R* Optimizer Validation and Performance Evaluation for Distributed Queries

Lothar F. Mackert [1]
Guy M. Lohman

IBM Almaden Research Center
K55-801, 650 Harry Road, San Jose, CA 95120-6099

Abstract

Few database query optimizer models have been validated against actual performance. This paper extends an earlier optimizer validation and performance evaluation of R* to *distributed* queries, i.e. single SQL statements having tables at multiple sites. Actual R* message, I/O, and CPU resources consumed — and the corresponding costs estimated by the optimizer — were written to database tables using new SQL commands, permitting automated control from application programs for collecting, reducing, and comparing test data. A number of tests were run over a wide variety of dynamically-created test databases, SQL queries, and system parameters. Both high-speed networks (comparable to a local area network) and medium-speed long-haul networks (for linking geographically dispersed hosts) were evaluated. The tests confirmed the accuracy of R*'s message cost model and the significant contribution of local (CPU and I/O) costs, even for a medium-speed network. Although distributed queries consume more resources overall, the response time for some execution strategies improves disproportionately by exploiting both concurrency and reduced contention for buffers. For distributed joins in which a copy of the inner table must be transferred to the join site, shipping the whole inner table dominated the strategy of fetching only those inner tuples that matched each outer-table value, even though the former strategy may require additional I/O. Bloomjoins (hashed semijoins) consistently performed better than semijoins and the best R* strategies.

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. The goal of our study was to investigate the performance and to thoroughly validate the optimizer against actual performance 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]. This paper extends our earlier validation and performance evaluation of local queries [MACK 86] to distributed queries over either (1) a high-speed network having speeds comparable to a local-area network (LAN) or (2) over a medium-speed, long-haul network linking geographically dispersed host machines. For brevity, we assume that the reader is familiar with System R [CHAM 81] and R* [LOHM 85], and with the issues, methodology, and results of that earlier study [MACK 86].

Few of the distributed optimizer models proposed over the last decade [APER 83, BERN 81B, CHAN 82, CHU 82, EPST 78, HEVN 79, KERS 82, ONUE 83, PERR 84, WONG 83, YAO 79, YU 83] have been validated by comparison with actual performance. The only known validations, for Distributed INGRES [STON 82] and the Crystal multicomputer [LU 85], have assumed only a high-speed local-area network linking the distributed systems. Also, the Distributed INGRES study focused primarily on reducing response time by exploiting parallelism using table partitioning and broadcast messages. In contrast, R* seeks to minimize total resources consumed, has not implemented table partitioning[2], and does not presume a network broadcast capability.

There are many important questions that a thorough validation should answer:

- Under what circumstances (regions of the parameter space) does the optimizer choose a suboptimal 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 distributed join techniques?
- Are there alternative distributed join techniques that are not implemented but look promising?

The next section gives an overview of distributed compilation and optimization in R*. Section 3 discusses how R* was instrumented to collect optimizer estimates and actual performance data at multiple sites in an automated way. Section 4 presents some prerequisite measurements of the cost component weights and the measurement overhead. The results for distributed joins are given in Section 5, and suggestions for improving their performance are discussed in Section 6. Section 7 contains our conclusions.

2. Distributed Compilation and Optimization

The unit of distribution in R* is a table and each table is stored at one and only one site. A ***distributed query*** is any SQL data manipulation statement that references tables at sites other than the ***query site***, the site to which an application program is submitted for compilation. This site serves as the ***master site*** which coordinates the optimization of all SQL statements embedded in that program. For each query, sites other than the master site that store a table referenced in the query are called ***apprentice sites***.

In addition to the parameters chosen for the local case:

---

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

2 Published ideas for horizontal and vertical partitioning of tables have not been implemented in R*.

(1) the order in which tables must be joined
(2) the join method (nested-loop or merge-scan), and
(3) the access path for each table (e.g., whether to use an index or not)

optimization of a *distributed* query must also choose for each join[3]:

(4) ***the join site***, i.e. the site at which each join takes place, and,
(5) if the inner table is not stored at the join site chosen in (4), the method for transferring a copy of the inner table to the join site:
(5a) ***ship whole***: ship a copy of the entire table once to the join site, and store it there in a temporary table; or
(5b) ***fetch matches*** (see Figure 1): scan the outer table and sequentially execute the following procedure for each outer tuple:
1. Project the outer table tuple to the join column(s) and ship this value to the site of the inner table.
2. Find those tuples in the inner table that match the value sent and project them to the columns needed.
3. Ship a copy of the projected matching inner tuples back to the join site.
4. Join the matches to the outer table tuple.

Note that this strategy could be characterized as a semijoin for each outer tuple. We will compare it to semijoins in Section 6.

If a copy of an outer (possibly composite) table of a join has to be moved to another site, it is always shipped in its entirety as a blocked pipeline of tuples [LOHM 85].

Compilation, and hence optimization, is truly distributed in R*. The master's optimizer makes all *inter-site* decisions, such as the site at which inter-site joins take place, the method and order for transferring tuples between sites, etc. *Intra-site* decisions (e.g. order and method of join for tables contiguously within a single site) are only *suggested* by the master planner; it delegates to each apprentice the final decision on these choices as well as the generation of an access module to encode the work to be done at that site [DANI 82].

Optimization in R* seeks to minimize a cost function that is a linear combination of four components: CPU, I/O, and two message costs: the number of messages and the total number of bytes transmitted in all messages. I/O cost is measured in number of transfers to or from disk, and CPU cost is measured in terms of number of instructions:

$$\text{R*\_\_total\_\_cost} = W_{CPU} * (\#\_\_\text{instrs}) + W_{I/O} * (\#\_\_\text{I/Os}) + W_{MSG} * (\#\_\_\text{msgs}) + W_{BYT} * (\#\_\_\text{bytes})$$

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 cost 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]. For our study we set these weights so that the R* total cost estimates the total time consumed by all resources, in milliseconds. Since all the sites in our tests had equivalent hardware and software configurations, identical weights were used for each site.

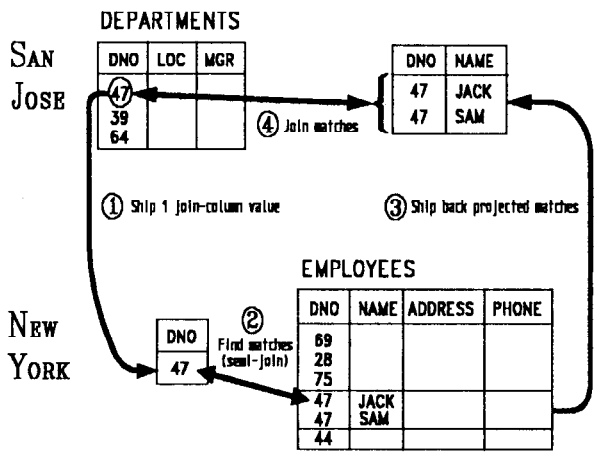

Figure 1: "Fetch-matches" transfer strategy for joining at site San Jose outer table DEPARTMENTS to inner table EMPLOYEES.

3. Instrumentation

An earlier performance study for System R [ASTR 80] demonstrated that extracting performance data using the standard database trace and debugging facilities required substantial manual interaction, severely limiting the number of test cases that could be run. Since we wanted to measure performance under a wide variety of circumstances, we added instrumentation that would automate measurements to a very high degree. The general design of this instrumentation and its application for the evaluation of local queries is described in [MACK 86], so that in this paper we recall only the main ideas and confine our discussion to its distributed aspects. Principals of our design were:

1. Add to the SQL language three statements for test control and performance monitoring which can be executed from an application program as well as interactively.
2. Develop pre-compiled application programs for automatically (a) testing queries using the SQL statements of (1) above, and (b) analyzing the data collected by step (a).
3. Store the output of the SQL statements of (1) and the application programs of (2) in database tables in order to establish a flexible, powerful interface between (1), (2a), and (2b).

We concentrate here on the first item — the SQL-level measurement tools — whose implementation was most complicated by the distribution of tables at different sites.

3.1. Distributed EXPLAIN

The EXPLAIN command writes to user-owned PLAN__TABLEs information describing the access plan chosen by the optimizer for a given SQL statement, and its estimated cost [RDT 84]. For a given distributed query, no single site has the complete access plan: the master site has the inter-site decisions and each apprentice has its local intra-site decisions. Hence the R* EXPLAIN command was augmented to store each apprentice site's plan in a local PLAN__TABLE, and the test application program was altered to retrieve that information from each apprentice's PLAN__TABLE.

3.2. Distributed COLLECT COUNTERS

This new SQL statement collects and stores in a user-owned table the current values of some 40 internal counters in the RSS* component (e.g., counts of disk reads and writes, lookups in the buffer, etc.), which R* inherited from System R, and some newly implemented counters of the communications component DC*. COLLECT COUNTERS automatically collects a (pre-defined) subset of these counters at all sites with which the user currently has open communication sessions, returns those counters to the master site, and inserts into a special user-owned table (COUNTER__TABLE) one tuple for each distinct counter at each site. Each counter value is tagged with its name, the component (RSS* or DC*) and site that maintains the counter, a timestamp, the invoking application program name, and an optional user-supplied sequence number.

The implementation of the COLLECT COUNTERS statement is dependent upon the mechanism for distributed query execution in R* [LIND 83]. The master site establishes communication sessions with all sites with which it has to have direct communication, and spawns children processes at these sites. The children may in turn establish additional sessions and spawn other children processes, creating a tree of processes that may endure through multiple transactions in an application program. Since descendant processes may spawn processes at any site, the tree may contain multiple descendant processes at a single site on behalf of the same master process (***loopback***). For collecting the counters from all sites that are involved in the current computation, we traverse the user's process tree. For each process, counters are collected at that process' site and are

3 The site at which any nested query (***subquery***) is applied must also be determined [LOHM 84], but consideration of subqueries is omitted from this paper to simplify the presentation.

returned to the master site. At the master site, each counter value is handled in the following way:

- If we have not yet inserted a tuple into the COUNTER_TABLE for the given counter from the given site (while executing the COLLECT COUNTERS statement of interest), the counter is inserted into the COUNTER_TABLE.
- RSS* counters from the given site that have already been inserted into the user's COUNTER_TABLE are discarded (loopbacks will cause redundant delivery of certain counters), because RSS* counters are database-site-specific.
- DC* counters are process-specific. If there is already a row in the COUNTER_TABLE for the given DC* counter at the given site, the counter value is added to the counter value in that row.

To be sure that sessions had been established with all sites relevant to a particular test, the test application program was altered to run the test sequence once before the first COLLECT COUNTERS statement.

3.3. FORCE OPTIMIZER

As in the local validation study, we had to be able to overrule the optimizer's choice of plan, to measure the performance of plans that the optimizer thought were suboptimal. This was done with the FORCE OPTIMIZER statement, which was implemented in a special test version of R* only. The FORCE OPTIMIZER statement chooses the plan for the *next* SQL data manipulation (optimizable) statement *only*. The user specifies the desired plan number, a unique positive integer assigned by the master site's optimizer to each candidate plan, by first using the EXPLAIN statement (discussed above) to discover the number of the desired plan. Apprentice optimization can be forced by simply telling each apprentice to utilize the optimization decisions recommended by the master's optimizer in its global plan.

3.4. Conduct of Experiments

Our distributed query tests were conducted in the same way and in the same environment as the local query tests [MACK 86], only with multiple database sites. All measurements were run at night on two totally unloaded IBM 4381's connected via a high-speed channel. Each site was initialized to provide 40 buffer pages of 4K bytes each, which were available exclusively to our test applications. This is approximately equivalent, for example, to a system with each site running 5 simultaneous transactions that are competing for 800K bytes of buffer space. The same effects of buffer size limitations that were investigated in [MACK 86] also apply to distributed queries, and thus are not discussed further in this paper. In order to vary database parameters systematically, synthetic test tables were generated dynamically, inserting tuples whose column values were drawn randomly from separate uniform distributions. For example, the join-columns' values were drawn randomly from a domain of 3000 integer values when generating the tables. All tables had the same schema: four integer and five (fixed) character fields. The tuples were 66 bytes long, and the system stored 50 of them on one page.

Each test was run several times to ensure reproduceability of the results, and to reduce the variance of the average response times. 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.

What follows is a sample of our results illustrating major trends for distributed queries; space considerations preclude showing all combinations of all parameters that we examined. 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 projection factor on the joined tables (50% or 100% of both tables) and the availability of totally unclustered indexes on the join columns of the outer and/or inner tables. Since unclustered index scans become very expensive when the buffer is not big enough to hold all the data and index pages of a table, the ratio between the total number of data and index pages of a table to the number of pages in the buffer is more important for the local processing cost than the absolute table size [MACK 85]. 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.

4. General Measurements

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

4.1. Cost of Measurements

The COLLECT COUNTERS statement, the means by which we measured performance, itself consumes system resources that are tabulated by the R* internal counters. For example, collecting the counters from remote sites itself uses messages whose cost would be reflected in the counters for number of messages and number of bytes transmitted. The resources consumed by the COLLECT COUNTERS instrumentation was determined by running two COLLECT COUNTERS statements with no SQL statements in between, and reducing all other observations by those resources.

4.2. 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. The per-message weight can be estimated by dividing the approximate number of instructions to initiate and receive a message by the MIP rate. And the per-byte weight estimate is simply the time to send 8 bits at the *effective* transmission speed of the network, which had been measured as 4M bits/sec for our nominally 24M bit/sec (3M Byte/sec) channel-to-channel connection. These estimates, and the corresponding actual weights for our test configuration, are shown in Figure 2.

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

WEIGHT	UNITS	HARDWARE/SOFTWARE	ESTIMATE	ACTUAL
W_{CPU}	msec/inst.	IBM 4381 CPU	0.0004	0.0004
$W_{I/O}$	msec/I/O	IBM 3380 disk	23.48	17.00[4]
W_{MSG}	msec/msg.	CICS/VTAM	11.54	16.5
W_{BYTE}	msec/byte	24Mbit/sec (nom.), 4Mbit/sec (eff.)	0.002	0.002

Figure 2: Estimated and actual cost component weights.

The actual per-message and per-byte weights were measured by moving to a remote site one table of a two-table query for which the executed plan and the local (I/O and CPU) costs were well known. We chose a query that nested-loop joined a 500-tuple outer table, A, and a 100-tuple inner table, B, having an index on the join column. The plan for the distributed execution of this query had to be one that was executed sequentially (i.e., with no parallelism between sites), so that the response time (which we could measure) equalled the total resource time. By SELECTing all the columns of B, we could require that the large (3500-byte) tuples of B had to be shipped without projection, thereby ensuring that both the number (1000) and size of messages sent was high and that the local processing time was a small part (less than 30%) of the total resource time. We could control the message traffic by varying the number of tuples in B matching values in A: when none matched, only very small messages were transferred (carrying fixed-size R* control information); when each tuple in A matched exactly one tuple in B, 500 small and 500 very large messages were transferred. For a given number

4 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.

of matching inner tuples, the query was run 10 times to get the average response (= total resource) time. The message cost was derived by subtracting from the total time the local cost, which was measured by averaging the cost of 10 executions of the same query when both A and B were at the *same* site. Knowing the number and size of the messages (using COLLECT COUNTERS) for that number of matching inner tuples allowed us to compute the per-message and per-byte weights for our test environment: 16.5 msecs. minimal transfer time, and an effective transfer rate of 4M bit/sec. Note that these figures include the instruction and envelope overheads, respectively, of R*, CICS, and VTAM [LIND 83, VTAM 85].

By varying the above per-message and per-byte weights, we could also use the observed number of messages and bytes transmitted on the high-speed channel-to-channel connection to simulate the performance for a medium-speed long-haul network linking geographically dispersed hosts: 50 msecs. minimum transfer time and effective transfer rate of 40K bit/sec (nominal rate of 56K bit/sec, less 30% overhead). The per-message weight differs because of the increased delay due to the speed of light for longer transmissions, routing through relays, etc. Unavailability of resources at remote sites unfortunately precluded validating on a real long-haul network these estimated weights.

5. Distributed Join Results

Having validated the weights used in the R* cost function, and having removed the cost of measuring performance, we were ready to validate the R* optimizer's decisions for distributed queries.

The simplest distributed query accesses a single table at a remote site. However, since partitioning and replication of tables is not supported in R*, accessing a remote table is relatively simple: a process at the remote site accesses the table locally and ships the query result back to the query site as if it were an outer table to a join (i.e., as a blocked pipeline of tuples). Since all of the *distributed* optimization decisions discussed earlier pertain to *joins* of tables at different sites, picking the optimal global plan is solely a local matter: only the access path to the table need be chosen. For this reason, we will not consider single-table distributed queries further, but focus instead entirely upon distributed join methods.

In R*, n-table joins are executed as a sequence of n-1 two-table joins. Hence thorough understanding and correct modeling of distributed two-table joins is a prerequisite to validating n-table distributed joins. Intermediate results of joins are called ***composite*** tables, and may either be returned as a pipeline of tuples or else materialized completely before the succeeding two-table join (e.g., if sorting is required for a merge-scan join). We will therefore limit our discussion in this section to that fundamental operation, the two-table join.

Our discussion will use a simple notation for expressing distributed access plans for joins. There are two different join methods: merge scan joins, denoted by the infix operator "-M-", and nested loop joins, denoted by "-N-". The operand to the left of the join operator specifies the outer table access, the right operand the inner table access. A table access consists of the table name, optionally suffixed with an "I" if we use the index on the join column of this table and/or a "W" or"F" if we ship the table whole or fetch only matching tuples, respectively. For example, AIW-M-B denotes a plan that merge-scan joins tables A and B at B's site, shipping A whole after scanning it with the index on the join column. Since the merge-scan join requires both tables to be in join-column order, this plan implies B has to be sorted to accomplish the join.

5.1. Inner Table Transfer Strategy

The choice of transfer strategy for the inner table involves some interesting trade-offs. Shipping (a copy of) the table whole ("W") transfers the most inner tuples for the least message overhead, but needlessly sends inner tuples that have no matching outer tuples and necessitates additional I/O and CPU for reading the inner at its home site and then storing it in a temporary table at the join site. Any indexes on the inner that might aid a join cannot be shipped with the table, since indexes contain physical addresses that change when tuples are inserted in the temporary table, and R* does not permit dynamic creation of temporary indexes (we will re-visit that design decision in Section 6). However, since the inner is projected and any single-table predicates are applied before it is shipped, the temporary table is potentially much smaller than its permanent version, which might make multiple accesses to it (particularly in a nested-loop join) more cost-effective.

The high-speed channel we were using for communication in our tests imposed a relatively high per-message overhead, thereby emphatically favoring the "W" strategy. Figure 3 compares the actual performance of the best plan for each transfer strategy for both the high-speed channel and the long-haul medium-speed network, when merge-scan joining[5] two indexed 500-tuple tables, C and D, shipping the inner table D and returning the result to C's site. Both tables are projected to 50% of their tuple length, the join column domain has 100 different values, and the ***join cardinality*** — the cardinality of the result of the join — was 2477. If we ship the inner table D as a whole, the best plan is CI-M-DIW, and if we fetch the matching inner tuples ("F"), CI-M-DIF is best.

For the W strategy, the message costs are only 2.9% of the total resource cost, partly due to the relatively high local cost because of the large join cardinality. For the F strategy, we spend 80.9% of the costs for communications, since for each outer tuple we have to send one message containing the outer tuple's value and at least one message containing the matching inner tuples, if any. The total of 1000 messages cannot be reduced, even if there are no matching tuples, since the join site waits for some reply from the inner's site. Note that the number of bytes transmitted as well as the number of messages is much higher for the F strategy, because each message contains relatively little data in proportion to the required R* control information. Another source for the higher number of bytes transmitted is the frequent retransmission of inner table tuples for the large join cardinality of this query. The penalty for this overhead and the discrepancy between the two transfer strategies is exaggerated by slower network speeds. For the medium-speed network in Figure 3, the per-message overhead is 49% of the cost, and the discrepancy between the two strategies increases from a factor of 4.4 to a factor of 11.6.

The importance of per-message costs dictate two sufficient (but not necessary) conditions for the F strategy to be preferred:

1. the cardinality of the outer table must be less than half the number of messages required to ship the inner as a whole, and
2. the join cardinality must be less than the inner cardinality,

after any local (non-join) predicates have been applied and the referenced columns have been projected out. The second condition assures that fewer inner tuples are transferred to the outer's site for F than for W. Since the join cardinality is estimated as the product of the inner cardinality, outer cardinality, and join-predicate selectivity, these two conditions are

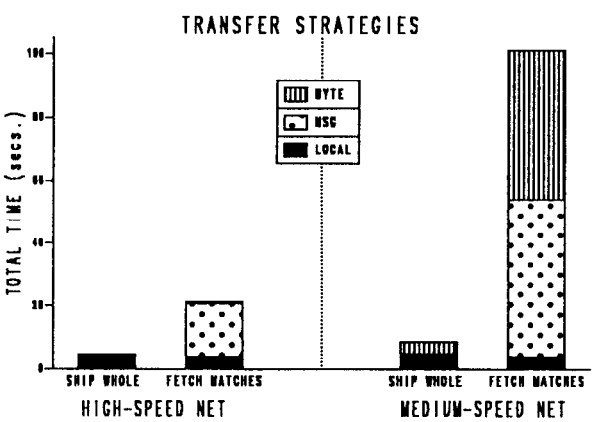

Figure 3: Comparison of the best R* plans, when using the ship-whole ("W") vs. the fetch-matches ("F") strategies for shipping the inner table, when merge-scan joining two indexed 500-tuple tables.

5 Nested loop joins perform very poorly for the "W" strategy, because we can not ship an index on the join column. For a fair comparison, we therefore only consider merge-scan joins.

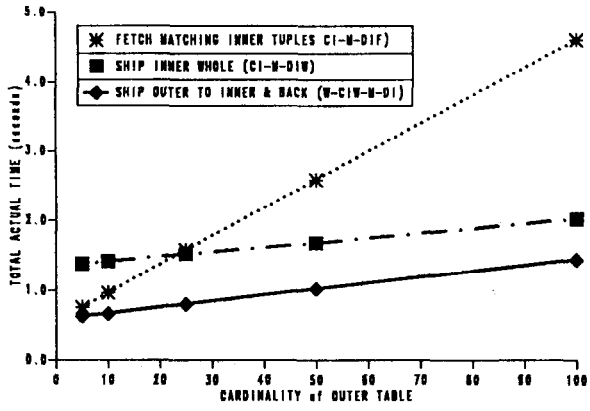

Figure 4: Shipping the outer table (C) to the inner's (D's) site and returning the result dominates both strategies for transferring the inner to the outer's site, even for small outer cardinalities (inner cardinality = 500 tuples).

equivalent to requiring that the outer cardinality be less than the minimum of (a) the inner's size (in bytes) divided by 8K bytes (the size of two messages) and (b) the inverse of the join-predicate's filter factor. Clearly these conditions are sufficiently strict that the F strategy will rarely be optimal.

Even when these conditions hold, it is likely that shipping the outer table to the inner's site and returning the result to the outer's site will be a better plan: by condition (1) the outer will be small, by condition (2) the result returned will be small, and performing the join at the inner's site permits the use of indexes on the inner. This observation is confirmed by Figure 4. The tests of Lu and Carey [LU 85] satisfied condition (2) by having a semijoin selectivity of 10% and condition (1) by cleverly altering the R* F strategy to send the outer-tuple values in one-page batches. Hence they concluded that the F strategy was preferred. Time constraints prevented us from implementing and testing this variation.

We feel that the conditions for the R* fetch-matches strategy to be preferred are so restrictive for both kinds of networks that its implementation without batching the outer-tuple values is not recommended for any future distributed database system. Therefore, henceforth we will consider only joins employing the ship-whole strategy.

5.2. Distributed vs. Local Join

Does distribution of tables improve or diminish performance of a particular query? In terms of total resources consumed, most distributed queries are more expensive than their single-site counterparts. Besides the obvious added communications cost, distributed queries also consume extra CPU processing to insert and retrieve the shipped tuples from communications buffers. In terms of response time, however, distributed queries may outperform equivalent local queries by bringing more resources to bear on a given query and by processing portions of that query in parallel on multiple processing units and I/O channels. Exploiting this parallelism is in fact a major justification for many distributed database systems [EPST 80, APER 83, WONG 83], especially multiprocessor database machines [BABB 79, DEWI 79, VALD 84, MENO 85].

The degree of simultaneity that can be achieved depends on the plan we are executing. Figure 5 compares the total resource time and the response time for some of the better R* access plans for a distributed query that joins two indexed (unclustered) 1000-tuple tables, A and B, at different sites, where the query site is A's site, the join column domain has 3000 different values, and each table is projected by 50%. For the plans shown, the ordering with respect to the total resource time is the same as the response time ordering, although this is not generally true. Plans shipping the outer table enjoy greater simultaneity because the join on the first buffer-full of outer tuples can proceed in parallel with the shipment of the next buffer-full. Plans shipping the inner table (whole) are more sequential: they must wait for the entire table to be received at the join site and inserted into a temporary table (incurring additional local cost) before proceeding with the join. For example, in Figure 5, note the difference between total resource time and response time for BIW-M-AI, as compared to the same difference for AI-M-BIW. Other plans not shown in Figure 5 that ship the inner table exhibit similar relationships to the corresponding plans that ship the outer (e.g., A-M-BW vs. BW-M-A, A-M-BIW vs. BIW-M-A, and AI-M-BW vs. BW-M-AI.). This assymmetry is unknown for local queries.

For merge joins not using indexes to achieve join-column order (e.g., A-M-BW, BW-M-A), R* sorts the two tables sequentially. Although sorting the two tables concurrently would not decrease the total resource time, it would lower the response time for those plans considerably (it should be close to the response time of BIW-M-A).

Comparing the response times for the above set of plans when the query is distributed vs. when it is local (see Figure 6), we notice that the distributed joins are faster. The dramatic differences between distributed and local for BIW-M-AI and AI-M-BIW stem from both simultaneity and the availability of two database buffers in the distributed case. However, by noting that for local joins the response time equals the resource time (since all systems were unloaded) and comparing these to the total resource times for the distributed query in Figure 5, we find that even the total resource costs for BIW-M-AI and AI-M-BIW are less than those for the local joins BI-M-AI and AI-M-BI, so parallelism alone cannot explain the improvement. The other reason is reduced contention: this particular plan is accessing both tables using unclustered indexes, which benefit greatly from larger buffers, and the distributed query enjoys twice as much buffer space as does the local query. However, not all distributed plans have

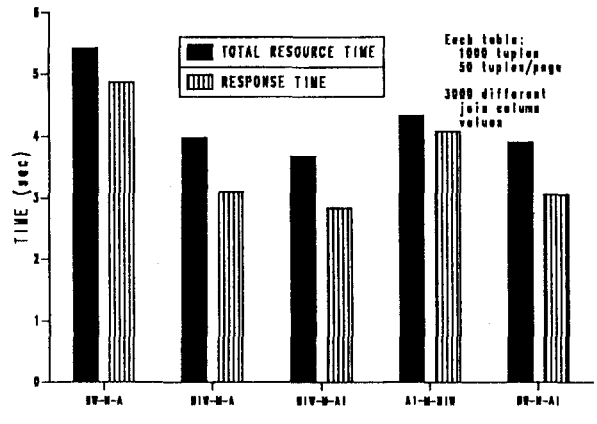

Figure 5: Resource consumption time vs. response time for various access plans, when joining 2 tables (1000 tuples each) distributed across a high-speed network.

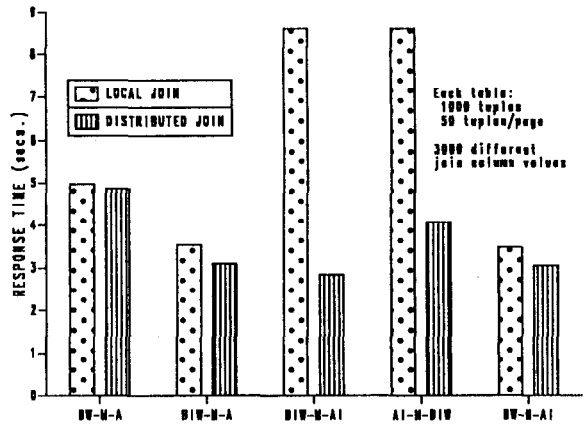

Figure 6: Response times for distributed (across a high-speed network) vs. local execution for various access plans, when joining 2 tables (1000 tuples each).

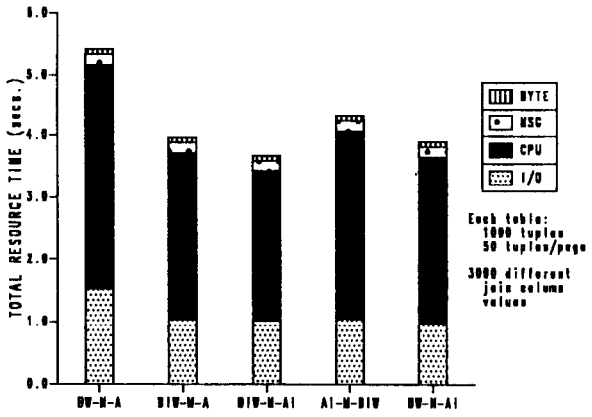

Figure 7: Relative importance of cost components for various access plans when joining 2 tables (of 1000 tuples each) distributed across a high-speed network.

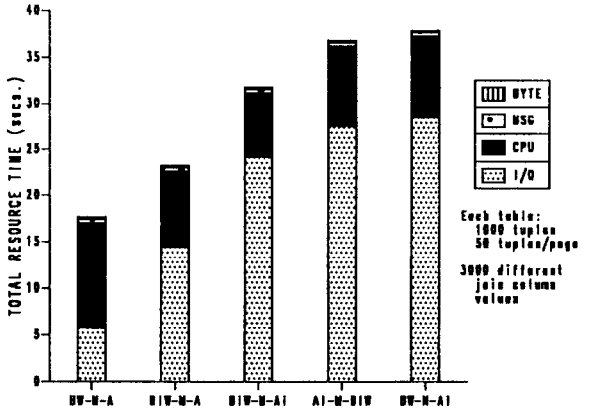

Figure 8: Relative importance of cost components for various access plans when joining 2 tables (of 2500 tuples each) distributed across a high-speed network.

better response times than the corresponding local plan; the increased buffer space doesn't much help the plans that don't access both tables using an index, and most of the distributed plans that ship the inner table to the join site (except for AI-M-BIW) are 15%-30% more expensive than their local counterpart because they exhibit a more sequential execution pattern.

For larger tables (e.g., 2500 tuples each), these effects are even more exaggerated by the greater demands they place upon the local processing resources of the two sites. However, for slower network speeds, the reverse is true; increased communications overhead results in response times for distributed plans being almost twice those of local plans. For a comparison of the resource times see Section 6.

5.3. Relative Importance of Cost Components

Many distributed query optimization algorithms proposed in the literature ignore the *intra*-site costs of CPU and I/O, arguing that those costs get dwarfed by the communication costs for the majority of queries. We have investigated the relative importance of the four cost components when joining two tables at different sites, varying the sizes of the tables and the speeds of the communication lines. Our results confirmed the analysis of Selinger and Adiba [SELI 80], which concluded that local processing costs are relevant and possibly even dominant in modelling the costs of distributed queries.

In a high-speed network such as a local-area network, message costs are of secondary importance, as shown by Figure 7 for the distributed join of two 1000-tuple tables. For our test configuration, message costs usually accounted for less (very often much less) than 10% of the total resource cost. This remained true for joins of larger tables, as shown in Figure 8 for two 2500-tuple tables. Similarly, message costs account for only 9% of the total cost for the optimal plan joining a 1000-tuple table to a 6000-tuple table, delivering the result to the site of the first table. This agrees with the measurements of Lu and Carey [LU 85].

When we altered the weights to simulate a medium-speed long-haul network, local processing costs were still significant, as shown in Figure 9 and Figure 10. In most of the plans, message costs and local processing costs were equally important, neither ever dropping under 30% of the total cost. Hence ignoring local costs might well result in a bad choice of the local parameters whose cost exceeds that of the messages. Also, the relative importance of per-message and per-byte costs reverses for the medium-speed network, because the time spent sending and receiving each message, and the "envelope" bytes appended to each message, are small compared to the much higher cost of getting the same information through a "narrower pipeline" than that of the high-speed network.

5.4. Optimizer Evaluation

How well does the R* optimizer model the costs added by distributed data? For the ship-whole table transfer strategy, for both outer and inner tables, our tests detected only minor differences (<2%) between actual costs and optimizer estimates of the number of messages and the number of bytes transmitted. The additional local cost for storing the inner table shipped whole is also correctly modelled by the optimizer, so that the

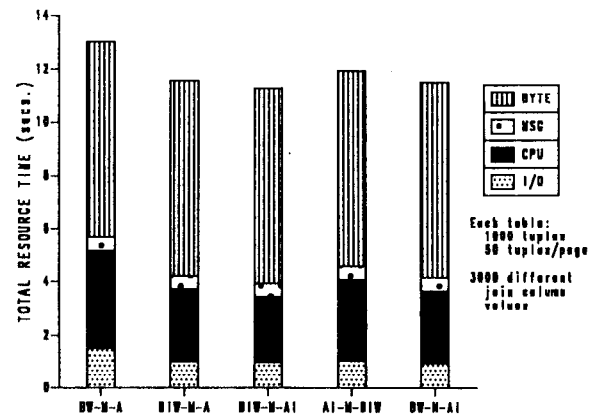

Figure 9: Relative importance of cost components for various access plans when joining 2 tables (of 1000 tuples each) distributed across a (simulated) medium-speed network.

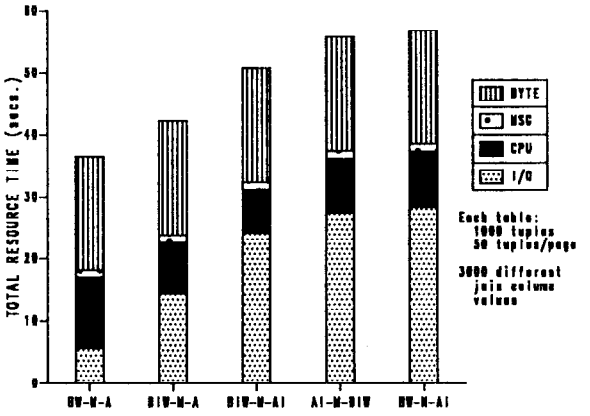

Figure 10: Relative importance of cost components for various access plans when joining 2 tables (of 2500 tuples each) distributed across a (simulated) medium-speed network.

system realizes, for example, that the plan AI-M-BIW is more expensive than BIW-M-AI. For the fetch-matches transfer strategy (for inner tables only), the expected number of messages was equal to the actual number in all cases, and the estimate for the bytes transmitted was never off by more than 25%. Although the number of bytes transferred is somewhat dependent on the join cardinality, the fixed number of bytes shipped with each message typically exceeds the inner-table data in each message, unless the inner's tuples are very wide (after projection) or are highly duplicated on the join-column value.

We encountered more severe problems in estimating the cost of shipping results of a join to the query site, because this cost is directly proportional to the join cardinality, which is difficult to estimate accurately. This problem is a special case of shipping a composite table to any site, so that these errors may be compounded as the number of tables to be joined at different sites increases.

In a high-speed network, where message costs are a small fraction of the total cost and the optimizer's decisions are based more on local processing costs, these errors (assuming that they are less than 50%) are not very crucial. For a given join ordering of tables, the choice of a site at which a particular composite table will be joined with the next inner table will depend mainly upon the indexes available on the inner, the sizes of the two tables, and possibly on the order of the composite's tuples (for a merge-scan join). However, in a medium-speed long-haul communication network, where the communications costs range from 30 to 70% of the total cost, the error in estimating the join cardinality is magnified in the overall cost estimate. In [MACK 86], we have already suggested replacing the current estimates of join cardinality with statistics collected while performing the same join for an earlier SQL statement.

Can we simplify the optimizer for high-speed local-area networks, under the assumption that message costs usually are less than 10% of the total cost? More precisely, can we, starting from the best ***hypothetical local plan*** (assuming all tables are available at the query site) for a given join, construct a distributed plan that is less than 10% more expensive than the optimum? This would considerably facilitate the optimization of distributed queries! Unfortunately the answer is no, because there may be distributed access plans that have a lower local cost than any hypothetical local plan. For example, the plan BIW-M-AI in Figure 5 has a lower local cost than any plan joining the two 1000-tuple tables locally. The corresponding hypothetical local plan BI-M-AI performs very poorly (cf. Figure 6), because the two tables do not fit into one database buffer together.

Estimates of the local processing costs for distributed queries suffered many of the same problems discovered for local queries by our earlier study. In particular, a better model is needed of the re-use of pages in the buffer when performing nested-loop joins using an unclustered index on the inner table [MACK 86]. However, the more distributed the tables participating in a join are, the better the R* optimizer estimates are. The reason for this is that join costs are estimated from the costs for producing the composite table and accessing the inner table, assuming these component costs are independent of each other. This assumption is most likely to be valid when the composite and inner tables are at different sites; tables joined locally compete for the same buffer space. For example, the estimated local costs (CPU and I/O) for joining two 1000-tuple tables locally (BI-M-AI) are the same as the estimated local costs for executing the distributed plan BIW-M-AI, but the first estimate considerably underestimates the actual local cost of BI-M-AI (see Figure 6), whereas it is very accurate for the actual local cost of BIW-M-AI (cf. Figure 5).

6. Alternative Distributed Join Methods

The R* prototype provides an opportunity to compare empirically the actual performance of the distributed join methods that were implemented in R* against some other proposed join methods for equi-joins that were not implemented in R*, but might be interesting candidates for an extension or for future systems:

1. joins using dynamically-created indexes
2. semijoins
3. joins using hashing (Bloom) filters (***Bloomjoins***)

None of these methods are new [BERN 79, DEWI 85, BRAT 85]. Our contribution is the use of performance data on a real system to compare these methods with more traditional methods. We will describe the join algorithms in detail and evaluate their performance using measured R* costs for executing sub-actions such as scans, local joins, sorting of partial results, creating indexes, etc. These costs were adjusted appropriately when necessary: for example, a page does not have to be fetched by a certain sub-action if it already resides in the buffer as a result of a previous sub-action. The alternative methods are presented both in the order in which they were proposed historically and in the order of increasingly more compact data transmission between sites. Although several hash-based join algorithms look promising based upon cost-equation analyses [DEWI 85, BRAT 85], we could not evaluate them adequately using this empirical methodology, simply because we did not have any R* performance figures for the necessary primitives.

Before comparing the methods, we will first analyze the cost for each one for a distributed equi-join of two tables S and T, residing at two different sites 1 and 2, respectively, with site 1 as the query site. Let the equi-predicate be of the form S.a = T.b, where a is a column of S and b is a column of T. For simplicity, we will consider only the two cases where both or neither S and T have an (unclustered) index on their join column(s). To eliminate interference from secondary effects, we further assume that: (1) S and T do not have any indexes on columns other than the join columns, (2) all the columns of S and T are to be returned to the user (no projection), (3) the join predicate is the only predicate specified in the query (no selection), and (4) S and T are in separate DBSPACES that contain no other tables. The extension of the algorithms to the cases excluded by these assumptions is straightforward.

6.1. Dynamically-Created Temporary Index on Inner

R* does not permit the shipment of any access structures such as indexes, since these contain physical addresses (TIDs, which contain page numbers) that are not meaningful outside their home database. Yet earlier studies of local joins have shown how important indexes can be for improving the database performance, and how in some situations creating a temporary index before executing a nested-loop join can be cheaper than executing a merge-scan join without the index [MACK 86]. This is because creating an index requires sorting only key-TID pairs, plus creation of the index structure, whereas a merge-scan join without any indexes on the tables requires sorting the projected tuples of the outer as well as the inner table. The question remains whether dynamically-created temporary indexes are beneficial in a distributed environment. The cost of each step for performing a distributed join using a dynamically-created temporary index is as follows:

1. **Scan table T and ship the whole table to site 1.** The cost for this step is equivalent to our measured cost for a remote access of a single table, subtracting the CPU cost to extract tuples from the message buffers.
2. **Store T and create a temporary index on it at site 1.** Since reading T from a message buffer does not involve any I/O cost, and either reading or writing a page costs one disk I/O, the I/O cost of writing T to a temporary table and creating an index on it will be the same as for reading it from a permanent table via a sequential scan and creating an index on that, except the temporary index is not catalogued. This cost was measured in R* by executing a CREATE INDEX statement, and then adding CPU time for the insert while subtracting the known and fixed number of I/Os to catalog pages.
3. **Execute the best plan for a local join at site 1.** Again, this cost is known from the measurements obtained by our earlier study for local joins. The I/O cost must be reduced by the number of index and data pages of T that remain in the buffer from prior steps.

6.2. Semijoin

Semijoins [BERN 79, BERN 81A, BERN 81B] reduce the tuples of T that are transferred from site 2 to site 1, when only a subset of T matches tuples in S on the join column (i.e., when the ***semijoin selectivity*** < 1), but at the expense of sending all of S.a from site 1 to site 2. The cost of each step for performing a distributed join using a semijoin when neither S.a nor T.b are indexed is as follows:

1. **Sort both S and T on the join column, producing S' and T'.** The costs measured by R* for sorting any table include reading the table initially, sorting it, and writing the sorted result to a temporary table, but not the cost of any succeeding read of the sorted temporary table.

2. **Read S'.a (at site 1), eliminating duplicates, and send the result to site 2.** This cost (and for the sort of S in the previous step) could be measured in R* for a remote "SELECT DISTINCT S.a" query, subtracting the CPU cost to extract tuples from the message buffers. If S' fits into the buffer, the previous step saves us the I/O cost; otherwise all cost components are included.
3. **At site 2, select the tuples of T' that match S'.a, yielding T", and ship them to site 1.** This cost is composed of the costs for scanning S', scanning T', handling matches, and shipping the matching tuples. Reading S'.a from the message buffer incurs no I/O cost, and scanning T' also costs only CPU instructions if T' fits into the buffer. Also, the pages of the matching tuples of T' can be transmitted to site 1 as they are found, and need not be stored, because we are using these tuples as the outer table in later steps. The cost for finding the matching tuples involves only a CPU cost that is roughly proportional to the number of matches found. The cost assessed here was derived from actual R* measurements for local queries, interpolating when the table sizes, projection factors, selection factors, etc. fell between values of those parameters used in the R* experiments.
4. **At site 1, merge-join the (sorted) temporary tables S' and T" and return the resulting tuples to the user.** This cost was measured in the same way as the previous step, less the communications cost. Note that T" inherits the join-column ordering from T'.

If there are indexes on S.a and T.b, we can either use the above algorithm or we can alter each step as follows:

1. This step and its cost can be eliminated.
2. Replace this step with a scan of S.a's index pages only (not touching any data pages) and their transmission to site 2. The cost was measured as in Step (2) above, but with an index existing on S.a; R* can detect that data pages need not be accessed.
3. Using the index on T.b, perform a local merge-scan or a nested-loop join, whichever is faster, at site 2, yielding T". Again, the cost for various local joins was measured in the earlier study; they were reduced by the cost of scanning S that was saved by taking it from the message buffer as pages arrived. Some interpolation between actual experiments was required to save re-running those experiments with the exact join cardinality that resulted here.
4. Join T" with S, using the index on S.a, again choosing between the merge-scan or nested-loop join plans whose costs were measured on R*. A known amount of I/O was subtracted for the index leaf pages that remain in the buffer from step (2).

6.3. Bloomjoin

Hashing techniques are known to be efficient ways of finding matching values, and have recently been applied to database join algorithms [BABB 79, BRAT 84, VALD 84, DEWI 85]. Bloomjoins use Bloom filters [BLOO 70] as a "hashed semijoin" to filter out tuples that have no matching tuples in a join [BABB 79, BRAT 84]. Thus, as with semijoins, Bloomjoins reduce the size of the tables that have to be transferred, sorted, merged, etc. However, the bit tables used in Bloomjoins will typically be smaller than the join-column values transmitted for semijoins. By reducing the size of the inner table at an early stage, Bloomjoins also save local costs. Whereas a semijoin requires executing an extra join for reducing the inner table, Bloomjoins only need an additional scan in no particular order. For simplicity, we use only a single hashing function; further optimization is possible by allowing multiple hashing functions [SEVE 76]. The cost of each step for performing a distributed join using a Bloomjoin when neither S.a nor T.b are indexed is as follows:

1. **Generate a Bloom filter, BfS, from table S.** The Bloom filter, a large vector of bits that are initially all set to "0", is generated by scanning S and hashing each value of column S.a to a particular bit in the vector and setting that bit to "1". As before, the cost of accessing S was measured on R*. We added 200 (machine-level) instructions per tuple (a conservative upper bound for any implementation) for hashing one value and setting the appropriate bit in the vector.
2. **Send BfS to site 2.** We assume that sending a Bloom filter causes the same R* message overhead as if sets of tuples are sent, and the number of bytes is obvious from the size of the Bloom filter.
3. **Scan table T at site 2, hashing the values of T.b using the same hash function as in Step (1). If the bit hashed to is "1", then send that tuple to site 1 as tuple stream T'.** This cost is calculated as in Step (1), but the number of tuples is reduced by the Bloom filtering. We need to estimate the reduced ***Bloomjoin cardinality*** of T, i.e. the cardinality of T'. We know it must be at least the ***semijoin cardinality of T***, SC_T, i.e. the number of tuples in T whose join-column values match a tuple in S. We must add an estimate of the number of non-matching tuples in T that erroneously survive filtration due to collisions. Let F be the size (in bits) of BfS, D_S the number of distinct values of S.a, D_T the number of distinct values of T.b, and C_T the cardinality of T. Then the number of bits set to "1" in BfS is approximated for large D_S by [SEVE 76]:

$$bits_S = F(1 - e^{-(\frac{D_S}{F})})$$

So the expected number of tuples in T', the Bloomjoin cardinality BC_T of table T, is given by

$$BC_T = SC_T + bits_S(1 - e^{-(\frac{\alpha D_T}{F})})$$

where

$$\alpha = (1 - \frac{SC_T}{C_T})$$

is the fraction of non-matching tuples in T.
4. **At site 1, join T' to S and return the result to the user.** This cost was derived as for semijoins, again using the Bloomjoin cardinality estimate for T'.

If there are indexes on S.a and T.b, we can either use the above algorithm or, as with semijoins, use the index on S.a to generate BfS -- thus saving accesses to the data pages in Step (1) — and use the index on both T.b and S.a to perform the join in Step (4).

As with semijoins, filtration can also proceed in the opposite direction: S can also be reduced before the join by sending to site 1 another Bloom filter BfT based upon the values in T. This is usually advantageous if S needs to be sorted for a merge-scan join, because a smaller S will be cheaper to sort. Filtration is maximized by constructing the more selective Bloom filter first, i.e. on the table having the fewer distinct join column values[6], and altering the Bloomjoin procedure accordingly:

- If we first produce BfS, then add step (3.5): while scanning T in step (3), generate BfT, send it to site 1, and use it to reduce S.
- If we first produce BfT, then add step (0.5): generate BfT, send it to site 1, and use it to reduce S while scanning S in step (1).

6.4. Comparison of Alternative Join Methods

Using the actual costs measured by R* as described above, we were able to compare the alternative join methods empirically with the best R* plan, for both the distributed and local join, for a two-table join with no projections and no predicates other than the equi-join on an integer column. The measured cost was total resource time, since response time will vary too much depending upon other applications executing concurrently.

Our experimental parameters for this analysis were identical to those in the previous section. We fixed the size of table A at site 1 at 1000 tuples, and varied the size of table B at site 2 from 100 to 6000 tuples. For the Bloomjoin we chose a filter size (F) of 2K bytes (16384 bits) to ensure that it would fit in one 4K byte page. Again, we assumed the availability of (unclustered) indexes on the join columns. We will discuss the impact of relaxing this and other assumed parameters where appropriate in the following, and at the end of this section.

As in the previous section, we compared the performance of the join methods under two classes of networks:

- a high-speed network (16.5 msecs. minimum transfer time, 4M bit/sec. effective transfer rate); and
- a medium-speed long-haul network (50 msecs. minimum transfer time, 40K bit/sec. effective transfer rate)

by appropriately adjusting the per-message and per-byte weights by which observed numbers of messages and bytes transmitted were multiplied. For each of these classes, we varied the query site between site 1 and site 2.

---

6 If this cannot be determined, simply choose the smaller table [BRAT 84].

6.4.1. High-speed Network

For a high-speed network (Figure 11), the cost of transmission is dominated by local processing costs, as shown by the following table of the average percentage of the total costs for the different join algorithms that are due to local processing costs:

Query Site	*R**	*R* + temp. index*	*Semijoin*	*Bloomjoin*
1 = site of A	88.9%	89.2%	96.5%	93.0%
2 = site of B	86.5%	91.4%	94.7%	90.1%

Temporary indexes generally provided little improvement over R* performance, because the inexpensive shipping costs permit the optimal R* plan to ship B to site 1, there to use the already-existent index on A to perform a very efficient nested-loop join. When there was no index on A, the ability to build temporary indexes improved upon the R* plan by up to 30%: A was shipped to site 2, where a temporary index was dynamically built on it and the join performed. Such a situation would be common in *multi-table* joins having a small composite table that is to be joined with a large inner, so temporary indexes would still be a desirable extension for R*.

Semijoins were advantageous only in the limited case where both the data and index pages of B fit into the buffer (*cardinality*$(B) \leq 1500$), so that efficient use of the indexes on A and B kept the semijoin's local processing cost only slightly higher than that of the optimal R* plan. Once B no longer fits in the buffer (*cardinality*$(B) \geq 2000$), the high cost of accessing B with the unclustered index precluded its use, and the added cost of sorting B was not offset by sufficient savings in the transfer cost.

Bloomjoins dominated all other join alternatives, even R* joining local tables! This should not be too surprising, because local Bloomjoins outperform local R* by 20-40%, as already shown in [MACK 86], and transmission costs represent less than 10% of the total costs. The performance gains depend upon the ratios, r_A and r_B, between the Bloomjoin cardinality and the table cardinality of A and B, respectively: r_B is relatively constant (0.31), whereas r_A is varying (e.g., 0.53 for *cardinality*$(B) = 2000$ and 1.0 for *cardinality*$(B) = 6000$). But even if those ratios are close to 1, Bloomjoins are still better than R*. For example, when r_A=1.0, r_B=0.8, and *cardinality*$(B) = 6000$, a Bloomjoin would still be almost two seconds faster than R*. Note that due to a much higher join cardinality in this case, the R* optimum would be more expensive than the plotted one.

Why are Bloomjoins — essentially "hashed semijoins" — so much better than semijoins? The message costs were comparable, because the Bloom filter was relatively large (1 message) compared to the number of distinct join column values, and the number of non-matching tuples not filtered by the Bloom filter was less than 10% of the semijoin cardinality. The answer is that the semijoin incurs higher local processing costs to essentially perform a second join at B's site, compared to a simple scan of B in no particular order to do the hash filtering.

The above results were almost identical when B's site (2) was the query site, because the fast network makes it cheap to ship the results back after performing the join at site 1 (desirable because table A fits in the buffer). The only exception was that temporary indexes have increased advantage over R* when A could be moved to the query site and still have an (dynamically-created temporary) index with which a fast nested-loop join could be done.

We also experimented with combining a temporary index with semijoins and Bloomjoins. Such combinations improved performance only when there were no indexes, and even then by less than 10%.

6.4.2. Medium-speed Network

In a medium-speed network, local processing costs represent a much smaller (but still very significant!) proportion of the cost for each join method:

Query Site	*R**	*R* + temp. index*	*Semijoin*	*Bloomjoin*
1 = site of A	38.5%	22.6%	46.3%	32.3%
2 = site of B	38.5%	36.0%	53.0%	41.6%

Regardless of the choice of query site, Bloomjoins dominated all other distributed join methods by 15-40% for *cardinality*$(B) > 100$ (compare Figure 12 and Figure 13). The main reason was smaller transmissions: the communications costs for Bloomjoins were 20-40% less than R*'s, and for *cardinality*$(B) \geq 1500$ shipping the Bloom filter and some non-matching tuples not filtered by the Bloom filter was cheaper than shipping B's join column for semijoins. Because of their compactness, Bloom filters can be shipped equally easily in either direction, whereas R* and R* with temporary indexes always try to perform the join at A's site to avoid shipping table B (which would cost approximately 93.2 seconds when *cardinality*$(B) = 6000$!).

Also independent of the choice of query site was the fact that temporary indexes improved the R* performance somewhat for bigger tables.

Only R* and semijoins change relative positions depending upon the query site. When the query site is the site of the non-varying 1000-tuple table A, semijoins are clearly better than R* (see Figure 12). When the query

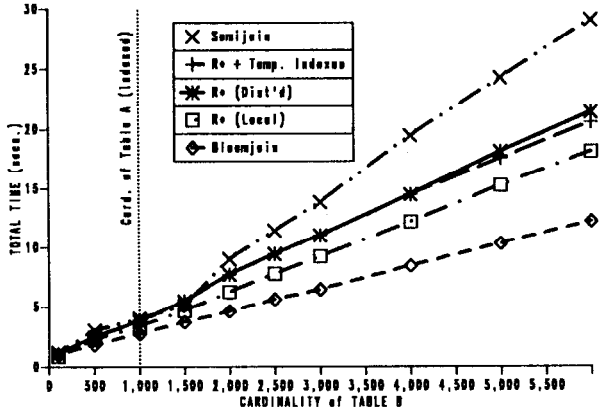

Figure 11: High-speed network; Query Site = 1 (A's site)

R*'s best distributed and local plan (measured) vs. performance of other join strategies (simulated) for a high-speed network, joining an indexed 1000-tuple table A at site 1 with an indexed table B (of increasing size) at site 2, returning the result at site 1.

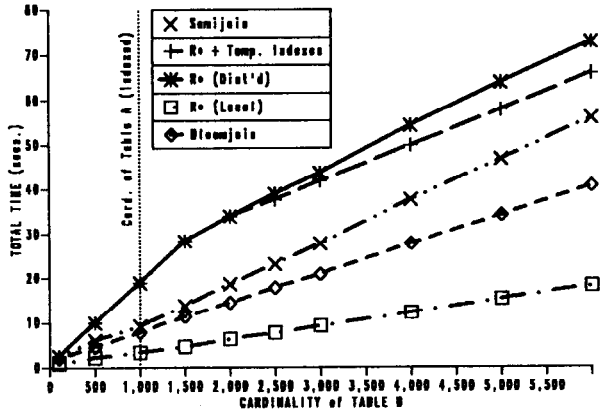

Figure 12: Medium-speed network; Query Site = 1 (A's site)

R*'s best distributed and local plan (measured) vs. performance of other join strategies (simulated) for a medium-speed network, joining an indexed 1000-tuple table A at site 1 with an indexed table B (of increasing size) at site 2, returning the result at site 1.

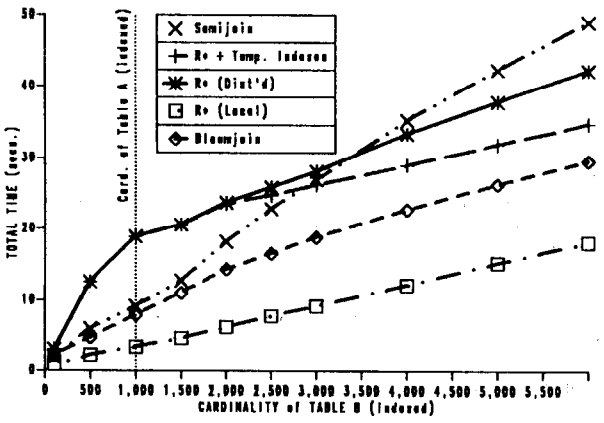

Figure 13: Medium-speed network; Query Site = 2 (B's site)

R*'s best distributed and local plan (measured) vs. performance of other join strategies (simulated) for a medium-speed network, joining an indexed 1000-tuple table A at site 1 with an indexed table B (of increasing size) at site 2, returning the result at site 2.

site is B's site, however, R* still beats semijoins when B is sufficiently large (cf. Figure 13). The reason is straightforward but important for performance on large queries. Since the join columns had a domain of 3000 different values, most of these values had matches in B when *cardinality*(*B*) $\geq$ 3000. Thus, the semijoin cardinality of A was close to its table cardinality, meaning that most of the tuples of A survived the semijoin and were shipped to site 2 anyway (as in the R* plan). With the additional overhead of sending the join column of B to site 1 and the higher local processing cost, semijoins could not compete.

Note that both the R* and the semijoin curves jumped when the index and data pages of table B no longer fit in the buffer (between 1500 and 2000 tuples), because they switched to sorting the tables.

6.4.3. Variation of the Experimental Parameters

Space constraints prevent us from presenting the results of numerous other experiments for different values of our experimental parameters:

- When indexes were clustered (rather than unclustered), semijoins beat R* by at most 10% (except when the query site = B's site and B is very large), but Bloomjoins still dominated all other distributed join techniques.

- Introducing a 50% projection on both tables in our join query did not change the dominance of Bloomjoins, but eliminated any performance advantage that temporary indexes provided over R* and, when the query site was A's site, reduced the local processing cost disadvantage of semijoins sufficiently that they beat R* (but by less than 10%). However, when the query site was B's site, the 50% projection reduced R*'s message costs more than those for semijoins, giving R* an even wider performance margin over semijoins.

- As expected, a wider join column (e.g., a long character column or a multi-column join predicate) decreased the semijoin performance while not affecting the other algorithms.

7. Conclusions

Our experiments on two-table distributed equi-joins found that the strategy of shipping the entire inner table to the join site and storing it there dominates the fetch-matches strategy, which incurs prohibitive per-message costs for each outer tuple even in high-speed networks.

The R* optimizer's modelling of message costs was very accurate, a necessary condition for picking the correct join site. Estimated message costs were within 2% of actual message costs when the cardinality of the table to be shipped was well known. Errors in estimating message costs originated from poor estimates of join cardinalities. This problem is not introduced by distribution, and suggestions for alleviating it by collecting join-cardinality statistics have already been advanced [MACK 86].

The modelling of local costs actually *improves* with greater distribution of the tables involved, because the optimizer's assumption of independence of access is closer to being true when tables do not interfere with each other by competing for the same resource (especially buffer space) within a given site. While more resources are consumed overall by distributed queries, in a high-speed network this results in response times that are actually less than for local queries for certain plans that can benefit from:

- concurrent execution due to pipelining, and/or
- the availability of more key resources — such as buffer space — to reduce contention.

Even for medium-speed, long-haul networks linking geographically dispersed hosts, local costs for CPU and I/O are significant enough to affect the choice of plans. Their relative contribution increases rather than decreases as the tables grow in size, and varies considerably depending upon the access path and join method. Hence no distributed query optimizer can afford to ignore their contribution.

Furthermore, the significance of local costs cannot be ignored when considering alternative distributed join techniques such as semijoins. They are advantageous only when message costs are high (e.g., for a medium-speed network) and any table remote from the join site is quite large. However, we have shown that a Bloomjoin — using Bloom filters to do "hashed semijoins" — dominates the other distributed join methods *in all cases investigated*, except when the semijoin selectivities of the outer and the inner tables are very close to 1. This agrees with the analysis of [BRAT 84].

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. Experimenting with n-table joins, in particular, is crucial to validating the optimizer's selection of join order. We hope to actually test rather than simulate semijoins, Bloomjoins, and medium-speed long-haul networks.

Finally, R* employs a homogeneous model of reality, assuming that all sites have the same processing capabilities and are connected by a uniform network with equal link characteristics. In a real environment, it is very likely that these assumptions are not valid. Adapting the optimizer to this kind of environment is likely to be difficult but important to correctly choosing optimal plans for real configurations.

8. 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. Toby Lehman (visiting from the University of Wisconsin) implemented the DC* counters. George Lapis helped with database generation and implemented the 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. Christoph Freytag, Laura Haas, Bruce Lindsay, John McPherson, Pat Selinger, and Irv Traiger constructively critiqued an earlier draft of this paper, improving its readability significantly. Finally, Tzu-Fang Chang and Alice Kay provided invaluable systems support and patience while our tests consumed considerable computing 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.

[BABB 79] E. Babb, Implementing a Relational Database by Means of Specialized Hardware, *ACM Trans. on Database Systems* **4,1** (1979) pp. 1-29.

[BERN 79] P.A. Bernstein and N. Goodman, Full reducers for relational queries using multi-attribute semi-joins, *Proc. 1979 NBS Symp. on Comp. Network.* (December 1979).

[BERN 81A] P.A. Bernstein and D.W. Chiu, Using semijoins to solve relational queries, *Journal of the ACM* **28,1** (January 1981) pp. 25-40.

[BERN 81B] 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.

[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. Morgan Kaufmann Publishers, Los Altos, CA.

[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, September 1982) pp. 54-61. Morgan Kaufmann Publishers, Los Altos, CA.

[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 79] D.J. DeWitt, Query Execution in DIRECT, *Procs. of ACM-SIGMOD* (May 1979).

[DEWI 85] D.J. DeWitt and R. Gerber, Multiprocessor Hash-Based Join Algorithms, *Procs. of the Eleventh International Conf. on Very Large Data Bases* (Stockholm, Sweden, September 1985) pp. 151-164. Morgan Kaufmann Publishers, Los Altos, CA.

[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.

[HEVN 79] A.R. Hevner and S.B. Yao, Query Processing in Distributed Database Systems, *IEEE Trans. on 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.

[LIND 83] B.G. Lindsay, L.M. Haas, C. Mohan, P.F. Wilms, and R.A. Yost, Computation and Communication in R*: A Distributed Database Manager, Proc. 9th ACM Symposium on Principles of Operating Systems (Bretton Woods, October 1983). Also in *ACM Transactions on Computer Systems 2*, 1 (Feb. 1984), pp. 24-38.

[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. Morgan Kaufmann Publishers, Los Altos, CA. 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* (Kim, Batory, & Reiner (eds.), 1985) pp. 31-47. Springer-Verlag, Heidelberg. Also available as IBM Research Report RJ4272, San Jose, CA, April 1984.

[LU 85] H. Lu and M.J. Carey, 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) pp. 292-304. Morgan Kaufmann Publishers, Los Altos, CA.

[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 (San Jose, CA, September 1985).

[MACK 86] L.F. Mackert and G.M. Lohman, R* Optimizer Validation and Performance Evaluation for Local Queries, *Procs. of ACM-SIGMOD* (Washington, DC, May 1986 (to appear)). Also available as IBM Research Report RJ4989, San Jose, CA, January 1986.

[MENO 85] M.J. Menon, Sorting and Join Algorithms for Multiprocessor Database Machines, *NATO-ASI on Relational Database Machine Architecture* (Les Arcs, France, July 1985).

[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.

[PERR 84] W. Perrizo, A Method for Processing Distributed Database Queries, *IEEE Trans. on Software Engineering* **SE-10,4** (July 1984) pp. 466-471.

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

[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* (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* (Univ. of Aberdeen, Scotland, July 1980 pp. 204-215. Deen and Hammersly, ed.

[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.

[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.

[VALD 84] P. Valduriez and G. Gardarin, Join and Semi-Join Algorithms for a Multiprocessor Database Machine, *ACM Trans. on Database Systems* **9,1** (March 1984) pp. 133-161.

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

[WONG 83] E. Wong, Dynamic Rematerialization: Processing Distributed Queries using Redundant Data, *IEEE Trans. on Software Engineering* **SE-9,3** (May 1983) pp. 228-232.

[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.