Query Optimization in the IBM DB2 Family

Peter Gassner
IBM Santa Teresa Laboratory
San Jose, CA 95161 USA
gassner@vnet.ibm.com

Guy M. Lohman
IBM Almaden Research Center
San Jose, CA 95120 USA
lohman@almaden.ibm.com

K. Bernhard Schiefer
IBM Toronto Laboratory
Toronto, Ontario, M3C 1W3 Canada
schiefer@vnet.ibm.com

Yun Wang
IBM Santa Teresa Laboratory
San Jose, CA 95161 USA
wang@stlvm14.vnet.ibm.com

Abstract

This paper reviews key query optimization techniques required by industrial-strength commercial query optimizers, using the DB2 family of relational database products as examples. The currently available members of this family are DB2/2, DB2/6000, and DB2 for MVS[1].

1 Introduction

Query optimization is the part of the query compilation process that translates a data manipulation statement in a high-level, non-procedural language, such as SQL, into a more detailed, procedural sequence of operators, called a plan. Query optimizers usually select a plan by modelling the estimated cost of performing many alternative plans and then choosing the least expensive amongst them.

Ever since the invention of the relational model in 1970, IBM has been actively involved in research on relational query optimization, beginning with the pioneering work in System R [SAC+79] and continuing with the R* distributed relational DBMS prototype [DN82], [LDH+84] and the Starburst extensible DBMS prototype [Loh88a], [LFL88], [HP88], [HCL+90], [PHH92]. The incorporation of this important technology into industry-leading products is documented primarily in IBM manuals, however, with a few recent exceptions [Mal90], [TMG93], [Moh93], [Wan92]. Without such documentation in the research literature, it is easy for researchers to lose sight of the challenges faced by product developers, who cannot just assume away the plethora of details in the SQL language and who must build "industrial-strength" optimizers for a wide variety of customer applications and configurations.

This paper summarizes some of these "industrial-strength" features of the query optimizers in IBM's DB2 family of products that evolved from this research, including the original DB2 which runs on the MVS operating system [DB293b] [DB293a] [DB293d], and the more recently released DB2 client/server products for the AIX and OS/2 operating systems: DB2/6000 and DB2/2 [DB293e] [DB293f]. To avoid confusion in this paper, DB2 for MVS will be used to refer to the original DB2. Due to space limitations, we must omit discussion of similar products for the VM/ESA and VSE/ESA operating systems (SQL/DS, IBM's first relational DBMS product [SQL93a] [SQL93b]) and for the OS/400 operating system (SQL/400). All of these products evolved

[1]The following are trademarks or registered trademarks of the International Business Machines Corporation in the United States and/or other countries: DB2, DB2/2, DB2/6000, SQL/DS, SQL/400, AIX, OS/2, OS/400, VSE/ESA, VM/ESA, Extended Edition, Database Manager, and Extended Services.

from the System R prototype, and thus incorporated many of the strengths of that work: optimizing queries once at compile time for repeated execution, using a detailed model to estimate the execution costs of alternative plans, and a wide repertoire of alternative access paths and join methods.

However, each product was tailored to its environment, often necessitating different implementations, especially of different execution strategies. The query optimizer for DB2 for MVS, which first appeared in 1983, largely followed that of System R but was totally rewritten to ensure robust error handling and performance, as well as complete coverage of SQL, correct handling of null values, and support for multiple languages and character sets. DB2 for MVS, written in a proprietary systems-development language, continues to be enhanced by IBM's Santa Teresa Laboratory in San Jose, California. In December of 1993, Version 3 was made generally available, with significant performance and availability enhancements.

The DB2 client/server products have evolved from the OS/2 Extended Edition Database Manager, which was written in C by IBM's Entry Systems Division in Austin, Texas. It used some preliminary code (mostly in the data manager) and concepts (an early version of the Query Graph Model [PHH92] and a cost-based version of a greedy algorithm [Loh88b]) from the Starburst prototype at that time. It was first released in 1988 as the OS/2 Extended Edition 1.0 Database Manager and was enhanced several times. In 1992, development and support for the OS/2 database product was moved to IBM's Toronto Laboratory in Canada, which in May 1993 released a 32-bit version of DBM, called DB2/2, for OS/2 2.x. Austin began, and Toronto completed, a port of this product to AIX (with major enhancements) and released it for general availability as DB2/6000 in November 1993. IBM is planning to extend the DB2 client/server products by incorporating significant portions of the Starburst prototype (the SQL compiler, including its rule-based query rewrite [PHH92] and optimizer [Loh88a] [LFL88] technologies), adding several important object-oriented extensions (joint work between the Database Technology Institute at Santa Teresa Laboratory and the Starburst project [LLPS91]), supporting parallel query execution (joint work between Toronto and the IBM T.J. Watson Research Center in Hawthorne, NY), and porting to non-IBM platforms, including Hewlett-Packard HP 9000 workstations and servers using HP-UX. Since DB2/2 and DB2/6000 share a common code base and will both offer the same functions, we will refer to both as DB2/* when there are no differences between the two products.

The remainder of this paper is structured as follows. We will first define what we mean by "industrial strength" in Section 2. Section 3 will cover transformations to the original SQL queries that standardize and simplify the query representation and give the cost-based optimizer more latitude in selecting an optimal plan. In Section 4, we present some of the more unusual access path and join execution strategies which the optimizer considers. Section 5 will discuss various algorithms for ordering joins, which determines the algorithmic complexity of query optimization. We present unusual aspects of the models for cardinality estimation and cost estimation in Section 6. Section 7 covers some more advanced features of the DB2 optimizers, including tricks used to avoid expensive operations, data manager features that the optimizer must model correctly, and run-time optimizations. Section 7.3 deals with other special considerations of which the optimizer must be aware. Finally, we conclude in Section 8.

In all of these sections any technique described without qualification applies to all the members of the DB2 family. In the MVS environment, all the references are to DB2 Version 3. The references to DB2/2 and DB2/6000 are to Version 1, except where noted.

2 "Industrial Strength" Optimization

```
''The devil of it is in the details.''
                    -- H. Ross Perot
```

Product optimizers must deal with many aspects of query optimization seldom even considered by research prototypes.

First of all they must, of course, optimize the *entire* SQL language. This includes support for predicates involving: complex AND, OR, and NOT clauses; IN and LIKE predicates; IS NULL and NOT NULL predicates; UNION, INTERSECT, and EXCEPT clauses; nested subqueries (and even *expressions* containing multiple subqueries!); correlation (references within a subquery to values provided in another $SELECT...FROM...WHERE...$ query block of the same query); and many other constructs that affect the optimizer to varying degrees.

Secondly, product optimizers must have robust performance, error handling, and the ability to work with constrained resource requirements so that very complex queries can be optimized reliably and in a reasonable amount of time. Applications may involve hundreds or even thousands of queries, ranging from single-table queries to complex nesting of views [TO91]. Without careful management, space-saving data representations, and re-use of fragments of alternative plans, optimization of queries involving more than a handful of tables, columns, and predicates will quickly consume a prohibitive amount of space and time. Most difficult of all, the result the user sees must be the same regardless of which plan the optimizer chooses or of any unusual situations that may arise in parts of any plan (e.g., finding no rows, NULLs, run-time exceptions, etc.).

Thirdly, product optimizers must deal with "nitty-gritty" details such as supporting national-language character sets (which often have different sorting sequences), ensuring correct answers under different isolation levels (especially cursor stability), handling nulls correctly (Do they sort high or low? Are they included in aggregate functions such as COUNT()? Is three-valued logic involved? etc.), and dealing with the aspects of the underlying data manager's implementation that must be specified in the plan and modeled accurately in the cost model. We will give examples of these throughout this paper, and emphasize the latter topic more in Section 7.2 below.

Fourthly, product optimizers must model and choose from a wide repertoire of execution strategies, each of which may be optimal for some query. In this paper we will focus on these strategies, highlighting some of the more unusual and less well-known features of IBM's DB2 family of relational DBMSs.

Lastly, and perhaps most importantly, optimizers for products must be very responsive to the changing requirements of their clientele, the customers, without adversely affecting existing applications. DB2 for MVS is a mature product with thousands of licenses on multi-user systems, many of which support thousands of users. Applications running on DB2 for MVS are often the information processing "bread and butter" of Fortune 500 companies. Customers expect only improvements (no degradation in the performance of *any* plan) with each new release of the product. DB2 for MVS tends to implement optimization strategies that will yield a decent access path most of the time, rather than risking performance problems on more obscure access paths that are more difficult to model accurately. Generally, a roughly 99-1 rule is used, meaning that a feature that is expected to offer improvement to 99% of queries but risks degradation of 1% of queries will not be put in the product.

In the highly competitive workstation world of DB2/*, users are demanding greater functionality, particularly for object-oriented features such as user-defined types and functions, binary large objects (BLOBs), constraints, triggers, and parallel execution. Besides the "industrial strength" features customers have come to expect from IBM DBMSs, DB2/* must also incorporate these new features quickly while complying with industry standards such as the recently published ANSI SQL92 standard and the emerging ANSI SQL3 standards. DB2/* is therefore aggressive in incorporating new functionality while striving to maintain industry-leading performance.

3 Query Transformation

A query can often be represented with several alternate forms of SQL. These alternate forms exist due to redundancies in SQL, the equivalence of subqueries and joins under certain constraints, as well as logical inferences that can be drawn from predicates in the WHERE clause. From their introduction, the DB2 family of products used cost-based optimization to choose the optimal access plan for a query, regardless of how it was originally formulated by the application programmer (e.g., ignoring the order in which tables were listed in the FROM list). However, they performed only a small number of semantic transformations. Documentation was provided to the

user with guidance in writing SQL. Some transformations were more difficult to describe, since they couldn't be shown to be universally beneficial.

Both DB2 for MVS and DB2/* prune the SELECT columns of an EXISTS subquery so that these (unneeded) SELECT columns will not have any effect on access path determination (see Section 4.1 below) or run-time performance. Both optimizers also merge views into the query whenever possible and evaluate predicates and project unused columns as soon as possible. In addition, DB2 for MVS will push predicates from the SELECT statement into a view that has to be materialized into a temporary table; materialization will be in-memory unless the table exceeds the available buffer. DB2/* transforms an IN predicate with a list of literals into a sequence of OR'ed predicates to enable the use of the "Index-ORing" table access strategy (see Section 4.1 below). It also evaluates at compile time certain expressions containing scalar functions with arguments that are constants, in order to avoid evaluating them at run-time.

All the DB2 products perform some logical transformations. For example, the NOT predicate is distributed whenever it is beneficial. Likewise, the pattern of a LIKE predicate is analyzed to see if a pair of bounding values (defining a range) can be extracted to reduce the number of rows for which the full pattern must be evaluated. DB2/* also employs DeMorgan's Law to produce conjuncts out of disjuncts, since this makes it easier for the optimizer to reorder predicate application.

DB2 for MVS generates the transitive closure of equality predicates, for both single-table and join predicates, to allow earlier filtration of rows and more potential join sequences. For example, if the user specified T1.C1 = T2.C2 AND T2.C2 = 5, DB2 will generate the implied predicate T1.C1 = 5 in order to reduce the number of T1 rows as soon as possible. Similarly, join predicates of T1.C1 = T2.C2 AND T2.C2 = T3.C3 will cause DB2 to generate T1.C1 = T3.C3, without which the join enumerator would have deferred considering a join between T1 and T3. When there are more than a preset number of tables in a join, join predicate transitive closure is not performed in order to keep the search space of the dynamic programming join enumeration strategy in check.

Transformations from subqueries to joins often improve performance dramatically but are also quite tricky (witness the number of papers that have published erroneous transformations). DB2 for MVS is very careful with query transformations. For example, subquery-to-join transformation is only done when there is only one table in the subquery, and only when that table has a unique index that is fully matched by equality predicates; this is a case where the transformation is always beneficial. DB2 for MVS employs other techniques to improve the performance of subqueries, such as caching results from correlated subqueries.

DB2/* has a slightly different set of subquery transformation techniques. DB2/* uses the lowest and highest values of an uncorrelated subquery to provide starting and stopping conditions (see Section 4.1 below) for the outer query block. As well, DB2/* will treat certain quantified (ALL, EXISTS) subqueries specially. For example,

```
SELECT C1
FROM T
WHERE C1 > ALL ( SELECT C2 FROM T2 )
```

will be treated as though it was written as

```
SELECT C1
FROM T
WHERE C1 > ( SELECT MAX(C2) FROM T2 )
```

The single value from the subquery can be used as a start condition on C1, and saves having to build a temporary table. Special care is taken to ensure that the transformation preserves the original semantics in the presence of null values or no rows.

DB2/* with Starburst will have a completely separate compilation step devoted to rewriting queries, based upon the Starburst rule-based query rewrite technology [PHH92]. This will add many additional transformations

to DB2/* Version 1's transformation repertoire, an adequate description of which demands more space than that available here.

4 Access Plan Strategies

Fundamental to any optimizer is the set of alternative strategies it supports for accessing individual tables and for joining them together.

4.1 Single Table

The two basic access paths available in the DB2 optimizers are access by index and access by table scan. Access by table scan simply examines every row in the tables, while optionally applying predicates. The optimizer must carefully separate and model both "SARGable"[2] and "residual" predicates. SARGable predicates are applied while the page is pinned in the buffer, to save the unnecessary CPU expense of copying a row. Predicates involving a nested subquery (another SELECT FROM WHERE query block) that depends upon (is *correlated to*) some value in the table being scanned must be recomputed for each such value. To avoid potential self-deadlock when additional pages are read into the buffer for the subquery, DB2 defers application of these "residual" predicates until each row is copied from the buffer.

DB2/* stores each table's records separately from those of any other table. DB2/2 Version 1 places them all in a single file, while DB2/6000 Version 1 splits them into partitions known as segments. This restriction is relaxed in DB2 for MVS, but generally customers find that it is more efficient to allocate each table in its own physical space.

Indexes are supported in DB2 and DB2/* by multi-column B+ trees, and are useful for ordering, grouping similar values, ensuring uniqueness, providing an alternative to accessing the base table (*index-only access*), and accessing directly only those rows satisfying certain predicates. The DB2 optimizers exploit a wide variety of very sophisticated index access strategies.

Perhaps the most important role of indexes is the latter one: applying predicates to minimize the data pages that must be visited. Predicates that reference only columns of the index can be applied as SARGs to the index key values as the index leaf pages are scanned. However, SARGs that may be used to define the starting and stopping key values for the scan of the index will further reduce I/O by limiting the scan of the index to some subtree. The DB2 optimizers therefore go to great lengths to exploit as many predicates as possible as start/stop conditions. Simple predicates involving the usual comparison of a column to a value or expression (e.g., DEPTNO = 'K55') are candidate start/stop conditions, as are join predicates that are "pushed down" on the inner table of a nested-loops join (see next subsection) and even a subquery that returns a single value (e.g., EMP.SALARY = (SELECT MAX(SALARY) FROM EMP)). More complex predicates such as BETWEEN (a pair of range predicates), LIKE (a pair of range predicates may be extracted if there are no "wild card" characters in the leading character of the LIKE pattern), and the IS NULL predicate can also be used as start/stop conditions. An $IN(< list >)$ predicate can even be exploited by either sorting the $< list >$ values and successively using each value as a start/stop key (in DB2 for MVS) or using row identifier union (in DB2/*), which is described below. To keep index operations efficient, expressions or even type conversions on an indexed column generally precludes using that column as a start or stop condition. However, in DB2/*, the datatypes in the comparison need not be identical; they need only satisfy the regular ANSI comparability requirements between numbers and characters.

Determining when predicates can be used as start/stop conditions, and when they also have to be reapplied as SARGs, is surprisingly tricky. Index keys are formed by concatenating column values from any number of columns of the index, but start/stop conditions will reduce an index scan only if the high-order (leading

[2]This acronym originated from Search ARGument.

or prefix) columns are first bound by predicates. For example, an index on columns X and Y cannot form a beneficial start/stop key from just one predicate on Y without a candidate predicate on X as well. As long as the the predicate comparison operator is "=", that predicate can be totally applied as a start/stop key only. So long as predicate comparison on a column is "=", ">=", or "<=", predicates on the succeeding column of the index are candidates for start/stop conditions. However, after the first inequality, predicates on successive columns of the index are not beneficial, and would have to be re-applied as SARGs anyway. For example, suppose we have a single index on columns X,Y, and Z. For predicates $X = 5$, $Y >= 7$, and $Z >=' B'$, DB2 would construct a starting key of $5||7$, a stopping key of 5, and would apply the predicate on Z as a SARG. DB2/* with Starburst will construct a key of $5||7||'B'$ and *will also re-apply the predicate on Z as a SARG*, because an index scan beginning at that key value would have included values such as $5||8||'A'$ that don't satisfy the predicate on Z. Had the predicate on Y been strict inequality, DB2/* Version 1 would have started with $5||7$ and applied the predicates on Y and Z as SARGs. Since a column of an index may individually be declared to be in descending order, descending index columns must flip start and stop conditions, e.g., $Z <' B'$ is a stopping condition when Z is descending. Finally, when a unique index is "fully qualified" (i.e., has equality predicates binding all columns), all DB2 optimizers recognize that at most a single row can be retrieved, so that a streamlined access may be used by the data manager and other processing short-cuts can be taken.

If the optimizer determines that an index contains all the columns of a table referenced in a query, or the query is of the form

```
SELECT MIN(SALARY) FROM EMP,
```

it can save fetching the data pages of the base table by using the index-only strategy. Otherwise, fetching these pages can be done immediately, or deferred until all the RIDs (Row IDentifiers) obtained from the index are accumulated. The latter case allows lists of RIDs to be further processed. For example, DB2 for MVS allows the RIDs to be sorted first, which arranges the RIDs in page (physical) order, improving the effective clustering of non-clustered indexes and hence minimizing the number of times a given page will be retrieved. DB2/* with Starburst will also support this strategy.

When the WHERE clause contains more than one predicate or combination of predicates that can be used as start/stop keys on an index, the DB2 optimizers will also consider multiple scans of the same index or even scanning multiple indexes for the same table. For example, if the predicate is

```
ZIPCODE BETWEEN 90100 AND 90199 OR
ZIPCODE BETWEEN 91234 AND 91247
```

and there is an index on ZIPCODE, DB2/* would consider a plan that accesses that index twice, once with start/stop keys of (90100,90199) and once with (91234,91247), and then unions the RID lists together, removing duplicates. This is sometimes referred to as "index ORing".

DB2/* with Starburst will extend this RID-list processing to include intersecting ("ANDing") RID lists, which will save data page fetches if there are indexes on all the referenced columns. For example, for the query:

```
SELECT C1, C2
FROM T
WHERE C1 = 10 AND C2 = 47
```

the RIDs from an index scan on C1, with start key 10 and stop key 10, will be sorted and then ANDed with those from another index scan on C2, with start key 47 and stop key 47. Since all the columns referenced in the query can be accessed via indexes, no data pages need be fetched.

DB2 for MVS currently performs both index ORing and index ANDing with many indexes in complicated combinations of AND and OR predicates, using the techniques of [MHWC90]. The same index can also be used many times. For example, if table T had a single index I on columns C1 and C2, then for the query:

```
SELECT C1,C2
FROM T
WHERE C1 = 10 AND (C2 = 32 OR C2 = 97)
```

DB2 for MVS could use index I twice, once with a start and stop key of $10||32$, and once with a start and stop key of $10||97$. Any level of AND/OR nesting is supported and any number of indexes may be used with the limitations due to memory constraints. For multiple index processing, the optimizer does a very detailed analysis to determine the best processing order that will minimize both memory usage (number of RIDs held) and table accesses. At run-time, this plan may be altered or stopped early, as described in Section 7.4.

One danger in using an index in an UPDATE statement is that updated rows might be placed ahead of an index scan and updated again, if an index on the updated column is chosen. For example, the following query might result in everyone getting an infinite raise if an index on SALARY is used to scan EMP:

```
UPDATE EMP
SET SALARY = SALARY * 1.1
```

This semantic anomaly was affectionately named the "Halloween problem" by the late Morton Astrahan because Pat Selinger discovered it on Halloween. It is a problem only because row accesses and updates are pipelined, which is normally beneficial but is easily prevented in this case by accumulating all RIDs to be updated before beginning the update.

4.2 Joins

Joining tables is one of the most critical operations to optimize well, because it is common, expensive, and comes in many flavors. Both DB2/* and DB2 for MVS support the nested-loops and (sort-) merge join algorithms, in their usual implementations. Nested-loop and merge join have started to "blend together" over the years, since both will bind the outer value of a join predicate and "push it down", as though it were a single-table predicate, to restrict access on the inner table. The basic difference remaining is that both of the merge join's inputs must be ordered (either by indexes or sorting), and the inner input must be from a single ordered table (either a base or temporary table), in order to use the row identifier (RID) as a positioning mechanism to jump back to an earlier position in the inner table when a duplicate in the outer table is encountered. For nested-loops join, the DB2 optimizers will consider sorting the outer table on the join-predicate columns, and DB2/* even considers sorting a table before the join for a later GROUP BY, ORDER BY, or DISTINCT clause.

The DB2 for MVS optimizer also supports the hybrid join [CHH+91], a mixture of the other two join methods. It is possibly best described as a nested-loops join with an ordered outer table and batched RID processing on the inner table". The hybrid join is often beneficial when only unclustered indexes are available on the inner table and the join will result in fewer tuples. Like a merge join, hybrid join usually requires the rows of both the inner and outer inputs to be ordered. Like a nested-loops join, it avoids having to put the inner table into a temporary table, but accesses the inner table's data pages efficiently, because it first sorts the list of RIDs and defers fetching any data pages until after the join. It can also be efficiently combined with other RID processing such as index ANDing.

When performing a nested-loops join using an index scan for the inner table, if the outer table is in join-predicate order, DB2 for MVS optimizer will utilize "index look-aside". This feature remembers the position in the leaf page last visited in the inner's index, thus saving the traversal of non-leaf index pages. The DB2 for MVS optimizer models this behavior where possible.

5 Join Enumeration

Using the access and join strategies of the previous section, the DB2 optimizers consider alternative join sequences, searching for a good plan in the same bottom-up fashion, but with different algorithms.

DB2 for MVS uses dynamic programming to enumerate different join orders. The basic algorithms are detailed in [SAC+79]. Generally, two tables will not be joined together if there is no join predicate between them. However, special-case heuristics are used to attempt to recognize cases where Cartesian joins can be beneficial. Any tables that are guaranteed to produce one row because of a fully-qualified unique index are fixed at the beginning of the join order. This is a "no lose" situation, is very safe, and reduces optimization time. DB2 for MVS will also take advantage of the fact that these "one row" tables do not affect the ordering of the result set.

As the number of tables participating in the join increases, the time and space requirements of a dynamic-programming join enumeration algorithm may be prohibitive on a small PC running OS/2. As a result, DB2/* uses a "greedy" algorithm [Loh88a] which is very efficient, since it never backtracks. Since the greedy algorithm always pursues the cheapest joins, it is possible for the optimizer to cache the result of a join in a temporary table and use it as the inner table of a later join. DB2/*, therefore, permits composite inners ("bushy tree" plans). As with DB2 for MVS, join predicates are used to guide possible joins. Only when no join predicates are left will a Cartesian product be performed.

In DB2/* with Starburst, the user will be able to expand the plan search space from a very limited space to one larger than that of DB2 for MVS. Compile-time options will permit the user to specify either the dynamic programming algorihm or the greedy heuristic to optimize each query, to allow composite inners or not, and to defer Cartesian products to the end or to permit them anywhere in the plan sequence. Searching a larger percentage of all the join combinations may allow the optimizer to find a more efficient plan but results in increased processing costs for the query optimization process itself [OL90]. DB2/* with Starburst will also consider any predicate referencing more than one table to act as a join predicate, avoiding potential Cartesian products. For example, a predicate of the form $X.1 + Y.2 > Z.4$ can be used to join tables X, Y, and Z. In addition, the implementation-dependent limits on the number of tables that may be referenced in a query will be removed; the only limit will be the amount of memory available to store plan fragments during optimization.

6 Modeling Plan Execution Costs

All the DB2 optimizers use a detailed mathematical model to estimate the run-time cost of alternative strategies and choose the cheapest plan. An important input to this cost model is a probabilistic model of how many rows will satisfy each predicate. These detailed models have been thoroughly validated and are essential to the selection of the best plan.

6.1 Cost Estimation

The DB2 optimizers use a cost-based model that estimates both I/O and CPU costs, which are then combined into a total overall cost. DB2 for MVS normalizes the CPU costs based on the processor speed of the CPU. DB2/* with Starburst will determine the appropriate I/O and CPU weights when DB2/* is installed by timing the execution of small programs with a known number of instructions and accessing a fixed number of pages.

The CPU cost formulas are arrived at by analyzing the number of instructions that are needed for various operations, such as getting a page, evaluating a predicate, traversing to the next index entry, inserting into a temporary table, decompressing a data row (which is done a row at a time), etc. The instruction costs of these operations have been carefully validated and are generally quite accurate as long as the I/O behavior and result size estimates are accurate.

I/O costs are more difficult to estimate reliably. Assumptions must be made about buffer pool availability and data clustering. Generally, DB2 for MVS will assume that a very small percentage of a buffer pool is available to any specific table. However, when there is high reference locality (such as an inner index, possibly the inner table of a nested-loops join or the inner table index of a hybrid join), a more liberal assumption is made about

whether a data or index page will remain memory-resident during the query. Indexes in DB2 for MVS normally have 3 or 4 levels in the index tree. These levels often have different buffer pool hit ratios and they each have different I/O cost formulas. Modelling the buffer pool hit ratios of these levels has proven to be very important for customers who do online transaction processing on large (greater than 10M rows) tables. Data page I/O is influenced by the reference pattern within the query, buffer pool size, and the degree to which data with common values is clustered together on data pages. The formulas for data access by sorted RID list are governed by the table size, cluster ratio of the index, and selectivity of the predicates on the index.

DB2 for MVS allows a user to specify the OPTIMIZE FOR N ROWS clause on any query to indicate that only N rows will be fetched from the cursor before closing it [DB293c]. In this case, DB2 for MVS will separate costs into those costs that are associated with a cursor being opened and costs that are incurred with each fetch. The access path picked will be the one that costs the least for an open call and N fetch calls, which may be very different from the access path without specifying this clause. If the "N rows" path is selected, DB2 for MVS may turn off sequential prefetch (discussed in Section 7.2 below) if it can determine that only a few pages will be needed to return N rows. Consider a customer running at 100 transactions per second. Prefetching may cause 64 index pages and 64 data pages to be read when only one fetch is done. This is about .5 MB of data, or 50 MB/sec of data being read from disk and brought into the buffer pool, most of it unnecessarily. This extra work may cost a customer tens of thousands of dollars per year in extra I/O, memory, and CPU costs. Because of the unavoidable inaccuracies of filter factors and statistics, the OPTIMIZE FOR 1 ROW case has been specially coded to choose an access path that avoids a sort (data or RID), if possible, regardless of whether the optimizer estimates the query will only return one row or not.

6.2 Statistics and Filter Factors

Cost estimates are directly dependent upon how many times operations are performed, which in turn depends on the database statistics and the use of those statistics to estimate the cardinality of any intermediate result. All the DB2 family members keep an extensive list of statistics about the tables and indexes. For tables, the number of rows and data pages is kept. The column cardinality and high/low values are kept for each column of a table. DB2 for MVS also computes non-uniform distribution statistics for every column that is a first column of an index. The top ten values and their relative frequencies are kept, and are very important in accurately calculating the selectivity of predicates. DB2/* with Starburst will, optionally, also collect non-uniform distribution statistics. These statistics can be collected for all columns of a table and the user can specify the number of values to collect; finer granularity gives better filter factors but is more expensive to collect.

Important statistics kept for indexes include the first column and full key cardinality, the number of leaf pages and levels, and a measure of how well the data is physically stored relative to the keys in the index. The DB2 family of products computes this quantity, called "cluster ratio", in order to predict I/O requirements for accessing a table using an index.

The exact formulas used by the products to derive this statistic differs. Since it is difficult to characterize the I/O reference pattern over various buffer sizes and with prefetch, the cluster ratio is the most important and most elusive statistic.

In DB2 for MVS, statistics can be updated by the user through SQL UPDATE statements. The ability to change statistics manually is very useful, especially in modelling production applications on test databases. DB2/* with Starburst will support this capability as well.

DB2 for MVS and DB2/* share the same basic filter factor formulas [DB293c]. The default values, used for expressions, host variables, and parameter markers, have been tuned to the usage patterns of typical customers and are essentially the same in all the products.

All the products use the classic formulas for compound predicates, which assume independence of conjuncts and hence multiply their respective filter factors [SAC+79]. Occasionally, accurate filter factors for compound predicates can be difficult to estimate due to predicate correlation. To reduce this problem, DB2 for MVS

generally will use only the most selective predicate for any particular column when computing result size. It will also use statistics on indexes to catch potential correlation between predicates. For example, the combined (multiplied) filter factors for the predicates that fully qualify an index to get just one key value cannot be more selective than the filter factor derived by inverting the number of distinct key values. Predicates of the form:

```
C1 > :hv1
   OR (C1= :hv1 AND C2 >= :hv2)
   OR (C1= :hv1 AND C2 = :hv2 AND C3 >= hv3)
```

(where :hv1, :hv2, and :hv3 are host variables) are difficult to analyze because they are often used to position a cursor on an index in response to an open call, or after a COMMIT, ROLLBACK WORK, or other interruption in batch processing (note that COMMIT WITH HOLD does not completely solve this problem as the cursor still needs the initial open, and cursor hold does not work with restart logic). DB2 for MVS uses special formulas to try to counter the effects of correlation when AND predicates are found under OR predicates.

The model for join size estimation in DB2/* with Starburst will incorporate an improved algorithm for determining filter factors that also accounts for redundant predicates on the same column [SSar]. Since query rewrite adds any implied predicates, redundant predicates are more likely to occur.

7 Advanced Features

Besides the rich repertoire of alternative plans considered by the DB2 optimizers, and the thorough modeling of their cost, DB2's optimizers also employ a number of more advanced techniques to ensure excellent performance. These include methods to avoid expensive operations (such as sorts and the creation of temporary tables), modeling sophisticated data manager features such as sequential prefetch, locking level and isolation level considerations, and even some run-time optimizations.

7.1 Techniques to Avoid Expensive Operations

The DB2 optimizers employ many techniques for minimizing operations known to be expensive.

When many predicates are connected by AND predicates, evaluating first those predicates that are most likely to return FALSE will give the best performance. When DB2 for MVS evaluates predicates on a table or index, it divides the predicates into "classes" or types. The types of predicates that are more likely to be false are evaluated first. Simple equal predicates (the equal class of predicates, which includes IS NULL) are evaluated first, range predicates next, other predicates next, and subquery predicates last. Within a class of predicates, the predicates are executed in order of appearance in the SELECT list. Customer experience has indicated that it is better to give the user control over execution order within a class, because filter factor calculations cannot always be trusted.

All the DB2 products avoid temporary tables and sorting (which must first accumulate all rows in a temporary table) as much as possible, as this is expensive and delays delivery of the first rows to the user. For example, nested-loops joins allow a single outer row to be tested against multiple inners and returned to the user immediately. When temporary tables are necessary, they are accumulated in memory and spill to disk only when the available buffer is exceeded.

Any sorts needed for GROUP BY and ORDER BY are combined into one by reordering the GROUP BY columns, if possible. A unique index on a set of columns S can be used to satisfy the SELECT DISTINCT on a superset of S, even if that index is not used in the query, because the unique index defines a key on S and ensures that duplicate values of S will never be inserted into the table. Similarly, the GROUP BY columns can be freely reordered to match those of an index that provides the necessary order for the GROUP BY, saving a sort.

Sometimes knowledge about an operation can be exploited to perform it more efficiently. For example, all the DB2 products stop evaluating EXISTS subqueries as soon as a single qualifying row is found. Also,

aggregation queries without a GROUP BY can exploit specific indexes to finish processing as soon as the first qualifying row is found. For example,

```
SELECT MAX(ORDER_NUMBER)
FROM SALES
```

need only retrieve the first row using a descending index on ORDER_NUMBER.

7.2 Modelling Advanced Data Manager Features

An optimizer's repertoire is, of course, limited by what features its underlying data manager supports. Conversely, the optimizer is often highly challenged to correctly model and construct plans for advanced data manager features that improve performance. The data managers of the DB2 family have implemented some sophisticated techniques to speed performance, which the optimizer must accurately model, as we discuss next.

In DB2/*, the table scan may be scanned both in a forward as well as a backward direction, (alternating from one direction to the other whenever the begining or end of the table is reached) This technique is known as a boustrophedonic scan [3]. This allows the next scan to utilize pages already in the buffer, rather than possibly force out those pages due to the least recently used (LRU) protocol of the buffer pool manager. DB2/6000 can also perform aggregation, or insertion into the sort heap, while the page is pinned in the buffer pool.

In DB2 for MVS, tables may be segmented in order to permit incremental backup, restore, and reorganization. Tables that are segmented have space map pages, so that only the data pages that actually contain data for that table are read. These space map pages can provide great performance improvements on a DELETE statement that does not have a WHERE clause. They can also indicate when the table is too small to benefit from sequential prefetch (see below).

DB2 for MVS has two types of page prefetching to minimize I/O time for sequential (or almost sequential) accesses: *sequential prefetch* and *skip sequential prefetch*. Sequential prefetch will read data asynchronously in contiguous blocks of 128K bytes. Sequential prefetch is enabled by the optimizer when it is scanning a table, a temporary table, or a clustered index. All table scans will use sequential prefetch unless the space map (for a segmented table) indicates that the table contains very few data pages or else the OPTIMIZE FOR N ROWS clause was used with a small value for N. Skip sequential prefetch uses special chained I/O features of MVS to read asynchronously a list of pages that are in order, but not necessarily contiguous. Such a list of pages occurs, for example, when fetching pages from a RID list that has been sorted. Normal, sequential and skip sequential I/O each have separate cost formulas.

The data manager of DB2 for MVS can also detect *de facto* sequential access during execution and thus enable *detection prefetch*. The optimizer attempts to model these situations where possible. For example, when the outer input to a nested-loops join is sufficiently ordered, prefetch is likely to be detected at execution time for the scan of the index and possibly the data pages of the inner table. Finally, DB2 for MVS allows the user to pick a 32K or 4K page size when a table is created. The page size affects the I/O characteristics, and this is modelled in the optimizer.

DB2 for MVS has the ability to compress or encrypt table data at the row level. The cost for decompression or decrypting is modeled by the optimizer.

DB2 for MVS may also optimize a query to use I/O parallelism. This feature is new to DB2 Version 3. Since it is a very involved and new feature, it is outside the scope of this article.

DB2/* with Starburst will also detect sequential access during execution, and prefetch data rather than relying on the operating system to detect sequential access, as is done by DB2/6000 Version 1. It may also choose to prefetch index leaf pages depending on their placement on the disk. This mechanism will be closely

[3] The origin of this term can be found in farming. A farmer plowing a field starts with a single furrrow and, upon reaching the far side of the field turns arround and starts a new furrow parallel to the original one.

coupled with an extent-based storage system to exploit multi-block I/O. This same mechanism can be exploited to provide skip sequential prefetch support.

7.3 Other Optimization Issues

The DB2 optimizers must deal with locking and the transactional semantics requested by the user (the *isolation level*). If the user has requested "cursor stability" isolation level, then locks acquired on indexes may be freed before the corresponding data pages are accessed. Since another user may change the underlying data pages in the interim, the optimizer must reapply any predicates already applied in the index.

As part of its plan, the optimizer selects a lock type and locking granularity for each table it accesses. This is determined using the type of query, the isolation level, and the access method for that table. Selecting the locking level presents an interesting trade-off: while larger locking granularities reduce the significant cost to acquire and free locks, they also reduce concurrent access to tables.

User-defined functions in DB2/* with Starburst present a number of new challenges. The definer of a new function has to convey the various characteristics of the function to the optimizer, including a cost function. To avoid requiring definers to write a cost function, DB2/* will use a generic function that is itself a function of a number of user-specified parameters which are optionally stored in the catalog with each function. In addition, user-defined functions may be non-deterministic, i.e., they may intentionally produce a different result each time they are invoked. An example of such a function is a random number generator. Such functions require that the optimizer store its result in a temporary table if it is to be reaccessed consistently, e.g., as the inner table of a nested-loops join.

7.4 Run-Time Query Optimization

DB2 for MVS has several run-time optimizations involving deferred index access. Processing of RID lists obtained from indexes can be abandoned in favor of a table scan if the RID list gets to be too large (as a percentage of the table size). If the RID list size gets so small that further RID processing will not be beneficial, the remaining RID access steps in the plan are skipped, and the data pages are accessed right away.

All the DB2 optimizers record dependencies upon any objects that are necessary to execute a query. These dependencies include indexes that are not used to access the data, but have provided information about uniqueness. If these indexes or tables do not exist at execution time, an automatic rebind of the query is performed and, if successful, the new plan is stored.

8 Conclusion

Unlike many research prototypes, industrial-strength commercial Query optimizers must support all the intricacies of the SQL language. They must accurately estimate the cost of using the large repertoire of access strategies provided by the data manager in order to select the access plans correctly. In this paper, we have described some of the important query optimization aspects that have been tackled by the DB2 family of relational database products.

However, despite more than two decades of work in this area, new discoveries continue to be made that allow us to extend our set of query transformations, let us model the CPU and I/O behavior of queries more accurately, and give us new techniques with which to evaluate relational operations. We will continue to make the best of these techniques available to our customers who rely on us to evaluate, ever more quickly, their increasingly complex queries on constantly increasing volumes of data.

Acknowledgements

The DB2 products are the result of years of work by a dedicated and innovative team of developers. The authors gratefully acknowledge the contributions of the optimizer group of DB2 for MVS at IBM Santa Teresa Laboratory, and especially Yun Wang, its chief architect. Tim Malkemus was the chief designer and team leader of the optimizer group for the development of the OS/2 Extended Edition Database Manager at the IBM Development Laboratory in Austin, Texas. Teams at Almaden Research Center, T.J. Watson Research Center, and Santa Teresa Laboratory are assisting the development of future versions of DB2/*. The authors wish to thank Chaitan Baru, Paul Bird, Hershel Harris, Richard Hedges, Glenn Laughlin, Bruce Lindsay, Tim Malkemus, John McPherson, Roger Miller, Hamid Pirahesh, Sheila Richardson, Pat Selinger, and Surendra Verma, who constructively reviewed an earlier version of this paper.

References

[CHH+91] J. Cheng, D. Haderle, R. Hedges, B. Iyer, T. Messinger, C. Mohan, and Y. Wang. An efficient hybrid join algorithm: A db2 prototype. In *Proceedings of the Seventh IEEE International Conference on Data Engineering, Kobe, Japan*, April 1991. Also available as IBM Research Report RJ7884, San Jose, CA, October 1990.

[DB293a] DB2. *Capacity Planning for DB2 Applications (GG24-3512)*. IBM Corp., 1993.

[DB293b] DB2. *DB2 V2.3 Nondistributed Performance Topics (GG24-3823)*. IBM Corp., 1993.

[DB293c] DB2. *DB2 V3 Administration Guide (SC26-4888), Chapter 7: Performance Monitoring and Tuning*. IBM Corp., 1993.

[DB293d] DB2. *Design Guidelines for High Performance (GG24-3383)*. IBM Corp., 1993.

[DB293e] DB2/2. *DB2/2 1.1.0 Guide (S62G-3663)*. IBM Corp., 1993.

[DB293f] DB2/2. *DB2/6000 1.1.0 Administration Guide (S609-1571)*. IBM Corp., 1993.

[DN82] D. Daniels and P. Ng. Query Compilation in R*. *IEEE Database Engineering*, 5(3):15–18, September 1982.

[HCL+90] L.M. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst Mid-Flight: As the Dust Clears. *IEEE Transactions on Knowledge and Data Engineering*, pages 143–160, March 1990. Also available as IBM Research Report RJ7278, San Jose, CA, Jan. 1990.

[HP88] Waqar Hasan and Hamid Pirahesh. Query Rewrite Optimization in Starburst. Research Report RJ6367, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, August 1988.

[LDH+84] G.M. Lohman, D. Daniels, L. Haas, R. Kistler, and P. Selinger. Optmization of Nested Queries in a Distributed Relational Database. In *Proceedings of the Tenth International Conference on Very Large Databases (VLDB), Singapore*, pages 218–229, August 1984. Also available as IBM Research Report RJ4260, San Jose, CA, April 1984.

[LFL88] M. Lee, J.C. Freytag, and G.M. Lohman. Implementing an Interpreter for Functional Rules in a Query Optimizer. In *Proceedings of the Fourteenth International Conference on Very Large Databases (VLDB), Los Angeles, CA*, pages 218–229, August 1988. Also available as IBM Research Report RJ6125, San Jose, CA, March 1988.

[LLPS91] G.M. Lohman, B. Lindsay, H. Pirahesh, and K.B. Schiefer. Extensions to Starburst: Objects, Types, Functions, and Rules. *Communications of the ACM*, 34(10):94–109, Oct. 1991. Also available as IBM Research Report RJ8190, San Jose, CA, June 1991.

[Loh88a] G.M. Lohman. Grammar-Like Functional Rules for Representing Query Optimization Alternatives. In *Proceedings of ACM SIGMOD 1988 International Conference on Management of Data, Chicago, IL*, pages 18–27. ACM SIGMOD, May 1988. Also available as IBM Research Report RJ5992, San Jose, CA, December 1987.

[Loh88b] G.M. Lohman. Heuristic Method for Joining Relational Database Tables. *IBM Technical Disclosure Bulletin*, 30(9):8–10, Feb. 1988.

[Mal90] Tim Malkemus. The database manager optimizer. In Dick Conklin, editor, *OS/2 Notebook: The Best of the IBM Personal Systems Developer*. Microsoft Press, 1990. Available from IBM as G362-0003, ISBN 1-55615-316-3.

[MHWC90] C. Mohan, D. Haderle, Y. Wang, and J. Cheng. Single Table Access Using Multiple Indexes: Optimization, Execution, and Concurrency Control Techniques. In *Proceedings of the International Conference on Extending Data Base Technology, Venice, Italy*, pages 29–43, March 1990. An expanded version of this paper is available as IBM Research Report RJ7341, San Jose, CA, March 1990.

[Moh93] C. Mohan. IBM's Relational DBMS Products: Features and Technologies. In *Proceedings of ACM SIGMOD 1993 International Conference on Management of Data, Washington, DC*, pages 445–448, May 1993.

[OL90] K. Ono and G.M. Lohman. Measuring the Complexity of Join Enumeration in Query Optimization. In *Proceedings of the Sixteenth International Conference on Very Large Databases (VLDB), Brisbane, Australia*, August 1990.

[PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 39–48, San Diego, June 1992.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and T.G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of ACM SIGMOD 1979 International Conference on Management of Data*, pages 23–34, May 1979.

[SQL93a] SQL/DS. *SQL/DS 3.4 General Information (GH09-8074)*. IBM Corp., 1993.

[SQL93b] SQL/DS. *SQL/DS Performance Tuning Handbook (SH09-8111)*. IBM Corp., 1993.

[SSar] K.B. Schiefer and Arun Swami. On the Estimation of Join Result Sizes. In *Extending Data Base Technology*, March 1994 (to appear). An expanded version of this paper is available as IBM Research Report RJ9569, San Jose, CA, November 1993.

[TMG93] Bruce Tate, Tim Malkemus, and Terry Gray. *Comprehensive Database Performance for OS/2 2.0 ES*. Von Norstrand Reinhold, 1993. Available from IBM as G362-0012, ISBN 0-442-01325-6.

[TO91] Annie Tsang and Manfred Olschanowsky. A Study of Database 2 Customer Queries. Technical Report TR 03.413, Santa Teresa Laboratory, Bailey Road, San Jose, CA, April 1991.

[Wan92] Yun Wang. Experience from a Real Life Query Optimizer (foils only). In *Proceedings of ACM SIGMOD 1992 International Conference on Management of Data, San Diego, CA*, page 286, May 1992.