。

更多的表连接方式意味着 DB2 可以有更多的选择来将表连接在一起，并从中选出最优的方案。如果连接谓词中的数据类型不一致，而使得 DB2 不得不放弃某些特定的连接方式，这将有可能导致 SQL 在执行时性能不够好。所以在不同的表之间建立连接关系时，应该避免连接谓词中的数据类型不一致。

看下面这个示例：

SQL 7： SELECT l\_comment, o\_comment FROM lineitem, order

WHERE l\_orderkey = o\_orderkey

对于 SQL 1，DB2 优化器采用了归并排序 (MSJoin) 的连接方法对两个数据表进行了连接操作 ( 如图 7 所示 )，注意根据 TPC-D 标准的定义，这里连接谓词 l\_orderkey = o\_orderkey 中的 2 个列的数据类型完全一致都为 integer 类型。如果改动其中一个列的数据类型为 double 类型，此时 DB2 就只能采用嵌套循环连接方法进行连接操作（如图 8 所示），而对比之后就会发现，使用嵌套循环连接的 Total Cost 较高，这意味着性能较差。

图 7. SQL 7 的访问路径图

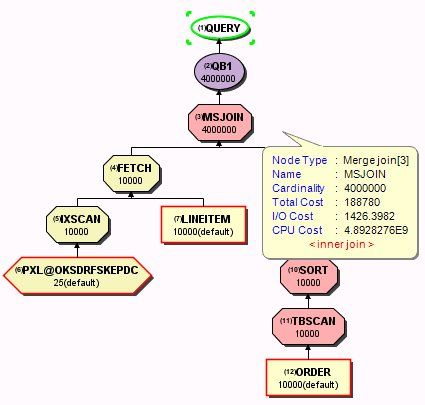
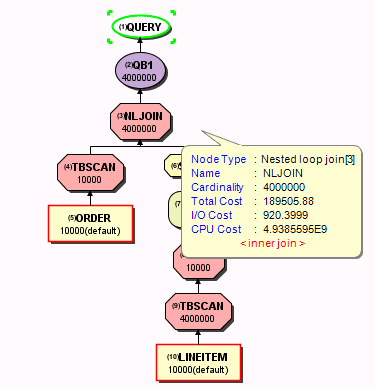


图 8. SQL 7 的访问路径图（修改 o\_orderkey 数据类型之后）



**确保在连接谓词中使用等号**

典型的连接谓词通常是形如 T1.C1=T2.C1 这样的形式，注意到这里是用“=”这个操作符将左右两边的列连接起来。理论上，也可以使用其他的操作符来构造连接谓词，比如“<”或者“>”这样的比较运算符。但是实际上基于性能的考虑，在连接谓词中应该只使用“=”，尽量避免使用其他的比较运算符。

对于如下的 SQL 语句：

Select T1.C2 From T1, T2 Where T1.C1 < T2.C1

在连接谓词 T1.C1 < T2.C1 中，使用了“<”这个比较运算符。 对于这样的 SQL 语句，DB2 只能采用 Nested-Loop-Join 这种最基本的方式来建立 T1 和 T2 之间的连接，而不考虑用其他的连接方式。如前面提到的，更多的表连接方式意味着 DB2 可以有更多的选择来将表连接在一起，并从中选出最优的方案。如果连接谓词中的连接符不是“=”，使得 DB2 不得不放弃某些特定的连接方式，从而也就无法选择最优的连接方式，这将有可能导致 SQL 在执行时性能不够好。

此外，对于这样没有使用“=”的连接谓词，DB2 在计算这个谓词的筛选率（selectivity）的时候，有可能计算的不够准确，而如果同样的连接谓词改为用“＝”连接，筛选率的计算就会准确很多。熟悉 DB2 的数据库管理员和数据库程序开发人员都会知道，筛选率的准确性对于 DB2 优化器非常重要。只有基于准确的筛选率，DB2 优化器才能从各种可能的访问路径中确定最优路径。而筛选率不准确，就有可能带来潜在的查询性能问题。

基于上述两点可以看出，“=”在构建连接谓词时很重要。在不同的表之间建立连接关系时，应该尽可能的使用“=”来构建连接谓词。

需要指出的是，在某些实际的应用场景当中，出于业务逻辑上的要求，出现 T1.C1 < T2.C1 这样的连接谓词可能是不可避免的。在这种情况下，**基于性能优化的考虑，应该在 T1 和 T2 上都建立适当的索引，使得 T1.C1 < T2.C1 这个谓词能够使用索引。**其中的原因在于，DB2 只能使用 Nested-Loop-Join 来建立 T1 和 T2 之间的连接，此时应该确保有合适使用的索引能够让 Nested-Loop-Join 采用 Index-Scan 这种数据访问方法，从而尽可能提高性能。但是对于上面提到的第二个筛选率问题，即使添加索引也不能很好的解决这个问题。

看下面的示例，

SQL 8： SELECT l\_comment, o\_comment FROM lineitem, order

WHERE l\_oderkey >o\_orderkey

SQL 9： SELECT l\_comment, o\_comment FROM lineitem, order

WHERE l\_oderkey =o\_orderkey

在 SQL 8 中连接谓词是通过大于号连接的，DB2 只能采用嵌套循环连接 (Nested-Loop-Join) 这种最基本的方式来建立两个表之间的连接（如图 9 所示）。在 SQL9 中连接谓词中采用“=”连接，此时 DB2 优化器选用了归并排序（MSJoin）的连接方式，它的 Total cost 比 SQL 8 的要低很多，具有较好的性能（如图 10 所示）。注意这里 SQL 8 与 SQL 9 在语义上是不等价的，在这里用这样的示例是为了说明连接谓词中不使用等号的写法会导致访问路径完全不同，从而影响查询性能。

图 9. SQL 8 的访问路径图

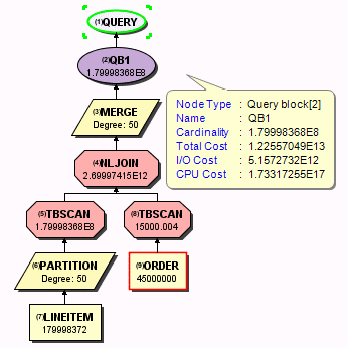
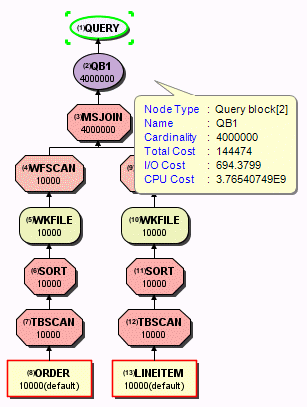


图 10. SQL 9 的访问路径图



**确保主键与外键之间的连接谓词**

**表之间的主键外键反映了表之间数据的依赖关系**。**如果一个 SQL 语句涉及两个表之间的连接，而这两个表存在主外键关系，那么通常情况下，该 SQL 语句中都应该有基于该主外键关系的连接谓词。**在写 SQL 语句的时候，也应该注意这一点，即根据主外键关系确保 SQL 中存在对应的连接谓词，否则的话，返回的查询结果中可能会包括大量无实际意义的记录，而返回这些记录又会给数据库执行带来额外的开销，造成性能问题。

来看一个简单的例子，假设在表 T1 与 T2 之间存在主外键的关系，其中 T1.C1 是主键，T2.C1 是外键。如果有如下的 SQL 语句：

SELECT T1.C2, T2.C2

FROM T1, T2

WHERE T1.C2 = 5

AND T2.C2 = ‘ IBM ’

AND T1.C3 = T2.C3

注意在这个 SQL 当中，T1 与 T2 之间有一个连接谓词 T1.C3 = T2.C3，但是却缺少 T1.C1 = T2.C1 这样的连接谓词。而通过 T1 与 T2 之间的主外键关系，可以合理推导出在通常情况下，用户想看到的结果中应该包括 T1.C1 = T2.C1 这样的逻辑关系。因此可以将上面的 SQL 改写为：

SELECT T1.C2, T2.C2

FROM T1, T2

WHERE T1.C2 = 5

AND T2.C2 = ‘ IBM ’

AND T1.C3 = T2.C3

**AND T1.C1 = T2.C1**

**通过添加这样的连接谓词，使得数据库可以有更多的选择来建立 T1 与 T2 直接的连接关系，并且避免了返回大量无意义的记录，从而使得整体性能得以提高。**但是需要注意的是，这样的写法改变了原先 SQL 的语义，从而改变了查询结果，所以在使用的时候需要用户来确认 T1.C1 = T2.C1 这样的逻辑条件对于其业务应用来说是正确的。

看下面的示例：

SQL10: SELECT l\_comment, o\_comment FROM lineitem, order

SQL11: SELECT l\_comment, o\_comment FROM lineitem, order **WHERE l\_orderkey = o\_orderkey**

其中 SQL10 没有包含 lineitem 和 order 表之间的主外键关系 l\_orderkey = o\_orderkey，从业务逻辑分析的角度出发来看，这样的写法很可能是由于人为的疏忽漏掉了这个连接谓词。从图 11 中也可以看出，SQL 10 返回了一个很大的结果集（图中 Cardinality 所示），可以合理推断，其中包含了大量 l\_orderkey ≠ o\_orderkey 的无效数据，这些数据是业务逻辑并不想要的，而 DB2 为了取得这些无效数据却要花费很高的代价。对比 SQL 11，具有 l\_orderkey = o\_orderkey 这样的主外键连接谓词，更符合逻辑。同时从图 12 也可以看出，它返回的结果集较小，所花费的成本（Total Cost）也小了很多，查询性能优化很多。

图 11. SQL 10 的访问路径图

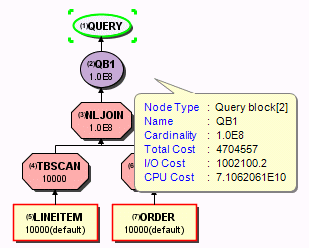
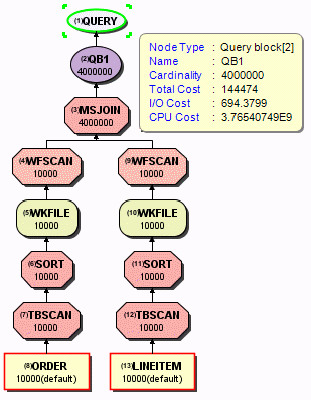


图 12. SQL 11 的访问路径图



**恰当安排 Group By 子句中列的顺序**

通常情况下，SQL 语句中的 GROUP BY 子句会导致数据库不得不通过一个排序（SORT）操作来实现对数据的分组，而排序被认为是一个比较耗费 CPU 和内存的操作。实际上某些情况下，如果写法得当，当中的排序操作是可以避免的。具体来说，在写 GROUP BY 子句的时候，应该考虑到数据库中已经存在的索引的情况。如果 GROUP BY 子句中所有的列恰好包括在某个索引的键（Key column）的范围之内而且是处于开始的位置，那么在写 GROUP BY 子句的时候，就应该按照该索引上键的先后顺序来写 GROUP BY 子句。

比如说有如下的 SQL 语句：

SELECT C2, C3, C1, AVG(C4)

FROM T1

GROUP BY C2, C3, C1

一般情况下，GROUP BY C2, C3, C1这样的写法都会导致数据库的一个排序操作。但假定表 T1 上已经存在一个索引 IX1(C1, C2, C3, C4), 这里注意到 GROUP BY 子句中引用到的列（C2，C3，C1）正好是索引 IX1 中的前三个键，那么就可以通过改变 GROUP BY 子句中列的顺序的办法来避免这个排序操作。

可以把 SQL 语句改写为如下所示：

SELECT C1, C2, C3, AVG(C4)

FROM T1

GROUP BY C1, C2, C3

通过这样改变 GROUP BY 子句中列的顺序使其与索引 IX1 中的键顺序一致，数据库就可以利用 IX1 来访问其已经排序的键值并直接返回进行下一步操作，从而避免额外的排序操作，从而带来查询性能上的提高。

需要指出的是，通过这样改写 GROUP BY 子句来避免排序，可能会导致最终返回结果的顺序不一致。在实际的业务逻辑当中，需要用户来确认是否其关注返回结果的顺序性。

下面来看一个具体的示例：

SQL 12：SELECT AVG(o\_shippriority) FROM order

GROUP BY o\_custkey , o\_orderkey, o\_orderdate

SQL 13：SELECT AVG(o\_shippriority) FROM order

GROUP BY o\_orderkey, o\_orderdate, o\_custkey

这里 2 个 SQL 唯一的差别就在于 GROUP BY 子句中列的顺序不同。根据 TPC-D 标准定义，order 表上存在一个索引 PXO@OKODCKSPOP (O\_ORDERKEY,O\_ORDERDATE,O\_CUSTKEY,O\_SHIPPRIORITY,O\_ORDERPRIORITY)。

由于 SQL 12 中的 GROUP BY 子句的列顺序与索引 PXO@OKODCKSPOP 的键顺序不一致，DB2 无法直接利用这个索引，所以 DB2 需要基于这 3 个列做一次排序（Sort），然后进行分组合并，排序的结果还需要通过临时文件（Wkfile）来保存，如图 13 所示。如果调整 GROUP BY 子句中的列顺序如 SQL 13 所示，使其与索引 PXO@OKODCKSPOP 的键顺序一致，DB2 通过这个索引返回的结果就已经是有序的，这样就省去了排序操作（如图 14 所示）。对比两者的访问路径图可以看出来，SQL 13 所花费的成本（Total Cost）会少很多，性能上有较大的提高。

图 13. SQL 12 的访问路径图

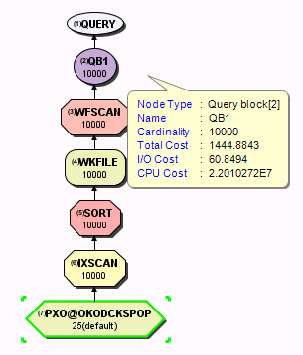
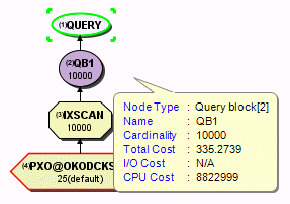


图 14. SQL 13 的访问路径图



**配合使用 OPTIMIZE FOR N ROWS 与 FETCH FIRST N ROWS ONLY**

在 DB2 的 SQL 语法中，**FETCH FIRST n ROWS ONLY 表示只取回结果集当中的前 n 条记录**。这在实际的业务逻辑中会经常用到，比如查找考试成绩在前三名的学生，或者是薪水最高的五位公司员工。而 **OPTIMZE FOR n ROWS 这个子句可能并不被一般用户所熟悉，它的作用是告诉 DB2 的优化器采用尽可能快的方式来返回结果集中的前 n 条记录**，但是注意最终结果集中的所有记录都会被返回，这是它与 FETCH FIRST n ROWS ONLY 的不同。

通常情况下，取得结果集中的全部记录（比如 1000000 条）与取出其中的前 n 条记录（比如第 1 条记录）相比，最优化的方法是不一样的。比如对于后者而言，通过索引来访问可能是最快的，而这种访问对于前者却未必是最佳的访问方式。也就是说，如果想要只取回结果集当中的前 n 条记录，应该使得 DB2 优化器知道这一点，从而选取最优的访问方式。

**所以如果 SQL 语句中带有 FETCH FIRST n ROWS ONLY 这个子句，那么应该同时加上 OPTIMZE FOR n ROWS 子句来配合使用。**比如对于如下的 SQL 语句：

SELECT e.name FROM employee e, department d

WHERE e.workdept = d.deptno

FETCH FIRST 10 ROWS ONLY

可以加上 OPTIMZE FOR n ROWS 子句变为如下的形式：

SELECT e.name FROM employee e, department d

WHERE e.workdept = d.deptno

**FETCH FIRST 10 ROWS ONLY**

**OPTIMIZE FOR 10 ROWS**

这样一来，DB2 优化器就会尽量采用最优化的方式来尽快返回前 10 条结果，比如避免采用一个临时表来存储中间结果，从而达到查询性能上的提升。

总结

本文介绍了在编写 SQL 语句时几种可能影响 DB2 查询性能的常见问题，并给出相应的编写高效 SQL 语句的最佳实践，它们对于数据库程序开发人员在 DB2 上进行 SQL 查询性能调优时非常重要。如果你的 SQL 语句写的高效简洁，你就可以有效的避免在 DB2 查询时可能出现的性能问题，从而减少总的成本并提高客户业务的投资回报。