R* Optimizer Validation and Performance Evaluation for Local Queries

Lothar F. Mackert [1], Guy M. Lohman

IBM Almaden Research Center, San Jose, CA 95120-6099

Abstract

Few database query optimizer models have been validated against actual performance This paper presents the methodology and results of a thorough validation of the optimizer and evaluation of the performance of the experimental distributed relational database management system R*, which inherited and extended to a distributed environment the optimization algorithms of System R Optimizer estimated costs and actual R* resources consumed were written to database tables using new SQL commands, permitting automated control from SQL application programs of test data collection and reduction A number of tests were run over a wide variety of dynamically-created test databases, SQL queries, and system parameters The results for single-table access, sorting, and local 2-table joins are reported here The tests confirmed the accuracy of the majority of the I/O cost model, the significant contribution of CPU cost to total cost, and the need to model CPU cost in more detail than was done in System R The R* optimizer now retains cost components separately and estimates the number of CPU instructions, including those for applying different kinds of predicates The sensitivity of I/O cost to buffer space motivated the development of more detailed models of buffer utilization unclustered index scans and nested-loop joins often benefit from pages remaining in the buffers, whereas concurrent scans of the data pages and the index pages for multiple tables during joins compete for buffer share Without an index on the join column of the inner table, the optimizer correctly avoids the nested-loop join, confirming the need for merge-scan joins When the join column of the inner is indexed, the optimizer overestimates the cost of the nested-loop join, whose actual performance is very sensitive to three parameters that are extremely difficult to estimate (1) the join (result) cardinality, (2) the outer table's cardinality, and (3) the number of buffer pages available to store the inner table Suggestions are given for improved database statistics, prefetch and page replacement strategies for the buffer manager, and the use of temporary indexes and Bloom filters (hashed semijoins) to reduce access of unneeded data

1. Introduction

One of the most appealing properties of relational data bases is their nonprocedural user interface Users specify only *what* data is desired, leaving the system optimizer to choose *how* to access that data The built-in decision capabilities of the optimizer therefore play a central role regarding system performance Automated selection of optimal access plans is a rather difficult task, because even for simple queries there are many alternatives and factors affecting the performance of each of them

Optimizers model system performance for some subset of these alternatives, taking into consideration a subset of the relevant factors As with any other mathematical model, these simplifications — made for modeling and computational efficiency — introduce the potential for errors Few of the optimizer models proposed over the last decade [APER 83, BERN 81, CHU 82, EPST 78, HEVN 79, KERS 82, ONUE 83, PALE 74, WONG 76, YAO 79, YU 83] have been validated by comparison with actual performance The only known validations, for INGRES [EPST 80, STON 80, STON 82] and System R [ASTR 80], have been quite limited in scope Carey and Lu have recently compared *distributed* join techniques empirically in some detail [CARE 85] There are many important questions that a thorough validation should answer

- Under what circumstances (regions of the parameter space) does the optimizer not choose the best plan, or, worse, a particularly bad plan?
- To which parameters is the actual performance most sensitive?
- Are these parameters being modeled accurately by the optimizer?
- What is the impact of variations from the optimizer's simplifying assumptions?
- Is it possible to simplify the optimizer's model (by using heuristics, for example) to speed up optimization?
- What are the best database statistics to support optimization?

Performance questions related to optimization include

- Are there possible improvements in the implementation of access paths?
- Are there alternative access mechanisms that are not implemented but look promising?

The goal of our study was to investigate the performance and to thoroughly validate the optimizer of a working experimental database system, R* [LOHM 85], which inherited and extended to a distributed environment [SELI 80, DANI 82] the optimization algorithms of System R [SELI 79] However, this paper considers only local queries, whose optimization and execution is almost

1 Current address University of Erlangen-Nürnberg IMMD-IV Martensstr 3 D-8520 Erlangen, West Germany

identical to that of System R, a companion paper addresses distributed queries [MACK 86] For brevity, we assume that the reader has an introductory familiarity with System R [CHAM 81] and R* [LOHM 85] We first examined local optimization in great detail because it is the basis for the more complicated problem of distributed optimization, the evaluation of which is still underway We have only investigated queries that consist of a single ***query block*** (SELECT FROM WHERE sequence), subqueries (in which the WHERE predicate may contain a query block) [LOHM 84] may be addressed by a future study

The next section discusses how R* was instrumented to collect optimizer estimates and actual performance data, and to reduce this data, in an automated way Section 3 presents some prerequisite measurements of the system and measurement overhead The results for single-table access and sorts are given in Section 4, and for joins in Section 5 Section 6 contains our conclusions

2. Instrumentation

Optimization in R* seeks to minimize a cost function that is a linear combination of four components CPU, I/O, and two message costs that are not relevant to this study and were always zero number of messages and number of bytes transmitted I/O cost is measured in terms of number of actual transfers to or from disk, not buffer looks To permit more detailed modeling, CPU cost in R* is in terms of number of instructions, whereas System R estimated the number of calls to its record-level storage system (Relational Storage System or ***RSS***) and converted it to I/Os

$$\text{SysR\_\_cost (in I/Os)} = (\#\_\_I/Os) + 1/3 * (\#\_\_RSS\_\_calls)$$

$$\text{R*\_\_cost (in msec)} = W_{CPU} * (\#\_\_instrs) + W_{I/O} * (\#\_\_I/Os) + W_{MSG} * (\#\_\_msgs) + W_{BYT} * (\#\_\_bytes)$$

Unlike System R, R* maintains the four cost components separately, as well as the total cost as a weighted sum of the components [LOHM 85], enabling validation of each of the components independently By assigning (at database generation time) appropriate weights for a given hardware configuration, different optimization criteria can be met Two of the most common are time (delay) and money cost [SELI 80]

R* inherited from System R some 40 internal counters of significant events (including number of buffer page reads and writes, and number of actual disk reads and writes), as well as some interactive tracing and debugging tools that enable a user to stop a process at a given address and examine and/or alter the values of system variables (including the counters) These tools might have been used without change to override the optimizer's choice of plan, and/or to examine the I/O counters before and after a query was executed However, an earlier performance study for System R [ASTR 80] demonstrated that the large amount of manual interaction to use these facilities severely limited the number of test cases that could be examined Since we wanted to measure performance under a wide variety of circumstances, we added instrumentation that would automate measurements completely in the following way

1 **Store all optimizer estimates and performance data in database tables**, permitting this data to be structured, subsetted, and ordered in different ways using the full power and flexibility of the SQL query language To achieve independence between multiple users, these tables are *not* catalog tables but user-owned tables They are created by the system on behalf of the user when needed the first time (presuming he has the appropriate authority), and they are treated as any other table owned by that user

2 **Add to the SQL language user-invoked statements for test control and performance monitoring, whose output is written to the tables described in (1) above** These statements permit tests to be both controlled and measured automatically from pre-compiled application programs

3 **Develop application programs for (a) testing queries and (b) analyzing the data collected by step (a)**

Details of the last two items are discussed below

2.1. SQL-level tools

We implemented in R* three new SQL statements — described below — that enable a user's application program to control performance tests and to write automatically into database tables data on the estimated and actual cost of that test The first two statements are generally available to any R* user, whereas the last statement was limited to a special version of R* for testing In fact, the EXPLAIN statement we implemented in R* is similar to the EXPLAIN statement in the SQL/DS product [RDT 84, SQL 84], with extensions for the distributed environment of R* All three statements are operational in a distributed environment, but, since this paper focuses on local queries only, we will describe their local functionality only

1 **EXPLAIN**

Originally invented for use in automated data base design [SCHK 79, FINK 82], the EXPLAIN command writes to user tables information about the access plan chosen by the optimizer for a given SQL statement, and its estimated cost The syntax of the EXPLAIN statement and the tables into which it inserts tuples are omitted due to space constraints, but are described in [RDT 84, SQL 84] The R* EXPLAIN can be executed from an application program as well as interactively, a requirement for automated testing of many different types of queries

2 **COLLECT COUNTERS**

This new SQL statement, designed specifically for this study, causes internal R* counters to be written to user-owned tables, tagged with a timestamp, the invoking application program name, and an optional user-supplied sequence number The user does not specify in the COLLECT COUNTERS statement which counters to collect the system automatically collects a subset of the internal R* counters that might be useful to any user as well as our study During our tests, only one test program was active at a time, so system-wide counters (such as disk I/Os) were equivalent to user-specific counters Execution of this instruction creates in the user's private DBSPACE the table <userid> COUNTER__TABLE, if it does not already exist, and inserts one tuple per counter The optional sequence number enables the user to group all such tuples by an integer identifying the COLLECT COUNTERS statement that generated them The columns of COUNTER__TABLE are defined as follows

TIME	A human-readable timestamp measured to hundredths of a second
APPLNAME	The application program name that invoked COLLECT COUNTERS
SEQUENCE	The optional, user-assigned sequence number, for relating each tuple to its source COLLECT COUNTERS statement
DBID	The database identifier of the R* system from which counters were collected (for distributed use only)

COMPONENT	The R* internal component (currently, RSS* or DC*) which maintains the counter
COUNTER_NAME	The name of the counter, e g "SORT CALLS", "RSS CALLS", "DATA PAGES READ", "DATA PAGES WRITTEN", "SENT MESSAGES", "SENT BYTES", etc
COUNTER_VALUE	The value of that counter (a monotone non-decreasing integer)

The net R* resources that are consumed executing any sequence of SQL statements can thus be calculated by subtracting the values of the counters COLLECTed before that sequence from those COLLECTed afterward, less the cost of measurement (2 COLLECT COUNTERS statements) The net elapsed time for the sequence is calculated by a similar subtraction of the TIME columns

Since there is no R* counter for the number of instructions executed by R*, nor any such operating system counters, we had to use an instruction trace tool based on the CP trace facility in the Virtual Machine (VM) operating system, to measure actual CPU costs

3 FORCE OPTIMIZER

In order to measure the performance of plans that the optimizer thought were suboptimal, we had to be able to overrule the optimizer's choice of plan This was done with the FORCE OPTIMIZER statement, which was implemented in a special test version of R* only The user specifies the desired plan number, a unique positive integer assigned by the optimizer to each candidate plan, by first using the EXPLAIN statement discussed above to discover the number of the desired plan The FORCE OPTIMIZER statement chooses the plan for the *next* SQL data manipulation (optimizable) statement *only* To ensure the complete generation of *all* candidate plans, the special test version of R* also had to be changed to override the routine pruning by the optimizer of dominated plans [SELI 79, LOHM 85], and to allocate space for storing more candidate plans

2.2. Application Programs

To study the system performance and validate the optimizer under a wide variety of conditions, we developed two kinds of application programs that totally automated the test process The first kind collected estimated and actual costs for a number of different queries under some particular test parameters (e g , column distributions) for a number of different test cases (e g , table sizes, existing indexes), using the SQL tools described above An outline of this kind of application program is given in Figure 1 Estimated costs for each candidate plan in each query are inserted into COST_TABLE by the EXPLAIN statement, actual costs are inserted into the ACTCOST table, and the values of parameters for the set of tests are recorded in the SITUATION table All buffer pages are flushed for each execution by reading a large, unrelated table from another DBSPACE, to ensure that no pages from a previous execution remain in the buffer

Once we had gathered the performance and monitor data, we reduced it to result tables using additional application programs that varied depending upon the topic of interest SQL SELECT statements were used to extract parameter data from the SITUATION table, estimated cost data from COST_TABLE, and actual cost data from the ACTCOST table The resulting tables were then either printed or plotted using an R* interface to the Graphical Data Display Manager (GDDM) graphics product [GDDM 84]

Figure 1 Outline of application program for automated testing.

```
/* create various-sized outer & inner tables, w/ & w/o indexes */
FOR ALL combinations of table size(s)
  Generate & insert tuples randomly (according to assumed distn )
  FOR ALL combinations of indices on table(s)

    CREATE/DROP INDEX(es)
    UPDATE STATISTICS FOR TABLE(s)
    INSERT INTO SITUATION VALUES (Sitn#, test parameters),

    /* try different queries for the same table configuration */
    FOR ALL queries Q
      FOR ALL candidate plans P for query Q

        FORCE OPTIMIZER TO PLN# P,   /* to do EXPLAIN */
        EXPLAIN PLAN, COST FOR query Q
        Flush all buffer pages
        FORCE OPTIMIZER TO PLN# P,   /* to do execution */

        COLLECT COUNTERS Sequence# = 1,
        Execute query Q to be tested (OPEN cursor and do FETCHes)
        COLLECT COUNTERS Sequence# = 2,

        FOR ALL counters C
           SELECT COUNTER_VALUE INTO  BEFORE FROM COUNTER_TABLE
              WHERE  COUNTER_NAME = C AND SEQUENCE = 1,
           SELECT COUNTER_VALUE INTO  AFTER FROM COUNTER_TABLE
              WHERE  COUNTER_NAME = C AND SEQUENCE = 2,

           /* X_C = Column in ACTCOST for C = net C resources */
           X_C =  AFTER -  BEFORE - cost of measuring
        INSERT INTO ACTCOST VALUES (Sitn#, Q, P, X_C values),
        DELETE FROM COUNTER_TABLE, /* delete all counter tuples */
```

2.3. Conduct of Experiments

Using the above tools, a more automated and systematic analysis of different variations of parameters was possible than the previous validation study of System R [ASTR 80] The test database could be (and was) dynamically generated to conform with given values of the parameters, whereas Astrahan et al varied only the queries to a fixed database of two tables (one of 145 pages and one of only 2 pages) Component-wise validation minimized the importance of choosing appropriate component weights for the cost function, and excluded the possibility of missing two modeling errors in separate components because they might offset each other in the total cost Finally, plots of our measurements readily revealed trends that otherwise might have been lost in the myriad observations in tabular form

All measurements were run at night on a totally unloaded IBM 4381 to preclude any interference from other users Hence R* total cost (resources consumed) in terms of time could be measured as response time, ignoring any overlap of processing and disk I/O R* was initialized to provide 40 pages of buffer pool, which were all available to that test user The value of each column in each tuple inserted in a table was drawn randomly from a uniform distribution, as assumed by the optimizer, except when we were explicitly investigating the impact of dropping that assumption (see section 4 2 below)

Each test was run several times to ensure reproduceability of the results, and to reduce the variance of the average response time However, the reader is cautioned that these measurements are highly dependent upon numerous factors peculiar to our test environment, including hardware and software configuration, database design, etc We made no attempt to "tune" these factors to advantage For example, each test table was assigned to a separate DBSPACE, which tends to favor DBSPACE scans The absence of other users competing for system resources probably yielded overly-optimistic results On the other hand, in more realistic environments, our test programs might actually run faster if they referenced pages that had been recently referenced by another user and hence remained in the buffer

What follows is a sample of our results illustrating major trends for local queries only, space considerations preclude showing all combinations of all parameters that we examined or any results for distributed queries For example, for joins we tested a matrix of table sizes for the inner and outer tables ranging from 100 to 6000 tuples (3 times the buffer size), varying the join-column domain size (500, 3000, or 10000 possible values), and projection factor on the joined tables (50% or 100% of both tables) Although these tests confirmed the accuracy of the overwhelming majority of the optimizer's predictions, we will concentrate here on those aspects of the R* optimizer that were changed or exhibited anomalous behavior

3. General Measurements

Several measurements pertaining to the optimizer as a whole were prerequisite to more specific studies These are discussed briefly below

3.1. Cost of Measurements

The COLLECT COUNTERS statement, the means by which we measured performance, itself consumes system resources that are reflected in the R* internal counters Hence the first measurement we made was to determine the resources consumed by two executions of the COLLECT COUNTERS statement This was accomplished by simply running two COLLECT COUNTERS statements with no SQL statements between them This test was run several times to get a good average value One has to be very careful here unless the statements between the two COLLECT COUNTERS fill either buffer, no *actual* (disk) I/O is required for the second COLLECT COUNTERS statement

3 2. Component Weights

The R* cost component weights for any given cost objective and hardware configuration can be estimated using "back of the envelope" calculations For example, for converting all components to milliseconds, the weight for CPU is the number of milliseconds per CPU instruction, which can be estimated as just the inverse of the MIP rate, divided by 1000 MIPS/msec The I/O weight can be estimated as the sum of the average seek, latency, and transfer times for one 4K-byte page of data These estimates, and the corresponding actual weights for our test configuration, are shown in Figure 2 The observed per-I/O rate is better than the estimate because the seek time was almost always less than the nominal average seek time, since R* databases are stored by VSAM in clumps of contiguous cylinders called extents

3.3. Cost of R* Environment

Our early measurements show a steep increase in the total cost as more tuples were requested by the test application program However, SQL queries requesting set functions (e g , AVERAGE) on 5000 tuples responded in one hundredth the time of the same query returning all 5000 tuples individually, even though R* did the same amount of I/O for both queries! The cause of this apparent anomaly turned out to be the overhead of returning tuples individually via "vanilla" VTAM (Virtual Telecommunications Access Method [VTAM 85]) from the address space of the R* system to the application program's address space, as shown in the "without blocking" curve of Figure 3 The queries of all subsequent tests SELECTed only aggregate functions of columns (e g , AVERAGE, MINIMUM, etc) to eliminate this environmental factor while still requiring the data to be accessed By blocking tuples returned to the application program into "mailboxes", as is done in the SQL/DS product, most of this overhead has now been eliminated in R* (cf the "with blocking" curve in Figure 3) This is the most dramatic example of a byproduct of this study thorough testing of R* under a variety of extreme conditions helped to isolate and correct any remaining bugs and performance anomalies, and thus to strengthen confidence in the robustness of R*

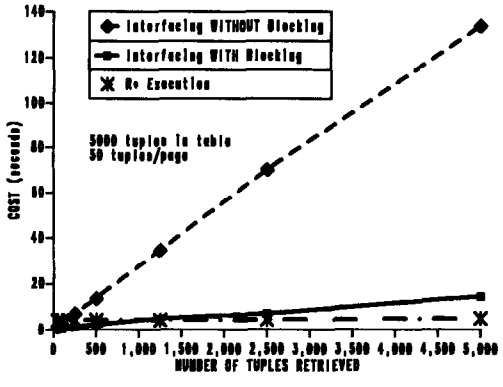

Figure 3 Overhead of CICS/VTAM application program interface, *with* and *without* tuple blocking

4. Single-Table Access

Queries accessing a single table are the simplest possible SQL query, and also the basis upon which multi-table (join) queries are optimized [SELI 79, SELI 80, DANI 82] Hence we next examined the single-table optimizer model to understand the simpler case

$$R^*\_total\_cost = W_{CPU} * (\#\_instrs) + W_{I/O} * (\#\_I/O) + W_{MSG} * (\#\_msgs) + W_{BYT} * (\#\_bytes)$$

WEIGHT	HARDWARE/SOFTWARE	ESTIMATE	ACTUAL
W_{CPU} (in msec/instr)	IBM 4381 processor	0 0004	0 0004
$W_{I/O}$ (in msec/I/O)	IBM 3380 disk	23 48	17 00
W_{MSG} (in msec/msg)	CICS/VTAM	11 54	16 5
W_{BYTE} (in msec/byte)	24M bit/sec (nom), 4M bit/sec (eff)	0 002	0 002

Figure 2 Estimated and actual cost component weights.

fully and to eliminate any modeling errors before dealing with the more complex interactions of multiple tables

We compared the estimated and actual costs of all possible ***access paths***, for various combinations of the ***table cardinality*** (number of tuples in the table) and the ***projectivity*** (proportion of each tuple SELECTed) Access paths implemented in R* (and System R) are briefly defined as follows The ***DBSPACE scan*** access path is a *physically* sequential scan of the DBSPACE containing the table An ***index scan*** traverses, in *logical* order, an index on one or more columns of the table The index is said to be ***clustered*** if the page references for an index scan are sufficiently close to their physical order [SCHK 85, MACK 85], otherwise it is ***unclustered*** The ***sorted scan*** access path reads the table — by the cheapest access path available — directly into a sort routine to logically order the tuples Logical order may be necessary for an ORDER BY or GROUP BY clause, a DISTINCT function, or (for multi-table queries) a merge-scan join Projectivity was found to be relevant primarily to the sorted access path, and was modeled correctly in all cases, so the effect of this parameter is not discussed further

4 1 Relative Importance of Cost Components

A fundamental question requiring empirical confirmation was whether CPU cost was significant enough to warrant being considered at all, much less modeled in more detail The contribution of CPU and I/O costs to total cost for different access paths is shown for the same query to a table of 1000 tuples (Figure 4) and 5000 tuples (Figure 5) The query has a simple (***SARGable***, i e executable by the RSS* [SELI 79]) predicate that eliminates half the tuples, plus a more complex (***residual***) predicate executable only by the RDS* The RDS* is the executive component of R* which — at run-time — invokes the RSS* to get tuples, and does conversions, arithmetic, and joins

Clearly CPU cost is significant, even when accessing large tables, although not enough to affect the choice of the optimizer for this query The additional CPU cost of the "sort" access path confirms that the CPU cost for sorting (modeled by System R as one RSS call plus some CPU for encoding and decoding column values to be sorted) is sufficiently different from other RSS* calls to justify a more detailed model, discussed in section 4 3 below

4.2. I/O Component

The System R formulas for I/O for a single-table query proved very accurate for all access paths but the unclustered index scan, which is complicated by the probability that successive tuples are on the same page and the probability that that page is still in the buffer (This explains, for example, why the cost of an unclustered index scan increases disproportionately between Figure 4 and Figure 5) The System R formula ignored the latter probability, pessimistically assessing one I/O per tuple whenever the table size exceeded the buffer size An improved I/O model for unclustered index scans was developed, implemented in R*, validated, and used for all tests presented in this paper, including those of Figure 4 and Figure 5 A detailed description of that model and its validation is given in [MACK 85] Hereafter, we will consider only unclustered indexes, since clustered index scans are modeled correctly and are unquestionably the best access path when they exist

As in System R, the R* optimizer's model assumes that

1 the values of all columns are uniformly distributed over their domain,
2 there are a constant number of tuples per page, and
3 tuples are either perfectly clustered or else randomly assigned to pages

While these assumptions were enforced by the programs creating tables for all tests in this paper, we also investigated the impact on single-table access of relaxing each of these assumptions A more general model that relates and exploits such deviations by collecting some appropriate statistics on the page reference string is the subject of another paper currently being drafted

The cost model for the I/O component proved so accurate that its validation revealed an implementation error and a major potential optimization The bug, now fixed, resulted from the RSS* default of reading each page into the buffer before it is written to disk This clearly is useless for pages of a temporary table newly created for accumulating intermediate results for a sort, and was easily overridden The potential optimization would combine the first merge scan with the initial scan that quicksorts each page of the table to be sorted Saving one scan of the table per sort could improve performance significantly (33% when 3 scans are required), but required a major restructuring of the sort module and hence was not implemented in R* The optimizer's I/O cost model for sorts required no changes

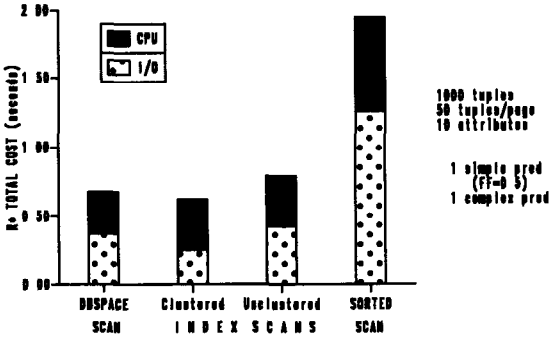

Figure 4 Relative importance of cost components for accessing a single table of 1000 tuples by various access paths.

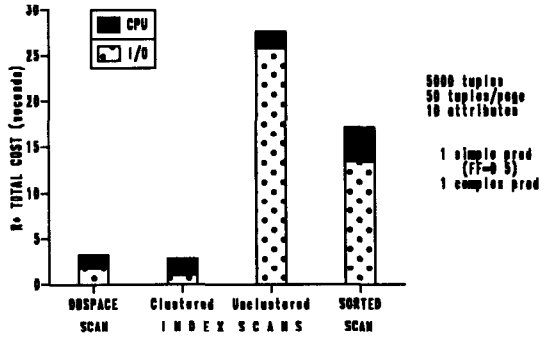

Figure 5 Relative importance of cost components for accessing a single table of 5000 tuples by various access paths.

4 3. CPU Component

The significant contribution of CPU to the total cost, coupled with known path-length differences for various RSS* calls, motivated us in R* to model CPU costs in more detail than just RSS* calls For example, although only 1 RSS* call is needed either to FETCH one tuple or to sort a table, the sort requires far more CPU for (1) allocating temporary disk space to store partially sorted strings of pages to be merged, (2) encoding and decoding the columns to be sorted, and (3) quicksorting individual pages in memory Our tests also revealed that the CPU cost of FETCHing a tuple was largely a function of the number of SARGable and residual predicates, as shown in Figure 6 Five SARGable predicates, being simpler to evaluate and applicable by the RSS*, consumed almost the same CPU as one residual predicate applied by the RDS* More selective SARGable predicates further reduce CPU by returning fewer tuples to the RDS*, saving both RSS* calls and residual predicate applications

New CPU cost models, validated by extensive testing and implemented in R*, are summarized by the following equations For a scan of a table, the number of instructions is given by

$$CPU_{SCAN} = OPENINST + FF_{KDOM} * TABLECARD * (TUPLINST + SARGINST) + FF_{SARGs} * TABLECARD * (RSSCALL + RESIDINST)$$

where[2]

$$TUPLINST = \begin{cases} 0 & \text{IF no index (DBSPACE scan)} \\ BUFFERINST & \text{IF only index pages accessed} \\ 2 * BUFFERINST & \text{OTHERWISE} \end{cases}$$

and

$$RESIDINST = \begin{cases} 0 & IF\ \#RES = 0 \\ RESID\_SETUP + \sum_{i=1}^{\#RES} ARINST_i & IF\ \#RES > 0 \end{cases}$$

for the following system constants

OPENINST —the number of instructions to OPEN and CLOSE the scan

SARGINST —the number of instructions to apply SARGable predicates (a linear function of the number of SARGable predicates, a query-dependent variable)

RSSCALL —the number of instructions to invoke the RSS*

BUFFERINST —the R* overhead for accessing a page in the buffer and extracting the desired tuple

RESID_SETUP —the number of instructions overhead for applying residual predicates

ARINST$_i$ —the number of instructions for calculating arithmetic expressions in residual predicate i (a linear function of the number of such expressions in predicate i, a query-dependent variable)

and the following query-dependent variables

TABLECARD —the cardinality of the table

FF$_{KDOM}$ —the estimated filter factor of any predicates that can be applied using the index (***key domains*** or ***KDOMs*** [SELI 79]), thus limiting the index scan to a subtree, =1 otherwise or if this is a DBSPACE scan

FF$_{SARGs}$ —the estimated filter factor of all SARGable predicates

#RES —the number of residual predicates

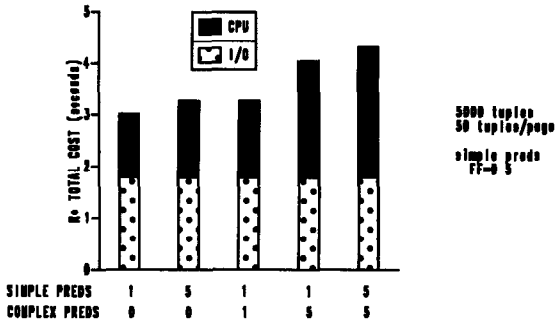

Figure 6 The effect of applying predicates on CPU cost for accessing one table

Intuitively, the first term of CPU_{SCAN} is the overhead of OPENing and CLOSEing the scan The second term estimates RSS* costs for applying SARGable predicates and for extracting the tuple from the buffer, for the portion of the table that any key domains limit the scan to In the case of a DBSPACE scan, the entire table is accessed ($FF_{KDOMs} = 1$), but the tuple extraction cost is negligible (TUPLINST $\approx$ 0) because it is done once per page rather than once per tuple For index scans, TUPLINST reflects the fact that only the index leaf page need be accessed whenever (a) the index contains all the columns of that table referenced in the query ***and*** (b) there are no predicates on columns not in the index, otherwise both the index leaf and data pages must be accessed The number of I/Os are also affected by whether the index and data pages need be accessed The final term of CPU_{SCAN} assesses RDS* costs for invoking the RSS* and applying residual predicates, for each RSS* call The number of RSS* calls is estimated as the table cardinality reduced by the estimated filter factor of all SARGable predicates (FF_{SARGs}), including KDOMs

Actual instruction counts were validated for each of the system constants, for both index scans and DBSPACE scans, but these details are implementation-dependent and are omitted here The only differences between index seans and DBSPACE scans for these constants are (1) OPENing and CLOSEing an index scan requires 8/15 the instructions of that for a DBSPACE scan, and (2) each RSS* call requires 1 2 times as many instructions for the index scan as for a DBSPACE scan

For sorts, the number of instructions is given by

$$CPU_{SORT} = ACQ\_TEMP + \#SORT * CODINGINST + \#PAGES * QUIKSORTINST + \#PASS * (ACQ\_TEMP + \#PAGES * I/O\_INST + \#SORT * NWAY * MERGINST)$$

where

$$\#PASS = \text{CEILING}(\log_{NWAY}(\#PAGES))$$

for the following system constants

---

2 For DBSPACE scans, this access is done only once per page rather than once per tuple i e BUFFERINST/TABLECARD and hence is minimal)

ACQ_TEMP —the number of instructions to acquire, OPEN, and CLOSE a temporary relation in which to store intermediate results

CODINGINST —the number of instructions to encode and decode the fields to be sorted (a linear function of the number of such fields, a query-dependent variable)

QUIKSORTINST —the number of instructions to quick-sort 1 page

I/O_INST —the R* overhead for accessing a page in the buffer, comparable to BUFFERINST earlier plus instructions for finding a buffer slot, obtaining locks, etc

NWAY —the number of strings of pages to be merged at a time (≤8 in Systems R and R*)

MERGINST —the number of instructions to compare the values of one tuple to be merged

and the following query-dependent variables

#SORT —the number of tuples to be sorted

#PAGES —the number of pages occupied by the table to be sorted

Here the first term is the overhead of the sort, primarily for allocating a temporary segment to store intermediate and final results of the sort The columns to be sorted in each tuple are encoded before sorting and decoded after sorting, and each page is initially quicksorted The last three terms reflect the overhead, per-page, and per-tuple costs for each merge pass

4.4. Potential Improvements

The optimizer and run-time routines in R* currently do not communicate to the buffer manager the type of scan (DBSPACE or index) of which a page request is part each page request is treated independently There is no pre-fetching, and paging out follows a strict least recently used (LRU) policy [EFFE 84, MACK 85] Hence old pages of a DBSPACE scan remain in the buffer even though they are unlikely to be referenced again, forcing out pages in unclustered index scans that *are* likely to be referenced again Knowing a page request is part of a DBSPACE scan, the buffer manager could pre-fetch the next few pages in physical order, re-using only a couple buffer pages For index scans, the index leaf pages contain a *run* of pointers to pages — in physical order — for each index value The buffer manager could pre-fetch all pages in a run and — by looking ahead to other runs already in memory — avoid paging out those pages that will be requested soon by another run

Chou and Dewitt [CHOU 85] have already pointed out these limitations of the LRU buffer management scheme, and have developed a taxonomy of database reference patterns and a promising buffer management scheme called DBMIN that dynamically allocates buffer pages and tailors the buffer management policy individually based upon the page reference pattern for each table reference in the query The allocation required is based upon the table's cardinality and the reference pattern type, which depends upon the plan chosen by the optimizer for a particular join the join method, order of tables, and access paths of both the inner and outer table A scheduler initiates execution of a new query only when enough free buffer pages are available for the buffer allocation for all of its table references, much as in the "hot set" model of Sacco and Schkolnick [SACC 82] Analytic models have shown DBMIN uniformly performs better than other database-oriented buffer management schemes Since DBMIN is a major departure from the R* buffer management policy, we have not implemented it in R* but are considering it for future work

5. Joins

Having thoroughly studied single-table access, we next investigated the R* (and System R) optimizer's model for joining two tables under various access methods (tables indexed or not) and join methods (nested loop vs merge scan [BLAS 77, SELI 79]) In R*, as in System R, only single tables are joined as the *inner* table to an *outer* table, which might be a *composite* table, i e an intermediate result Composites need be completely materialized only when they must be sorted, otherwise they are materialized as needed in a "pipelined" fashion [CARE 85] Since n-table joins are composed of n-1 two-table joins, inaccurate modeling and plan choices by the optimizer for one of those two-table joins could cause it to choose a suboptimal join order Thus our discussion will concentrate on two-table joins, of which a thorough understanding and correct modeling is prerequisite to validating n-table joins

5.1 Cost Components' Importance

First, we needed to confirm that CPU cost was a significant component of total cost during joins Since all the I/O cost of a join derives from the cost of accessing the two tables and/or sorting them to do a merge join, and these operations had significant CPU costs, one would expect an even larger CPU contribution to total cost for joins This reasoning is confirmed by Figure 7 for joining two 1000-tuple tables (cf Figure 4) In fact, the portion of the total cost contributed by CPU is higher for joins than for single table accesses for two reasons (1) joins must OPEN and CLOSE scans on the inner table for each outer tuple, and (2) when sorting any outer table (composite or not) for a merge join, R* always inserts it first into a temporary table Both of these are very CPU-intensive For two 2500-tuple tables, CPU cost still dominates for a merge join when both tables remain unindexed, but is only 13% - 35% of the total otherwise

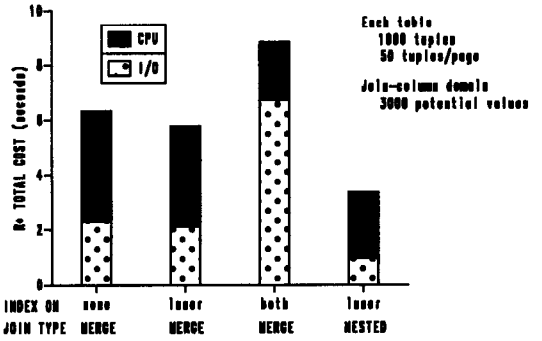

Figure 7 Relative importance of cost components when locally joining two 1000-tuple tables, for different combinations of (unclustered) indexes and join types.

5 2 Actual vs. Estimated Costs

Does the optimizer model reality correctly, i e , does it order the alternatives correctly? The answer is "most of the time, but not always" As stated earlier, the optimizer correctly chooses clustered index scans whenever they exist, and thus these will not be discussed further in this section When the inner table is unindexed, the optimizer correctly estimates *in all circumstances* that a nested-loop join will be hopelessly expensive unless the inner table fits in the

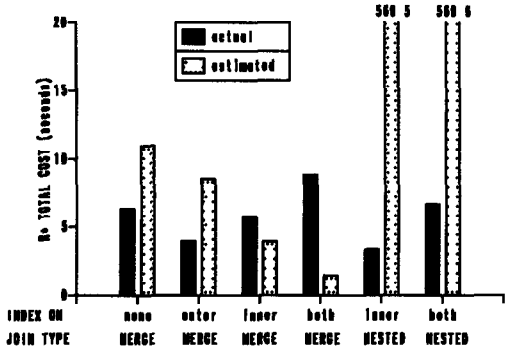

Figure 8 Actual vs. estimated cost for joining 2 tables of 1000 tuples each, for varying access methods & join types.

buffer for the *entire* join Clearly the merge-scan join improves performance for joins of tables that are both larger than the buffer and unindexed on the join column, a quite common situation This agrees with the results of Carey and Lu [CARE 85] We will not discuss these cases further, concentrating instead on cases for which modeling anomalies occur

When there is an (unclustered) index on one or both of the tables, the optimizer has more difficulty modeling the complex I/O situation, particularly for smaller tables Figure 8 shows a particularly anomalous situation for which the best actual plan is estimated to be almost the worst plan (and vice versa) Since tuple order is unimportant to a nested-loop join, an index on the outer table only serves to get in the way by consuming additional buffer pages for the index pages (compare the actuals for the two nested-loop cases shown) However, if the outer table is a very small portion of the DBSPACE or if a highly selective predicate can be applied using an index, accessing the outer with that index can be far cheaper than a DBSPACE scan Our experiments dedicated a DBSPACE to each table The prediction powers of the optimizer improve as the tables get larger, as can be seen from Figure 9 for two 2500-tuple tables, but a few anomalies remain Why does adding an index sometimes *increase* the actual cost (as in Figure 8 and Figure 9), contrary to the optimizer's estimates? And why are the estimates so large for nested-loop join? The answer to the first question is that the optimizer models the scan of each table independently, ignoring the actual competition for buffer space among the concurrent scans of index and data pages for each table Taken alone, the outer table and its index appear to fit in the buffers, but interference from the inner table reduces the effective number of buffer pages available to the outer The answer to the second question is that the optimizer pessimistically models the worst case each outer tuple starts a completely new scan of the inner that cannot re-use any of the inner pages in the buffer if the estimated number of inner pages accessed per outer exceeds the size of the buffer In reality, there is a non-zero probability that some inner pages that remain in the buffer could be useful to the next outer tuple as well Index pages often need not be re-fetched, particularly when the join predicate is highly selective, and no re-fetches are needed when the buffer can contain both tables and indexes

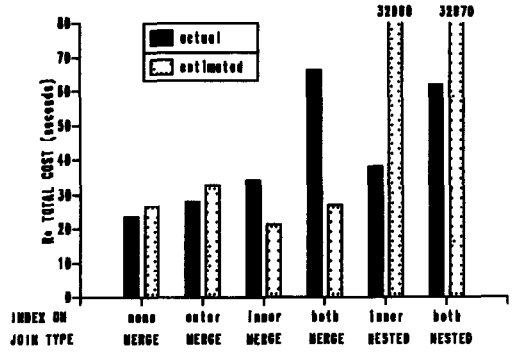

Figure 9 Actual vs. estimated cost for joining 2 tables of 2500 tuples each, for varying access methods & join types.

As the relative sizes of the two tables (arbitrarily named A and B) are varied, does the optimizer usually pick the best plan, or at least avoid bad plans? This was examined by fixing Table A's cardinality at 1000 tuples while varying Table B's cardinality, when each table was indexed on the join column or not As might be expected from the above discussion, the optimizer predicts best when there are no indexes on either table, as shown in Figure 10 In fact, the optimizer always picks the best plan in this case (Note that the best plan and R* plan curves overlap) The plan is shown in the figure using "*" to denote nested-loop join[3] and "+" for merge-scan join, and the first table as the outer-most For example, the R* and best plan in Figure 10 uses a merge-scan join, with the smaller table always as the outer table

Adding an index on Table A (see Figure 11) has counter-intuitive and counter-productive results practically the worst plan is chosen! ("(I)" after a table denotes that the plan uses the (unclustered) index on that table) This is again due to the overestimation of nested-loop join costs and the under-estimation of index competition for buffer space Adding an index to the table that is growing (Table B), however, is not nearly so bad, and may even result in *better* performance as the table grows (cf Figure 12 between 2500 and 3000 tuples)! Why? The answer lies in the trend in actual costs observed between Figure 8 and Figure 9, for the "inner indexed" case As the indexed table — likely to be the inner in any optimal plan — grows, at some point the actual cost of a

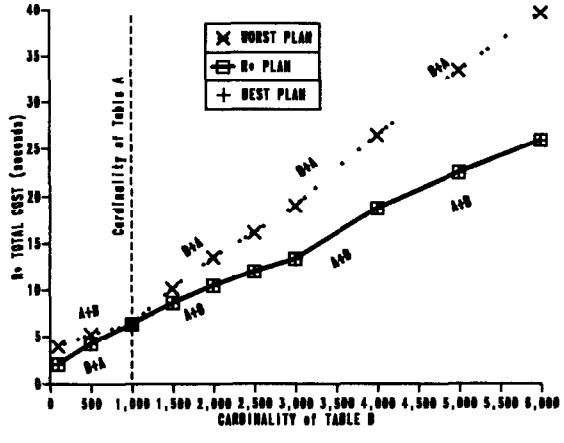

Figure 10 R* optimizer's plan vs. actual worst and best plan, for join of unindexed 1000-tuple table A and unindexed table B of increasing size (* ≡ Nested-loop join, + ≡ Merge-scan join)

3 Plans joining unindexed tables with the nested-loop join were omitted from Figure 10 through Figure 12 because the uniformly poor performance of those plans (whenever the smallest table exceeded 100 tuples) expanded the scale of those figures too much!

nested-loop join exceeds a merge-scan join, in agreement with the optimizer's relative estimates (compare the difference of the actuals for the two join types in the two figures) This is due to the decreased likelihood that the next outer tuple of the nested-loop join can benefit from inner pages remaining in the buffer We suspect — but were unable to confirm — that joining T tables with T-1 concurrent nested-loop joins would diminish this likelihood further, thus making the nested-loop join less attractive and the optimizer's estimates better In contrast, the merge-scan join cost goes up more slowly due to sorting and merging more tuples in B When both A and B are indexed, the results are similar to those of Figure 12

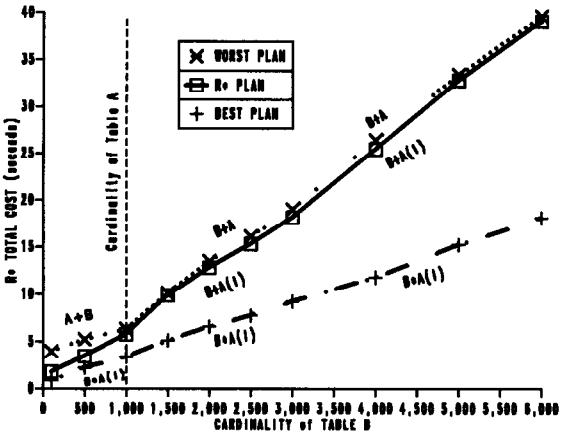

Figure 11 R* optimizer's plan vs. actual worst and best plan, for join of indexed 1000-tuple table A and unindexed table B of increasing size (* ≡ Nested-loop join, + ≡ Merge-scan join)

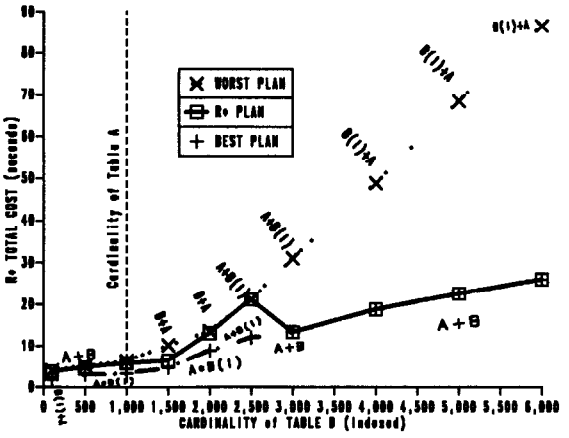

Figure 12 R* optimizer's plan vs. actual worst and best plan, joining unindexed 1000-tuple table A and indexed table B of increasing size (* ≡ Nested-loop join, + ≡ Merge-scan join)

5.3. Suggested Improvements

Clearly the modeling of nested-loop joins needs to be improved We suggest several solutions here First, the optimizer needs a more detailed model of how the index and data pages for each table interact with each other and those of other tables during concurrent scans, as in a join We have developed and are refining such a model that treats the scan of each table as two concurrent processes — one for the index (if any) and one for the data pages — competing for buffer space The share of the buffer that each process consumes is a function of (1) single-table access selectivities, (2) join selectivities, and (3) type of join This model should also be extendible to access methods other than indexes

Second, a more accurate estimate of join selectivity is needed As noted earlier and as shown graphically by Figure 13, the nested-loop join cost is far more sensitive to the ***join cardinality*** — the cardinality of the result of the join — than is the merge-scan join The System R and R* estimate, obtained by multiplying the product of the two tables' cardinalities by the smaller (more selective) estimate of the join predicate for the two tables [SELI 79], could be replaced (or at least biased) by statistics on actual join cardinality and the semijoin cardinalities for both the inner and outer tables, i e the number of tuples in the other table matching values in that table These statistics could be collected while performing the same join for an earlier SQL statement, and updated in the catalog only when significant differences from previous values are found, to avoid contention for catalog pages

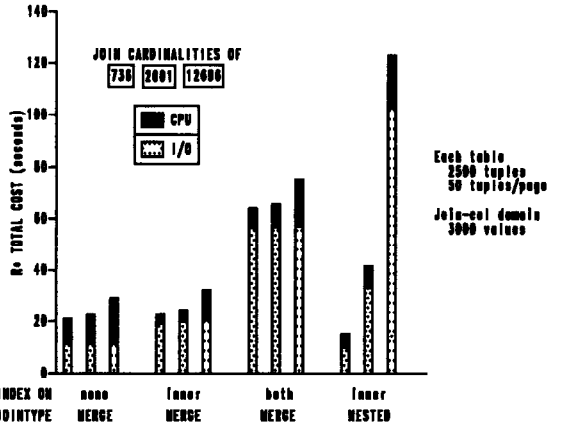

Figure 13 Impact of actual join cardinality on actual cost, for varying access methods & join types.

The performance advantage of having an index on the inner table suggests that we add to R* the ability to create temporary indexes on tables The additional cost of creating the index after sorting the inner is more than offset by not having to sort the outer table for a merge-scan join allowing temporary indexes will usually "tip the scales" in favor of a cheaper nested-loop join when the outer table is sufficiently small (see Figure 8) This was confirmed by tests which included the cost of a CREATE INDEX and a DROP INDEX on the inner table using the standard SQL commands, a process far more costly (because of catalog I/Os) than if temporary indexes were actually implemented Plans creating and using such a simulated "temporary" index nonetheless for this test series *always* performed better than the best "status quo" plan that did not permit such indexes, as shown in Figure 14

Another technique for improving join performance is using a Bloom filter [BLOO 70] as a "hashed semijoin" The Bloom filter is built anew at run-time for each join, and costs one scan each of the outer and inner tables plus the cost to store the filtered inner table in a temporary table A bit in a vector of M bits, initially all 0, is set to 1 if a join-column value from the outer table in a join hashes to that bit The join-column value for each tuple in the inner table is then hashed using the same hash function If the bit hashed to is 1, it is *likely* that this is due to a matching value in the outer table, so the inner tuple is copied to the temporary table, which then replaces the inner in the join The bit hashed to *may* be 1 due to a ***hash collision***, in which case the inner tuple will unnecessarily be copied during filtering to the temporary table

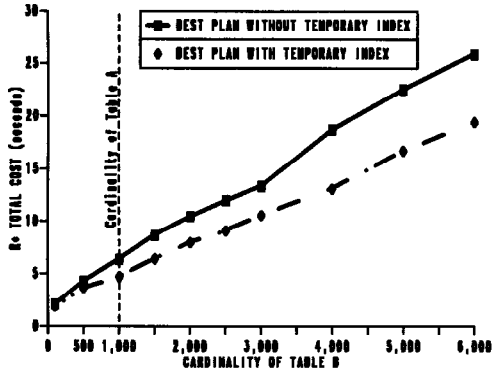

Figure 14 Local join performance improvement using a (simulated) temporary index on table A

but later will be excluded during regular join processing Such coincidental hashings of multiple values to the same bit can be minimized by hashing each value using multiple hash functions, and by judiciously choosing M and the number of hash functions [SEVE 76] Though not quite as selective (due to collisions) as a semijoin, a Bloom filter significantly reduces the size of relations to be sorted for a merge-scan join or to be re-scanned for a nested-loop join, and the bit vector requires less memory than the vector of join-column values used by a semi-join The potential performance improvement in R* was simulated and is shown in Figure 15 Assuming a single hash function and a Bloom filter of only one page (M=32K bits), our test tables of up to 6K tuples would have a probability of collision of less than 0 06, and tables would have to exceed 54K tuples to exceed our conservative collision probability of 0 30 We are also investigating the potential performance benefits of hash joins [DEWI 84, BRAT 84, DEWI 85] as an alternative to sorting and merge-scan joining unindexed tables in a new, experimental database system that we are implementing for the local area network environment

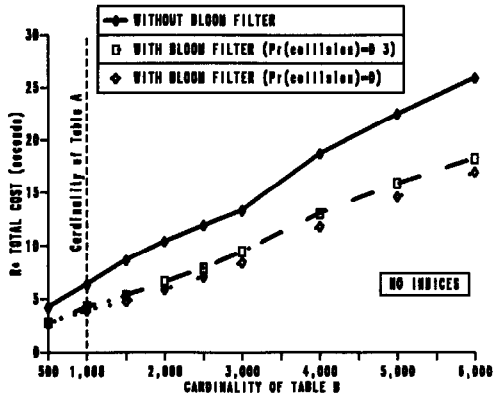

Figure 15 Local join performance improvement using a Bloom filter having Pr{collision} = 0 3 and ≈ 0

6. Conclusions

Our testing substantiated the contribution of CPU cost to total cost in virtually all queries, and the need to model CPU in more detail than was done in System R In some cases, CPU cost significantly affected the choice of plan (see below) While retaining the cost components separately consumes marginally more memory during optimization, it enables the user to determine if his query is I/O- or CPU-bound, and greatly facilitated validation

Both index and DBSPACE scans would benefit from pre-fetching pages and a page replacement strategy tailored to the type of scan This would require the optimizer to give the buffer manager more context with each page request and/or some "hints" on when a page may be discarded

When the inner table has no index on the join column or exceeds the size of the buffer, the nested-loop join is rarely optimal, and the optimizer correctly avoids the nested-loop join The merge-scan join is, therefore, an essential part of the optimizer's repertoire, confirming the results of Blasgen and Eswaran [BLAS 77] The nested-loop join method is preferred to merge-scan join, however, when the inner table fits in the buffer and the inner's join column is both indexed and reasonably selective This is in agreement with the results of Carey and Lu [CARE 85] Hence the ability to dynamically create temporary indexes could save much sorting of composite results necessitated by merge-scan joins

Although the optimizer's estimates are good when all tables fit in the buffer or all tables exceed the buffer, in the sensitive and hard-to-model region inbetween the optimizer overestimates the cost of the nested-loop join when indexes are involved, and therefore "switches" too soon from nested-loop to merge-scan join A better model is needed of nested-loop join performance, which is very sensitive to three parameters that are extremely difficult to estimate *a priori* (1) the join cardinality (size of the result of the join), (2) the outer table's cardinality, and (3) the buffer utilization The first parameter controls how many data pages of the inner table need be accessed per outer tuple Its estimate currently depends upon several questionable assumptions such as values being distributed uniformly and independent of those in other tables, maintaining statistics on actual join cardinality would significantly improve its estimation The second parameter determines how many times the inner must be scanned When the outer table is the result of a previous join, its cardinality is now estimated using the first parameter, so estimation errors can propagate Insufficient buffer space to contain the referenced portions of the inner table has a quantum effect on nested-loop join performance Yet it is impossible at optimization time to predict the potential buffer contention at run-time from other users *Within a given query*, however, it should be possible to develop a more general model for the buffer contention among the concurrent scans of index and data pages of tables being "pipeline" joined The authors are currently developing such a model

Regardless of the join method, the utility of using an index to scan the outer table depends upon (1) the proportion of the DBSPACE occupied by the table, (2) whether the index can be used to apply a predicate that limits the scan, and (3) the degree to which the index pages compete for buffer space with its data pages and with pages from scans of other tables Although the optimizer currently models the first two factors correctly, again an improved model of buffer utilization is needed for the third

All other things being equal (i e , if both tables or neither table is indexed on the join column), the optimizer correctly chooses the smaller table to be the outer table of either join type, while the I/O cost should be (and is) symmetric, the CPU cost for opening and closing scans on the inner table turns out to be the tie-breaker!

When only one of the tables is indexed, the primary criterion for choosing the inner table is which table is indexed

Does the prospect of increasingly inexpensive main memory make optimization unnecessary, or at least eliminate the problems associated with buffer limitations? We think not Competition for that memory by other users and other software will ensure scarcity of buffer space — and the need to model it carefully — for the foreseeable future

Do the trends found by our study suggest abandoning increasingly complex optimizer models in favor of a simple heuristic, e g one that orders tables by increasing size and joins them all by a nested-loop join using existing or temporary indexes? Definitely not We have eliminated extreme cases in the interest of clearer exposition, but in practice the optimizer cannot enjoy the luxury of catering to "typical" cases only If anything, this study *increased* our commitment to modeling the various plans in sufficient detail to predict performance correctly, rather than assuming away complexities with simplifying heuristics Every time we tried to devise a general rule, it was easy to construct a counter-example Ultimately, optimization is judged by users more by how well it avoids bad plans *in all situations* than how well it chooses "the optimal" plan in *most* situations This study significantly improved the capability of our optimizer technology to pick the best plan in all situations

There remain many open questions which time did not allow us to pursue We did not test joins for very large tables (e g , 100,000 tuples), for more than 2 tables, for varying buffer sizes, or for varying tables per DBSPACE Our investigations thus far lead us to suspect that the crucial parameter is the ratio of the table size(s) to the buffer size Selected tests comparing 10,000-tuple joins (5 times the buffer size) with 6,000-tuple joins (3 times the buffer size) had identical plans whose performance scaled predictably, so we limited our test tables to 6,000 tuples Once the shortcomings in the current optimizer model for two-table joins are corrected, we feel confident — but wish to verify thoroughly — that the optimizer will choose the correct order of join for more than two tables Also, we would like to measure the impact on response time and overall throughput of the DBMIN buffer management scheme of Chou and DeWitt [CHOU 85]

7. Acknowledgements

We wish to acknowledge the contributions to this work by several colleagues, especially the R* research team, and Lo Hsieh and his group at IBM's Santa Teresa Laboratory We particularly benefitted from lengthy discussions with — and suggestions by — Bruce Lindsay George Lapis helped with database generation and implemented the improved "mailbox" application program interface and an R* interface to GDDM that enabled us to graph performance results quickly and elegantly Paul Wilms contributed some PL/I programs that aided our testing, and assisted in the implementation of the COLLECT COUNTERS and EXPLAIN statements Luis Cabrera, Christoph Freytag, Laura Haas, John McPherson, Pat Selinger, and Irv Traiger constructively critiqued an earlier draft of this paper, improving its readability significantly We are indebted to the anonymous referees for providing constructive critique and additional references Brad Wade developed the instruction trace tool that proved invaluable for measuring instruction counts Finally, Tzu-Fang Chang and Alice Kay provided considerable systems support and patience while our tests consumed vast amounts of their resources

Bibliography

[APER 83] P M G Apers, A R Hevner, and S B Yao, Optimizing Algorithms for Distributed Queries, IEEE Trans on Software Engineering **SE-9** (January 1983) pp 57-68

[ASTR 80] M M Astrahan, M Schkolnick, and W Kim, Performance of the System R Access Path Selection Mechanism, *Information Processing* **80** (1980) pp 487-491

[BERN 81] P A Bernstein, N Goodman, E Wong, C L Reeve, J Rothnie, Query Processing in a System for Distributed Databases (SDD-1), *ACM Trans on Database Systems* **6,4** (December 1981) pp 602-625

[BLAS 77] M W Blasgen and K P Eswaren, Storage and Access in Relational Data Bases, IBM Systems Journal **16,4** (1977) Also available as "On the Evaluation of Queries in a Relational Data Base System", IBM Research Report RJ1745, San Jose, CA, April 1976

[BLOO 70] B H Bloom, Space/Time Trade-offs in Hash Coding with Allowable Errors, *Communications of the ACM* **13,7** (July 1970) pp 422-426

[BRAT 84] K Bratbergsengen, Hashing Methods and Relational Algebra Operations, *Procs of the Tenth International Conf on Very Large Data Bases* (Singapore, 1984) pp 323-333

[CARE 85] M J Carey and H Lu, Some Experimental Results on Distributed Join Algorithms in a Local Network, *Procs of the Eleventh International Conf on Very Large Data Bases* (Stockholm, Sweden, August 1985)

[CHAM 81] D D Chamberlin, M M Astrahan, W F King, R A Lorie, J W Mehl, T G Price, M Schkolnick, P Griffiths Selinger, D R Slutz, B W Wade, and R A Yost, Support for Repetitive Transactions and Ad Hoc Queries in System R, *ACM Trans on Database Systems* **6,1** (March 1981) pp 70-94

[CHAN 82] J-M Chang, A Heuristic Approach to Distributed Query Processing, *Procs of the Eighth International Conf on Very Large Data Bases* (Mexico City, VLDB Endowment, Saratoga, pp 54-61

[CHOU 85] H-T Chou and D J DeWitt, An Evaluation of Buffer Management Strategies for Relational Database Systems, *Procs of the Eleventh International Conf on Very Large Data Bases* (Stockholm, Sweden, September 1985) pp 127-141

[CHU 82] W W Chu and P Hurley, Optimal Query Processing for Distributed Database Systems, *IEEE Trans on Computers* **C-31** (September 1982) pp 835-850

[DANI 82] D Daniels, P G Selinger, L M Haas, B G Lindsay, C Mohan, A Walker, and P Wilms, An Introduction to Distributed Query Compilation in R*, *Procs Second International Conf on Distributed Databases* (Berlin, September 1982) Also available as IBM Research Report RJ3497, San Jose, CA, June 1982

[DEWI 84] D J DeWitt, R H Katz, F Olken, L D Shapiro, M R Stonebraker, and D Wood, Implementation Techniques for Main Memory Database Systems, *Proc SIGMOD 84* (Boston, MA, May 1984) pp 1-8

[DEWI 85] D J DeWitt and R Gerber, Multi-Processor Hash-Based Join Algorithms, *Procs of the Eleventh International Conf on Very Large Data Bases* (Stockholm, Sweden, September 1985) pp 151-164

[EFFE 84] Effelsberg, W and Haerder, T , Principles of Database Buffer Management, *ACM Trans Database Systems* **9** (December 1984) pp 560-595

[EPST 78] R Epstein, M Stonebraker, and E Wong, Distributed Query Processing in a Relational Data Base System, *Procs of ACM-SIGMOD* (Austin,TX, May 1978) pp 169-180

[EPST 80] R Epstein and M Stonebraker, Analysis of Distributed Data Base Processing Strategies, *Procs of the Sixth International Conf on Very Large Data Bases* (Montreal,IEEE, October 1980) pp 92-101

[FINK 82] S Finkelstein, M Schkolnick, and P Tiberio, DBDSGN — a Physical Database Design Tool for System R, *IEEE Database Engineering* **5,1** (1982) pp 228-235

[GDDM 84] *Graphical Data Display Manager Presentation Graphics Feature Interactive Chart Utility User's Guide, Release 4.* IBM Reference Manual SC33-0111-3 (IBM Corp , October 1984)

[HEVN 79] A R Hevner and S B Yao, Query Processing in Distributed Database Systems, *IEEE Trans Software Engineering* **SE-5** (May 1979) pp 177-187

[KERS 82] L Kerschberg, P D Ting, and S B Yao, Query Optimization in Star Computer Networks, *ACM Trans on Database Systems* **7,4** (December 1982) pp 678-711

[LOHM 84] G M Lohman, D Daniels, L M Haas, R Kistler, P G Selinger, Optimization of Nested Queries in a Distributed Relational Database, *Procs of the Tenth International Conf on Very Large Data Bases* (Singapore, 1984) pp 403-415 Also available as IBM Research Report RJ4260, San Jose, CA, April 1984

[LOHM 85] G M Lohman, C Mohan, L M Haas, B G Lindsay, P G Selinger, P F Wilms, and D Daniels, Query Processing in R*, *Query Processing in Database Systems, Springer-Verlag* (Kim, Batory, & Reiner (eds), 1985) pp 31-47 Also available as IBM Research Report RJ4272, San Jose, CA, April 1984

[MACK 85] L F Mackert and G M Lohman, Index Scans using a Finite LRU Buffer A Validated I/O Model, IBM Research Report RJ4836 (Almaden Research Center, San Jose, CA,

[MACK 86] L F Mackert and G M Lohman, R* Optimizer Validation and Performance Evaluation for Distributed Queries, IBM Research Report RJ5050 (Almaden Research Center, San Jose, CA,

[ONUE 83] E Onuegbe, S Rahimi, and A R Hevner, Local Query Translation and Optimization in a Distributed System, *Procs NCC 1983* (July 1983) pp 229-239

[PALE 74] F P Palermo, A Data Base Search Problem, *Information Systems COINS IV (Procs of COINS-72 Symp)* (Miami, FL, 1974) pp 67-101 Plenum Press, Julius T Tou, ed

[RDT 84] *RDT Relational Design Tool,* IBM Reference Manual SH20-6415 (IBM Corp , June 1984)

[SACC 82] G M Sacco and M Schkolnick, A Mechanism for Managing the Buffer Pool in a Relational Database System using the Hot Set Model, *Procs of the Eighth International Conf on Very Large Data Bases* (Mexico City, 1982) pp 257-262

[SCHK 79] M Schkolnick and P Tiberio, Considerations in Developing a Design Tool for a Relational DBMS, *Proc IEEE COMPSAC Conf* (Chicago, November 1979) pp 228-235

[SCHK 85] M Schkolnick and P Tiberio, Estimating the Cost of Updates in a Relational Database, *ACM Trans on Database Systems* **10,2** (June 1985) pp 163-179 Also available as IBM Research Report RJ3327, San Jose, CA, 1981

[SELI 79] 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, *Procs of ACM-SIGMOD* **20** (1979) pp 23-34

[SELI 80] P G Selinger and M Adiba, Access Path Selection in Distributed Database Management Systems, *Procs International Conf on Data Bases* (Deen and Hammersly, ed , Univ of Aber pp 204-215

[SEVE 76] D G Severance and G M Lohman, Differential Files Their Application to the Maintenance of Large Databases, *ACM Trans on Database Systems* **1,3** (September 1976) pp 256-267

[SQL 84] *SQL/Data System Application Programming for VM/ System Product, Release 3,* IBM Reference Manual SH24-5068-0 (IBM Corp , December 1984)

[STON 80] M Stonebraker, Retrospection on a Data Base System, *ACM Trans on Database Systems* **5,2** (June 1980) pp 225-240

[STON 81] M Stonebraker, Operating System Support for Database Management, *Communications of the ACM* **24** (July 1981) pp 412-418

[STON 82] M Stonebraker, J Woodfill, J Ranstrom, M Murphy, J Kalash, M Carey, K Arnold, Performance Analysis of Distributed Data Base Systems, *Database Engineering* **5** (IEEE Computer Society, December 1982) pp 58-65

[VTAM 85] *Network Program Products Planning (MVS, VSE, and VM),* IBM Reference Manual SC23-0110-1 (IBM Corp , April 1985)

[WONG 76] E Wong and K Youssefi, Decomposition — a Strategy for Query Processing, *ACM Trans on Database Systems* **1,3** (September 1976) pp 223-241

[YAO 79] S B Yao, Optimization of Query Algorithms, *ACM Trans on Database Systems* **4,2** (June 1979) pp 133-155

[YU 83] C T Yu, and C C Chang, On the Design of a Query Processing Strategy in a Distributed Database Environment, *Proc SIGMOD 83* (San Jose, CA, May 1983) pp 30-39