A QUERY PROCESSING STRATEGY FOR THE DECOMPOSED STORAGE MODEL

Setrag Khoshafian George Copeland Thomas Jagodits
Haran Boral Patrick Valduriez

Microelectronics and Computer Technology Corporation
Austin, Texas

Abstract

Handling parallelism in database systems involves the specification of a storage model, a placement strategy, and a query processing strategy. An important goal is to determine the appropriate combination of these three strategies in order to obtain the best performance advantages. In this paper we present a novel and promising query processing strategy for a decomposed storage model. We discuss some of the qualitative advantages of the scheme. We also compare the performance of the proposed "pivot" strategy with conventional query processing for the n-ary storage model. The comparison is performed using the Wisconsin Benchmarks.

1 Introduction

It is becoming increasingly clear that in order to maintain substantial performance improvements in computing systems, parallel processing technology is a must. It is also becoming increasingly clear that although it is relatively easy to propose or even build parallel machines, it is much harder to develop algorithms which exploit the available parallelism. This paper addresses this issue in database systems.

For processing database queries in a highly parallel environment we need the specification of:

(i) *the storage model* which specifies the sets of internal objects manipulated by the system,

(ii) *the data placement strategies*, which specify how the objects of the storage model are distributed over the components (i.e. the multiple repositories) of the architecture, and

(ii) *the query processing algorithms* with respect to the given architecture, data placement, and storage model.

Most database systems use an n-ary storage model (NSM) for a set of records. This approach stores data as seen in the conceptual schema. Also, various inverted files or cluster indexes might be added for improved access speeds. The key concept in NSM is that all attributes of a conceptual schema record are stored together.

An alternative to NSM is the Decomposed Storage Model[5]. DSM is a fully transposed storage model with surrogates included. DSM pairs each attribute value with the surrogate of its conceptual schema record in a binary relation. Furthermore, two copies of each binary relation are stored, one clustered (i.e. sorted or hashed with an index) on each of the two attributes. This provides an alternative to mirroring for data recovery, as well as improved access times since every field has a cluster index.

For data placement in a parallel environment with multiple repositories, it has been shown[10 11] that a declustered organization provides many advantages, both in terms of response time and load balancing. In the declustered placement scheme a set of records pertaining to the same internal object is partitioned horizontally and distributed across a prescribed number of repositories. Declustering is orthogonal to the storage model. In other words, it could be used with both NSM and DSM. DSM provides more opportunities for distributing vertical partitionings of the data. However, since the set objects in NSM (i.e. sets of tuples storing relations) are usually larger than those in DSM (sets of binary tuples storing attributes), we believe declustering (which is strictly a horizontal partitioning strategy) provides more opportunities in parallelism to NSM. It is an interesting research issue to analyze the interplay of these two forms of parallelism on the performance of (parallel) algorithms for NSM and DSM.

For query processing strategies, many proposals have been made for the efficient uni-processor execution of some database operations. In particular, many solutions have concentrated on the relational join because it is an important and costly operation[6]. One recent proposal for the efficient evaluation of join queries is the concept of join indexes as presented in[15 16]. A join index abstracts the join through a binary relation using surrogates.

Overall, in a uni-processor context, query processing strategies based on NSM are well understood[12 18].

There has been some proposals for parallel algorithms in distributed databases[3 13] and database machines[2 14]. However, we believe much more work needs to be done for developing efficient parallel algorithms for database query evaluation.

In this paper we present what we believe to be a novel and promising algorithm (called the pivot algorithm), for processing generic queries with DSM and join indexes. The two main advantages of the pivot algorithm are 1) provision for parallel execution in the multiple phases of the algorithm and 2) generation of small temporary results.

Our ultimate goal is to determine the storage model, placement strategy, and query processing strategy which gives us the best performance advantages in a parallel system. This will entail the specification of a complete list of reasonable approaches, with a comprehensive comparative analysis. We find this noble goal to be too overwhelming. There are too many combinations and strategies. Instead, we chose to implement and compare two representative strategies: 1) NSM with a simple conventional query processing strategy and 2) DSM with the pivot query processing strategy. Furthermore, our implementation was with a uniprocessor VAX 780 system running UNIX. Therefore the comparison will not

CH2407-5/87/0000/0636$01.00 ©1987 IEEE

demonstrate the real and important comparison of the strategies, namely their performance in a parallel environment. Nevertheless we believe it is an important starting point: besides telling us how these two strategies will behave in a uniprocessor environment, it will also help us observe relative performance changes as we increase the parallelism (either through simulation or through developing and running the algorithms on a parallel machine). Also the implementation helped us validate some of the analytical results in[5].

The pivot algorithm and the DSM were implemented on an extension of WiSS (the Wisconsin Storage System)[4] called D-WiSS. WiSS itself provides an NSM storage model. The benchmarks used to compare NSM and DSM were the Wisconsin Benchmarks[1].

The paper is organized as follows: Section 2 describes how relations and queries are represented in DSM. Section 3 describes the pivot query processing algorithm. Section 4 describes the Wisconsin Benchmarks along with our extensions to them. Section 5 presents and discusses the implementation results. Section 6 provides some conclusions.

2 Representation of Relations And Queries In DSM

In this section we present an overall framework for the representation of relations and queries.

We start with a number of "conceptual" relations R1, ..., Rn which constitute the database. We assume each relation is of the form:

Ri(Si,Ai1,...,Aim),

where Si is the *surrogate attribute* of the relation and Ai1, ..., Aim are either atomic (i.e., integer, real, string, etc.) valued or surrogate valued attributes of Ri. There is a unique surrogate value of the surrogate attribute for each tuple of the relation. The surrogate is used to implement the identity of the tuple[8]. The value of the surrogate attribute of a tuple is (system-wide) unique. In most cases, a surrogate valued attribute Aij stores the surrogate attributes of another relation. Therefore surrogate valued attributes are similar to "foreign key" valued attributes in the relational model.

With NSM all the attributes Si, Ai1, ..., Aim of Ri are stored together contiguously. The relation Ri is either clustered or hashed on the surrogate attribute. For the other attributes we have inverted (secondary) indexes.

With DSM, Ri is stored as m binary relations:

Ri1(Si,Ai1),...,Rim(Si,Aim).

Furthermore, there are two copies of each Rik(Si, Aik): one clustered on the surrogate Si and the other on Aik.

Finally, whenever there are two attributes of two relations Ri and Rj which could be joined, we shall assume there exists a binary join relation Rij(Si,Sj) which represents the join. More specifically, if attribute A of Ri and attribute B of Rj are join attributes then:

Rij(Si,Sj) = {(Si,Sj) | ti is a tuple in Ri,
tj is a tuple in Rj,
ti.A = tj.B,
Si is the surrogate of ti,
Sj is the surrogate of tj}.

Rij is called a "join index"[15] of Ri and Rj.

For example, assume the schema of employee and department relations is

Empl(Se: surrogate,
Name: string,
Dept: integer,
Sal: real,
SS#: integer)

Dept(Sd: surrogate,
Num: integer,
Name: string,
Budget: real),

where each Empl.Dept is equal to some Dept.Num to establish the entity relationship that each employee works for some department. The corresponding DSM representation is

Empl-Name(Se: surrogate, Name: string)
Empl-Dept(Se: surrogate, Sd: surrogate)
Empl-Sal(Se: surrogate, Sal: real)
Empl-SS#(Se: surrogate, SS#: integer)

Dept-Num(Sd: surrogate, Num: integer)
Dept-Name(Sd: surrogate, Name: string)
Dept-Budget(Sd: surrogate, Budget: real)

where Empl-Dept is a join index. There are two copies of each of these binary relations, one clustered on each of the two attributes. For value-based relations (all except the single join index Empl-Dept), the first copy is clustered on the surrogate (Se or Sd) and the second copy is clustered on the attribute value. For join index relations (Empl-Dept), one copy is clustered on the Se surrogate and the other on the Sd surrogate. As we shall see, the Pivot algorithm will specify which of the copies to choose.

As an example query (in a Prolog style notation), we will use

ANS(SS#, DN) <==
Empl(Se, Name, DNum, "10k", SS#),
Dept(Sd, DNum, DN, Budget).

which yields the social security number and department name of all employees whose salary is "10k". With DSM, the same query is represented as

ANS(SS#, DN) <==
Empl-Dept(Se, Sd),
Empl-Sal(Se, "10k"),
Empl-SS#(Se, SS#),
Dept-Name(Sd, DN).

3 The Pivot Query Processing Strategy

This section describes the pivot query processing strategy (QPS). First, we provide the rationale behind the pivot QPS. Then, we describe the algorithm. Finally, an example is given.

3.1 Rationale

Most QPS's are developed for single-processor architectures using NSM. These query processing strategies have deep and narrow query trees because they totally order the operations. The pivot QPS is designed for parallel se architectures using DSM. DSM provides the opportunity to have more inter-operation parallelism, as well as smaller intermediate relations which act as operands for subsequent operations.

The pivot QPS has four phases: the select phase, the pivot phase, the value materialization phase and the composition phase. Below, we discuss the pivot QPS advantages in each phase. We assume a declustered placement strategy.

The select phase executes a select operation for every predicate binding in the query, and these are done in parallel. Operands are small because they are binary. There is also parallelism within each select operation when the number of qualifying objects is large (e.g., poor selectivity or range queries) due to declustering. In QPS's designed for single processor machines, select operands are small, a second select which is large (i.e., with a large number of qualifying objects or with no index) is often omitted, and joins are serialized starting with the first select result. With DSM, this second select always has an index and is done in parallel with the first select. Parallelism within each select operation and declustering puts an upper bound on the time to execute large selects.

The pivot phase picks a pivot surrogate and executes an m-way join. The pivot surrogate is any one of the surrogates appearing on the right hand side of the query. However, we can show the most selective surrogate attribute of the relations in the m-way join is the optimal.

The pivot surrogate is used to group the attributes for the final result. The join operands of the m-way join are small because they are only unary or binary relations containing only surrogates. In QPS's designed for single processor machines, joins typically have large operands, do not always have the correct indexing, and are fully serialized. Using DSM, there is a small join index[15] clustered on the desired surrogate for all entity-based joins, so that a full scan is avoided. Declustering causes much parallelism within each join operation. Some join operations can be done in parallel. Communications between processors is minimized by having small operands.

The value materialization phase executes several independent joins in parallel. These join operands are small because they are only binary relations containing only surrogates and their cardinality has been fully reduced due to the combination of all selects. There is always an attribute value relation clustered on surrogate. Declustering causes much parallelism within each join operation. Communications between processors is minimized by having small operands.

The composition join executes an m-way merge join with a large degree of parallelism. The operands are small because they are only binary relations containing only surrogates and their cardinality has been fully reduced due to the combination of all selects.

3.2 The Algorithm

To present the pivot query processing strategy, we start with a generic query. Let

$$R1, \ldots, Rn$$

be conceptual relations. We give an informal definition of a generic Select/Join/Project generic query (SJP), where we assume all the selects are exact matches and all the joins are equijoins. A generic SJP query is of the form:

```
ANS(X1,...,Xp) <== R1(S1,
                      J11,...,J1a,
                      "v11",...,"v1c",
                      X11,...,X1e, ...)
                   ...

                   Rn(Sn,
                      Jn1,...,Jnb,
                      "vn1",...,"vnd",
                      Xn1,...,Xnf, ...).
```

where the Si, Jij, "vij", and Xijattributes are reordered for convenience. The Si's are the surrogates of the relations, the Jij's are join attributes, the "vij"'s are (exact match) values of select attributes, and the Xij's are project attributes. Therefore, each Jij is used to join with at least one Ji'j', and each Xij on the right hand side corresponds to an Xi on the left hand side.

The pivot algorithm proceeds in four phases: the Select Phase, Pivot Phase, Value Materialization Phase, and Composition Phase. Next, we describe the functionalities of each of these phases.

I. The Select Phase: Performs all the selects of the tuples through the DSM copies clustered on the selected attribute values. The output of this phase is a collection of temporaries, where each temporary is a set containing the surrogates of the selected conceptual objects. Therefore, for each attribute A of a relation R which is selected we will evaluate (italics indicates which copy of the DSM sets; thus an italicized value indicates the value clustered copy, an italicized surrogate indicates the surrogate clustered copy):

$$S\text{-}A(S) \;\texttt{<==}\; R\text{-}A(S, \text{“}v\text{”}).$$

II. The Pivot Phase: Selects a pivot surrogate and performs the main m-way join of the query, incorporating selected tuples produced in the select phase, as well as join indexes which are involved in joins for producing the final result. The pivot surrogate could be any of the

surrogates appearing on the right hand side of the query. Although we have developed some heuristics, the determination of the "optimal" pivot surrogate is still an open issue. The pivot surrogate is used to group attributes for the final result. Therefore the output of this phase is a collection of temporaries where each contains a pivot surrogate and the surrogate of a relation which has project attributes. Therefore, if S-A1, ..., S-An constitute all the outputs of the select phase, and JI1, ..., JIm are all the join indexes which are needed to produce the result, then the expression to evaluate the temporaries in the pivot phase is:

```
I1(SP,S1),...,Ik(SP,Sk) <==
            S-A1(SS1),...S-An(SSn),
            JI1(SJ1l,SJ1r),  ...,JIm(SJml,SJmr),
```

where SP is the pivot surrogate, and S1, ..., Sk are surrogates of relations which have project attributes (note that if any of these is SP, the corresponding Ii will be unary). Here we have used a convenient non-first-order notation which has several temporaries on the left hand side of the expression. The interpretation of the expression is that the evaluation of the right hand side produces all the temporaries on the left hand side. Each SSi corresponds to an SJij in a join index on the right hand side, and each Si on the left hand side also corresponds to an SJij on the right hand side of the expression.

We have purposefully ignored specifying two important (and interrelated) aspects of the multiway join: the choice of the cluster copy of the JIi(SJil, SJir) join indexes (i.e., the one clustered on SJil or the one clustered on SJir), and the ordering of the joins. This problem by itself is quite involved and currently we are investigating a number of recent and innovative solutions[9]. We should mention, however, that this ordering and the choice of the cluster copy of the join indexes has no bearing on the rest of the pivot algorithm. It is just a local optimization at the pivot phase. Also note that the pivot algorithm itself is imposing some ordering on the choice of the base relations due to the four phases.

III. The Value Materialization Phase: Materializes the attribute values for the tuples whose surrogates were produced in Phase II. The pivot surrogates are maintained. The output is a collection of temporaries where each temporary is a binary relation containing (pivot-surrogate, project-attribute-value) pairs. Therefore, for each projected attribute A, we will produce a temporary V-A, which contains the projected attribute values, as well as the corresponding pivot surrogate values:

```
V-A(SP,A) <== Ii(SP,Si), R-A(Si,A),
```

where Ii was produced in the pivot phase and R-A is the DSM copy for attribute A clustered on surrogate.

IV. The Composition Phase: Performs an m-way join of all the temporaries produced in Phase III. The join is done on the pivot surrogate. The result is the final answer of the query.

```
ANS(A1,...,An)<==V-A1(SP,A1),...,V-An(SP,An).
```

3.3 A Pivot Example

For the example in Section 2, the four phases are:

I. Select Phase:

```
S-Sal(Se) <- Empl-Sal(Se,"10k").
```

II. Pivot Phase:

```
I1(Se,Sd),I2(Se) <== S-Sal(Se),
        Empl-Dept(Se,Sd),
```

where we have chosen Se as our pivot.

III. Value Materialization Phase:

```
V-SS#(Se,SS#) <== I2(Se),
     Empl-SS#(Se,SS#).

V-DN(Se,DN) <== I1(Se,Sd),
     Dept-Name(Sd,DN).
```

IV. Composition Phase:

```
ANS(SS#,DN) <== V-SS#(Se,SS#),
          V-DN(Se,DN).
```

4 The Wisconsin Benchmarks

In this section we present a brief overview of the Wisconsin Benchmark (WB), and indicate the introduction of an important parameter for our comparative performance analysis. A detailed presentation of WB is given in[1].

The database of the WB consists of 1k tuple, 2k tuple, 5k tuple and 10k tuple relations. Each relation has 15 attributes. Two of these attributes (called "unique1" and "unique2") have distinct values for each tuple. These attributes take 2 byte integer values. There are also 11 other 2 byte attributes called "two", "four," "ten" , "twenty", "hundred" , "thousand", "twothousand", "fivethousand", "tenthousand", "odd", "even". The first nine of these assume as many distinct values as indicated by their names (of course if the cardinality of the relation is smaller than the name of the attribute, the attribute will assume a unique value per tuple. Finally, there are three string valued attributes: "stringu1", "stringu2", and "string" of 52 bytes each. Hence, the total size of a tuple is 182 bytes.

Since the DSM is a surrogate based storage model, we chose unique1 to be the surrogate in each of the relations. Furthermore, we assumed the NSM relations were clustered on unique1. With NSM, we had inverted indexes for selects and joins.

The Wisconsin Benchmarks include select, join, project, as well as update, and aggregate operations. In the select and join queries of these benchmarks the projections were on all of the attributes of the relations. We don't believe this is the typical case, and furthermore this is not reasonable in comparing NSM with DSM (we know DSM will come out a loser each time). This is an important short-coming of these benchmarks, as they favor non-decomposed storage models. Therefore, we introduced an important modification to the select/project queries of the WB: we varied the number of projected

attributes. This enabled us to identify the high watermark value of the number of projected attributes, below which DSM is a winner.

Finally, the 2-way joins in the WB were always 1-1 joins, preceded by a select. Assuming the selected relation was the first relation, we augmented the binary join benchmarks with 1-10, 10-10 and 10-1 joins on 10k tuple relations.

In APPENDIX I we present generic queries for all the runs analyzed in this paper.

5 Implementation Results

To experiment with the pivot algorithm as well as to substantiate some of the claims made in[5], we implemented a fully decomposed storage scheme based on DSM and join indexes.

In[17] we summarized the relative advantages of DSM with Join Indexes[15] over NSM. Our conclusions were 1) that DSM provides better retrieval performance when the number of retrieved attributes is low or the number of retrieved records is medium to high (i.e., greater than the number stored in a disk block), while NSM provides better retrieval performance when the number of retrieved attributes is high and the number of retrieved records is low, and 2) the performance of single attribute modification is the same for both DSM and NSM, while NSM provides better record insert/delete performance.

The implementation results confirmed our analysis. In Figure 1 we give the ratio for number of blocks accessed by NSM/DSM for 1% and 10% selects on 10K tuple relations. The x-axis is the fraction of the total number of projected attributes (the total number is 16 - 1 surrogate and 15 attributes). First observe that the relative performance of DSM is better when the selectivity is worse. In fact with 10% selectivity, DSM will perform better if the fraction of projected attributes is less or equal to approximately 70% of the total number of attributes. With 1% selectivity DSM is better only if the total number of projected attributes is about 30% of the total number of attributes.

To show the performance of the joins with DSM versus NSM, we have analyzed the performance of several types of 2-way joins, namely: 1-1, 1-10, 10-1, and 10-10. These are illustrated in Figure 2. For all these joins, there is a 1% select on the first relation and both relations are 10k tuple relations. First, we note that the total number of tuples retrieved from both relations decreases in the order 1-10, 10-10, 1-1,and 10-1. For example, with 1-1 a total of 200 (100 from each relation) tuples will be retrieved. However, with 1-10 a total of 1100 tuples will be retrieved (100 from the first relation (i.e., the one on which the selection is performed) and 1000 from the second). For the 1-10 and 10-10 joins DSM performs better if the number of projected attributes is approximately less than or equal to 50% of the total number of attributes in both relations. For the 1-1 and 10-1 cases DSM performs better if the total number of projected attributes is less or equal to approximately 30% of the total number of attributes. However, we should emphasize that the x-axis here is the fraction of the total number of attributes from both relations.

The implementation runs showed the superiority of NSM in update queries. However, the NSM implemented had only three inverted indices versus fifteen for DSM. The append query (in the Wisconsin benchmark) requires 7 page accesses with NSM versus 66 with DSM. The delete query requires 7 page accesses with NSM versus 71 with DSM. Finally, the modify query needs 6 accesses with NSM versus 5 with DSM. Append and delete queries are clearly very expensive with DSM. However, considering a fully inverted NSM (i.e., 15 indices in our benchmark) would lead to a DSM performance worse by only a factor 2 for append and delete. DSM always outperforms NSM for modify.

Finally, tables 1-7 provide the total CPU times for each benchmark query (select (1%), 2-way join, 3-way join, append, delete, modify and aggregate sum). The numbers show that, for a given query, the CPU time of DSM is proportionally worse than the IO number. This is because our implementation trades off data compression for CPU overhead. This tradeoff is desirable since the relative speeds of logic and RAM are increasing at a much faster rate than disk speeds. Therefore, in drawing our conclusions we concentrated primarily on the number of page accesses.

6 Summary and Conclusions

In this paper we have presented a novel query processing algorithm called the pivot algorithm. The algorithm is designed for parallel architectures which use the decomposed storage model (DSM). The algorithm consists of four phases: the select phase, the pivot phase, the value materialization phase, and the composition phase. Each of these phases provides opportunities for parallelism either through the possibility of executing multiple operations concurrently (clearly seen in the select phase and the value materialization phase), or through pipelining the temporary results within an execution of the given phase (pivot and composition phase). The philosophy of the pivot algorithm is to partition the execution of a Select/Join/Project query into numerous operations of small operands, attempting to increase the degree of parallelism as much as possible. The total amount of work done could actually increase with the pivot algorithm. With this strategy the total number of joins will be more than that of, say, conventional query processing strategy with the n-ary storage model (NSM). However, each join of the pivot algorithm usually involves smaller operands. The performance and comparison of the pivot algorithm with an NSM query processing strategy on a uni-processor elucidates this point, since it is a comparison of total time.

Therefore we implemented a DSM storage model, ran some benchmarks with the pivot query processing strategy, and compared it with NSM. The implementation of DSM and the pivot strategy was based on WiSS (the Wisconsin Storage Systems), and the benchmarks used to compare the two strategies were the Wisconsin Benchmarks. The results confirmed some of our previous analytical results comparing DSM and NSM. In fact, it is interesting to note that in this uni-processor environment the pivot algorithm seems to be quite competitive with conventional NSM based strategies. We have argued qualitatively, and we believe that the relative advantages of DSM with the pivot algorithm will increase in a parallel architecture. Of course this remains to be shown. This exercise provides just the starting point in determining efficient storage models, placement strategies and algorithms for database systems in parallel architectures. Besides specifying more

precisely the strategy for selecting the pivot surrogate and ordering the m-way join in the pivot phase, we need to analyze the performance of the pivot algorithm in a parallel architecture. Finally, we shall attempt to extend the pivot algorithm to more complex storage models.

Acknowledgement

We would like to thank Marc Smith for his helpful comments on an initial draft of this paper.

Appendix I

In this appendix we present representatives of the queries and updates analyzed in the paper, in an SQL syntax. In these examples we have two 10,000 tuple relations tenktup1 and tenktup2 and one 1,000 tuple relation onektup. For each of these relations unique1 is the surrogate. All joins are preceded by selects

(a) 1% select with project on 10 attributes:

```
range of t is tenktup1

retrieve into ANS(t.uniqu2, t.two, t.four, t.ten,
t.twenty,                              t.hundred,
t.thousand, t.twothousand, t.string1,
       t.string2)
         where (t.unique2 > 301) and (t.unique2 <
402)
```

(b) 1-1, 2-way join query with project on 5 attributes of each relation:

```
range of t is tenktup1
range of w is tenktup2

retrieve into ANS(t.uniqu2, t.two, t.four, t.ten,
t.twenty,                              t.hundred,
t.thousand, t.twothousand, t.string1,
       t.string2, w.uniqu2, w.two, w.four, w.ten,
                          w.twenty, w.hundred,
w.thousand, w.twothousand,
w.string1, w.string2)
         where (t.unique2 = w.unique2) and
(t.unique2 < 1000)
```

(c) 1-10 join with project on 2 attributes:

```
range of t is tenktup1
range of x is tenktup2

retrieve into ANS (t.unique2, t.two, x.unique2,
x.two)
         where (t.unique2 = x.thousand) and
(t.unique2 < 100)
```

(d) 10-10 join with project on 3 attributes.

```
range of t is tenktup1
range of x is tenktup2.

retrieve into ANS (t.unique2, t.two, t.four,
x.unique2, x.two,
x.four)
         where (t.thousand = x.thousand) and
(t.unique2 < 100)
```

(e) 10-1 join with project on 2 attributes:

```
range of t is tenktup1
range of w is tenktup2

retrieve into ANS (t.unique2, t.two, w.unique2,
w.two)
         where (t.thousand = w.unique2) and
(t.unique2 < 100)
```

(f) 3-way join query with projection on 5 attributes:

```
range of o is onektup
range of t is tenktup1
range of w is tenktup2

retrieve into ANS (o.unique2, o.two, o.four,
o.ten, o.string1,
t.unique2, t.two, t.four, t.ten, t.string1)
         where (o.unique2 = t.unique2) and
(t.unique2 = w.unique2)
    and (w.unique2 < 1000) and (t.unique2 <
1000).
```

(g) Append

```
append to tenktup2 (unique1 = 10003, unique2 =
10003, two = 0,
four = 2, ten = 0, twenty = 10, hundred = 50,
          thousand = 688, twothous = 1950,
fivethous =
4950, tenthous = 9950, odd = 1, even = 100,
string1 =
"MxxxxxxxxxxxxxxxxxxxxxxxxxGxxxxxxxxxxxxxxxxxxx
xxxC",
string2 =
"GxxxxxxxxxxxxxxxxxxxxxxxxxCxxxxxxxxxxxxxxxxxxx
xxxA",
string  =
"OxxxxxxxxxxxxxxxxxxxxxxxxxOxxxxxxxxxxxxxxxxxxx
xxxO".).
```

(h) Modify

```
range of 1 is tenktup1

replace t(unique2 = 10001)
         where (t.unique2 = 1491)
```

(i) Deletion

```
range of t is tenktup1

delete t where (t.unique2 = 1004)
```

(j) Aggregate 50M

```
range of x is tenktup2.

retrieve into ANS (x = sum (x.twothous by
x.hundred).
```

References

1. [Bitton et al. 1983] Bitton D., DeWitt D. J., Turbyfill C., "Benchmarking Database Systems: A Systematic Approach", Int. Conf. on VLDB, Florence, September 1983.

2. [Bitton et al. 1983] Bitton D., Boral H., DeWitt D. J., Wilkinson W. K., "Parallel Algorithms for the Execution of Relational Operations", ACM TODS, vol. 8, no. 3, September 1983.

3. [Ceri and Pelagatti 1984] Ceri S., Pelagatti G., "Distributed Databases: Principles and Systems", McGraw-Hill Ed., 1984.

4. [Chou et al. 1985] Chou H. T., DeWitt D. J., Katz R., Klug A., "Design and Implementation of the Wisconsin Storage System", Software Practice and Experience, vol. 15, no. 10, October 1985.

5. [Copeland and Khoshafian 1985] Copeland G., Khoshafian S., "A Decomposition Storage Model", ACM-SIGMOD, Austin, Texas, May 1985.

6. [DeWitt et al. 1984] DeWitt D. J., et al., "Implementation Techniques for Large Memory Database Systems", ACM-SIGMOD Int. Conf., Boston, June 1984.

7. [Ibaraki 1984] Ibaraki T., Kameda T., "Optimal Nesting for Computing N Relational Joins", ACM TODS, vol. 9, no. 3, September 1984.

8. [Khoshafian and Copeland 1986] Khoshafian S., Copeland G., "Object Identity", Proc. of ACM Conf. on OOPSLA, Portland, Oregon, October 1986.

9. [Krishnamurthy, et al. 1986] Krishnamurthy R., Boral H., Zaniolo C., "Optimization Of Nonrecursive Queries", to appear in VLDB-1986, Kyoto, Japan, August 1986.

10. [Livny et al. 1986] Livny M., Khoshafian S., Boral H., "Multi-Disk Management Algorithms", Database Engineering, vol. 9, no. 1, March 1986.

11. [Salem and Molina] Salem K., and Garcia-Molina H., "Disk Striping," International Conference on Data Engineering, Los Angeles, California, February 1986.

12. [Selinger et al. 1979] Selinger et al., "Access Path Selection in a Relational Database Management System", ACM-SIGMOD Int. Conf., Boston, May 1979.

13. [Valduriez 1982] Valduriez P., "Semi-Join Algorithms for Distributed Database Systems", 3rd Int. Symposium on Distributed Databases, Berlin, West Germany, September 1982.

14. [Valduriez and Gardarin 1984] Valduriez P., Gardarin G., "Join and Semi-Join Algorithms for a Multiprocessor Database Machine", ACM TODS, vol. 9, no. 1, March 1984.

15. [Valduriez 1985] Valduriez P., "Join Indices", MCC Tech. Report 1985, to appear in ACM TODS.

16. [Valduriez and Boral 1986] Valduriez P., Boral H., "Evaluation of Recursive Queries Using Join Indices", Proc. of 1st Int. Conf. on Expert Database Systems, Charleston, South Carolina, April 1986.

17. [Valduriez et al. 1986] Valduriez P., Khoshafian S., Copeland G., "Implementation Techniques of Complex Objects", Proc. of 12th Int. Conf. on VLDB, Kyoto, Japan, August 1986.

18. [Wong and Youssefi 1976] Wong E., Youssefi K., "Decomposition -- A Strategy for Query Processing", ACM TODS, vol. 1, no. 3, September 1976.

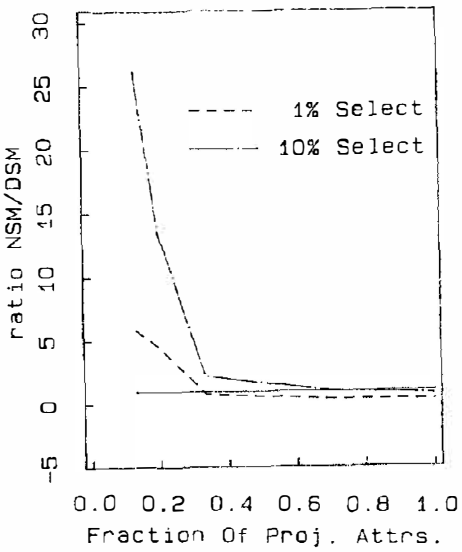

Figure 1

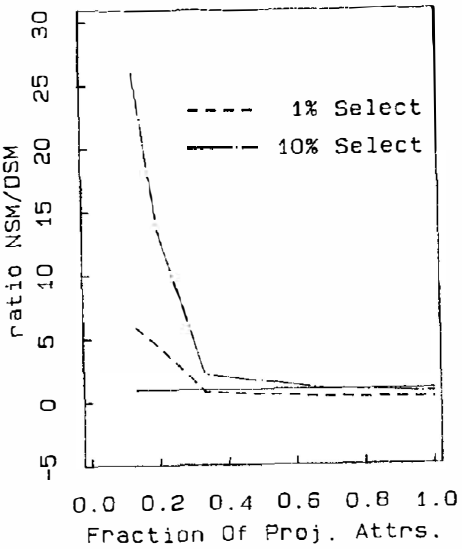

Figure 2

Approximate Time for each Selection Query					
NSM	DSM				
	2attr	3attr	5attr	10attr	Allattr
1.25	0.39	0.74	1.87	3.73	6.39

Table 1

Approximate Time for each 2-Relation Join Query					
NSM	DSM				
	4attr	6attr	10attr	20attr	30attr
23.98	7.53	8.90	16.18	29.14	43.88

Table 2

Approximate Time for each 3-Relation Join Query					
NSM	DSM				
	4attr	6attr	10attr	20attr	30attr
25.08	11.16	14.51	22.41	43.16	95.75

Table 3

Approximate Time for each Append Query	
NSM	DSM
0.17	1.41

Table 4

Approximate Time for each Deletion Query	
NSM	DSM
0.20	8.71

Table 5

Approximate Time for each Modify Query	
NSM	DSM
0.21	0.10

Table 6

Approximate Time for each Aggregate Sum Query	
NSM	DSM
85.98	18.62

Table 7

* Time is in seconds