Fundamental Techniques for Order Optimization

David Simmen
IBM Santa Teresa Lab
dsimmen@vnet.ibm.com

Eugene Shekita
IBM Almaden Research Center
shekita@almaden.ibm.com

Timothy Malkemus
IBM Austin Lab
malkemus@vnet ibm com

Abstract

Decision support applications are growing in popularity as more business data is kept on-line. Such applications typically include complex SQL queries that can test a query optimizer's ability to produce an efficient access plan. Many access plan strategies exploit the physical ordering of data provided by indexes or sorting. Sorting is an expensive operation, however. Therefore, it is imperative that sorting is optimized in some way or avoided all together. Toward that goal, this paper describes novel optimization techniques for pushing down sorts in joins, minimizing the number of sorting columns, and detecting when sorting can be avoided because of predicates, keys, or indexes. A set of fundamental operations is described that provide the foundation for implementing such techniques. The operations exploit data properties that arise from predicate application, uniqueness, and functional dependencies. These operations and techniques have been implemented in IBM's DB2/CS.

1 Introduction

As the cost of disk storage drops, more business data is being kept on-line. This has given rise to the notion of a data warehouse, where non-operational data is typically kept for analysis by decision support applications. Such applications typically include complex SQL queries that can test the capabilities of an optimizer. Often, huge amounts of data are processed, so an optimizer's decisions can mean the difference between an execution plan that finishes in a few minutes verses one that takes hours to run.

Many access plan strategies exploit the physical ordering of data provided by indexes or sorting. Sorting is an expensive operation, however. Therefore, it is imperative that sorting is optimized in some way or avoided all together. This leads to a non-trivial optimization problem, however, because a single complex query can give rise to multiple *interesting orders* [SAC+79]. Here, an interesting order refers to a specification for any ordering of the data that may prove useful for processing a join, an ORDER BY, GROUP BY, or DISTINCT. To be effective, an optimizer must detect when indexes provide an interesting order, the optimal place to sort if sorting is unavoidable, the minimal number of sorting columns, whether two or more interesting orders can be combined and satisfied by a single sort, and so on. This process will be referred to as *order optimization.*

At first glance, it might seem like hash-based set operations [BD83, DKO+84] make order optimization a non-issue, since hash-based operations do not require their input to be ordered. An index may already provide an interesting order for some operation, however, making the hash-based alternative more expensive. This is particularly true in warehousing environments, where indexes are pervasive. As a result, an optimizer needs to be cognizant of interesting orders. It should always consider both hash- and order-based operations and pick the least costly alternative [Gra93].

Although people have been building SQL query optimizers for close to twenty years [JV84, Gra93], there has been surprisingly little written about the problem of order optimization. This paper describes novel techniques to address that problem. One of the paper's key contributions is an algorithm for reducing an interesting order to a simple canonical form by using applied predicates and functional dependencies. This is essential for determining when sorting is actually required. Another important contribution is the notion of *sort-ahead,* which allows a sort for something like an ORDER BY to be pushed down in a join tree or view. All of these techniques have been implemented in the query optimizer of IBM's DB2/CS, which is the client-server version of DB2 that runs OS/2, Microsoft Windows NT, and various flavors of UNIX. Henceforth, DB2/CS will be referred to as simply DB2. Much of the discussion in this paper is framed in the context of the DB2 query optimizer. The techniques that are described have general applicability, however, and could be used in any query optimizer.

The remainder of this paper is organized as follows: In Section 2, related work is described. This is followed by a brief overview of the DB2 optimizer in Section 3. Next, fundamental operations for order optimization are

SIGMOD '96 6/96 Montreal, Canada
© 1996 ACM 0-89791-794-4/96/0006...$3.50

described in Section 4. In Section 5, the architecture of the DB2 optimizer that has been built around those fundamental operations is described. An example is then provided in Section 6 to illustrate how things tie together. Advanced issues beyond the scope of this paper are mentioned in Section 7. Finally, performance results are presented in Section 8, and conclusions are drawn in Section 9.

2 Related Work

The classic work on the System R optimizer by Selinger et al. [SAC+79] was the first research to look at the problem of order optimization. That paper coined the term "interesting orders". In System R, interesting orders were mainly used to prevent subplans that satisfy some useful order from being pruned by less expensive but unordered subplans during bottom-up plan generation.

A recent paper on the Rdb optimizer [Ant93] talked about combining interesting orders from ORDER BY, GROUP BY, and DISTINCT clauses, if possible, so at most one sort could be used. That paper was primarily an overview of the Rdb optimizer, however. It did not specifically focus on order optimization.

Other, more loosely related papers include those on predicate migration [Hel94] and group-by push-down [YL93, CS93]. Predicate migration considers whether an expensive predicate should be applied before or after a join. Similarly, group-by push-down considers whether GROUP BY should be performed before a join. In each case, an optimizer determines which is the better alternative using its cost estimates. Both techniques are similar to the notion of sort-ahead, as described in this paper.

3 Overview

The DB2 optimizer is a direct descendent of the Starburst optimizer described in [Loh88, HFLP89]. Among other things, the DB2 optimizer uses much more sophisticated techniques for order optimization. This section provides an overview of the DB2 optimizer to establish some background and terminology. More details will be given later.

The DB2 optimizer actually has several distinct optimization phases. Here, we are mainly concerned with the phase where traditional cost-based optimization occurs. Prior to this phase, an input query is parsed and converted to an intermediate form called the *query graph model* (QGM).

The QGM is basically a high-level, graphical representation of the query. *Boxes* are used to represent relational operations, while arcs between boxes are used to represent *quantifiers*, i.e., table references. Each box includes the predicates that it applies, an input or output order specification (if any), a distinct flag, and so on. The basic set of boxes include those for SELECT, GROUP BY, and UNION. Joins are represented by a SELECT box with two or more input quantifiers, while ORDER BY is represented by a SELECT box with an output order specification.

After its construction, the original QGM is transformed into a semantically equivalent but more "efficient" QGM using heuristics such as predicate pushdown, view merging, and subquery-to-join transformation. [PHH92]. Finally, cost-based optimization is performed. During this phase, the QGM is traversed and a *query execution plan* (QEP) is generated.

A QEP can be viewed as a dataflow graph of *operators*, where each node in the graph corresponds to a relational operation like a join or a low-level operation like a sort. Each operator consumes one or more input records (i.e., a table), and produces an output set of records (another table). We will refer to these as input and output *streams*. Figure 1 illustrates what the QGM and QEP might look like for a simple query.

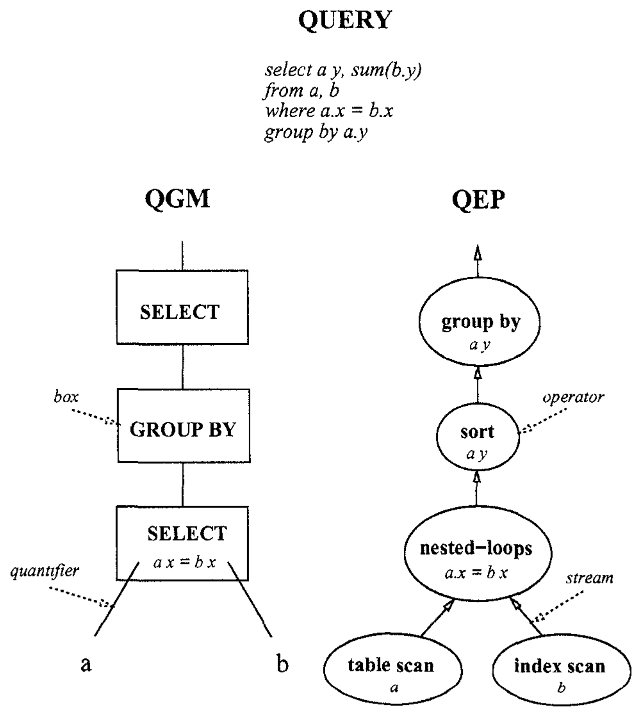

Figure 1: Simple QGM and QEP Example

Each stream in a QEP has an associated set of *properties* [GD87, Loh88]. Examples of properties include the columns that make up each record in the stream, the set of predicates that have been applied to the stream, and the order of the stream. Each operator in a QEP determines the properties of its output stream. The properties of an operator's output stream are a function of its input stream(s) and the operation being applied by the operator. For example, a sort operator passes on all the properties of its input stream unchanged except for the order property and cost. Note that a stream's order, if any, always originates from an ordered index scan or a sort.

During the *planning phase* of optimization, the DB2 optimizer builds a QEP bottom-up, operator-by-operator, computing properties as it goes. At each step, different alternatives are tried and more costly subplans with comparable properties are pruned [Loh88]. At strategic points during planning, the optimizer may decide to build a QEP which satisfies an interesting order. A sort may need to be added to a QEP if there is no existing QEP with an order property satisfying the interesting order.

Interesting orders are generated in a top-down scan of QGM prior to the planning phase. This is referred to as the *order scan* of QGM. Interesting orders arise from joins, ORDER BY, GROUP BY, or DISTINCT, and are hung off the QGM. Here, both order properties and interesting orders will be denoted as a simple list of columns in major to minor order, i.e., $(c_1, c_2, ..., c_n)$. Without loss of generality, we will always assume that an ascending order is required for each column c_i.

Interesting orders are pushed down and combined in the order scan whenever possible. This allows one sort to satisfy multiple interesting orders. As interesting orders are pushed down they can turn into *sort-ahead* orders. These allow the optimizer to try pushing down a sort for, say, an ORDER BY to an arbitrary level in a join tree. Different alternatives are tried, and only the least costly one is kept. The next section looks at the fundamental operations on interesting orders needed to accomplish these tasks.

4 Fundamental Operations for Order Optimization

4.1 Reduce Order

The most fundamental operation used by order optimization is something referred to as *reduction*. Reduction is the process of rewriting an order specification (i.e., an order property or interesting order) in a simple canonical form. This involves substituting each column in the specification with a designated representative of its equivalence class (called the *equivalence class head*) and then removing all redundant columns. Reduction is essential for testing whether an order property satisfies an interesting order.

As a motivating example, consider an arbitrary interesting order $I = (x, y)$, and suppose an input stream has the order property $OP = (y)$. A naive test would conclude that I is not satisfied by OP, and a sort would be added to the QEP. Suppose, however, that a predicate of the form *col* = *constant* has been applied to the input stream, e.g., $x = 10$. Then the column x in I is redundant since it has the value 10 for all records. Hence, I can be rewritten as $I = (y)$. After being rewritten, it is easy to determine that OP satisfies I, so no sort is necessary. Note that a literal expression, host variable, or correlated column qualify as a constant in this context.

Reduction also needs to take column equivalence classes into account. These are generated by predicates of the form *col* = *col*. For example, suppose $I = (x, z)$ and $OP = (y, z)$. Further suppose that the predicate $x = y$ has been applied. The equivalence class generated by $x = y$ allows OP to be rewritten as $OP = (x, z)$. After being rewritten, it is easy to determine that OP satisfies I.

Reduction also needs to take keys into account. For example, suppose $I = (x, y)$ and $OP = (x, z)$. If x is a key, then these can be rewritten as $I = (x)$ and $OP = (x)$. Here, y and z are redundant since x alone is sufficient to determine the order of any two records.

Keys are really just a special case of functional dependencies (FDs) [DD92]. So rather than keys, FDs are actually used by reduction, since they are more powerful. In the DB2 optimizer, a set of FDs are included in the properties of a stream. The way FDs are maintained as a property will be discussed in more detail later.

The notation used for FDs is as follows: A set of columns $A = \{a_1, a_2, ..., a_n\}$ functionally determines columns $B = \{b_1, b_2, ..., b_m\}$ if for any two records with the same values for columns in A, the values for columns in B are also the same. This is denoted as $A \rightarrow B$. The *head* of the FD is A, while the *tail* is B.

It is important to note that all of the above optimizations can be framed in terms of functional dependencies. This is because a predicate of the form $x = 10$ gives rise to $\{\} \rightarrow \{x\}$, i.e., the "empty-headed" FD [DD92]. Moreover, a predicate of the form $x = y$ gives rise to $\{x\} \rightarrow \{y\}$ and $\{y\} \rightarrow \{x\}$. If $x = y$ is a join predicate for an outer join, then $\{x\} \rightarrow \{y\}$ holds if x is a column from a non-null-supplying side. In addition, $\{x\} \rightarrow \{all\ cols\}$ when x is a key. Finally, $\{x\} \rightarrow \{x\}$ is always true.

The mapping of predicate relationships and keys to functional dependencies makes it possible to express reduction in a very simple and elegant way. The algorithm for Reduce Order is shown in Figure 2. In the algorithm, note that the equivalence class head is chosen from those columns made equivalent by predicates already applied to the stream. Also note that $B \rightarrow \{c_i\}$ if there exists some $B' \rightarrow C$ where $B' \subseteq B$ and $C \supseteq \{c_i\}$. This follows from the algebra on FDs [DD92]. Consequently, simple subset operations can be used on the input FDs to test whether $B \rightarrow \{c_i\}$.

```
Reduce Order
input:
  a set of FDs, applied predicates, and
  order specification $O = (c_1, c_2, ..., c_n)$
output:
  the reduced version of $O$

1) rewrite $O$ in terms of each column's
   equivalence class head
2) scan $O$ backwards
3) for (each column $c_i$ scanned)
4)   let $B = \{c_1, c_2, ..., c_{i-i}\}$, i.e.,
     the columns of $O$ preceding $c_i$
5)   if ( $B \rightarrow \{c_i\}$ ) then
6)     remove $c_i$ from $O$
7)   endif
8) endfor
```

Figure 2: Reduce Order Algorithm

The correctness proof for Reduce Order is straightforward. Consider what happens when two records r_1 and r_2 are compared. The only time the value of c_i affects their order is when r_1 and r_2 have the same values for all columns in C. But then $r_1.c_i$ and $r_2.c_i$ must also have the same value because $B \rightarrow \{c_i\}$. Consequently, removing c_i will not change the order of records produced by O.

Before moving on, note that an order specification can become "empty" after being reduced. For example, suppose the predicate $x = 10$ has been applied and the interesting order $I = (x)$ is reduced. The predicate $x = 10$ gives rise to $\{\} \rightarrow \{x\}$. Consequently, I will reduce to the empty interesting order $I = ()$, which is trivially satisfied by any input stream.

4.2 Test Order

As it generates a QEP, the optimizer has to test whether a stream's order property OP satisfies an interesting order I. If not, a sort is added to the QEP. The algorithm for Test Order is shown in Figure 3. Note that when a sort is required, the reduced version of I provides the minimal number of sorting columns, which is important for minimizing sort costs.

4.3 Cover Order

As mentioned earlier, the DB2 optimizer tries to combine interesting orders in the top-down order scan of QGM. This often allows one sort to satisfy multiple interesting orders. When two interesting orders are combined, a *cover* is generated. The cover of two interesting

```
Test Order
input:
  an interesting order $I$ and an order
  property $OP$
output:
  true if $OP$ satisfies $I$, otherwise false

1) reduce $I$ and $OP$
2) if ( $I$ is empty or the columns in $I$
        are a prefix of the columns in $OP$ ) then
3)   return true
4) else
5)   return false
6) endif
```

Figure 3: Test Order Algorithm

orders I_1 and I_2 is a new interesting order C such that any order property which satisfies C also satisfies both I_1 and I_2. For example, the cover of $I_1 = (x)$ and $I_2 = (x, y)$ is $C = (x, y)$.

Of course, it is not always possible to generate a cover. For example, there is no cover for $I_1 = (y, x)$ and $I_2 = (x, y, z)$. As in Test Order, however, interesting orders need to be reduced before attempting a cover. Suppose the predicate $x = 10$ has been applied in this example. Then the interesting orders would reduce to $I_1 = (y)$ and $I_2 = (y, z)$, giving the cover $C = (y, z)$. The algorithm for Cover Order is shown in Figure 4.

```
Cover Order
input:
  interesting orders $I_1$ and $I_2$
output:
  the cover of $I_1$ and $I_2$; or a return code
  indicating that a cover is not possible

1) reduce $I_1$ and $I_2$
2) w.l.o.g., assume $I_1$ is the shorter interesting order
3) if ( $I_1$ is a prefix of $I_2$ ) then
4)   return $I_2$
5) else
6)   return "cannot cover $I_1$ and $I_2$"
7) endif
```

Figure 4: Cover Order Algorithm

4.4 Homogenize Order

As mentioned earlier, an attempt is made to push down interesting orders in the order scan of QGM so that sort-ahead may be attempted. When an interesting order I is pushed down, some columns may have to be substituted with equivalent columns in the new context. This is referred to as *homogenization*. For example, consider the following query:

```
select *
from a, b
where a.x = b.x
order by a.x, b.y
```

Here, the ORDER BY gives rise to the interesting order $I = (a.x, b.y)$. The order scan will try to push down I to the access of both table a and table b as a sort-ahead order. For the access of table b, the equivalence class generated by $a.x = b.x$ is used to homogenize I as $I_b = (b.x, b.y)$.

I cannot be pushed down to the access of table a, since $b.y$ is unavailable until after the join. However, suppose $a.x$ is a base-table key that remains a key after the join [DD92]. If so, $\{a.x\} \rightarrow \{b.y\}$. This allows I to be reduced to $I = (a.x)$, which can be pushed down to the access of table a. As this example illustrates, an interesting order needs to be reduced before being homogenized. The algorithm for Homogenize Order is shown in Figure 5.

```
Homogenize Order
input:
  an interesting order $I$ and target
  columns $C = \{c_1, c_2, ..., c_n\}$
output:
  $I$ homogenized to $C$, that is, $I_C$; or a return
  code indicating that $I_C$ is not possible

1) reduce $I$
2) using equivalence classes, try to substitute each
   column in $I$ with a column in $C$
3) if ( all the columns in $I$ could be substituted ) then
4)   return $I_C$
5) else
6)   return "cannot homogenize $I$ to $C$"
7) endif
```

Figure 5: Homogenize Order Algorithm

Note that unlike Reduce Order, Homogenize Order can choose any column in the equivalence class for substitution. Moreover, there is no need to choose from just the columns that have been made equivalent by predicates applied so far. Columns that will become equivalent later because of predicates that have yet to be applied can also be considered. This is because homogenization is concerned with producing an order that will eventually satisfy I.

5 The Architecture for Order Optimization in DB2

This section describes the overall architecture of the DB2 optimizer for order optimization. Only a high-level summary of the architecture is provided. The focus will be those parts of the architecture that have been built around the fundamental operations discussed in the previous section.

5.1 The Order Scan of QGM

As mentioned earlier, interesting orders are generated during the order scan, which takes place prior to the planning phase of optimization. Interesting orders arise from joins, ORDER BY, GROUP BY, or DISTINCT, and are hung off the QGM.

Each QGM box has an associated output order requirement, and each QGM quantifier has an associated input order requirement. In contrast to an interesting order, an *order requirement* forces a stream to have a specific order. Either the input or output order requirement can be empty. Output order requirements come from ORDER BY, while input order requirements currently come from GROUP BY. (Note that this does not preclude hash-based GROUP BY from being considered during the planning phase of optimization.) Each QGM box also has an associated list of interesting orders, which can double as sort-ahead orders.

Conceptually, the order scan has four stages. In the first stage, input and output order requirements are determined for each QGM box. Then, interesting orders for each DISTINCT is determined. Next, interesting orders for merge-joins and subqueries are determined. Finally, the QGM graph is traversed in a top-down manner.

In the top-down traversal, interesting orders are recursively pushed down along quantifier arcs. When an interesting order is pushed down to a quantifier Q, it gets homogenized to Q's columns and then covered with Q's input order requirement, if any. Similarly, before an interesting order can be pushed into a box B and added to B's list of interesting orders, it gets covered with B's output order requirement.

One subtlety in the order scan is that the algorithms for Cover Order and Homogenize Order require their inputs to be reduced. This in turn requires a set of applied predicates and FDs. Unfortunately, these are not known in the order scan since they are computed as properties during the planning phase of optimization.

This problem is resolved by proceeding optimistically. When an interesting order I is pushed down, the order scan simply assumes that all the predicates below a given box have been applied. Furthermore, if I cannot be fully homogenized to a quantifier, the largest prefix of I that can be homogenized is used. This is done in the hope that some FD will make the suffix redundant. The planning phase can detect when these assumptions turn out to be false.

5.2 The Planning Phase of Optimization

During the planning phase of optimization, the DB2 optimizer walks the QGM bottom-up, box-by-box, and incrementally builds a QEP. For each box, alternative subplans are generated, and more costly subplans with comparable properties are pruned [Loh88]. The input and output interesting orders associated with each box are used to detect when a sort is required.

As a QEP is built, the interesting orders that hang off a QGM box are used for both pruning and to generate sort-ahead orders. During join enumeration, for example, the optimizer will try sorting the outer for each interesting order it finds. This allows a sort for, say, an ORDER BY to be pushed down an arbitrary number of levels in a join tree or view. If no sort is actually required at any level, this will be detected, of course. Note that this is only done for join methods where the order of the outer stream is propagated by the join.

When an interesting order is pushed down to the outer of a join, it has to be homogenized to the quantifier(s) that belong to the outer. This cannot be done during the order scan, since the order in which joins are enumerated is not known then. In the case of a merge-join, a cover with the merge-join order is also required.

Unfortunately, the process of pushing down sort-ahead orders increases the complexity of join enumeration [OL90]. This is because two join subtrees with the same tables but different orders are not compared and pruned against each other. It is possible to show that the complexity of join enumeration increases by a factor of $O(n^2)$ for n sort-ahead orders. In practice, this has not been problem, since typically $n \leq 3$.

5.2.1 Properties

For order optimization, the most important properties are the order property, the predicate property, the key property, and the FD property. Each of these is discussed in detail below. For any property x, the two primary ssues are how x propagates through operators and how two plans are compared on the basis of x.

How the different properties propagate will be discussed shortly. In terms of the way properties are compared, the DB2 optimizer treats everything uniformly. Let P_1 and P_2 be two plans being compared. Also, overload the symbol "$\leq$" for properties to mean less general or equivalent. Then P_2 prunes P_1 if $P_2.cost \leq P_1.cost$ and for every property x, $P_1.x \leq P_2.x$. In other words, P_1 can be pruned if it costs more than P_2 and has less general properties. Thus, for pruning, it suffices to define $\leq$ for each property.

The Order Property

The order property (if any) of a stream always originates from an ordered index scan or a sort. The way it propagates for most relational operators is straightforward except for projections and joins. If any column c_i of an order property $OP = (c_1, c_2, ..., c_n)$ is projected, then only the prefix $OP' = (c_1, c_2, ..., c_{i-1})$ is propagated.

For both nested-loops and merge-join [BE76], the order of the outer stream is propagated. In the special case when the outer has only one record, however, the inner order is propagated. There are also circumstances where the outer and inner orders can be concatenated, but that discussion is beyond the scope of this paper. For hash-join [DKO+84], neither the outer nor inner order is propagated.

The Test Order algorithm given in Section 4.2 is used to compare the order properties OP_1 and OP_2 of two plans during pruning. Let $int(OP_1)$ denote OP_1 cast as an interesting order. Then "$\leq$" can be defined for the order property as follows: $OP_1 \leq OP_2$ if OP_2 satisfies $int(OP_1)$.

The Predicate Property

The predicate property is simply the set of conjuncts which have been applied to a stream. Each operator propagates the predicate property by taking the predicate property of its input stream and unioning it with any conjuncts applied by the operator. For the predicate property, "$\leq$" is defined as follows: Let PP_1 and PP_2 be the predicate properties of two plans being compared. Then, $PP_1 \leq PP_2$ if $PP_1 \subseteq PP_2$.

The predicate property is used to determine both column equivalences and functional dependencies that arise from the application of equality predicates. In the DB2 optimizer, FDs that arise from predicates are not actually added to the FD property, however, since this information would be redundant and would only add to the complexity of maintaining the FD property (see below).

The Key Property

The key property of a stream is the set of unique keys for the stream. Each key K is represented as a set of columns $K = \{c_1, c_2, ..., c_n\}$. Keys are useful for a variety of reasons beyond their role in order optimization. One example is their use in DISTINCT elimination [PL94]. Consequently, in the DB2 optimizer, keys are maintained as a separate property.

Keys originate from base-table constraints or can be added via a GROUP BY or DISTINCT operation. If any column c_i of a key $K = \{c_1, c_2, ..., c_n\}$ in a key property KP is projected by an operator, then K is removed from KP.

Whether a key propagates in a join requires analysis of the join predicates and the keys of the join's input streams. Consider the join of two streams S_1 and S_2 on join predicates JP. Let the key properties of S_1 and S_2 be denoted as KP_1 and KP_2 respectively. If a given row of S_1 can match at most one row of S_2 (i.e., the join is n-to-1), then KP_1 is propagated. This is true if any key $K = \{c_1, c_2, ..., c_n\}$ of KP_2 is *fully qualified* by predicates in JP of the form $S_1.col = S_2.c_i$ for all c_i. Similarly, if the join is 1-to-n, then KP_2 is also propagated. If neither KP_1 nor KP_2 can be propagated, then the key property of the join is formed by generating all concatenated key pairs $K_1 \cdot K_2$, where $K_1 \in KP_1$ and $K_2 \in KP_2$. For example, if $K_1 = \{a_1, a_2, ..., a_n\}$ and $K_2 = \{b_1, b_2, ..., b_m\}$ then $K_1 \cdot K_2 = \{a_1, a_2, ..., a_n, b_1, b_2, ..., b_m\}$.

An attempt is made to keep each key property as "succinct" as possible by removing keys that have become redundant because of projections and/or applied predicates. Each key is rewritten in a canonical form by substituting each column with its equivalence class head and removing redundant columns. If the DB2 optimizer detects that some key has become fully qualified by equality predicates during this process, then the entire key property is discarded and a *one-record condition* is flagged. This condition serves as the key property and indicates that at most one record is in the stream.

After simplifying each key in the property, redundant keys are removed from the key property using the definition of "$\leq$" that follows: Let key $K_1 = \{a_1, a_2, ..., a_n\}$ and let key $K_2 = \{b_1, b_2, ..., b_m\}$. Then $K_1 \leq K_2$ if $\{b_1, b_2, ..., b_m\} \subseteq \{a_1, a_2, ..., a_n\}$. If $K_1 \leq K_2$, then K_1 is implied by K_2. In that case, K_1 is redundant and can be removed.

The definition of "$\leq$" is also used to compare the key properties KP_1 and KP_2 of two plans during pruning. More specifically, $KP_1 \leq KP_2$ if for all $K_1 \in KP_1$ there exists some $K_2 \in KP_2$, where the relationship $K_1 \leq K_2$ holds. In other words, each $K_1 \in KP_1$ is implied by some $K_2 \in KP_2$.

The FD Property

In the DB2 optimizer, the FD property is simply a set of FDs, which can be empty. Each FD originates from a key. A key becomes an FD when it fails to propagate through a join. The columns of the key become the new FD's head, and the remaining columns in the key's stream become the new FD's tail. As an example, assume $K = \{c_1\}$ is a key in the join stream S with columns $\{c_1, c_2, ...c_n\}$. Further assume that the key property KP of S does not propagate in the join. Then, $\{c_1\} \rightarrow \{c_2, ..., c_n\}$ is added to the FD property of the join. The same is done for all keys in KP. Note that if S had a one-record condition, then the empty-headed FD $\{\} \rightarrow \{c_1, c_2, ..., c_n\}$ would be generated.

The effect of projection on the FD property is similar to the effect of projection on the key property. Let $A = \{a_1, a_2, ..., a_n\}$ and $B = \{b_1, b_2, ..., b_m\}$. Then let F be a member of the FD property FP, where F is defined as $A \rightarrow B$. If any column a_i in A is projected, then F is removed from FP. In contrast, if any column b_i in B is projected, then F' replaces F, where F' is identical to F but with b_i removed from B.

Except for projection, FDs almost always propagate unchanged. In a join, the FDs of the outer and inner stream are combined and keys that do not propagate are used to generate new FDs, as described above. The resulting set of FDs can then be used to infer still more FDs [DD92]. This is not done in the DB2 optimizer because of its complexity, which is NP-complete in the general case [BB79].

Like the key property, an attempt is made to keep each FD property as "succinct" as possible by removing FDs that have become redundant because of projections and/or applied predicates. First, each FD is rewritten in a canonical form by substituting each column with its equivalence class head and removing redundant columns from both the head and tail. Then redundant FDs are removed from the FD property using the definition of "$\leq$" that follows: Let F_1 be defined as $A_1 \rightarrow B_1$ and let F_2 be defined as $A_2 \rightarrow B_2$. Then, $F_1 \leq F_2$ if $A_2 \subseteq A_1$ and $B_2 \supseteq B_1$. If $F_1 \leq F_2$, then F_1 is implied by F_2. In that case, F_1 is redundant and can be removed from the FD property.

The definition of "$\leq$" is also used to compare the FD properties FP_1 and FP_2 of two plans during pruning. More specifically, $FP_1 \leq FP_2$ if for all $F_1 \in FP_1$ there exists some $F_2 \in FP_2$, where the relationship $F_1 \leq F_2$ holds. In other words, each $F_1 \in FP_1$ is implied by some $F_2 \in FP_2$.

6 An Example

An example that illustrates how some of the techniques tie together is shown in Figure 6. In the example, the ORDER BY's interesting order $OB = (a.x)$ was pushed down and covered with the GROUP BY's interesting order $GB = (a.x, a.y, b.y)$. The resulting cover was then pushed down and itself covered with the merge-join's interesting order $MJ = (b.x)$. The key on $b.x$ gives rise to $\{b.x\} \rightarrow \{b.y\}$, which propagates through all the joins. This FD and the equivalence class generated by the predicate $a.x = b.x$ allowed the optimizer to detect that GB can be reduced to $GB = (a.x, a.y)$. As a result, the sort on $a.x, a.y$ simultaneously satisfies all interesting orders.

As shown, the optimizer determined that pushing

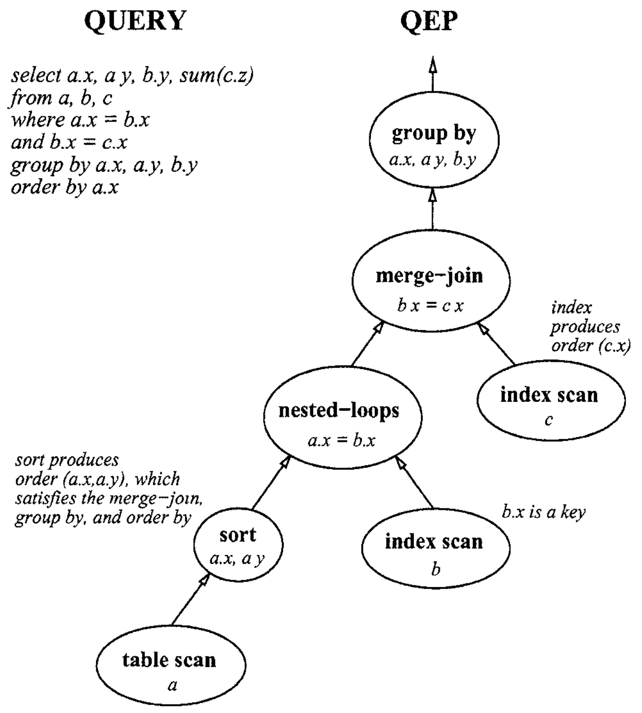

Figure 6: Query Example

down the sort before the first join results in the most efficient QEP. This is likely to be true if the size of table a is smaller than the result of either join. Because of the indexes on $b.x$ and $c.x$, the resulting QEP would probably beat one that used hash-based operators. Finally, note that the sort could be eliminated if there was an ordered index on $a.x, a.y$.

7 Advanced Issues

One of the issues that we have tacitly avoided in this paper is the fact that the order-based GROUP BY and DISTINCT operators do not dictate an exact interesting order. For example, consider a GROUP BY for $x, y, sum(distinct\ z)$. This can be satisfied by (x, y, z) or (y, x, z). Moreover, x, y, and z can be in ascending or descending order. In fact, a total of sixteen different orders can satisfy the order-based GROUP BY.

Rather than generate sixteen different interesting orders, one general interesting order is used in the real implementation. It includes information about which columns can be permuted and which columns can be in ascending or descending order. Using this information, the DB2 optimizer can correctly detect any order that satisfies the order-based GROUP BY. Accounting for these "degrees of freedom" adds a non-trivial amount of complexity to all operations on orders. It probably doubled the amount of code. In general, though, the same underlying logic that has been described in this paper still prevails.

8 Performance Results

Clearly, the techniques described in this paper for order optimization can only improve the quality of execution plans produced by an optimizer. In cases where an execution plan's performance would degrade, which can happen with sort-ahead, an optimizer would simply pick a better alternative using its cost estimates. Therefore, the only question is whether the improvement in performance offered by our techniques is worth the implementation effort. More specifically, are there a lot of "real world" queries where the improvement in performance is significant?

IBM maintains a number of internal benchmarks that have been inspired by real DB2 customers over the years. On those benchmarks and at customer sites, we have observed substantial improvement in the performance of many queries because of the techniques described in this paper. The biggest improvements are typically seen in decision-support environments with lots of indexes. Often, applications in these environments cannot fully anticipate the predicates that will be specified by end-users at runtime. Nor can they anticipate schema changes, such as the addition of a new index or key. As a result, queries in these environments frequently include a lot of redundancy – grouping on key columns, sorting on columns that are bound to constants through predicates, and so on. Order optimization is able to eliminate this kind of redundancy, which in turn usually leads to a better exection plan.

8.1 TPC-D Results

Unfortunately, the benchmarks described above are unknown outside of IBM. Therefore, we turn to the TPC-D benchmark [1] to illustrate how much our techniques for order optimization can improve performance. A description of the TPC-D benchmark and its schema is omitted. For details, readers are directed to [Eng95].

TPC bylaws prohibit us from disclosing a full set of unaudited TPC-D results. Moreover, IBM was reluctant to let certain results be published when this paper was written. Consequently, the focus here will be on just Query 3 of the TPC-D benchmark. Query 3 was chosen because it is (relatively) simple and benefits from several of the techniques that have been described in this paper. Query 3 retrieves the shipping priority and potential revenue of the orders having the largest revenue among those that had not been shipped as of a given date. It is defined as follows:

[1] TPC-D is a trademark of the Transaction Processing Concil.

select l_orderkey,
  *sum(l_extendedprice * (1 - l_discount)) as rev,*
  o_orderdate, o_shippriority
from customer, order, lineitem
where o_orderkey = l_orderkey
and c_custkey = o_orderkey
and c_mktsegment = 'building'
and o_orderdate < date('1995-03-15')
and l_shipdate > date('1995-03-15')
group by l_orderkey, o_orderdate, o_shippriority
order by rev desc, o_orderdate

To gather performance results, we built a modified version of DB2 with order optimization disabled. Then we ran queries on both the production and disabled version of DB2. Results were obtained on a 1GB TPC-D database using a single IBM RS/6000 Model 59H (66 Mhz) server with 512MB of memory and running AIX 4.1. A real benchmark configuration was used, with data striped over 15 disks and 4 I/O controllers. Using a combination of big-block I/O, prefetching, and I/O parallelism, this configuration was able to drive the CPU at 100% utilization.

The results for Query 3 are shown in Table 1. The numbers in the table correspond to the elapsed time to run Query 3, averaged over five runs. As shown, the elapsed time for the version of DB2 with order optimization disabled was significantly slower than the production version of DB2 (by a ratio of 2.04).

Production DB2	Disabled DB2	Ratio
192 sec.	393 sec.	2.04

Table 1: Elapsed Time for Query 3

The execution plan chosen by the production version of DB2 is shown in Figure 7. Using a combination of Reduce Order, Cover Order, and Homogenize Order, the DB2 optimizer was able to determine that it was beneficial to push the sort for the GROUP BY below the nested-loop join. This sort not only provided the required order for the GROUP BY, but it also caused the index probes in the nested-loop join to become clustered. We refer to these as *ordered* nested-loop joins. Here, an ordered nested-loop join is especially important because it allows prefetching and parallel I/O to be used on the *lineitem* table, which is the largest of all the TPC-D tables.

In Figure 7, note that the sort on *o_orderkey* satisfied the GROUP BY because of the equivalence class generated by the predicate *o_orderkey* = *l_orderkey* and because of the FD $\{o\_orderkey\} \rightarrow \{o\_orderdate, o\_shippriority\}$. In SQL queries, there is often no choice but to include functionally dependent

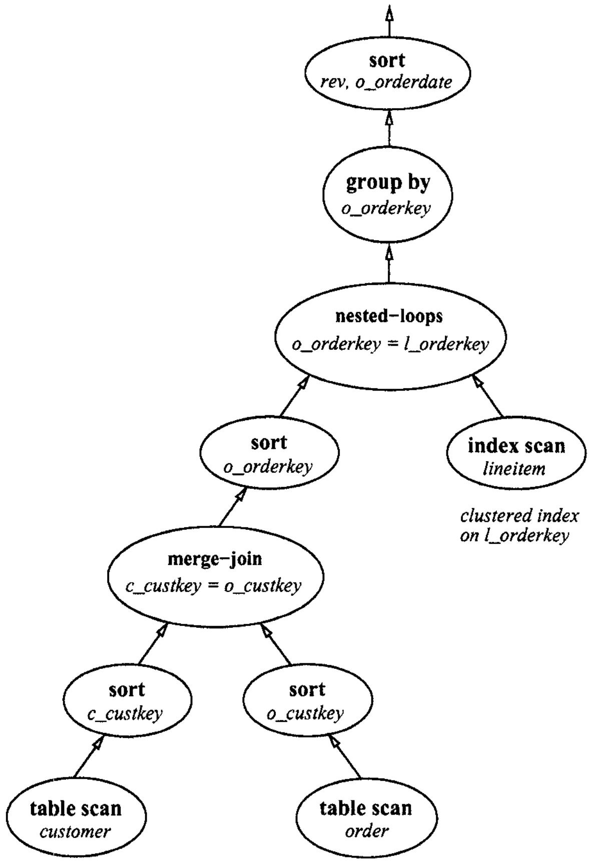

Figure 7: Query 3 in Production Version of DB2

(i.e., redundant) columns like these in a GROUP BY, since that is the only way to have them appear as output.

For comparison, the execution plan chosen by the version of DB2 with order optimization disabled is shown in Figure 8. In this case, the DB2 optimizer was unable to detect that the sort on *o_orderkey* satisfies the GROUP BY. Moreover, without an awareness of equivalence classes, the optimizer was unable to determine that the same sort could be used to generate an ordered nested-loop join for the *lineitem* table. Consequently, a more costly merge-join was used.

9 Conclusion

This paper described the novel techniques that are used for order optimization in the query optimizer of IBM's DB2. These general techniques, which can be used by any query optimizer, make it possible to detect when sorting can be avoided because of predicates, keys, indexes, or functional dependencies; the minimal number of sorting columns when a sort is unavoidable; whether a sort can be pushed down into a view or join tree to make it cheaper; and whether two or more sorts can be

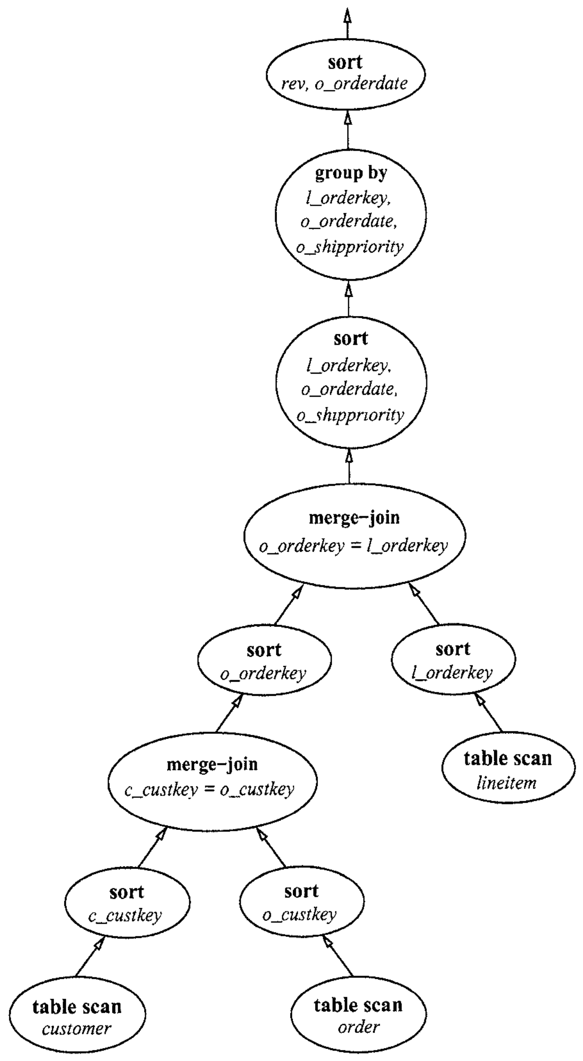

Figure 8: Query 3 with Order Optimization Disabled

combined and satisfied by a single sort. For complex queries in a data warehouse environment, these techniques can mean the difference between an execution plan that finishes in a few minutes verses one that takes hours to run.

This paper's main contribution was a set of fundamental operations for use in order optimization. Algorithms were provided for testing whether an interesting order is satisfied, for combining two interesting orders, and for pushing down an interesting order in a query graph. All of these hinge on a core operation called *Reduce Order*, which uses functional dependencies and predicates to reduce interesting orders to a simple canonical form.

This paper also described the overall architecture of the DB2 optimizer for order optimization. In particular, the paper described how order, predicates, keys, and functional dependencies can be maintained as access plan properties. The importance of maintaining functional dependencies as a property goes beyond just order optimization. Functional dependencies can be used for other optimizations as well [DD92].

Finally, results for Query 3 of the TPC-D benchmark were provided to illustrate how much the techniques described in this paper can improve performance. On a 1GB TPC-D database, a version of DB2 with order optimization disabled ran Query 3 roughly 2x slower than the production version of DB2 with order optimization enabled.

Acknowledgements

The authors would like to thank Bobbie Cochrane, Guy Lohman, and Jeff Naughton for reading earlier drafts of this paper. Thanks also go to Bernie Schiefer for generating TPC-D benchmark results.

References

[Ant93] G. Antosheknov. Query processing in dec rdb: Major issues and future challenges. In *IEEE Bulletin on the Technical Comittee on Data Engineering*, December 1993.

[BB79] C. Beeri and P. Bernstein. Computational problems related to the design of normal form relational schemas. In *ACM Transactions on Database Systems*, March 1979.

[BD83] D. Bitton and D. DeWitt. Duplicate record elimination in large data files. In *ACM Transactions on Database Systems*, June 1983.

[BE76] M. Blasgen and K. Eswaran. On the evaluation of queries in a relational data base system. Technical Report 1745, IBM Santa Teresa Lab, April 1976.

[CS93] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proceedings of the 19th International Conference on Very Large Data Bases*, August 1993.

[DD92] H. Darwen and C. Date. The role of functional dependencies in query decomposition. In *Relational Database Writings 1989-1991*. Addison Wesley, 1992.

[DKO$^+$84] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, June 1984.

[Eng95] S. Englert. Tpc benchmark d. In *Transaction Processing Performance Council*, 777 N. First St, Suite 600, San Jose CA 95112-6311, October 1995.

[GD87] G. Graefe and D. DeWitt. The exodus optimizer generator. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, June 1987.

[Gra93] G. Graefe. Query evaluation techniques for large databases. In *ACM Computing Surveys*, June 1993.

[Hel94] J. Hellerstein. Pratical predicate placement. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, June 1994.

[HFLP89] L. Haas, J. Freytag, G. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, June 1989.

[JV84] M. Jarke and Y. Vassiliou. Query optimization in database systems. In *ACM Computing Surveys*, June 1984.

[Loh88] G. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, June 1988.

[OL90] K. Ono and G. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the 16th International Conference on Very Large Data Bases*, August 1990.

[PHH92] H. Pirahesh, J. Hellstein, and W. Hasan. Extensible rule based query rewrite optimization in starburst. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, June 1992.

[PL94] G. Paulley and P. Larson. Exploiting uniqueness in query optimization. In *International Conference on Data Engineering*, February 1994.

[SAC$^+$79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, June 1979.

[YL93] P. Yan and P. Larson. Performing group-by before join. In *International Conference on Data Engineering*, February 1993.