Modelling Costs for a MM-DBMS

Sherry Listgarten Marie-Anne Neimat
lastname@hpl.hp.com
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94301

Abstract

Research in main-memory database management systems (MM-DBMS's) has addressed several aspects of data management such as indexing, transaction management, and query processing. However, query optimization has been by and large a neglected component of these systems, perhaps because the prototypical queries requiring real-time response are fairly simple. Experience has shown, however, that many of the systems employing MM-DBMS's have a heterogeneous query base, and that there is a need for real-time response to complex queries. MM-DBMS's can also be used as caches to disk-based systems, and queries over such two-tiered DBMS's range from simple to complex. Finally, we expect that MM-DBMS's will gradually move into the mainstream, whereupon they will need to support applications that incorporate all types of queries.

While each aspect of query optimization needs to be reexamined for a MM-DBMS, we focus on cost model development in this paper. Because the costs underlying a main-memory system are inherently more complex and more variable than those underlying a disk-based DBMS, a new approach is needed to develop an accurate and robust cost model. We compare three different approaches, based on hardware costs, application costs, and execution engine costs, respectively, and argue that only the latter provides for rapid development of an accurate and portable cost model.

1 Introduction

Database researchers have been investigating aspects of main-memory database management systems (MM-DBMS's) since the mid-eighties, but recently the interest has taken on a new urgency as cheap memory and 64-bit addressing are becoming a reality. Several fairly complete systems have been developed in the last few years, and recent investigations have taken a fresh look at a variety of issues in the context of main-memory: recovery, indexing, parallelism, and concurrency control, for example. However, the issue of query optimization has largely been neglected, perhaps because the queries requiring high-performance processing, such as those needed for telecom switching and financial trading, tend to be quite simple (e.g., a hash lookup on a single table). Yet very few applications require only such queries. Complex queries having to do with fraud detection, financial analysis, hot billing, and so forth, must be done over the same data and under constraints on response time that require the use of a MM-DBMS. This necessitates the use of appropriate optimization techniques for memory-resident data. In addition, main-memory systems may be used as caches for disk-based DBMS's; such systems need to handle all sorts of queries. Finally, we consider main-memory databases to be a "disruptive technology" [Bow95] and so we anticipate that as the technology becomes more widely adopted, MM-DBMS's will be used in increasingly general-purpose situations, which will require query optimization. Indeed, the recent announcement that Oracle will be including a copy of an in-memory database with each Oracle7 system [Ora95] goes some way towards justifying this belief.

A cost-based query optimizer has three main components, each of which needs to be reexamined in a main-memory context: the set of feasible plans; the search strategy used to navigate through this set of plans; and the objective function used to choose among the plans.

The traditional notions of feasibility are directly affected by the assumption of memory residence. Certain join methods may no longer be practical, while alternative indexing methods [Leh86] and join methods [Wil95] may be preferable. For example, it has been suggested (and our results confirm) that it is not necessary to provide sort-merge or merge-join methods in MM-DBMS's [Wha90]. The set of feasible plans is constrained by more than just the constructs provided by the execution engine. Limited memory availability, for example, may restrict the set of feasible plans to those that do not create temporary indexes or materialize intermediate tables. The domain may also be extended by differentiating between plans that are considered identical in a disk-based system. For example, since predicate evaluation can be a dominant cost in main-memory systems [Wha90], accounting for different predicate orders will enlarge the search space.

A variety of search strategies have been implemented in query optimizers — exhaustive search of a constrained

space, heuristic search, and randomized search, for example. While all of these are applicable in a main-memory context, their suitability may be quite different depending on how the search space itself has changed. For example, if only one join method is supported, then a more exhaustive search of the plan space may be practical, especially if one can neglect interesting orders. If the compiler has very limited memory to work with, the commonly-used dynamic programming approach may be difficult or impossible to implement. Heuristics that seem reasonable in a disk-based system are no longer so obvious in a main-memory context. For example, predicates are routinely pushed down by traditional query optimizers, but if predicate evaluation is relatively expensive, it is not clear that this is always the best thing to do [Cha95]. Predicate costs need to be considered in addition to predicate selectivity.

However, the main-memory assumption arguably affects the optimizer's objective function more than any of the other components. With the single dominant cost of I/O absent in MM-DBMS's, the cost function and the model on which it is based change completely. A main-memory system implements a large number of operations with comparable costs. No single cost stands out, and so an appropriate cost model will be more complex and likely more difficult to construct. Furthermore, the costs are quite variable from system to system as they are subject to changes in hardware and system software. Since MM-DBMS's are often embedded systems, it must be possible to adapt the cost model to a variety of substrates.

We consider three different types of cost models in this work — those based on hardware costs, application costs, and execution engine costs, respectively — and argue that only the latter provides for an accurate and easy-to-maintain cost model. We have integrated such a model into the cost-based optimizer of Smallbase [Hey95a], a MM-DBMS developed at HP Labs that is currently in use in a variety of applications, including telecom switches [Hey95b] and financial trading [Mun94]. In this work, we compare the three different types of cost models, and explain how we used an engine-based approach to develop, instantiate, and verify Smallbase's cost model.

In Section 2, we describe the three types of cost models, and discuss their utility in the face of the high complexity and variability of a MM-DBMS's costs. In Section 3, we present our implementation of the engine-based approach, and a test we devised to instantiate and verify the model. And in Section 4 we discuss some of the results of the verification process.

2 A Taxonomy and Comparison of Main-Memory Cost Models

In this section, we consider three fundamental types of cost models that can be developed for main-memory systems, and discuss how they deal with the inherently complex and variable nature of the costs incurred in main-memory query processing.

A *hardware-based* cost model is formulated in direct analogy to the traditional I/O-based models for disk-based systems; it simply counts CPU cycles instead of I/O operations. This approach, while conceptually simple, is difficult to implement. An accurate cycle count likely depends on hardware policies such as cache-replacement and pre-fetching, which can be difficult to model. Furthermore, since such policies vary from machine to machine, and since the number of cycles involved in even basic arithmetic operations may change, a hardware-based cost model is very difficult to port. Once constructed, however, such a model is likely to be accurate and reliable, in the sense that costs are unlikely to change over time on a given system.

A second type of cost model, such as that used in [Wha90], is what we call *application-based*. An application-based cost model is expressed in terms of "bottleneck costs", namely the costs of those sections of code that are found to be bottlenecks when the system is run on one or more target applications. A profiler is used to determine where the application is spending most of its time, and the relevant DBMS code is written as tightly as possible so that the relative costs will remain reasonably constant when hardware or software are changed.

A model based on such costs is much simpler to develop than the hardware-based model, but it is less general since the bottleneck costs do not fully reflect the cost of a plan. Because the model is based on those costs reflected by certain applications or test programs, it may not be meaningful or practical for all applications. For example, the cost of an index lookup is not modelled at all in [Wha90] since it was not a bottleneck for their applications, and it cannot be expressed in terms of the few base costs that were identified as such. This cost may well be a significant factor in the cost of certain query evaluations, and may play a part in determining which of several plans should be executed.

Application-based cost models are easier to port than hardware-based models, since profiling information can be regenerated on each system of interest. However, a new profile may result not only in modified relative values for the bottleneck costs, but also in the addition or removal of such costs; that is, the model itself may change, and not just the instantiation. This then involves

re-expressing the costs of all the database operations in terms of the new set of underlying "bottleneck" costs. In summary, an application-based cost model is simpler to develop than the more complex hardware models, and it is based on a relatively small set of costs that have very reliable values on a given architecture. However, such a model may be accurate only for a certain range of applications, and it can be somewhat difficult to port.

Finally, a third type of cost model is based not on the fundamental hardware costs, and not on the fundamental application costs, but on the costs of the primitive operations supported by the MM-DBMS's execution engine itself. We call this an *engine-based* model. The unit costs are the low-level operations that are implemented by the system's execution engine. These operations may be identifiable as nodes of an internal tree structure or as instruction types, depending on whether the execution engine operates recursively or iteratively.

An engine-based cost model is not application-specific, and will naturally reflect different kinds of expressions, even argument types, since they are reflected in the operations of the execution engine. There is likely to be some ambiguity about the level of detail to which the engine's operations should be reflected in the model; the analysis tool we describe later offers a way to determine to what degree the model should be refined.

Engine-based models are easier to develop than hardware-based models, more general than application-based models, and easy to port to different systems. Developing such models requires knowledge of the internals of the DBMS, and the cost model will change as operations are added to (or removed from) the execution engine. However, such changes are likely to be rare, and the model is easy to re-instantiate as necessary due to more frequently occurring changes in the operations' implementations. The simplicity of model construction and ease of adaptability to various platforms led us to incorporate this type of model in the Smallbase system.

3 Constructing an Engine-Based Cost Model

There are two steps involved in constructing an engine-based model: model design and model instantiation/verification. The first stage involves identifying the base costs and expressing the query processing costs in terms of these costs. The second involves determining the relative values for the base costs and then verifying the model. Model design is done manually, and requires knowledge of the internals of the DBMS's execution engine. It may not always be clear to what level of detail the model should reflect the execution engine. For example, should the model have distinct costs for evaluating greater-than and equality predicates? How about for evaluating the same predicate on different data types? Initially, the model should be made as detailed as practical; it can then be simplified, or further refined, as determined during verification. Since the basic model needs to be updated only when functionality is added to (or removed from) the execution engine, construction and maintenance of the model itself is fairly simple.

Model instantiation, on the other hand, should be automated as much as possible, since costs may change frequently. New cost values need to be determined for each system the database is run on and for each new release of the DBMS itself. We developed a program, or cost test, that instantiates and verifies the model using a subtractive method to identify the cost values, as well as statistical techniques to determine their reliability. This test has enabled us to easily adapt the model as our implementation has changed, for example when we moved from using recursive evaluation to iterative evaluation. It has also been helpful in validating optimization on several architectures. We review the design of the cost test in the remainder of this section, and discuss some of the results and discrepancies in Section 4.

The cost test estimates a value for each unit cost given a formula that specifies the cost's dependency on table size. In many cases (e.g., the cost of adding two integers) the cost is simply a constant, but in others (e.g., the cost of creating a hash index) it may have the form $A * n + B$, or even (e.g., the cost of sorting) something like $A * nlogn + B * n + C$, where n is the number of tuples. In general, any expression linear in the cost parameters can be accepted by the cost test, though we currently restrict expressions to three terms. (We allow only linear expressions so that it is simple to solve the resulting system of equations.)

For each unit cost, two queries are identified whose execution costs differ in that value, plus perhaps other costs whose estimates are already available. If no such pair of queries can be constructed, a single query is used and all costs but the one being measured are subtracted from the execution time. Note that the order in which the costs are estimated is important, since certain costs may be needed to estimate other costs. In general, such dependencies should be limited to avoid propagating errors.

Each query or pair of queries is run on several different table sizes. For a given table size, the query is run twice in a row, with only the second execution time being used. If the operation being measured occurs only once during that execution (e.g., a scan close operation), then the query is executed several times, with the median time being used. The effect of these repeated measurements is

to load the cache, so that measurements are made with cached instructions and data (to the extent permitted by the query). This is done to minimize the variance of the resulting values, and because it is likely that the most time-consuming operations will occur in cache.

Once a value has been derived for each table size, a least-squares fit is used to determine estimates for the parameters of the appropriate cost formula. The fit is done to minimize relative error, as opposed to absolute error, in order to give equal weight to the smaller tables. The estimates are then output along with the relative errors for each table size. For example, if a T-tree lookup is modeled as a constant (a T-tree is B-tree-like index designed for MM-DBMS's), then the output might look as follows if seven different table sizes are used:

```
Est. for T-tree lookup (one integer key):     444.000
  20.00   13.85   13.85    5.71  -12.94  -12.94  -20.71
```

The progressive and significant change in the relative errors (percentage deviation of derived estimate from actual measurements) on the second line indicates that a better model would reflect table size in the lookup cost.

On the other hand, the following output validates the linear model specified for creating a hash index:

```
Est. for creating hash index:    60.359 * n + 466.221
    2.36   -1.36   -5.23    2.33    3.52   -6.17    3.52
```

In addition to running each query on several different table sizes, the cost test also iterates through various data types to ensure that the costs are measured in an application-independent fashion. Queries may also be modified to vary other parameters, such as the length (i.e., number of values) of an index key. Cost estimates are derived for each type of query, and then the relative errors can be analyzed to determine where the model needs to be refined or otherwise modified.

Continuing with the previous example, the full output for an index lookup might be as follows:

```
Est. for T-tree lookup (one integer key):     444.000
  20.00   13.85   13.85    5.71  -12.94  -12.94  -20.71
Est. for T-tree lookup (two integer key):     612.000
  27.50    7.37    5.52    3.73   -2.86  -13.80  -11.30
Est. for T-tree lookup (one double key):      564.000
  22.61   12.80    6.42    4.44  -11.88   -7.54  -13.23
Est. for T-tree lookup (two double key):      608.000
  14.72   10.55   16.92    4.83   -9.25  -14.37  -16.71
Est. for T-tree lookup (one string key):      674.000
  34.80   20.36    5.31    0.60   -6.39  -13.59  -21.63
Est. for T-tree lookup (two string key):      774.000
  35.79   13.82   13.82    3.20   -7.86  -15.87  -17.66
```

4 Results and Discussion

In this section we look at some of the results of using the verification process on our engine-based cost model.

The cost test helped us to refine and improve Smallbase's cost model in a number of ways. For example, our initial model had a single cost for T-tree lookups, and did not differentiate based on key length or type. However, the cost test indicated a large discrepancy in cost based on data type, as shown in the previous example. It turns out that even a simple plan can run about 25% faster if the appropriate index is chosen for a predicate with several data types. Figure 1 shows the result of running the following SQL query on a 10,000-tuple table, where A is an integer column and B is a char(20) column:

```
SELECT * FROM TBL1 WHERE
A < :V1 AND B < :V2;
```

Indexes exist on both columns A and B. Column values are chosen uniformly and independently from the intervals shown in Table 1, where k varies with the desired selectivity of the result. Query parameters are chosen so that the predicates have equal selectivities.

TABLE 1. Schema for TBL1

Col.	Low	High
A	0	*k*
B	'0000'	*'k'*

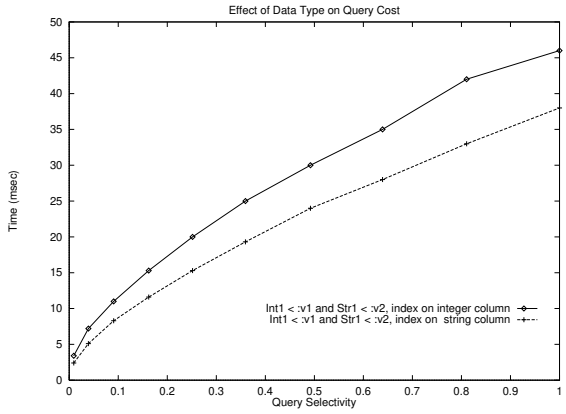

FIGURE 1. Effect of Data Type on Query Cost (Choice of Index)

Note that when the index on the char(20) column is used, the query runs about 25% faster than when the integer index is used, for a wide range of selectivities.

In addition, even if indexes are not present, optimization is more accurate if predicate costs are broken down according to type. The cost test noted a significant difference in the cost of evaluating a simple inequality on string and integer data types (the former taking about twice as long). If the cost model takes data type into account, the more expensive predicates can be evaluated after cheaper predicates, assuming equivalent selectivities. Figure 2 shows the results of running the following query on a 10,000-tuple table, where A and B are integer columns and C and D are char(20) columns:

```
SELECT * FROM TBL2 WHERE
A < B AND C < D;
```

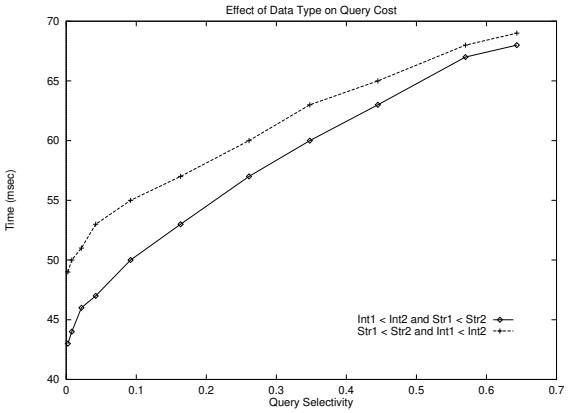

FIGURE 2. Effect of Data Type on Query Cost (Predicate Reordering)

Column values for TBL2 are chosen uniformly and independently from the intervals shown in Table 2, where *k* again varies with the desired selectivity of the result. Predicates are constructed to have equal selectivities, so only the evaluation cost differs.

TABLE 2. Schema for TBL2, TBL3, and TBL4

Col.	Low	High
A	0	9999
B	0	*k*
C	'0000'	'9999'
D	'0000'	'*k*'

Predicate reordering based on data type results in a small but consistent improvement in performance, up to 12%. The improvement is greater for small selectivities because there is less overhead needed to produce result tuples (even though the tuples are not printed or otherwise examined).

The cost test also showed, surprisingly, that it is cheaper in Smallbase to retrieve tuples from either a temporary table or a T-tree index than it is from a permanent table, due to the overheads of traversing certain directory structures when scanning a permanent table. Hence, it can be advantageous in our system, even in the absence of predicates, to create T-trees or build temporary tables. Consider the following simple join query, where A and B are integer columns as described in Table 2:

```
SELECT * FROM TBL3, TBL4 WHERE
TBL3.A < TBL4.B;
```

Results from running this query on two 1000-tuple tables with no indexes on the join columns are shown in Figure 3. Note that T-tree scans can be over 20% faster than table scans, even though they do not index the join column! Furthermore, if the optimizer is allowed to create plans that generate temporary tables[1], then performance is even better if a temporary table is built to hold the inner relation.

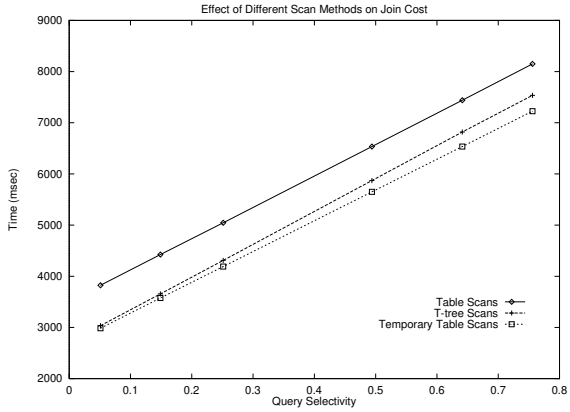

FIGURE 3. Effect of Scan Method on Join Cost

Finally, the cost test was helpful in verifying our intuition and that in [Wha90] that predicate costs are relatively high in MM-DBMS's, and that an optimizer should model these costs carefully in addition to modeling predicate selectivities. Figure 4 shows the result of running the following queries on a 10,000-tuple table. The queries are constructed so that all predicates have equal selectivity and differ only in the cost of performing the arithmetic operations.

1. An application may choose to prohibit such an operation due to memory constraints.

```
SELECT * FROM TBL5 WHERE
A + B < C AND D < E;

SELECT * FROM TBL5 WHERE
(A+365*B)/20 < C AND D < E;
```

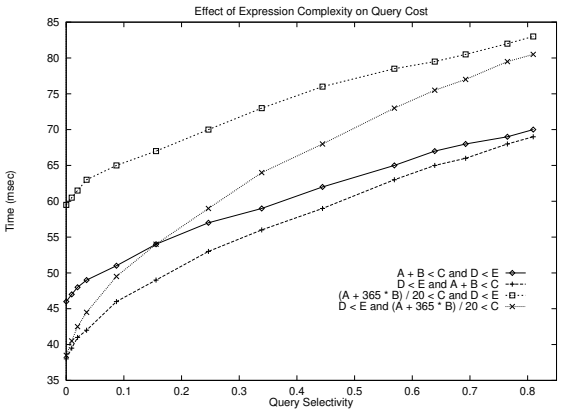

FIGURE 4. Effect of Expression Complexity on Query Cost

Column values are integers chosen uniformly and independently from the intervals shown on the left half of Table 3. The constant value 100 for column B is chosen arbitrarily. The value of k varies between 4,000 and 50,000, depending on the desired selectivity of the result.

TABLE 3. Schema for TBL5

Col.	Low	High	Low	High
A	0	9,999	0	$20 * 9{,}999$[a]
B	100	100	100	100
C	100	$100 + k$	1825	$1825 + k$
D	0	9,999	0	9,999
E	0	k	0	k

a. The 10,000 multiples of 20 are used for column A in the second query.

The predicate A+B < C is more time-consuming to evaluate than D < E. A cost model that reflects this fact enables the optimizer to reorder the predicates so that the less expensive but equally selective predicate D < E is evaluated first. Figure 4 shows that this reordering can yield as much as a 17% improvement, even for these simple predicates. The more complex expression shown, similar to one that might be encountered in financial analysis, yields up to a 35% difference.

Thus the engine-based approach combined with the analysis provided by the cost test provided us with an accurate but not unwieldy cost model. There remain a few difficulties with the cost test that would be useful to resolve in the future, mainly related to the variability of the underlying costs. For example, we found that we were unable to get reliable values when we specified fairly complex cost functions (e.g., $A * nlogn + B * n + C$), and hence we restricted our cost model to simpler functions. In addition, the cost test revealed that values that theoretically should have cost independent of table size, such as the cost of adding two columns, were actually increasing with table size. It turns out that this is an effect of caching; since larger tables were not completely in cache, operations on data in those tables take more time. We would like to have better control over this effect.

5 Conclusions and Future Work

In this work, we have constructed a taxonomy of cost models for MM-DBMS's, and have presented a practical approach for developing an appropriately detailed cost model for a MM-DBMS. We also outlined a cost test that has proven useful for instantiating and validating a fairly complex cost model. The plans that result are often surprising (for example, it proves efficient to use range indexes to scan a table even when no predicates are present!), but they are always among the best for our system.

Work remains to be done on the other aspects of optimization for a MM-DBMS, namely search strategies and the notion of feasibility. We expect that we need not restrict an exhaustive search to left-deep join trees, but it remains to be seen whether such plans can be efficiently implemented, and how much of an improvement may result. It will also be interesting to consider which heuristics are applicable in this new context, and how they may be integrated with other search strategies.

6 References

[Bow95] J. L. Bower and C. M. Christensen. Disruptive technologies: catching the wave. *Harvard Business Review*, 73(1): 43-53, Jan./Feb. 1995.

[Cha95] S. Chaudhuri, personal communication, April, 1995.

[Hey95a] M. Heytens, S. Listgarten, M.-A. Neimat, and K. Wilkinson. Smallbase: A main-memory DBMS for high-performance applications. Research report, Database Technology Department, Hewlett-Packard Laboratories, September, 1995.

[Hey95b] M. Heytens, S. Listgarten, M.-A. Neimat, and K. Wilkinson. Customizing Smallbase for service control points. Submitted to *HP Journal*.

[Leh86] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the 12th International Conference on Very Large Databases,* Kyoto, Japan, pp. 294-303, Aug. 1986.

[Mun94] M. Cochinwala and J. Bradley. A multidatabase system for tracking and retrieval of financial data. In *Proceedings of the 20th International Conference on Very Large Databases,* Santiago, Chile, pp. 714-721, Sept. 1994.

[Ora95] In-memory database sets store by Alpha technology. *Computing*, Jan. 1995.

[Wha90] K.-Y. Whang and R. Krishnamurthy. Query optimization in a memory-resident domain relational calculus database system. *ACM Transactions on Database Systems*, 15(1): 67-95, Mar. 1990.

[Wil95] A. Wilschut, J. Flokstra, and P. Apers. Parallel evaluation of multi-join queries. In *Proceedings of ACM SIGMOD Conference,* San Jose, California, pp. 115-126, May 1995.