Bulletin of the Technical Committee on

Data Engineering

December, 1993 Vol. 16 No. 4

IEEE Computer Society

Letters

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems.

Letter from the TC Chair

Dear Colleagues:

It is most gratifying to see that the resurrected Data Engineering Bulletin has now been published for one full year. Still more gratifying is the interest of database practitioners and researchers in writing for the Bulletin, the willingness of our new Editors to put together very informative issues in a short time, and the enthusiastic response of the members of the database community to the Bulletin. It reconfirms in our mind the value that the Bulletin brings to us. We would have liked to see the Bulletin on a firmer financial footings by now, but I am hopeful that a combination of free electronic distribution and paper distribution at cost to those who want this medium will ensure long term survival and growth of the Bulletin.

Besides the Data Engineering Bulletin, we focussed this year on fostering closer co-operation between TCDE and other database professional bodies that organize conferences. We are happy to report that TCDE continued sponsorship of the Data Engineering Conference, we are sponsoring the Third International Conference on Parallel and Distributed Information Systems. In addition, we provided co-operation status for the VLDB-93 conference this year, and we plan to continue this relationship with VLDB conferences. The FODO'93 conference on Foundations of Data Organization and Algorithms was provided with co-operation status in 1993 and this was extended to the Database and Expert Systems Applications Conference in 1994. Finally, we are working on establishing closer ties with the EDBT conference and foundation.

If you have any suggestions for new activities for TCDE, please write a note to any member of the Executive Committee. One final request — if you find that some of your colleagues are not receiving this Bulletin, please instruct them to send e-mail to `tcdata@crl.dec.com` with the word `enroll` in the subject line, and an electronic application will be sent to them."

Rakesh Agrawal
TCDE Chair

Letter from the Editor-in-Chief

The current issue is on query processing in commercial database systems. This is the first issue of the Bulletin published during my tenure as editor-in-chief to focus on the current state of practice in the commercial database world. As I have indicated before, it is my hope that one role for the Bulletin is to keep the database technical community informed about the state of commercial practice. In this regard, the March 1994 Bulletin, to be edited by Mei Hsu jointly with Ron Obermark, will be on transaction monitors and distributed transaction systems. It too will contain papers describing the attributes of current commercial offerings.

Issues such as this one require special editorial determination on the part of the issue editor as product development engineers work under very tight schedules as a way of life. These engineers also do not, as a matter of course, write technical papers for a wide audience. Goetz Graefe, our issue editor, has succeeded in overcoming these difficulties. He has collected four papers from prominent database vendors. These papers introduce us to the inside world of ”real” query processing. This is a world in which all queries must be dealt with, in which a new release must not break previous queries, in which no query can be permitted to be a serious embarassment. I want to thank both Goetz and our authors for bringing this inside view to the wider community.

Technical Committee members should all have received the announcement that the Bulletin is now available electronically via our bulletin mail server. To find out what issues are available electronically and how to acquire them, simply send mail to `tcdata@crl.dec.com` with the word `index` in the subject line. Win Treese, our networks and mailer wizard at CRL performed the magic that brings this service to you and I want to express my thanks to him for his substantial contribution to the success of the Bulletin.

I wish that I could announce that we have resolved the difficulties that stand in the way of our providing paid hardcopy subscriptions for the Bulletin. I cannot. Hence, I cannot now tell you under what circumstances you will be able to receive the March 1994 issue of the Bulletin in hardcopy. (Though, rest assured that it will be available electronically via our bulletin mail server.) This seemingly simple problem has been difficult to deal with because the Computer Society does not have policies in place that cover this situation. Hence, **policy decisions** have to be made, and there continues to be disagreement as to the terms of those decisions.

On a happier note, it is my pleasure to announce the appointment of two new associate editors for the Bulletin, Jennifer Widom and Shahram Ghandeharizadeh. Jennifer recently moved from the IBM Almaden Research Center to Stanford University. Hence her experience bridges the worlds of industry and academia. Her strong publishing record in the area of active databases and constraints has won her a prominent place among researchers in the area. Shahram is a recent University of Wisconsin Ph.D who is now at the University of Southern California. He has a strong publication record in the areas of parallelism, multimedia, and physical database design issues, which is impressive given how recently he graduated. Success in attracting such outstanding associate editors for the Bulletin provides continuing evidence of the value and the stature of the Data Engineering Bulletin as a resource for our community.

David Lomet
Editor-in-Chief

Letter from the Special Issue Editor

It has been my pleasure to edit this special issue of the IEEE Database Engineering Bulletin. Among many interesting topics, I selected query processing in commercial systems. Two observations led to this choice.

First, the competition in the field of decision support seems to be increasing at a dramatic pace. All relational database companies are committing significant resources to improving query optimization and processing in their products. Thus, I felt that the research and industrial communities should get a glimpse at the interesting developments currently under way, without forcing authors to go through the highly selective review process of journals and conferences such as ACM SIGMOD, VLDB, and Data Engineering.

Second, in some aspects of query processing, the industrial reality has bypassed academic research. By asking leaders in the industrial field to summarize their work, I hope that this issue is a snapshot of the current state of the art. Undoubtedly, some researchers will find inspirations for new, relevant work of their own in these articles.

The following articles summarize current technology and some future plans in IBM's DB2, a family of database systems running on mainframes, workstations, and personal computers; IBM's AS/400 system, which in the past has not been exposed much in the literature but includes a number of very interesting design decisions; Tandem's NonStop SQL, which exploits distributed-memory parallelism; and Digital's RDB/VMS, which recently added unique technology that permits dynamic optimization decisions at run-time. For me, each of these articles has been fascinating reading, because I learned new and interesting things in each one of them.

Unfortunately, I have not been able to obtain contributions from Informix, Ingres, Oracle, and Sybase, partially because all possible authors' time had already been committed to system development during the time this issue was compiled. In order to keep a level playing field and to allocate space that permits at least somewhat detailed descriptions, we omitted query processing technology in object-oriented database systems. We believe that ad-hoc query processing is required for object-oriented database systems to gain further market share, including value-based matching that does not rely on precomputed access structures; thus, query optimization and processing technology developed in the relational context will gain importance in the object-oriented field. We hope to invite the database companies not represented in the present issue to contribute to a second special issue on industrial, state-of-the-art database query processing.

Goetz Graefe
Portland State University
Portland, OR

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 from the

[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.

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 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

[2]This acronym originated from Search ARGument.

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 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.

[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.

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.

Query Processing in the IBM Application System/400

Richard L. Cole
University of Colorado at Boulder and IBM Rochester, Minnesota
rickcole@cs.colorado.edu

Mark J. Anderson
IBM Rochester, Minnesota
mja@vnet.ibm.com

Robert J. Bestgen
IBM Rochester, Minnesota
rbestgen@vnet.ibm.com

Abstract

The IBM Application System/400 uses many advanced query processing techniques within its layered, integrated system architecture. Techniques such as selectivity estimation using indices, dynamic binding of query plans to data, re-optimization of plans at run-time, predicate index creation, subquery transformation to join, optimizer controls, adaptive optimization of I/O buffers, and so on, were motivated by the need to provide a powerful, high performance, and integrated database that was easily used and administered by small and large enterprises alike[1].

1 Introduction

Effective and efficient query processing has been an important component of the success of the IBM Application System/400 (AS/400), with over 200,000 machines installed. Every one of these machine integrates support for powerful database mangement and advanced query processing. Yet, the databases on these systems must be easy to administer and use as they are employed by both small and large enterprises[2]. Satisfying these diverse requirements has resulted in the development of many innovative and advanced query processing techniques, some of which have been addressed in research forums and some that have not[3]. In this paper, we describe these techniques and relate them (when possible) to existing research in database query processing.

Query processing may be classified into two phases: query optimization and query evaluation [Gra93], independently of a system's overall architecture. However, in the AS/400, database support and query processing are part of a tightly integrated system architecture. Therefore, we first need to consider the system's architectural elements that support and enhance query processing before describing specific query processing techniques.

Working from the hardware out, the system architecture of the AS/400 has the following layers [ScT89]: the true system hardware; Horizontal Licensed Internal Code (HLIC) and Vertical Licensed Internal Code (VLIC) which together comprise the Licensed Internal Code; Operating System/400 (OS/400); and finally compilers and

[1]The following are trademarks or registered trademarks of the International Business Machines Corporation in the United States and/or other countries: Application System/400, AS/400, Operating System/400, OS/400.

[2]Consider that the AS/400 has the following configurations, from low-end (model B10) through high-end (model F95): 1–54 "MIPS" relative to a model B10; 1–7 I/O buses; 1–4 processors (shared memory architecture); 0.6–188.8 GB of disk storage; 16–1,280 MB maximum internal memory; and supporting up to 30–1,659 concurrent users. Performance numbers for these systems include: 6,750–373,275 RAMP-C transactions per hour, and 67.7–885.3 TpMC (transactions per minute for TPC-C) for $1,772–$3,086 dollars per transaction per minute.

[3]Few of these techniques have been described in any publication; previous AS/400 related publications [AnC88, ACD89, ClC89, CRR93, Faw89, Moh93, SMT93] have not focused on query processing techniques in detail.

utilities such as SQL/400 (SQL language and run-time support), Query/400 (a decision support tool), AS/400 Office (document storage, retrieval, etc.), and so on. Support for query processing is present in the HLIC, VLIC, OS/400, and (for example) SQL/400. Performance-critical functions such as database record locking, storage management, join operations, predicate evaluation, index management, and so on are located within the Licensed Internal Code; in fact, more than half of the query processing support is located within the Licensed Internal Code.

This "hardware" support of query processing and other system functions defines a high level machine. The interface to this high level machine is referred to as the machine interface or simply the MI. The MI is opaque in the sense that the software layers above it may not directly reference the internals of MI objects. The primary database functions available at the MI include: creation, activation, and manipulation of database cursor objects (through which query evaluation functions are defined and executed, e.g., all predicate evaluation is performed at this level); and creation of database indices (which are also entirely maintained below the MI).

The query processing portion of the OS/400 layer serves to translate abstract query requests, formulated in a "query definition template," into the appropriate MI functions. Several host language compilers and precompilers are supported, all of which use the query interface supplied by OS/400. Query optimization is also located within the OS/400 layer. This partitioning of function between two software layers is a natural way of building a database system (cf. [CAB81]); the important difference here is that the AS/400's Licensed Internal Code layers comprise a very large percentage of database and query processing.

The advantages of this architecture are that performance critical functions may be placed below the MI, higher layers may build upon the support provided by the MI, providing function abstraction, and overall support for query processing (and other functions) is provided in a highly integrated fashion. For example, the Licensed Internal Code provides hardware-optimized and integrated support for both navigational and set-oriented query processing primatives. The integration of these two data models (hierarchical and relational) continues through the OS/400 layer such that users may simultaneously and correctly use both data models on shared data [AnC88]. A further example of integrated support is that of one-step commands for compilation and binding of SQL programs. SQL language compilation, host language compilation, query optimization, and query execution are fully integrated components of the AS/400's system architecture.

The rest of this paper is divided among two main sections: in Section 2, we discuss a number of innovative query optimization techniques, and in Section 3 we describe advanced techniques for query evaluation. All techniques discussed are present in currently available systems. Finally, we summarize our presentation and outline future plans.

2 Query Optimization

Query optimization in the AS/400 is primarily located within the OS/400 portion of the system's architecture. It is modeled on the System R query optimizer [SAC79] and like System R is based on a "cost" consisting of a normalized combination of estimated I/O and CPU components. The optimizer performs an exhaustive search through the set of potentially optimal plans for single relations (subject to the optimizer policing criteria described in section 2.6), and uses a greedy search algorithm for the optimization of join plans. Predicates are evaluated as soon as their references are in scope. The greedy join search algorithm first finds the best two relations to join, then the best third relation to join to this composite relation, and so on. Therefore, bushy trees [GrD87] are not considered, only left-deep trees; however, Cartesian products of relations are considered. This search algorithm does not guarantee optimality, since the search space has been reduced a priori and the remaining search space is not searched exhaustively. However, in our experience, the algorithm and search space definition produces good join plans for joins of up to seven or so relations. In addition, the join optimization is very fast.

The set of innovative and advanced query optimization techniques employed in the AS/400 can be classified as follows: statistics and selectivity estimation, predicate handling, index creation and usage, subquery optimizations, adaptive access plans, and optimizer controls. These classes are described in further detail within the following subsections.

2.1 Statistics and Selectivity Estimation

The AS/400 uses several innovative techniques for statistics gathering and selectivity estimation. In particular, the Machine Interface (MI) provides an Estimate Data Space Index Key Range (ESTDSIKR) instruction [ACD89] that OS/400 uses to compute a real-time estimate of the number of keys within a specified range of key values. Such an index search is useful for predicates realizable as a range, and can be performed whenever there exists an index having one or more keys matching a selection predicate consisting of a comparison of the key attribute to a constant (or a run-time value supplied for a host variable). The ESTDSIKR instruction provides an accurate estimate of the number of key values within the specified range of keys by probing the index structure and counting page pointers. The instruction supports an "early exit percentage," that allows the OS/400 optimizer to state the maximum percentage of key range values within the index beyond which processing should terminate. The optimizer uses this fast-path capability to limit ESTDSIKR processing for selection predicates containing host variables during the start of query run-time when verifying access plans (see Section 2.5 for more details on the re-optimization of access plans).

The ultimate value of the ESTDSIKR capability lies in the timeliness of its statistics. Typically, statistics gathering is a background or off-line process, e.g., by using a "RUNSTATS" command, and therefore statistics become out-of-date, resulting in the production of suboptimal plans. The real-time estimates of ESTDSIKR take into account the often dynamic nature of database systems.

Another dynamic aspect of statistics gathering in the AS/400 is the maintenance of statistics, such as "total number of records," "number of distinct attribute values," and so on. This meta-data is kept within the database objects themselves and maintained in real-time by the Vertical Licensed Internal Code (VLIC) whenever changes are made to these objects that affect their statistics. This information can then be materialized by the OS/400 optimizer, which may assume that the information is always up-to-date. Because the statistics of each database object, and in fact all data dictionary information[4], are distributed among the objects themselves, contention is reduced.

2.2 Predicate Optimizations

Predicates may be applied to data in base relations as well as to data in applicable indices having keys that match a predicate's attributes. Such predicates may be applied (sequentially) to every key in an index, or (when possible) may be used as a key range that limits an index's candidate keys to just those that collate between the start and stop key values. Not only are limiting key ranges supported for value ranges defined by ANDed predicates, but ORed range expressions are supported as well, e.g., $A < 10$ OR $A > 1000$.

Key range predicates may be more complex than simple (non-derived) key fields. Indices may exist having derived key expressions such as XLATE(A,XTBL), where XTBL defines a translate table used for an alternative collating sequence. The optimizer exploits the existence of these indices whenever possible and optimal.

The OS/400 optimizer performs two additional optimizations with regards to selection predicate processing. First, it attempts to rectify illogical predicate combinations, such as $A > 10$ AND $A < 9$ or $A > 10$ AND $A > 100$, by replacing them with their minimum covering equivalents, i.e., FALSE and $A > 100$ respectively. (Also, predicates such as $A > 10$ AND $A < 20$ are replaced by an equivalent BETWEEN or range predicate.) This optimization not only improves a query's run-time performance, but also increases the accuracy of selectivity estimation and the optimality of the resulting plans. Second, the optimizer computes the transitive closure of simple selection predicates with join predicates, producing additional, implied predicates that may significantly improve query performance. For example, the restriction TABLE2.B = 10 on table 2 is implied by TABLE1.A

[4]There is also a centralized SQL-type catalog for SQL application compatibility.

= 10 AND TABLE1.A = TABLE2.B. The optimizer keeps track of these implied predicates and prevents them from adversely affecting selectivity estimation by assigning them a selectivity of 1.

2.3 Index Creation and Usage

The OS/400 query optimizer (and AS/400 query evaluation) makes extensive use of index structures (also see Section 3.1). Not only does the optimizer consider using indices that currently exists, it may also consider the cost of creating and using a temporary index, one created expressly for the query being optimized. Data access via a temporary index is often useful when algorithms such as index-nested-loops join are optimal [DNB93]. Data access using an index maintained in real-time also provides a mechanism for query re-use. For example, queries containing SQL host variables may be closed and reopened with different host variable values without requiring resorting of data for join algorithms or ordering specifications. Access using an index also facilitates opening queries for update, even when they have ordering criteria, e.g., an SQL ORDER BY.

The utility of a temporary or persistent index (maintained in real-time) is greatly increased by support for creating a "predicate index," an index that addresses only those records that satisfy a certain predicate or predicates, also called a "partial index" [Sto89]. A predicate index is very useful during query processing because if an index's predicates cover a query's predicates, the index may be used to apply these covered predicates. In effect, the work to define the restricted set of records has already been accomplished. In order to ensure correctness the optimizer performs a covering analysis of the index's and query's predicates. This implementation alternative is considered in a cost-based fashion along with other alternatives available to the optimizer.

The performance of creating predicate indices is improved by optimization (and run-time) support for predicate evaluation using currently existing indices. For example, when building a predicate index for the selection expression A $>$ 10 AND B $<$ 100, an existing index with keys on attribute "A" may be used to evaluate the first predicate using a limiting key range just as for regular query evaluation.

Another important point at which indices are used in the AS/400 is when a National Language Sort Sequence (NLSS) has been specified. This is a system-wide specification of an alternative collating sequence for a particular language. Indices play an important role because ordering criteria and data comparisons can be built into an index's collating sequence, negating the need for on-the-fly translations. The AS/400 optimizer performs a cost based analysis of available indices that may satisfy an NLSS, is capable of using translated keys for equal/not-equal predicates for which no NLSS was specified, and may even consider using non-translated keys for equal/not-equal predicates for which an NLSS was specified.

2.4 Subquery Optimizations

As efficient subquery processing is of key importance in production databases, significant effort has been expended in this area. This has also been a focus of research efforts [GaW87, Kim82]. The following optimization techniques are employed:

- Simple, non-correlated subqueries are often evaluated in two steps: produce the subquery result, and substitute the result for the subquery predicate. This avoids repetitive execution of subqueries that return the same result.

- A similar optimization involves the use of MIN or MAX aggregate functions for SQL ANY and ALL subqueries compared using greater-than, less-than, and so on. Computing the minimum or maximum value of the subquery reduces the answer set to a single record that is substituted for the subquery in the referencing selection expression.

- Correlated subqueries are transformed into join queries whenever possible. Care must be taken with this transformation due to the well documented [GaW87] differences in semantics between subqueries and join queries, i.e., for an SQL IN predicate, duplicates are of no concern, but duplicates do contribute to a relational join result. A technique employed in the AS/400 for avoiding the problem of duplicates is to recognize when the join is over primary key values, for which we can assume that all keys are unique. If this is not the case, the AS/400 optimizer may still often use join transformations because the VLIC supports an "unique-fan-out" join option. This special-purpose join option prevents the production of incorrect, duplicate join values by skipping over them during join processing. (The unique-fan-out-join is essentially a duplicate preserving semi-join.)

Similarly, correlated subqueries referenced by the SQL NOT IN predicate may be transformed to join queries with the "exception join" option. This option produces those join records that do not match the join criteria, i.e, the anti-semi-join, and duplicates are avoided by also specifying a unique-fan-out.

- Whenever correlated subqueries cannot be transformed into more efficient join queries, the run-time engine maintains a table of, correlation value – subquery result, pairs that can be used to return subquery results without actually (re-)executing the subquery when correlation values are repeatedly used.

2.5 Adaptive Access Plans

Access plans built by the AS/400 optimizer contain specifications of MI objects to be created and their linkages. Thus, optimization at compile-time is amortized over many query executions by re-using access plans at run-time. However, there are situations in which we should re-optimize the access plan at run-time, beyond the standard re-optimization scenarios, e.g., a dropped dependent index. For example, because of changes in the cardinality of the participating relations, the current plan may become suboptimal. At start-up-time the optimizer notes significant cardinality changes and uses this and other criteria to determine whether it is beneficial to re-optimize an access plan. The re-optimization criteria are as follows:

- Significant data changes, not simply an increase or decrease in the cardinality of relations [DMP93], but also significant data updates (those not modeled as a delete followed by an insert). Such statistics are maintained by the VLIC, materialized at start-up-time, and compared to their compile-time counterparts. When they have changed by more than a given factor, the query is re-optimized and the existing access plan is replaced.

- Another important criterion for re-optimization is the creation of "interesting indices" since compile-time optimization. When at start-up-time there is a new, potentially useful index, the plan is re-optimized in order to consider the advantages of using it.

- Access plan re-optimization is also considered for queries containing host variables, i.e., embedded SQL queries that refer to program variables. Because the optimizer cannot know the selectivity of predicates containing host variables at compile-time, the access plan is initially built using default selectivities. However, such a plan is always reoptimized upon first use by the program, i.e., the first time host variable values are supplied it is likely that the initial plan is suboptimal. From then on, at each subsequent use of the access plan, the ESTDSIKR instruction is used (when possible) to reestimate the selectivity of these predicates. If the predicates' selectivity is similar to that previously estimated for the access plan, then the current plan is used, otherwise the query is re-optimized and the access plan is replaced.

- The final criterion for plan re-optimization involves the use of run-time statistics, i.e., feedback from the execution engine to the optimizer. The primary statistic is a simple decaying average of the number of actual output records per use. This statistic is returned by the execution engine and stored into a statistics profile[5]. The statistic is used to verify selectivity assumptions, and is factored into a start-up-time decision procedure that determines whether re-optimization is desirable.

2.6 Optimizer Controls

The AS/400 optimizer provides several "control knobs" that allow users and system applications to influence the optimizer's behavior. In order to provide flexibility in cost goals, the optimizer provides an optimization goal specification. The optimization goal may take on one of several values with the following implications:

- The most restrictive value is optimize goal "complete". The point of this option value is to completely eliminate unnecessary I/O during a user's next request. Therefore, all predicates will have been applied before any data is returned. The advantage of this is that the query may be re-used, i.e., repositioned to "start" and re-read, amortizing the extra effort required at query open time. To carry out this directive, the optimizer may place all output records in a temporary result or force access via a predicate index. A drawback is that changes to database records will not appear in the result should the query be re-used.
- Optimize goal "minimize I/O" is a slightly less restrictive version of the previous optimization goal. The optimizer considers the use of predicate indices, but avoids the use of temporary results in order to provide "live" data.
- Optimize goal "all I/O" tells the optimizer to minimize the total cost of a query, including the output of all result records, a useful option for batch applications that wish to maximize system throughput.
- Optimize goal "first I/O" results in minimization of query execution cost for producing the first output buffer's worth of result records. This option is often used by interactive applications, e.g., the Query/400 decision support product uses this option. An extended version is also available using SQL/400 as an option to optimize for an arbitrary number of output records.

Another control knob is called "allow copy data". This knob allows the user or program to specify if static (temporary copies) of data are allowed based on one of the following three values. Allow copy data "no" informs the optimizer that temporary copies of data are not allowed during query processing. For example, sort operations would be disallowed, replaced by the use of indices. The advantage of this option value is that access to data is "live" in that updates made by this and other processes may be visible (depending on isolation levels). In addition, query cursor's may be re-used, and still reflect the current state of the database. Allow copy data "yes" informs the optimizer that copies of data may be made only if absolutely necessary to the query's implementation, and allow copy data "optimize" indicates that temporary results may be used as necessary to minimize cost.

The optimizer (optionally) attempts to control or police the amount of effort it expends on the optimization of a particular query evaluation plan. The idea here is to avoid "over-optimization" of ad-hoc queries, queries that are executed only once, in which case the combined effort of optimization and execution should be minimized. (Dynamic SQL queries are policed as well since they are also optimized at run-time.) To this end, the optimizer quantifies the effort expended during optimization, in the same units as cost, and compares this quantity to the cost of the current best query evaluation plan. Optimization is terminated when the optimization effort exceeds a given percentage of the plan's cost, assuming a best plan currently exists (cf. [Koo80]).

The last control knob we will discuss is that of "close SQL cursor". This option allows the user to specify whether to keep cursors open only as long as the program is active (end-of-program), open as long as any SQL program is active (end-SQL), or open as long as the job is active (end-of-job). Keeping SQL cursors open between program invocations amortizes the SQL open and query start-up overhead over many more uses.

[5]Should the statistics profile be in use, it is simply not updated; not having the latest statistic value does not appreciably degrade the optimization process.

3 Query Evaluation

As previously described, query evaluation is primarily performed by the Licensed Internal Code, which supports several alternative execution algorithms. These algorithms include: sequential table scans; index scans with arbitrary selection expressions, including key range specification and key (non-range based) predicates; join operations including several flavors of inner-, outer-, semi- and anti-joins; and join algorithms consisting of index-nested-loops join and joins based upon the use of Cartesian products.

The execution engine supports a rich set of data manipulation language operations. For example, a "previous" as well as a "next" iterator is supported, i.e., a scrollable cursor, and query cursors may be repositioned to a virtual start or end position. Blocked I/O, i.e., multiple record I/Os from and to disk as well as multi-record program application program logic is employed whenever possible, and is also an option for SQL applications.

In the following subsections we describe some of these query evaluation techniques in further detail, having classified them into the following categories: indices, sorting, join operations, aggregate and scaler functions, and I/O buffering.

3.1 Indices

Indices in the AS/400 are binary radix trees with front end key compression. However, they are balanced at the page level, as in a more typical balanced tree. Thus, disk I/O is minimized while search key comparisons are quickly made on the basis of single bit values.

As previously discussed in Sections 2.2 and 2.3, indices are a heavily used feature of the AS/400. To briefly recap, temporary and persistent indices may be created having elaborate selection expressions, predicate indices, and derived keys. Selection expressions may refer to base relations as well as to any other indices currently existing over that data (however, multiple index ANDing and ORing is a future enhancement). In addition, indices created expressly for a particular operation, i.e., temporary indices, take advantage of their lack of need for crash recovery by reducing the number of forces to disk and by using temporarily allocated storage segments.

3.2 Sorting

Non-index based sorting in the AS/400 uses a tournament sort, i.e., replacement selection, for run creation and the subsequent merging of sorted runs from disk. It also implements early duplicate removal [BBD83] for SQL DISTINCT requests. The sort interface provides for multiple, concurrent inputs, which is particularly effective for the processing of SQL UNION requests. Here, the entire set of relations may be unioned in a single step instead of in a series of binary merges. Another sort interface enhancement allows for re-use of the tournament tree and allocated sort space should multiple, sequential sort invocations be required.

3.3 Join Operations

The AS/400 Licensed Internal Code supports several types of join operations. Along with the standard inner join, the system also supports semi-join, left-outer-join, left-anti-join, and the unique-fan-out-join previously described in Section 2.4. The primary join algorithm is index-nested-loops join [DNB93], which requires the use of an index over the inner or right join input. If such an index does not currently exists, a temporary one must be created, an alternative considered by the query optimizer. The implementation of the index-nested-loops join algorithm supports multi-way joins, not simply a sequence of binary joins. Joins based directly upon producing a Cartesian product are also supported, because, in our experience they sometimes significantly outperform predicate based joins.

3.4 Aggregate and Scaler Functions

Many non-standard aggregate and scaler functions are supported, all of which are available using SQL/400. In addition to the standard minimum, maximum, sum, count, and average aggregate functions, the AS/400 supports standard deviation and variance, and the following scaler functions: absolute value, arc cosine, anti-log, arc sine, arc tangent, hyperbolic arc tangent, hyperbolic sine, hyperbolic cosine, hyperbolic tangent, sine, cosine, tangent, natural log, cotangent, exponential, common log, square root, logical AND, logical OR, logical NOT, logical XOR, maximum and minimum of a set of values, (e.g., MAX(A,B,C,0)), translate to upper case, strip leading or trailing blanks, return hex value, and remainder. Direct support for these additional functions simplify the user's and application programmer's job.

3.5 I/O Buffering

Database systems and operating systems often have different assumptions and goals regarding many aspects of their operation. Of particular importance to database systems is the operating system support for buffering the database contents in memory [Sto81]. In the AS/400, the database support of the Licensed Internal Code and OS/400 is part of the operating system. This means that during query evaluation, the Licensed Internal Code may use synchronous and asynchronous read-ahead, write-behind, and other advanced I/O techniques that do not conflict with operating system assumptions.

For example, the Licensed Internal Code supports dynamic, run-time optimization of memory buffer allocation during multiple record insertions [CRR93]. The size of allocated buffers while double buffering inserts is tuned to the state of the I/O request to the "second buffer." Insert (CPU) processing of the "first buffer" should ideally complete just as the I/O to the second buffer completes. This very specific run-time optimization works in concert with the system-wide optimization of buffer allocation for database as well as non-database objects. For each database object, the Licensed Internal Code also tracks physical I/Os, i.e., page faults, as well as logical I/O requests, and attempts tune the size of the logical requests so that page faults are minimized. The updated logical request sizes are used when the next I/O is requested. Thus, the system attempts to maximize the utility of all I/O requests for all paged objects.

Another example of database and operating system integration is support for explicit object paging requests using the MI Set Access State (SETACST) instruction, available to OS/400 users as the Set Object Access (SETOBJACC) command. For example, database objects such as tables and indices may be explicitly brought into memory before the execution of a database application, thus reducing disk I/O.

4 Summary and Conclusions

In this paper, we have described several advanced and innovative query processing features, all of which are implemented and available in current AS/400 systems. Many of these techniques were driven not only by performance concerns, but by ease of administration and ease of use requirements, e.g., real-time statistics maintenance, automatic access plan re-optimization, and integrated data models.

We are continuing to improve the functionality and performance of the AS/400's database and query processing. In the near future, we plan to enhance (i) index support, by adding the capability for using multiple keys with key range limits, and index ANDing and ORing techniques [MHW90]; (ii) the use of hash based techniques, in addition to indices and sorting, particularly for performing aggregate operations; (iii) support for distributed SQL processing using stored procedures and compound statements, and distributed data extensions for DRDA2 (distributed remote unit of work); (iv) join optimization using an alternative polynomial time join optimization algorithm [SwI93]; (v) and finally, exploitation [PMC90] of the AS/400's multi-processor architecture machines.

References

[AnC88] M. J. Anderson and R. L. Cole, “An Integrated Data Base,” in IBM Application System/400 Technology, V. J. Gervickas (editor), IBM, Document Number SA21-9540, 1988.

[ACD89] M. J. Anderson, R. L. Cole, W. S. Davidson, W. D. Lee, P. B. Passe, G. R. Ricard, and L. W. Youngren, Index Key Range Estimator, U.S. Patent 4,774,657, IBM, 1989.

[BBD83] D. Bitton, H. Boral, D. J. DeWitt, and W. K. Wilkinson, “Parallel Algorithms for the Execution of Relational Database Operations,” ACM Trans. on Database Sys. 8, 3 (September 1983), 324.

[CAB81] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolo, P. G. Selinger, M. Schkolnik, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost, “A History and Evaluation of System R,” Comm. of the ACM 24, 10 (October 1981), 632. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

[ClC89] B. E. Clark and M. J. Corrigan, “Application System/400 Performance Characteristics,” IBM Sys. J. 28, 3 (1989).

[CRR93] M. J. Corrigan, G. R. Ricard, R. M. Rocheleau, and L. W. Youngren, Optimized I/O Buffers, U.S. Patent 5,179,662, IBM, 1993.

[DNB93] D. DeWitt, J. Naughton, and J. Burger, “Nested Loops Revisited,” Proc. Parallel and Distr. Inf. Sys., San Diego, CA, January 1993.

[DMP93] M. A. Derr, S. Morishita, and G. Phipps, “Design and Implementation of the Glue-Nail Database System,” Proc. ACM SIGMOD Conf., Washington, DC, May 1993, 147.

[Faw89] J. E. Fawcette, “Understanding IBM’s Relational Database Technology,” Inside IBM’s Database Strategy, an IBM Sponsored Supplement to DBMS, Redwood City, CA, September 1989, 9.

[GaW87] R. A. Ganski and H. K. T. Wong, “Optimization of Nested SQL Queries Revisited,” Proc. ACM SIGMOD Conf., San Francisco, CA, May 1987, 23.

[GrD87] G. Graefe and D. J. DeWitt, “The EXODUS Optimizer Generator,” Proc. ACM SIGMOD Conf., San Francisco, CA, May 1987, 160.

[Gra93] G. Graefe, “Query Evaluation Techniques for Large Databases,” ACM Computing Surveys 25, 2 (June 1993), 73-170.

[Kim82] W. Kim, “On Optimizing an SQL-like Nested Query,” ACM Trans. on Database Sys. 7, 3 (September 1982), 443.

[Koo80] R. P. Kooi, “The Optimization of Queries in Relational Databases,” Ph.D. Thesis, Case Western Reserve Univ., September 1980.

[MHW90] C. Mohan, D. Haderle, Y. Wang, and J. Cheng, “Single Table Access Using Multiple Indexes: Optimization, Execution and Concurrency Control Techniques,” Lecture Notes in Comp. Sci. 416 (March 1990), 29, Springer Verlag.

[Moh93] C. Mohan, “IBM’s Relational Database Products: Features and Technologies,” Proc. ACM SIGMOD Conf., Washington, DC, May 1993, 445.

[PMC90] H. Pirahesh, C. Mohan, J. Cheng, T. S. Liu, and P. Selinger, “Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches,” Proc. Int’l. Symp. on Databases in Parallel and Distributed Systems, Dublin, Ireland, July 1990, 4.

[ScT89] D. L. Schleicher and R. L. Taylor, “System Overview of the Application System/400,” IBM Sys. J. 28, 3 (1989).

[SAC79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, “Access Path Selection in a Relational Database Management System,” Proc. ACM SIGMOD Conf., Boston, MA, May-June 1979, 23. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

[SMT93] S. Scholerman, L. Miller, J. Tenner, S. Tomanek, and M. Zolliker, “Relational Database Integration in the IBM AS/400,” ACM SIGMOD Record 22(4), December 1993, 5.

[Sto81] M. Stonebraker, “Operating System Support for Database Management,” Comm. of the ACM 24, 7 (July 1981), 412. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

[Sto89] M. Stonebraker, “The Case for Partial Indexes,” ACM SIGMOD Record 18, 4 (December 1989), 4.

[SwI93] A. N. Swami and B. R. Iyer, “A Polynomial Time Algorithm for Optimizing Join Queries,” Proc. IEEE Int’l. Conf. on Data Eng., Vienna, Austria, April 1993, 345.

Query Processing in NonStop SQL

Albert Chen *Yung-Feng Kao* *Mike Pong* *Diana Shak* *Sunil Sharma*
Jay Vaishnav *Hansjorg Zeller*
Tandem Computers, Inc.
Cupertino, CA

1 Introduction

NonStop SQL is a fault-tolerant, distributed implementation of the query language SQL for Tandem NonStop TM computer systems [Tand87, Tand89].

Tandem NonStop computer systems are loosely-coupled (i.e. non-shared-memory) multi-processor systems. A system comprises of clusters of processors up to a maximum of 224 processors per cluster. The system can be incrementally expanded as the computing requirements grow. The processors are interconnected by a duplexed fiber-optic bus. Most I/O devices, including disks, are duplexed and may be connected to two processors in order to provide redundant access paths to a device. Most of the critical system processes are supported as process pairs with one process acting as the primary process and the other process acting as a hot standby. Thus, the system is able to survive any single point failure without losing access to any software or hardware resource.

The Tandem NonStop Kernel TM operating system is a message-based operating system. Access to I/O devices, including disks, is obtained by messages to server processes that manage a specific I/O device. Any process in the system can access any I/O device in the system by sending a message to the appropriate server process.

Applications are typically developed using the client-server model. A transaction monitor (PATHWAY) is provided to manage the servers and to manage communication between clients and servers. The application servers may be written in a variety of languages (C, COBOL, Pascal, TAL) and use embedded SQL statements to access data.

NonStop SQL TM builds upon the message-based fault-tolerant architecture of a Tandem computer system. The system provides a global name space and a program can access any table in the entire system (subject to the access rights of the program). Transactions may be distributed and transactions may access any table in the entire system. A 2-Phase commit protocol is used to coordinate transaction commit. NonStop SQL incorporates node autonomy: a query on a horizontally partitioned table will tolerate some unavailable partitions, as long as it does not require data from those parti- tions. Node autonomy ensures that a user can always access local data regardless of whether or not remote partitions of a table are accessible. An extensive and complete versioning support permits programs, tables, catalogs and system software of different versions to coexist in a distributed system.

1.1 The Query Processing Components of NonStop SQL

The query processing system for NonStop SQL consists of the SQL compiler, the SQL executor, the SQL file system, disk processes and the SQL dictionary. The SQL compiler compiles static and dynamic SQL statements. It references the SQL dictionary to perform name binding and retrieves statistical information about the contents of tables in order to compile an execution plan (also called the access plan) for each statement. In contrast to other RDBMS implementations, the execution plans for static SQL statements are stored in the same file as the object code produced by the host language (3GL) compilers.

.

The executor is a set of system library procedures that are invoked by an application program to execute an SQL statement. For static SQL, the executor retrieves the execution plan from the object file, which contains not only the machine code produced by the host language (3GL) compiler but also the execution plans produced by the SQL compiler . The executor compiles an execution plan for DML supplied in dynamic SQL statements by invoking the SQL compiler at run time. It interprets each execution plan and uses the file system to access a specific table via a specified access path. The file system sends a message to the appropriate disk process to retrieve the data. Each disk process provides access to objects on a specific disk. Fig. 1 illustrates the NonStop SQL process architecture.

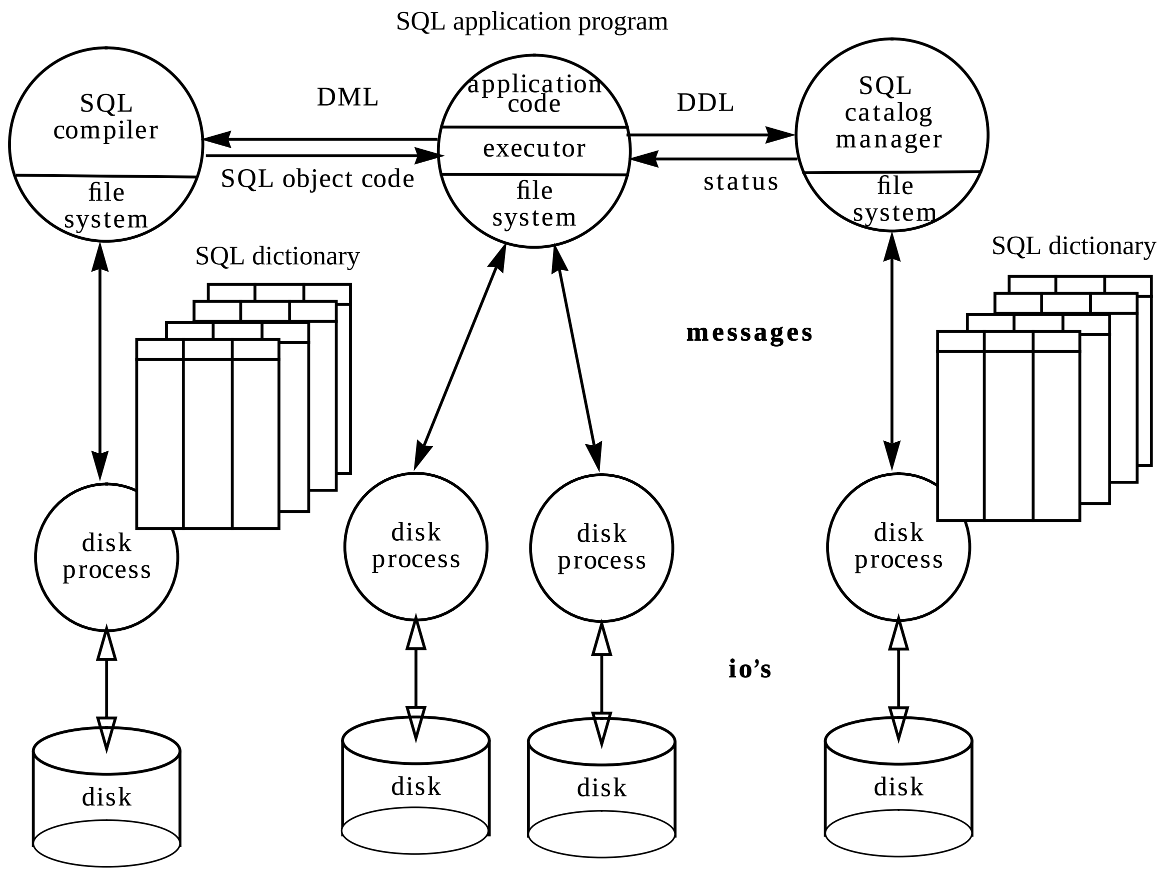


Fig. 1: The NonStop SQL process architecture

2 Query Optimization

NonStop SQL possesses a cost-based query optimizer [Seli79] that performs single-site optimization as well as some multi-site optimization. The optimizer generates an execution plan by using cost, expressed in terms of the number of equivalent I/Os performed on a certain hardware configuration, as a measure of system resources consumed by each operation [Pong88]. The resources consumed are cpu instructions, input/output operations (I/Os), and messages. The purpose of expressing the cost in terms of the number of equivalent I/Os is to combine disparate units of measure into a single one. The optimizer chooses the execution plan that has the least cost relative to all the other execution plans that it generates for a given query. The costing for an execution plan accounts for remote data accesses and for the cost of performing sorts.

2.1 Selectivity estimation

The optimizer estimates the selectivity of predicates using the column-level statistics that are stored in the data dictionary. The relevant statistics are the SECONDLOWVALUE, SECONDHIGHVALUE and the UNIQUEENTRYCOUNT. They are gathered and stored in the data dictionary when a user executes the UPDATE STATISTICS command. The difference between the second high value and the second low value defines the range of values occurring within a column. The optimizer assumes that the values are uniformly distributed within that range. The selectivity of an equality predicate is the inverse of the unique entry count. Range predicate selectivity is estimated using interpolation. A predicate that involves an inequality comparison and contains a parameter or a host variable is assigned a default selectivity of 1/3.

For join selectivity estimation, the optimizer considers transitive relationships between equi-join predicates. For example, if A = B and B = C then the optimizer infers that A = C. The optimizer maintains equivalence classes of columns belonging to equi-join predicates and uses them for estimating join selectivity. The unique entry count is synthesized for a join column after the table to which it belongs is added to a join composite. The cardinality of the resulting composite is used as an upper bound on the unique entry count. The selectivity of an equi-join predicate is the inverse of the synthesized unique entry count.

2.2 Plan Generation

The optimizer does an exhaustive enumeration of execution plans for evaluating a query. For each table, it generates a plan that utilizes the primary access path as well as one plan for each alternate access path. Each of the single-table execution plans provide the basis for generating join plans.

The optimizer builds a tree of access paths called the search tree for enumerating join plans. Logically, all possible paths are considered. However, the search space is reduced by using the following heuris- tics:

1. It considers only right-linear join trees. This means that a join of four tables (ABCD) can be realized by (((AB) C)D) or (((AC) B)D), but not by ((AB)(CD)).

2. It considers three join methods, namely, nested-loops, hash and sort-merge join.

3. A new table can be joined to an existing composite (joined) table either if there is an equi-join predicate that relates the composite and the new table, or none of the remaining tables have an equi-join predicate relating them with the composite.

4. Only one plan that performs a cartesian product is added to the search tree at each stage. The smallest table that does not already belong to the composite and does not have an equi-join predicate relating it with the composite is added to the search tree. This plan allows optimization of certain queries (star-queries) that perform best if a cartesian product between small tables is formed first before joining the result with the larger tables.

Each node in the search tree represents an access path for a set of tables joined together. It is characterized by :

1. the set of tables it contains

2. the access path for reading rows from it

3. the set of evaluated predicates
4. the sort order of the output
5. the cardinality of the output

The notion of an order is important. For example, a scan on a table that retrieves rows through an index will yield rows in a particular order. The order decides whether a subsequent sort (for an ORDER BY or a GROUP BY) can be eliminated, or whether a predicate can be used in an index key or whether a sort-merge join should be performed.

The optimizer may prune the search tree whenever it adds a new composite. Given two search trees that have the same characterization, it retains the one that has a lower cost.

The optimizer evaluates all non-correlated subqueries once as part of the initialization step. A correlated subquery is executed at the stage where values from the outer reference are first available.

2.2.1 Serial Execution Plans

Given two tables (or a composite and a single table), the optimizer generates every nested-loops join plan that can be generated according to the heuristics described above. Sort-merge join and hash join plans are generated only when the two tables have (at least) one equi-join predicate relating them. It generates two sort-merge join plans. The first one uses the composite that has the lowest cost and adds the cost of sorting so that the composite and the table will yield rows in the same order to the merge phase. The next one uses the composite, if one exists, that yields rows in the same order as the table. Additionally, it generates two hash join plans: a simple hash join and a hybrid hash join plan. The hash join methods are described in section 3.4.

2.2.2 Parallel Execution Plans

NonStop SQL can perform SQL SELECTS, INSERTS, UPDATES, DELETES, joins, grouping and aggregation in parallel. A comprehensive discussion on parallel execution plans appears in section 3.5. The goal of using a parallel execution scheme for queries is to partition the work load over multiple CPUs and achieve an improvement in response time at the cost of resource consumption. Parallel execution plans are generated by the optimizer whenever the user enables parallel execution plans.

To determine the cost of executing a query in parallel the optimizer computes the number of horizontal partitions of a (composite) table that will be operated on by a given operation. It walks through a parallel execution plan and computes its cost. It ascribes a fixed start-up cost for starting each executor server process (ESP) and the communication costs necessary for initializing the execution envi- ronment for each operation. Then, it amortizes the cost of executing the operation in serial over the number of partitions that will be operated on and adds it to the fixed start-up cost. This is the cost of performing the operation per partition. It accumulates the total cost for the plan per partition.

The optimizer first chooses the cheapest of all parallel execution plans. It supports tie-breaking rules when the costs of two plans differ by no more than 5%. It chooses a plan according to the following precedence relationship:

1. Choose a matching partitions plan in preference over any other parallel execution plan.
2. Choose any parallel execution plan in preference over the hash-repartitioned plan.
3. Choose the cheaper of two hash-repartitioned plans

The optimizer chooses a parallel execution plan if its cost is lower than the cost of the best sequential plan.

2.2.3 Multiple Index Access

For a single table query, if the predicates of the WHERE clause are represented in the disjunctive normal form and no subtree that is rooted in a disjunct contains another disjunct within it, then the optimizer attempts to use a multiple index access plan. According to this plan, one or more indices that belong to the same table are scanned in sequence and a logical union is performed on the rows retrieved by them. This plan competes with other plans based on its cost.

2.3 Selection of the Best Plan

After the optimizer has completed the enumeration of execution plans, each plan in the search tree has a cost associated with it. The optimizer intends to choose the execution plan that has the least cost (such a plan is considered to be the best plan). Often, the search tree contains two or more execution plans whose costs are very close. The optimizer therefore implements a hierarchy of tie-breaking rules for comparing two plans whose costs are within 10% of each other. The rule hierarchy is enumerated below:

1. Choose a local access path in favor of a remote one
2. Choose a plan that uses an index as an access path when each key column of the index has an equality predicate specified on it.
3. If there are two plans that are of the type described in item 2 above, then choose the one that provides a keyed access to the base table (the table is implemented in a key-sequenced file) instead of an alternate index access path.
4. Choose a plan that shows a lower index selectivity.
5. Choose a plan that has a lower cost.
6. Choose a plan that employs more predicates on key columns of an index.
7. Choose a plan that uses an index that is UNIQUE.
8. Choose the plan that accesses the base table directly, everything else being equal.

3 Query Processing Features

This section describes some of the features that allow NonStop SQL to process large amounts of data and complex queries very quickly.

3.1 Set-oriented Operations in the Disk Process

The disk process provides single-table set-oriented access to tables [Borr88]. A set is defined by a restriction expression and a list of attributes to project. A request (message) from the file system describes the data to be retrieved in terms of a set (i.e., the begin and end keys, the restriction expression and the attributes to be projected). In response, the disk process may reply with one or more buffers containing the data that corresponds to the set. Set-oriented updates are optimized by sending the update expression to the disk process (in addition to the begin and end keys, and the predicates which define the set). Sequential inserts are optimized by buffering inserts in a file system buffer. Set-oriented deletes are optimized by sending the begin and end keys and the predicates (which define the set to be deleted) to the disk process in a single message. If the data in certain columns of a table is already ordered according to the GROUP BY clause, then grouping and aggregation are subcontracted to the disk process. The disk process also supports prefetch of multiple blocks and post-writes of updated blocks. These features reduce the number of messages between the application (file system) and the disk process.

3.2 Horizontal Partitioning

A table or an index can be horizontally partitioned by specifying the key ranges of each partition. The partitioned table appears as a single table; the file system redirects any SQL SELECT, INSERT, DELETE or UPDATE requests to the proper partition depending on the key range specified in statement. Users can instruct the system to open partitions only when they are needed. This feature improves the performance of queries. by reducing the number of file opens that are needed. It serves as the basis for processing a query in parallel.

3.3 Combination and Elimination of Sorts

A sort may be performed either for eliminating duplicate values or for ordering the data. The optimizer considers a SELECT DISTINCT, an aggregate function that contains DISTINCT, a GROUP BY clause or an ORDER BY clause as a request to sort data. Since sorting is one of the most time-consuming operations in evaluating a query, the optimizer attempts to eliminate redundant sorts. One or more sort requests can be satisfied by a single sort under certain conditions; the optimizer combines sorts when the following occur:

- the select list of a SELECT DISTINCT statement is a subset of the ordering columns of the ORDER BY clause or the grouping columns of the GROUP BY clause
- the grouping columns of the GROUP BY clause form a subset of the select list of a SELECT DISTINCT statement.
- the grouping columns of the GROUP BY clause form a subset of the ordering columns of the ORDER BY clause; the grouping columns form the leading prefix of the ordering columns.
- the ordering columns of the ORDER BY clause form a subset of the select list of a SELECT DISTINCT statement or the grouping columns of a GROUP BY clause

Additionally, the optimizer eliminates a sort when it can use an index as described below:

- An index provides data in the order that satisfies the order required by a GROUP BY or an ORDER BY clause.
- The grouping columns or the entire select list for a SELECT DISTINCT form the leading prefix of an index key for an index that is unique.

3.4 Hash Joins

A variety of prototype implementations and academic papers have shown that hash join algorithms [DeWi85, Schn89] are the join algorithm of choice when a sufficient amount of main memory is available and an index that localizes the number of probes on the inner table does not exist. Usually the square root of the inner table size (in blocks) is considered a sufficient amount of memory.

To take advantage of the large amount of main memory common in modern systems, NonStop SQL supports hash join algorithms in addition to the traditional nested-loops and sort-merge join. The Adaptive Hash Join algorithm used in NonStop SQL [Zell90b] is a variation of the well-known Hybrid Hash Join developed in the GAMMA parallel database machine project of the University of Wisconsin [DeWi85].

NonStop SQL has refined the Hybrid Hash Join algorithm such that it becomes more robust against estimation errors of the input relation size and the amount of available memory. This allows the optimizer to choose a hash join even in cases where the actual values for these parameters are not well known in advance. When selecting a join algorithm, the optimizer considers, among others, the following factors:

- an estimate of the size of the input relations, and
- an estimate of the amount of memory available for the join.

These estimates are used for determining the cost of different join algorithms. The one with the least cost is selected. A nested-loops join or a merge join will not experience a performance degradation if the actual values for the input relation size and memory size differ from the optimizer's estimates. This is because their execution plans are not dependent of the table size or the available memory. For the execution of a hash join, however, some parameters like the number of hash classes and the amount of information that is maintained in main memory are predetermined. A change in the environmental conditions can cause either a performance degradation or even a failure of the join. The Adaptive Hash Join uses several methods to become more robust and to achieve good performance over a large range of memory sizes:

- variable-length I/O transfers to and from the disks,
- dynamic swapping of parts of the hash table to disk files,
- dynamic splits of "hash buckets" into smaller fractions, and
- a "hash loop" algorithm (an algorithm with a cost of O(nwhere n and m are the table sizes) for cases of extreme overflow.

Consequently, the Adaptive Hash Join is able to respond to changing system load by adjusting its memory consumption at various points through its execution. This is described in more detail in [Zell90b].

3.5 Parallel Execution

Parallel execution in NonStop SQL is transparent to the application. This means, that a query does not have to be formulated in a special way to be executed in parallel. The optimizer considers sequential as well as parallel execution plans as a routine part of execution plan generation, costs each one of them and selects the plan with the least cost. This approach is in contrast with the strategy followed by some other RDBMSs that generate a sequential execution plan but execute the operations in parallel. NonStop SQL's optimizer is able to allow both serial and parallel execution plans to compete on the basis of their costs because both classes of plans have costs assigned to them on the same basis. The plan that is finally chosen for execution is the best amongst those considered. Furthermore, NonStop SQL's optimizer has a repertoire of parallel execution plans. Costing allows the optimizer to systematically select the particular type of parallel execution plan that offers the best performance for the given query.

A component called the Executor Server Process (ESP) is introduced for bringing parallelism to relational operations. The executor component in the application program can start a set of ESP's to execute queries or parts of queries in parallel. Thus, there is a master-slave relationship between the application (the master executor) and the ESP's. Each ESP consists of a small main program, executor, and file system. Master and server executors communicate via messages. Fig.2 shows that the master in addition to using the file system that it is bound to uses the server executors to execute requests.

This architecture allows a broad variety of parallel execution plans where parallelism can be handled by the file system for single-variable queries and by the executor for all types of relational operations. The file system always accesses partitioned tables serially. ESP's are used for executing operations for SQL SELECT, INSERT, DELETE and UPDATE statements on partitioned files in parallel. However, if a table has indices that reside on

Figure 2: Executor Server Processes

different disks, then an update to a base table record and the corresponding index updates are done in parallel by the file system. Since the start-up cost for the ESP's may be significant, ESP's are shared between SQL statements in a process and are retained beyond transaction boundaries.

3.5.1 Parallelism Without Repartitioning

A parallel execution plan is a natural way of operating on a NonStop SQL table that is horizontally partitioned. An index may be horizontally partitioned to avoid a bottleneck when processing GROUP BY or ORDER BY clauses. For SQL SELECT, INSERT, DELETE or UPDATE statements, two or more partitions can be processed in parallel. The number of ESP's used for single-table queries is always equal to the number of partitions of the table. For aggregate queries, each ESP computes a local aggregate and returns its result to the master executor. The master computes the query result in the final step.

3.5.2 Dynamic Repartitioning

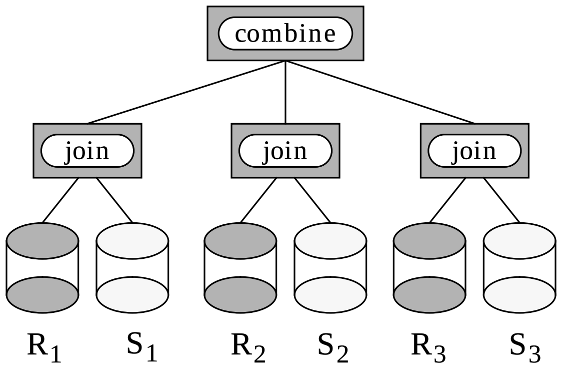
NonStop SQL is able to perform equi-joins and other data-intensive operations such as duplicate elimination or grouping in parallel even when the tables are not partitioned. This is done by reading a non-partitioned table and using a hash-based partitioning mechanism for building a temporary partitioned table.

3.5.3 Parallel Join Execution Plans

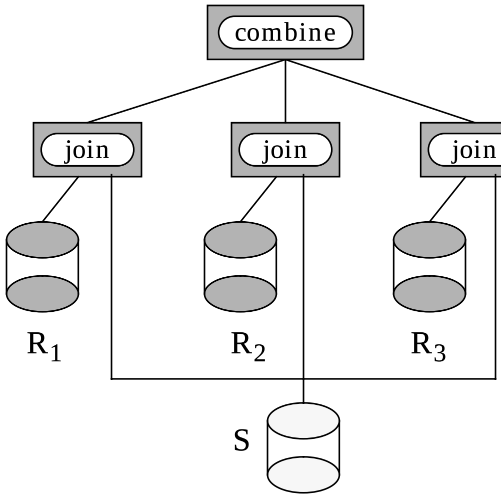
NonStop SQL offers three parallel execution plans for performing joins. The plans can use one or more of the three join methods nested-loops, hash or sort-merge. Fig. 3 illustrates these plans. They are described in the following.

Joining Tables With Matching Partitions: When the two tables are partitioned in such a way that a join of pairs of individual partitions can be performed, then the optimizer considers the performance of a join of each such pair in parallel. For example, consider a banking application that uses the following tables:

Account(Branch, AccountNo, Owner, CreditLimit)
and
Deposits(Branch, AccountNo, Date, Amount)

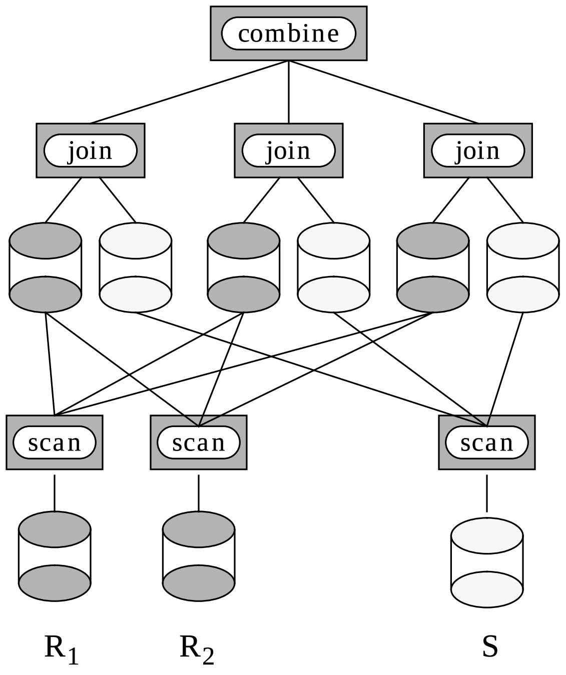


Matching Partitions



Joining a Partitioned Table

With Any Other Table



Hash-partitioned Join

Fig. 3: Parallel Join Execution Plans

Both the tables are partitioned on the same key ranges in the Branch attribute. Hence a join over

Account.Branch = Deposits.Branch AND Account.AccountNo = Deposits.AccountNo

will find matching rows stored in corresponding partitions. It is therefore possible to join each pair of corresponding partitions independently and in parallel. The optimizer recognizes this special but nevertheless quite common case and generates the appropriate parallel execution plan. Designing the database such that joins of matching partitions are possible is a very common and simple means of achieving parallelism.

Joining a Partitioned Table with Any Other Table Parallelism is achieved for this plan by reading each partition of a partitioned table R in parallel. Each partition of R is subsequently joined with the entire table S in parallel. This plan uses either the nested-loops or the hash join method. A nested-loops join is chosen only if an index access path exists for the table S. In order for this plan to achieve high parallelism, either table S should be partitioned and have an index on the join columns or the cost of reading the table S should be very small in comparison with the cost of reading from table R. If this plan performs a hash join, then the entire table S (after applying local selection predicates) is sent to each ESP that processes a partition of the outer table R (fragment and replicate). Each ESP then builds a hash table from S and performs a hash join. Hash joins work very well for this plan in those cases where table S is relatively small. Unlike nested-loops join plans of this type, a hash join plan is able to avoid contention on the inner table and to achieve a high degree of parallelism.

Hash-Partitioned Join Unlike the two parallel execution join plans described earlier, a hash-partitioned join plan is not confined to a specific partitioning scheme of the base tables. It is applicable, however, for equijoins only. Fig. 3 shows two base tables, having one and two partitions respectively, are joined by three parallel join processes. In general, an arbitrary number of join processes can be used. In the current version of NonStop SQL, the number of join processes is equal to the number of CPUs in the system.

This join plan operates in two phases. In the first phase, each of the two tables to be joined are repartitioned as described in section 3.5.2. The repartitioning is realized by applying the same hash func- tion to the values contained in the join columns from each of the tables. This ensures that rows from the outer and inner tables that have matching join columns are stored in corresponding partitions. The second phase of the hash-partitioned join is identical to the parallel join with matching partitions. A pair-wise join of partitions that are built in the first phase is performed. Note that the hash-based partitioning algorithm ensures that matching rows are stored in corresponding partitions.

4 Grouping and Aggregation

Since grouping and aggregation are very common in decision support queries, NonStop SQL uses several techniques that allow grouping and aggregation to be performed efficiently.

When the operand of the MIN or the MAX aggregate function is a column that belongs to an index key and the query contains equality predicates on other key columns that precede it, NonStop SQL computes the aggregate by positioning on either the first or the last row of a given key range for an index. This row always contains the MIN or the MAX value respectively for that key column. The executor simply reads the first row returned by the file system and returns the MIN or MAX value to the application.

NonStop SQL also performs grouping and aggregation in parallel as was described in section 3.

In Release 3 (currently in beta), the disk process computes aggregates for single variable queries. Any number of SQL aggregate functions may be computed using this method for a given query. The optimizer automatically chooses this strategy for single variable queries and when the aggregate function references the innermost table of a join sequence. This optimization reduces the number of messages exchanged by the SQL file system with a disk process and hence reduces the network traffic. It is also used for grouping if rows read by the disk process are in grouping order, that is, whenever the grouping columns form the left prefix of an index key.

In Release 3, the executor supports new algorithms for performing grouping and aggregation using a hash-based method instead of sorting. The method is more efficient and less time-consuming than sorting. If the grouped data can fit in main memory, then grouping is performed in one pass over the raw data. If the grouped data does not fit in main memory, then the raw data is written out to disk using large transfers according to the hash value of the grouping columns. Subsequently, sections of the raw data are read sequentially according to their serial hash value. Grouping and aggregation are performed after the raw data is re-read. This technique ensures that the raw data is read at most twice. This technique is used with both serial and parallel execution plans. When this technique is used by ESP's, each slave ESP hashes and groups data belonging to the local partition. After the elimination of duplicates, resulting groups and partial aggregates are then sent to the master executor process who finalizes the result. This performance enhancement has the effect of reducing network traffic by applying grouping and aggregation locally and reducing inter-network data transfer.

5 User Influence

The optimizer is designed to optimize for the total response time of a given SQL query. Decision support applications often require a few rows very fast, for example, to display the first 10 of them on a screen. They don't care about the overall time to return the entire result of the query. NonStop SQL therefore provides a directive for influencing the optimizer to choose an execution plan that is biased towards returning the first few rows quickly, usually by reading from an index. In this case the optimizer avoids sort operations wherever possible.

For sophisticated users, NonStop SQL provides a mechanism that causes an execution plan to use a specific index, a user-prescribed join sequence, and even the join method to be used, per-force. This mechanism allows users to force a particular execution plan in those cases when the optimizer is unable to generate the access plan that the user wishes to obtain.

6 Performance

In a massively parallel environment it is critical for a DBMS to demonstrate linear scalability in terms of speedup and scaleup.Speedup is a measure of how fast a parallel processing system solves a problem of a fixed size. In other words, users can increase the speed of a query executing with NonStop SQL almost linearly by adding processors and disk drives to their system. Scaleup is a measure of how well a parallel processing system deals with a growing database. In other words, when a batch job grows in size, users can keep its processing time constant by adding more equipment to their system. NonStop SQL demonstrated near linear speedup and scaleup for applications in a benchmark that was audited by the Codd and Date Consulting Group. This benchmark is documented by [Engl89, Engl 90].

We cite a few results from some performance tests that were run in-house as a part of testing our software. Note that the results are not officially validated by an independent auditor. They are presented here simply to underscore the gains of supporting more efficient algorithms for routine processing of queries. A performance improvement ranging from 12% through 60% is observed when the Wisconsin benchmark queries use the hash join method instead of the nested-loops or sort-merge join.When grouping and aggregation are performed using a hash-based method instead of a sort-based one, the Wisconsin benchmark queries show a performance improvement in the range of 60%. When aggregation is performed by the disk process, queries that apply aggregate functions on the leading key columns of a 1000 row table implemented in a key-sequenced file show an improvement in the range of 75%. When the same table contains 100,000 rows, the same query shows an improvement in the range of 230%. The row size in both cases is 208 bytes.

7 Summary and Outlook

NonStop SQL is a robust distributed RDBMS that achieves high performance through low-level integration of relational functions as well as by exploiting the parallelism that is inherent in the Tandem architecture. It incorporates a cost-based query optimizer that optimizes both serial and parallel query execution plans for simple as well as complex queries. It offers query processing features such as parallel execution plans, hash joins, hash based grouping and aggregation in order to allow a large volume of data to be processed very efficiently.

There is an awakening interest for utilizing massively parallel processing on very large databases for commercial applications. NonStop SQL's query processing architecture and the linear speedup and scaleup it has demonstrated make it ideally suited for such applications. In addition, a suite of high-performance utilities for managing a large local or distributed database are built into the product. They also incorporate parallel processing capabilities. The high degree of manageability offered by NonStop SQL has been as important a reason for its commercial success as its query processing capabilities. Furthermore, NonStop SQL's query processing architecture is such that NonStop SQL can very easily be used as a data server for a large number of clients executing on workstations. We believe that NonStop SQL is an excellent solution for the new class of client-server applications as well as those that will be developed for utilizing massively parallel processing commercially.

References

[Borr88] A. Borr, F. Putzolu, High Performance SQL Through Low-level System Integration, Proc. ACM SIGMOD 1988, pp. 342-349.

[DeWi85] D. DeWitt, R. Gerber, Multiprocessor Hash-based Join Algorithms, Proc. 1985 VLDB, pp. 151-164.

[Engl89] S. Englert, J. Gray, T. Kocher, P. Shah, A Benchmark of NonStop SQL Release 2 Demonstrating Near-linear Speedup and Scaleup on Large Databases, Tandem Computers, Technical Report 89.4, May 1989, Part No. 27469.

[Engl90] S. Englert, J. Gray, T. Kocher, P. Shah, NonStop SQL Release 2 Benchmark, Tandem Systems Review 6, 2 (1990), pp. 24-35, Tandem Computers, Part No. 46987.

[Engl91] S. Englert, Load Balancing Batch and Interactive Queries in a Highly Parallel Environment, Proc. IEEE Spring COMPCON 1991, pp. 110-112.

[Lesl91] Harry Leslie, Optimizing Parallel Query Plans and Execution, Proc. IEEE Spring COMPCON 1991, pp. 105-109.

[Moor90] M. Moore, A. Sodhi, Parallelism in NonStop SQL, Tandem Systems Review 6, 2 (1990), pp. 36-51, Tandem Computers, Part No 46987.

[Pong88] Mike Pong, NonStop SQL Optimizer: Query Optimization and User Influence, Tandem Systems Review 4,2 (1988), pp. 22-38, Tandem Computers, Part No. 13693.

[Schn89] D. Schneider, D. DeWitt, A Performance Evaluation of Four Parallel Join Algorithms in a Shared-nothing Multiprocessor Environment, Proc. ACM SIGMOD 1989, pp. 110-121.

[Seli79] P. Selinger et. al., Access Path Selection in a Relational Database Management System, Proc. ACM SIGMOD Conference 1979, pp. 23-34.

[Tand87] The Tandem Database Group, NonStop SQL: A Distributed, High-performance, High-availability Implementation of SQL, Proc. 2nd Int. Workshop on High Performance Transaction Systems, Springer Lecture Notes in Computer Science No. 359.

[Tand88] The Tandem Performance Group, A Benchmark of NonStop SQL on the Debit Credit Transaction, Proc. ACM SIGMOD 1988, pp. 337-341.

[Tand89] The Tandem Database Group, NonStop SQL: A Distributed, High-performance, High-availability Relational DBMS, Proc. First Annual Distributed Database Event, The Relational Institute, 2099 Gateway Place #220, San Jose, CA 95110.

[Zell90a] H. Zeller, Parallel Query Execution in NonStop SQL, Proc. IEEE Spring COMPCON 1990, pp. 484-487.

[Zell90b] H. Zeller, J. Gray, An Adaptive Hash Join Algorithm for Multiuser Environments, Proc. 16. VLDB, 1990, pp. 186-197.

Query Processing in DEC Rdb: Major Issues and Future Challenges

Gennady Antoshenkov
Database Systems Group, Digital Equipment Corporation
110 Spit Brook Road, Nashua, NH 03062, USA
antoshenkov@nova.enet.dec.com

Abstract

DEC Rdb[1] is a relational database management system developed by Digital Equipment Corporation for demanding applications in both OLTP and decision support domains. Discussed in this paper are major issues we have faced during design of the query processing component and what we have learned through the production experience. Also, areas requiring substantial research are outlined, including dynamic optimization, estimation, and gap skipping on ordered data sets.

1 Introduction

It is currently an explicit DEC Rdb goal to efficiently process complex and simple queries against very large databases for a variety of applications and data types. Today fifty gigabyte databases are handled routinely by Rdb and a number of customers have already built and keep building few hundred gigabyte databases.

Database size explosion is happening within the OLTP and decision support domains that often coincide. This puts under stress the existing query evaluation mechanisms whose weaknesses become exposed because of the increased volume of processed data. Also, a demand is growing for support of large objects, texts, and multimedia by the same database engine which handles the traditional relational applications. In this domain we already offer the binary objects and multimedia features and are now in the development cycle for extended data types, user-defined functions, and text retrieval functionality.

In the course of experiencing a rapid db size growth, we learned two lessons:

1. the traditional query optimization technology is weak and its cost model is inadequate for optimizing even moderately complex queries against tables containing 100K, 1M, or more records because of the estimation error amplification.
2. a thorough propagation of new technologies into each area where they can be applied prevents optimization disbalance and potential performance decrease.

The first lesson was particularly hard and fruitful since until recently neither manual optimization nor the query plan store/load were available in the product. As a consequence, in 1990, we built a new, dynamic optimizer for single table retrieval which has a much more refined architecture with dynamic strategy switches based on the new "competition" cost model. This dynamic cost model is derived from the properties of transformation of selectivity/cost probability distributions caused by Boolean and relational operators. The transformation happens to be diametrically opposite to the one assumed in the traditional static optimizers [SACL79]: instead of being

[1]DEC Rdb is a trademark of Digital Equipment Corporation.

convex (e.g. bell-shape, single point surge) and providing cost estimation stability, the Boolean and relational transformations in fact deliver highly concave (e.g. hyperbolic) distributions which express extreme uncertainties and estimation instability [Ant93].

In the new uncertainty-tuned architecture, several execution options are tried in parallel to exhaust high probability regions of processes, leading to the result delivery or to the uncertainty reduction and a reliable execution direction selection. Advancement and switching of processes is controlled by the dynamic optimizer at frequent execution points. Manipulation of record sets is supplemented with heavy usage of record ID lists, bitmap filters (Babb's arrays [Babb79]), and estimation-supporting structures.

The second lesson has to do with integration of new technologies into the existing product. Query processing and optimization are known to be complex and deeply interconnected. Hence, execution strategies selected with a newly introduced feature tend to be based on similar connectivity patterns in all areas where this feature may have a potential impact. If a given feature is not fully propagated in all such areas, the missing pieces may cause performance suboptimality and, along with that, performance instability across similar queries. The latter constitutes the worst system behavior from the customer point of view since the unexpected query slowdown on the production workload is the most disruptive. In critical applications, the user can tolerate suboptimality during prototyping and initial workload trials, but not in the middle of the production run.

The rest of this paper is organized as follows. Section 2 surveys major Rdb features related to query processing. Section 3 describes the extent, roots, and shape of estimation uncertainty of intermediate query data flows. Section 4 discusses short- and long-term solutions for defeating uncertainty. Section 5 presents the "competition" cost model used in Rdb as a parallel alternative strategies driver. In Section 6, architectural issues of the Rdb dynamic optimizer are addressed and compared with other dynamic approaches. Section 7 contains a discussion and examples of technology propagation. Section 8 concludes the paper.

2 Query Processing Overview

Following is a concise description of DEC Rdb features related to query processing which are either in the product release V6.0 (currently in beta testing) or in the previous releases (currently sold to the customers). Features implemented in V6.0 will be explicitly marked throughout this section.

Rdb runs on shared disk cluster composed of any mixture of VAX and Alpha AXP processors [2]. All database operations can be simultaneously performed on a given database by applications or utilities from any of VAX or AXP machines. Data pages can be shared in main memory of a processor by several applications using "global buffers" feature. Distributed processing is part of a different product (not explored here) with Rdb being a server. In client-server architecture, the message traffic can be largely reduced by using "stored procedures" facility (V6.0) which allows a programming-language-style arrangement of many SQL statements to be stored and interpreted by an Rdb server.

Data records and indexes are stored in multiple files where each storage area may accommodate any mixture of tables/indexes, each table or index can be partitioned over many storage areas and thus across multiple files. Both sorted and hashed indexes are supported with a capability of record clustering via each index type. For sorted indexes, the underlying B+ trees support prefix and tail compression in all their nodes, and in addition, order-preserving repeated byte compression (V6.0), bitwise key compression, and irreversible long string truncation. Composite keys can be stored in any asc/desc combination of attribute sequences. Each sorted index can be scanned in direct and reverse (V6.0) order.

Pages, as store units, have their sizes set independently of I/O buffer sizes for more flexible performance tuning. Disk access parallelism is achieved by asynchronous prefetches at sequential scan during query processing (V6.0) and in backup/restore. Striping disk controllers are supported transparently to the query engine (with no

[2]VAX and Alpha AXP are trademarks of Digital Equipment Corporation.

device-specific optimization). Large binary objects are stored as segmented strings and are pointed to from within the records. All or portions of a database can be written to and accessed from the write-once devices.

Query evaluation can be requested to be optimized either for total execution time reduction or for fast first few records delivery. The optimization request choice can cause a substantial performance difference. Update queries have user-controlled update lock timing: (1) when all or majority of records touched during retrieval are expected to be updated, the user can request the update locks to be placed at record retrieval time, otherwise (2) records are read-locked upon retrieval and these locks are escalated to update locks at update time only for a subset of the records to be actually updated.

Rdb query optimizer performs a combinatorial search for best execution plan using exhaustive search and producing a left-deep execution tree. The best execution plan can be stored, altered, and reused by means of "query outline" facility (V6.0). Join execution strategies include: nested loop, merge, and special merge flavors for outer- and full-join, union-join is done via full join (last three in V6.0). Selections are pushed down through joins, unions, and aggregates into index and table retrievals, including extra conjuncts derived from equivalence classes (the last in V6.0).

Retrieval strategies are: sequential table scan, index retrieval with/without record fetches, retrieval by record ID, and Leaf strategy dynamically optimizing access through one or more indexes. OR optimization concatenates streams and eliminates duplicates from multi-index scans and performs a single index scan over ORed list of ranges using zig-zag skip (see example in Section 7). AND optimization is part of the Leaf strategy described in [Ant93].

Rdb employs two different cost models. At compilation time, the cost model drives a single plan selection based on assumptions of independence of operands' data distribution and uniformity (and sometimes skewness) for selectivity calculation. At execution time, a "competition" cost model (see Section 5) is used assuming hyperbolic distribution (see Section 3), with cost estimates upgraded continuously so that execution arrangement can be radically changed at any point of execution.

Statistics for table and index cardinalities are maintained dynamically with $O(log(n))$ precision (utility runs are not necessary, but can be performed at any time to refresh statistics to their precise values). Statistics for selectivity estimation are collected at "start retrieval" time during execution by a quick "no lock" index lookup.

Sort uses quick sort in main memory and replacement selection to merge the sorted runs, both tag sort and whole record swap are employed and picked automatically (V6.0). Integrity constraints have an option to be enforced either at update or at commit time. Constraint enforcement is optimized by omitting redundant checks and by using record ID lists of modified records to narrow down a scope of verification.

Parallelism in Rdb is exploited for the mentioned-above asynchronous record prefetches and for driving the competing processes in the Leaf retrieval strategy. In latter case, parallelism is emulated by very fast switching within a single process, supporting a proportional speed of advancement of participating index scans in accordance with the competition model.

3 Estimation of Intermediate Data Flows

Optimality of query performance largely depends on the ability to estimate the size and cost of producing the intermediate query data flows. All major commercial query engines use a single scalar value to express these estimates for the flows produced by execution plan operators. In addition, the independence of operand data distributions is typically assumed for each operator. The industry-standard cost model [SACL79] uses single value estimates under the operand independence assumption and is widely believed to be sufficiently stable within realistic estimation error limits.

Our production experience with Rdb contrasts the estimation stability belief. Roughly half of all problem reports in the query processing component are related to estimation instability and the subsequent incorrect plan selection that causes performance suboptimality of a few orders of magnitude.

The operand independence assumption does not match our experience either. Having looked regularly at many execution trace prints of customer queries, we were able to see the real effect of ANDing, ORing, etc. on the result sizes. 10% of table ANDed with another 10% of this table would produce either around 0%, or 1%, or 10% of the table, each with substantial probability. These three cases correspond to -1, 0, +1 correlations respectively indicating that probabilities of different operand data correlations are spread across [-1,+1] correlation interval and are not all concentrated around zero point of this interval.

Not having performed a massive statistical experiment for determining a shape of correlation distribution over [-1.+1] interval, we assumed a uniform correlation distribution over [-1,+1] to be much more realistic than zero correlation assumption. We will further talk about **mixed correlation** assuming equal probability of each correlation on [-1,+1].

In a quest for the estimation instability cause, we conducted the following experiment. Given arbitrary probability distributions for estimates at the lowest (table/index scan) level of the execution plan tree, we looked at the intermediate data flow estimate distributions using the mixed correlation assumption. To see the patterns of distribution propagation up through the execution tree, we first investigated distribution transformations caused by different operators and then looked at a cumulative effect of nested operators.

The results of this experiment (published in [Ant93]) demonstrated the following distribution transformation patterns:

1. Uniform selectivity distribution of AND operands are transformed by ANDing into a distribution closely approximated by a truncated hyperbola.
2. ORing produces a "mirror" hyperbola with a surge at selectivity=1.
3. Joins and unions follow AND/OR patterns respectively, presence of duplicate join keys increases hyperbola skewness.
4. Nesting of ANDs and joins produces hyperbolas with quickly increasing skewness, mirror symmetry holds for nesting ORs and unions.
5. At some point of the nested AND/OR balance, the transformed distribution is nearly uniform.
6. Any initial distribution patterns, including a precise (single value) estimate, tend to transform into hyperbolas, often with one or few nested operators.

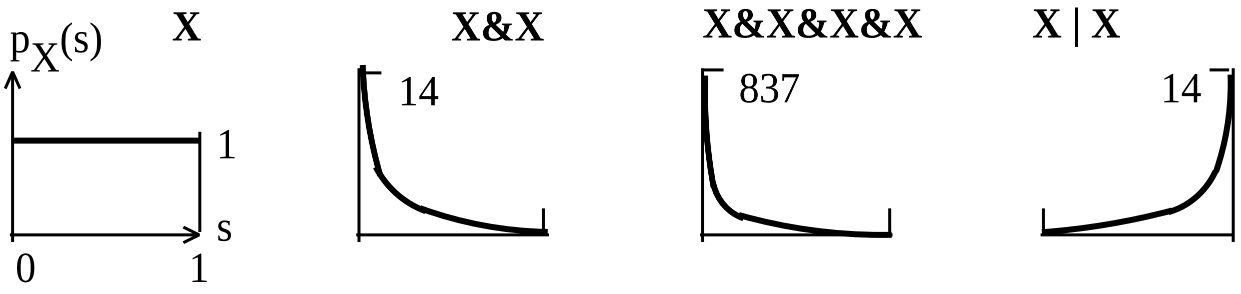
The basic transformation patterns of uniform distribution are illustrated in Figure 1. AND/OR operators assume mixed correlation between their operands.

Based on our production experience, we also found that zero-surge hyperbolas (1/n Zipf distribution [Zipf49]) dominate all other intermediate estimate distributions. A simple explanation for this is that queries searching through the entire database are rare and that small queries and subqueries are frequent.

4 How to Defeat High Uncertainty

Hyperbolic estimate distribution manifests high uncertainty and thus offers an explanation of the traditional cost model instability we observed. Indeed, a selectivity/cost region around the mean point of hyperbola will cover the actual selectivities/costs with the same probability as the narrow region around zero and the wide region stretching to the right end. With these extremes, a validity of the traditional single value (mean point) cost model is largely reduced. Similar result of the exponential error propagation for n-way joins is reported in [IoCh91].

A "quick" solution for compensating this fundamental optimizer limitation has been offered by many vendors in the form of manual optimization tools which allow overriding the optimizer decision by a user-specified



The uniform distribution $p_X(s)$ of selectivity s for predicate X is transformed into a truncated hyperbola for predicate $X\&X$, into a highly skewed hyperbola for predicate $X\&X\&X\&X$, and into a mirror hyperbola for $X|X$.

Figure 1: Transformation of a Uniform Distribution

execution plan. Rdb now has a similar offer called "query outlines". Unlike other manual optimization tools, a query outline accepts any generic recommendations for the desirable execution plan and uses those which are feasible.

Query outline enforces user specification of:

1. full or partial join order,
2. join and retrieval strategies, and
3. a set of indexes to choose from for a given retrieval.

Specifications are optional; any unspecified plan portions are calculated
by the regular optimization procedure. Such flexibility of query outline specification speeds up and simplifies manual optimization. Storing partial recommendations prolongs applicability and usefulness of query outlines in the event of SQL schema modification.

A truly satisfactory solution for unreliable optimization, however, is automatic query optimization capable of efficient estimation uncertainty removal. The first task here is to separate portions of a query expression with sufficiently accurate estimates from the uncertain areas. This requires replacement of the traditional single point estimation with the estimation probability density function represented numerically (the approach we took in our experiments) or by analytical approximation (e.g. by series like in [SunL93]). Uncertainty separation task is not resolved today and requires a substantial research effort.

Using estimation methods based on data stored in indexes/tables delivers sufficient accuracy for a few nested operators above the retrieval nodes. Sampling techniques for a variety of stored data structures are described in [OlRo89], [OlRo90], [Ant93], [OlRo93]. Algorithms and stop rules for sampling estimation of joins and selects are presented in [LiNS90], [HaSw92].

The more operators are involved in a subquery subject to direct sampling estimation, the larger certainty areas can be potentially uncovered. However, restrictions on sample sizes aiming to keep estimation cost lower than execution cost limit the number of nested operators to a few for any reasonable precision goal. In addition, for the combinatorial best solution search, n-factorial number of subsets of n-way operation needs to be estimated to find the best execution tree. Simultaneous multi-goal estimators with a common stop rule could be a good future vehicle to handle this problem.

Suppose at this point that we effectively separate an uncertainty area of a query expression after applying a desirable multi-goal estimation procedure. The well-estimated query portions can be optimized with the known methods, but how do we process the operators fallen into the uncertainty area? One can derive the answer to this question from the shape of the predominant hyperbolic estimation distribution. Based on the hyperbolic distribution properties, in 1990, we developed a competition model [Ant91] for dynamic query optimization and implemented it in Rdb release V4.0.

5 Competition Model

Suppose two alternative strategies s_1 and s_2 can resolve a subquery and at least one of them (say s_1) has a hyperbolic distribution of its cost estimate. With high enough hyperbola skewness, a substantial probability of s_1 cost is concentrated in a small interval $[0, \hat{c}]$ around zero cost. If the subquery evaluation always runs s_1 first until incurring cost $\hat{c}$ or until a successful completion, then with a small initial investment $\leq \hat{c}$, a substantial proportion of the fast s_1 completion cases is fully exhausted. If both strategies have a hyperbolic cost distribution, then the best way to proceed is to run s_1 and s_1 in parallel with the proportional speed of cost advancements and at some point $\hat{c}$ switch to a single strategy s_1 or s_2 whichever has the least estimated average cost.

Described above is a **direct competition** model for running two alternative strategies. For optimal $\hat{c}$ selection defined in [Ant91], direct competition guaranties the lowest average cost arrangement of s_1/s_2 evaluation advancement.

Query execution strategies often contain two or more phases, for instance: (phase 1) index scan delivers a set of record IDs, (2a) this set is sorted, (2b) data records are retrieved in a physical location order touching each page only once. In this example, phase 1 cost is much lower than the combined phase 2 cost. Also, knowing a record ID set, the phase 2 cost can be accurately estimated. If two indexes are available, a competition between the above strategy application to two indexes can be arranged with a parallel run completion defined by the extrapolated cost of both strategy phases. Compared to a direct competition, this **two-phase competition** is much more productive since its “decisive” first phase costs only a fraction of the total strategy cost.

A similar approach for concurrent evaluation of alternative strategies capitalizing on uncertainty reduction is proposed in [Roy91]. The uncertainty reduction in this approach is achieved by randomizing the strategy execution and using a part of intermediate results as random samples for calculating cost estimates. In contrast to the competition model, the randomized execution and sampling do not cover several important cases:

1. n records are requested from a subquery, with ($n < resultcardinality$ – no randomization or sampling can improve knowledge of n when n is unknown, e.g. if n is picked by a user application during execution.

2. Some strategies, like merge over indexes, are usually implemented using a continuous index scan, randomization of which would destroy the merge order.

3. The short subqueries, involving few I/Os, would incur a too high proportion of the randomizing overhead.

We have considered using randomized evaluation in Rdb to speed up the competition uncertainty reduction whenever randomization is applicable. Unfortunately, according to our experiments, a randomized index access based on the acceptance/rejection method incurred three thousand rejections per one successful B-tree descent on 1M record table. Fortunately, we solved this problem by introducing pseudo-ranked B-trees [Ant92] so that with 1% overhead of maintaining approximate branch cardinalities, rejection rate was reduced to 50%. In addition, for index ranges, a single pseudo-ranked tree descent to a smallest branch covering the range may have a precision of approximate cardinalities sufficient for a strategy switch decision.

Parallel processing of alternative strategies should be used with care in heavy update environments because it places more than a minimum amount of locks. A pilot no-lock sampling, which gives up and tries again upon lock encounter, can resolve some uncertainties and reduce alternatives before real processing starts. In Rdb, we perform such pilot no-lock estimation by a single B-tree descent to a range-containing branch.

6 Architectural Issues

Uncertainty of an estimation distribution transformation built into the major relational/Boolean operators is not the only kind of uncertainty to be dealt with. Values of user variables, attributes of already resolved query portions, available space size, contention rate, etc. become available at different execution stages and thus have to be dealt with dynamically. This type of uncertainty can be largely resolved by value binding at the moment these values become available, often at the start of operator evaluation.

A method of the dynamic query subplan selection at the value bind time is proposed in [GrWa89]. Dynamic plans provide a switching mechanism for traditionally-generated alternative subplans exercised at subplan start time depending on the available variable values. Competition-based dynamic optimization, on the other hand, incorporates all the same variables into its cost model, and provides strategy switching at the start and at any other point of subquery execution. It is obvious, however, that the statistically certain switching decisions made at start time with a small computational overhead help to avoid a larger competition overhead without sacrificing optimality. This makes a subquery start time a perfect moment for uncertainty separation (discussed in Section 4) and for the initial arrangement of single or parallel execution threads.

At the level of a single table access, a strategy switching mechanism is described in [MoHa90]. It performs switches during index scans upon reaching some thresholds with a goal to correct occasional suboptimal evaluation directions chosen at compile time. In this approach, the thresholds are also calculated during compilation time (making it similar to the dynamic plans), but the threshold-controlled switches are done within scan execution (making it different than the dynamic plans). In Rdb, the thresholds are partially calculated at compilation time, are finalized at retrieval start time, and are adjusted to the changing "currently best" strategy during retrieval.

In Rdb, retrievals from tables and indexes are controlled by the dynamic optimizer in three stages:

1. Initial stage - does inexpensive calculations and index estimates resulting in the initial competition or a single thread execution arrangement.

2. Index scan stage - a complex mechanism incorporating several cooperative and competing strategies, delivering the requested records and/or record ID list or a decision to perform a sequential table retrieval.

3. Final stage - does the unswitchable record fetches and delivery either from the record ID list or sequentially.

A detailed description of this architecture can be found in [Ant91] and [Ant93]. Dynamic architecture as part of the product is presented in [HoEn91]. At the index scan stage, several kinds of competition take place simultaneously, involving up to three indexes. Fast first record delivery intent may compete with building the record ID list. A sequence of index processing may change in the course of direct competition. Some index scans can be abandoned as losers in two-phase competition for the cheapest final stage. Index scan can compete with the best complete retrieval strategy already discovered.

Despite the novelty and extreme refinement, Rdb user's community grasped the dynamic concept very quickly and offered many suggestions on further architecture refinement. For instance, many users asked for a new competition between index retrieval delivering a desired order and retrieval by record ID list followed by sort. But the most encouraging thing was that when looking through many customer cases, we found almost every conceivable combination of dynamic switches, which led to performance improvements. Since all implemented competitions were tuned to a hyperbolic cost distribution, the consistent presence and variety of switches clearly indicate the hyperbolic distribution dominance.

Through the dynamic optimizer we discovered frequency and unpredictability of data clustering in addition to the one defined in the SQL schema. To exploit such hidden clustering, we added a reliable dynamic clustering predictor which made the final stage I/O estimation near precise.

When we first included dynamic optimization into the production release, we were alert for unexpected problems caused by switches in the middle of execution. The new switching behavior did not bother anyone - imprecise switches were reported and then adjusted by us, but typically, more elaborate switching schemes were requested. One exception was that a switch to sequential retrieval on a table with heavy update activities caused the retrieval transaction to wait and then abort in the middle of execution. We learned from this that for heavy updates gradual locking must be done in all circumstances.

The most striking characteristics of the customer feedback were that (a) almost all general query processing mechanisms applicable to dynamic optimization but not included into it yet were identified and requested to be integrated and (b) within the dynamic optimizer, applicability of new principles to the areas where they were not applied yet was discovered and reported to us as an improvement request. Most of the time, such technology propagation requests were based on particular application cases.

7 Technology Propagation

Seeing opportunities of propagation of technological features and principles into different product areas is usually a mental exercise. But in demanding database environments, missing such opportunities is a practical matter of unexpected performance degradation causing a great deal of customer dissatisfaction. Database applications usually contain a set of queries exercising multiple variations of a few specific features, putting certain query processing mechanisms under stress. When a new feature is introduced in a given stressed area, query strategies change to make use of the new feature. If feature propagation is incomplete, the uncovered cases may not blend with new execution strategies, causing performance degradation.

Technology propagation is needed during each substantial new feature integration. It is also beneficial in the design and research activities. For example, a detailed account of multiple feature propagation (a duality) between hash and sort-merge join techniques is presented in [Gra94]. If both join strategies are to coexist, duality would be desirable not only at the conceptual level but also in implementation.

During the Rdb development cycle, we exercised numerous concept propagation opportunities. With the introduction of “descending” indexes, we allowed any ascending/descending combination of the composite key attributes. Some critical customer applications, however, had numerous queries like

“following a sequence of X, deliver records with ten X just below and ten X just above C”

which can be efficiently resolved with a help of two (ascending and descending) indexes on X. Despite this resolution, the price of double space consumption and extra index updates for two-index approach was too steep. That forced us to implement a reverse index scan and thus deliver a single-index resolution of the above problem.

In 1987, we introduced an efficient merge join of relations A and B over attribute X with index available on B.X. If for any gap between two consecutive values of A.X there are several B.X values to skip, these B.X values are skipped in a zig-zag fashion by trying first all keys on the current index page, and then (for big gaps) descending from the B-tree root to the gap end.

Then we observed that passing the gap end value from A to B can also be done through the aggregate defined on B.X, and implemented zig-zag gap skip over one or several aggregates. Then, when optimizing a multiple range selection, we applied zig-zag skip to the gaps in the ordered range list. Finally, merge join and range list optimizations were integrated into the dynamic retrieval strategy called “Leaf”. If indexes other than B.X (say B.Y) were available for enforcing an additional selection, a competition arrangement called “Sorted” tactic would be picked at the initial stage which:

1. scans B.X and B.Y in parallel,
2. fetches and delivers records on X side in the desirable sorted order,
3. builds record ID filter on Y side for additional Y selection,
4. starts skipping X-side records not in Y-filter when filter is complete.

```
select A.X,
    ( select avg(B.X) from B where B.X=A.X
                                  and B.X in (2,4,8,16,32)
                                  and B.Y<1 )
    from A;
```

```
Match                                  merge join B.X=A.X
 Outer loop
   Index only retrieval on relation A
   Index name AX [0:0]                 full index scan
 Inner loop (zig-zag)
   Aggregate Cunjunct                  avg(B.X)
   Leaf#01 Sorted B Card=4096          retrieval on B
     FgrNdx BX [1:1...]5 Fan=19        B.X in (2,4,8,16,32)
     BgrNdx1 BY [0:1] Fan=19           B.Y<1
```

Figure 2: SQL Query and Execution Plan using Skip Techniques

Figure 2 shows an SQL query combining all the skip techniques described above in a single execution plan, printed by the Rdb strategy dump facility. The gap end values are passed from the Outer loop to the Inner loop through the Aggregate into the Leaf#01 to the FgrNdx (foreground index) BX skip. Simultaneously, the gap ends in the range list [2:2], [4:4], [8:8], [16:16], [32:32] are passed to index BX for skip. At the same time, BgrNdx1 (background index) BY is scanned and its record IDs restricted by B.Y ¡ 1 are used for constructing the "B.Y ¡ 1" filter. If the background BY scan finishes before BX, the complete filter produced by BY is further used to avoid record fetches in BX scan. Otherwise, BX succeeds faster without filtering, and retrieval on B is accomplished.

The example above illustrates how the clustering property of the index structure can be exploited in multiple ways to pick needed data from continuous disk portions and to skip the unneeded records by gap propagation and by using filters. For very large databases, pressure remains to develop more sophisticated and integrated processing methods based on indexes and their order-maintenance, clustering, and declustering properties. The challenge is how to efficiently separate the requested continuous and ordered pieces from gaps and chaos, and then to decide when to apply hash join and aggregation to non-clustered and no-useful-order regions.

8 Conclusion

Query processing tends to transform intermediate data flows in a way that the flow cardinality estimates increase their uncertainties quickly and drastically. This limits a traditional mean-point cost model usage and calls for dynamic optimization methods that deal with uncertainty directly. We found that the dominant shape of uncertainties produced by nested query operators is hyperbola and developed a hyperbola-tuned competition cost model for use in the DEC Rdb dynamic optimizer.

During dynamic optimization, at operator start time, execution of a single or competition of a few alternative strategies is arranged. Parallel run of alternative strategies reduces uncertainty and facilitates a reliable choice of the best way to continue. ¿From customer feedback, we learned about the adequacy of the dynamic approach and about the need for further competition refinement. Within the same dynamic framework, we see significant potentials for speeding up the uncertainty reduction by dynamic sampling and randomized query processing.

The gap-skipping techniques for clustered and ordered data are widely used in commercial RDBMSes today. Their propagation through the query engine, however, is not sufficiently covered in the literature, though their practical importance is on equal parity with such concepts as pushing selects down and using hash join and hash aggregation for non-clustered non-ordered data.

Parallel query processing is the obvious way to employ a quickly growing hardware power for handling very large volumes of data. We expect that the Rdb dynamic optimizer, with its parallel architecture, will match the challenges of parallel processing related to high uncertainty, skewness, and unknown correlations.

With large volumes, incorporation of compression technology becomes important in many areas, including large texts, spatial objects, and traditional structures like B+ trees. In these areas we investigate ways of compression that provide a high speed processing of compressed data.

References

[Ant91] G. Antoshenkov, "Dynamic Optimization of a Single Table Access," Technical Report DBS-TR-5, DEC-TR-765, DEC Data Base Systems Group, (June 1991).

[Ant92] G. Antoshenkov, "Random Sampling from Pseudo-Ranked B+ Trees," Proceedings of the 18th VLDB conference), (August 1992).

[Ant93] G. Antoshenkov, "Dynamic Query Optimization in Rdb/VMS," Proceedings of Ninth Int. Conference on Data Engineering, (April 1993).

[Babb79] E. Babb, "Implementing a Relational Database by Means of Specialized Hardware," ACM Transactions on Database Systems, Vol. 4, No. 1, (March 1979).

[HaSw92] P. Haas and A. Swami, "Sequential Sampling Procedures for Query Size Estimation," Proceedings of the ACM SIGMOD Conference, (June 1992).

[HoEn91] L. Hobbs and K. England, "Rdb/VMS: A Comprehensive Guide," Digital Equipment Corporation, Chapter 6 (1991).

[Gra94] G. Graefe, "Sort-Merge-Join: An Idea Whose Time Has(h) Passed?" to appear in Proceedings of Int. Conference on Data Engineering, (1994).

[GrWa89] G. Graefe and K. Ward, "Dynamic Query Execution Plans," Proceedings of the ACM SIGMOD Conference, (May 1989).

[IoCh91] Y. Ioannidis and S. Christodoulakis, "On the Propagation of Errors in the Size of Join Results," Proceedings of the ACM SIGMOD Conference, (June 1991).

[LiNS90] R. Lipton, J. Naughton, D. Schneider, "Practical Selectivity Estimation through Adaptive Sampling," Proceedings of the ACM SIGMOD Conference, (June 1990).

[MoHa90] C. Mohan, D. Haderle, Y. Wang, J. Cheng, "Single Table Access Using Multiple Indexes: Optimization, Execution, and Concurrency Control Techniques," Advances in Database Technology - EDBT'90, (March 1990).

[OlRo89] F. Olken and D. Rotem, "Random Sampling from B+ Trees," Proceedings of the 15th VLDB conference, (1989).

[OlRo90] F. Olken and D. Rotem, "Random Sampling from Hash Files," Proceedings of the ACM SIGMOD Conference, (June 1990).

[OlRo93] F. Olken and D. Rotem, "Sampling from Spatial Databases," Proceedings of Ninth Int. Conference on Data Engineering, (April 1993).

[Roy91] S. Roy, "Adaptive Methods in Parallel Databases," PhD Dissertation, Dept. of Computer Science, Stanford University, (August 1991).

[SACL79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, T. Price, "Access Path Selection in a Relational Database Management System," Proceedings of the ACM SIGMOD Conference, Boston, (June 1979).

[Zipf49] G.K. Zipf, Human Behavior and the Principle of Least Effort, Addison-Wesley, Reading, MA, (1949).

CALL FOR PARTICIPATION

DATA ENGINEERING

Tenth International Conference on
Data Engineering

February 14-18, 1994
Doubletree Hotel, Houston, Texas
Sponsored by the IEEE Computer Society

IEEE

SCOPE

Data Engineering deals with the modeling and structuring of data in the development and use of information systems, as well as with relevant aspects of computer systems and architecture. The Tenth Data Engineering Conference will provide a forum for the sharing of original research results and engineering experiences among researchers and practitioners interested in automated data and knowledge management. The purpose of the conference is to examine problems facing the developers of future information systems, the applicability of existing research solutions and the directions for new research.

TECHNICAL PROGRAM HIGHLIGHTS

Research Papers Sessions on:
- Disk Storage Management • Management of Distributed Data
- Query Processing • Analytical Modeling • Temporal Databases
- Multidatabase Systems • Knowledge and Rule Management
- Indexing Techniques • Data Mining • Parallel Databases
- Heterogeneous Information Systems.

Invited Presentations:
- Al Aho (Bellcore): “Engineering Universal Access to Distributed Interactive Multimedia Data”.
- Gio W ieder hold (ARPA): “From Data Engineering to Information Engineering”.

Panel Discussions:
- Mobile Computing
- Business Applications of Data Mining
- Future Database Technologies.

Technology and Application Track:
- Practice-oriented presentations of applications of database technologies.

TUTORIAL PROGRAM HIGHLIGHTS

1. **Active Database System:** Klaus Dittrich (Zurich Univ.) & Jennifer Widom (IBM Almaden), Sunday, Feb. 13 1994, (full day).
2. **New Developments in SQL Standards:** Krishn Kulkarni (Tandem Computers Inc.) and Andrew Eisenberg (DEC), Monday morning, February 14, 1994, (half day).
3. **User Interfaces and Databases:** Prasun Dewan (University of North Carolina), Tuesday Morning, Feb. 15, 1994, (half day).
4. **Multimedia Database Systems:** Arif Ghafoor (Purdue University), Monday Afternoon, February 14, 1994, (half day).
5. **Object-oriented Modeling of Hypermedia Documents:** Wolfgang Klas, Karl Aberer (GMD - IPSI), Monday Morning, February 14, 1994, (half day).
6. **Medical Databases:** Lynn L. Peterson and J. C. G Ramirez (University of Texas), Tuesday Afternoon, Feb. 15, 1994, (half day).
7. **Object-Oriented Systems Development: From Analysis to Implementation:** Gerti Kappel (University of Vienna) and Gregor Engels (Leiden University), Monday Afternoon, February 14, 1994, (half day).

HOTEL RESERVATION INFORMA TION

Call the Houston Doubletree Hotel Post Oak at (713) 961-9300 or toll-free 1-800-528-0444 to make your hotel reservation. To obtain the special conference rate of U.S. $85.00 per night for Single or Double, you must mention you are attending the ICDE-94 conference. The cut-off date for guaranteed guest room reservations is Feb. 1, 1994. If you have any questions on registration, tutorials, or program, please send e-mail to icde94@cs.uh.edu or fax to (713) 743-3335.

REGISTRATION FORM AND FEES SCHEDULE

• ADVANCE (Received by January 10, 1994) •
- **Registration**: Member($290), Non-Member($360), Student($70)
- **Tutorials (Full /Half day):** Member($200/$120), Non-Member($250/$150)

• LATE/ON-SITE (Received after January 10, 1994) •
- **Registration:** Member($320), Non-Member($460), Student($110)
- **Tutorials (Full /Half day):** Member($240/$150), Non-Member($300/$185)

The conference registration fee covers the proceedings, conference reception, and refreshments, but not the banquet. Banquet tickets for the Texas Evening, Thursday Feb. 17, 1994 are $38/ea. Additional reception tickets may be purchased for $30/ea. Payment should be in US dollars ONLY (check drawn on a US bank, International Money Order, or credit card) Please complete this form, and return by Jan. 10, 1994 with your payment (payable to ICDE-94) to:

Dr. Albert Cheng, ICDE-94,
Computer Science Department, University of Houston
Houston, TX 77204-3475, USA, Fax: (713) 743-3335

For information contact the Program Chair, Marek Rusinkiewicz: icde94@cs.uh.edu, (713) 743-3350

Name: ______________________________
Address: ____________________________
____
____
E-mail: _____________________________
Fax: ________________________________
•Advance Registration: $________
•Advance Tutorials: $________
Please check the tutorial Number(s):
1__ 2__ 3__ 4__5__ 6__ 7__
•Add’l Reception Tickets ($30/ea): $_______
•Banquet Tickets ($38/ea): $________
•Total Amount Enclosed: $_______ Signature:_______________
• Credit Card: o Visa o Mastercard o American Express
• Credit Card Number: _________________ • Exp. Date:_______