Providing Better Support for Quantified Queries

Sudhir G. Rao **Antonio Badia** **Dirk Van Gucht**

Abstract

Relational database systems do not effectively support complex queries containing quantifiers (*quantified queries*) that are increasingly becoming important in decision support applications. *Generalized quantifiers* provide an effective way of expressing such queries naturally. In this paper, we consider the problem of processing quantified queries within the generalized quantifier framework. We demonstrate that current relational systems are ill-equipped, both at the language and at the query processing level, to deal with such queries. We also provide insights into the intrinsic difficulties associated with processing such queries. We then describe the implementation of a quantified query processor, Q^2P, that is based on multidimensional and boolean matrix structures. We provide results of performance experiments run on Q^2P that demonstrate superior performance on quantified queries. Our results indicate that it is feasible to augment relational systems with query subsystems like Q^2P for significant performance benefits for quantified queries in decision support applications.

1 Introduction

Several recent papers have pointed out that users often use embedded and correlated queries [6, 18]. These queries aggregate data into sets of entities and - (1) produce a statistical summary by applying aggregate functions on these sets or (2) show complex relationships amongst such sets of entities. The first type of queries are an important subset of the `aggregate queries` that are commonly used in decision support applications. The second type of queries are often expressed as `quantified queries`, i.e., the queries are expressed using quantifiers [8]. Quantified queries are becoming increasingly important in decision support applications in general, and in the insurance and health-care sectors in particular. Typical applications involve determining the `most frequently (least frequently, etc.) used health-care services`, the `percentage degree of donor-recipient type matches` for `organ transplants`, the `cases of diseases not seen in the last five years`, and so on.[1]

In this paper, we address the issue of supporting the class of quantified decision support queries.[2] Our position is that quantified queries are inadequately supported in current relational database management systems. The support that is lacking is both at the language level and in the underlying query processing system. There are two essential problems:

- *SQL's syntax is too restricted to express quantified queries.* While SQL allows subqueries to form sets, the relationships that can be expressed over sets are limited, and must be written in awkward and complicated ways.
- *SQL queries that simulate quantifiers are frequently ill-supported by existing relational query processors.* While relational systems are already equipped with multiple access methods, join algorithms, etc., allowing very good performance on a wide range of queries, there is very little support for quantified queries.

In two recent papers, Hsu and Parker [16] and, independently, Badia, Van Gucht, and Gyssens [1], showed that *generalized quantifiers* in query languages are an effective way of expressing quantified queries naturally. It then becomes important to demonstrate that such quantified queries can be supported effectively. In this paper, we demonstrate the feasibility of augmenting existing relational database systems with appropriate data structures and algorithms for supporting the critical

[1]A recent report on the future of database systems has identified health-care information systems as part of the next generation application areas [12].

[2]Quantified queries are expressed in SQL using the GROUP BY, (NOT) EXISTS, and (NOT) IN clauses. It is significant that the TPC-D (Decision Support) Benchmark Specification by the Transaction Processing Council includes two quantified queries (Q4 and Q16) that involve these clauses for measuring performance in a decision support environment [28].

operations necessary in the evaluation of `quantified queries`.

The main contributions of this paper are - 1) an implementation of a Quantified Query Processor, Q^2P, employing sophisticated algebraic primitives to support generalized quantifiers, 2) an adoption of novel techniques and variations of known structures and algorithms to efficiently support a wide range of generalized quantifiers, and 3) results of performance experiments demonstrating superior performance on quantified queries. The Q^2P system has been designed with a view towards the facilitation of its integration into relational systems.

The paper is organized as follows. In Section 2, we provide a discussion of quantification, and show how it is currently expressed and supported in relational languages. In Section 3, we discuss results of performance experiments run on several relational database systems that demonstrate SQL's shortcomings. In Section 4, we describe the implementation of Q^2P, its data structures and algorithms, and compare the performance of our system with that of relational systems. In Section 5, we describe related work and point out its limitations. Finally, in Section 6 we offer some preliminary conclusions that seem to be supported by our tests and point out further work.

2 Quantification

We are often interested in queries that seek relationships between objects by performing complex operations on sets that are *parameterized* by said objects (i.e. the sets represent information associated with the given object). Calling these queries `quantified` comes from the fact that it is usually necessary to use some form of *quantification* in order to express the proper relationship between the sets. The relationships, however, are not limited to the standard first order logic quantifiers, exists ($\exists$) and all ($\forall$). While they are obviously very important[3], the quantifiers $\exists$ and $\forall$ are only two among many possible. Other examples are: *(exists) exactly one, (exists) at least ten, at least 10%, exactly as many.*

2.1 Generalized Quantifiers

Declarative query languages use embedded sub-queries and some form of *quantification* or *set predicates* to express relationships between sets of the sort just described (SQL [17], OQL [5], CORAL [23], RC/S [21] etc.). It is often argued that these features enhance the declarativeness of the query language. In two recent papers, Hsu and Parker [16] and, independently, Badia, Van Gucht, and Gyssens [1], validated this argument by establishing a link between the phenomenon of subquery syntax in query languages and the theory of *generalized quantifiers* as it was introduced by Barwise and Cooper [2] for natural language formalization.

We illustrate the use of generalized quantifiers through an example. Consider, as part of a health-care study, a database with the relations

```
disease(dname,dtype)
patient-symptom(pname,symptom)
disease-symptom(dname,symptom)
```

We first need the notion of a *set term*. Since generalized quantifiers are relationships between sets, the language needs to express set formation and manipulation in a simple and direct manner. For this we use the standard set abstraction mechanism, where the set of x such that $\varphi(x)$, with $\varphi(x)$ some well-formed formula (with x its free variable) is written as $\{x \mid \varphi(x)\}$.

We will also use *parametric* sets, in which there are *free* variables or *parameters* which will be captured later by a larger set expression. For instance, the set $\{s \mid \texttt{patient-symptom(x,s)}\}$ has the parameter x, and is intended to represent a *set of sets*: for any x that appears as a first argument of the relation `patient-symptom`, the set of all symptoms that x has. Intuitively, it corresponds to a grouping of the relation `patient-symptom` by the first argument.

Our queries will then be set expressions, where the formula can be formed by applying a generalized quantifier to a series of sets. We view quantifiers as binary *set predicates*.[4] For example, the quantifier `some` is such that "`some`(S,T)" is true if and only if "$S \cap T \neq \emptyset$", the quantifier `no` is such that "`no`(S,T)" is true if and only if "$S \cap T = \emptyset$", the quantifier (`not`) `all` is such that "(`not`) `all`(S,T)" is true if and only if "$(\neg)S \subseteq T$",[5] and the quantifier `at least 10%` is such that "`at least 10%`(S,T)" is true if and only if "$|S \cap T| \geq |S| * 0.1$".

For example, consider the queries "*Find the patient-disease pairs* (`p`, `d`) *such that patient* `p` *has* [`some` | `all` | `at least 10%`] *of the symptoms associated with disease* `d`." The `some` query can be formalized as follows (the `all` and `at least 10%` queries are similar and

[3]One may argue that uses of *NOT EXISTS* (which is used to express universal quantification in SQL) are rarely seen. We suspect that, while this is true, the reason is not a lack of usefulness, but the fact that programmers avoid them for two reasons: (1) the usage of *NOT EXISTS* subqueries can get tricky, and (2) they are very inefficiently supported. So most quantifiers, like *all*, *(exists) at least ten*, *at least 10%*, and so on, are *simulated* with the use of the GROUP BY and HAVING clauses in combination with the COUNT function, which goes against the declarative spirit of the language.

[4]All the generalized quantifiers in the example sentences are *binary*. There are, however, natural *unary*, as well as higher arity, generalized quantifiers [30]. A well-known example in the database community of a unary generalized quantifier is SQL's `EXISTS`. In SQL, `EXISTS`(S) is true if and only if the set $S \neq \emptyset$.

[5]The `all` quantifier described in this paper should not be confused with SQL's ALL predicate which is used for quantified comparison [17]. SQL's ALL predicate is used to compare a scalar value with all the values returned by a sub-query.

```
SELECT P.pname, D.dname
FROM   patient-symptom P, disease-symptom D
WHERE [some | all | no | not all]
        (SELECT P.symptom
         FROM   patient-symptom p
         WHERE P.pname = p.pname) IS A
        (SELECT d.symptom
         FROM   disease-symptom d
         WHERE D.dname = d.dname)
```

Figure 1: The `some`, `all`, `no`, and `not all` queries expressed using Hsu and Parker's extended-SQL.

can be obtained by replacing `some` by the appropriate quantifiers).

$$\{p,\ d|\ \texttt{some}(\{s \mid \texttt{patient-symptom(p,s)}\}, \{s \mid \texttt{disease-symptom(d,s)}\})\}$$

Intuitively, these queries are evaluated as follows: group the relation `patient-symptom` by the first argument, so that for each patient name p, we have the set of all the related symptoms; group the relation `disease-symptom` by its first argument, so that for each disease name d, we have the set of all the related symptoms. Finally, for each pair of patient-disease values p, d (as obtained in the previous step), find out if the set associated with p is in the relation [`some` | `all` | `at least 10%`] with the set associated with d. If it is, then the pair p, d qualifies as part of the answer; otherwise it doesn't. Formulas like these, with parameters (free variables), set terms and generalized quantifiers, will be called *GQ-queries.* They are intended to express the `Quantified Queries` that were described before.

We can have any level of nesting in *GQ-queries* for expressing complicated queries. For instance, the query "*Find the patients that have at least 10% of the symptoms of any disease except those symptoms belonging to heart diseases*" is expressed as

$$\{p \mid \texttt{at least 10\%} (\{s \mid \texttt{patient-symptom(p,s)}\} \{s_n \mid \texttt{no}\ (\{d \mid \texttt{disease-symptom(d,}s_n\texttt{)}\})\} \{d \mid \texttt{disease(d,heart-disease)}\})\})\}$$

Hsu and Parker [16] discussed the syntactic limitations of SQL to express GQ-queries. To overcome these limitations, they syntactically extended SQL and provided a translation mechanism from extended-SQL to SQL2 [17].

Reconsider the health-care database and the queries we used as an example: "*Find the patient-disease pairs* (`p`, `d`) *such that patient* `p` *has* [`some`|`all`|`no`|`not all`] *symptoms associated with disease* `d`." In Hsu and Parker's extended-SQL, these queries can be formulated as shown in Figure 1. These queries are used for discussion in the rest of the paper.

```
SELECT P.pname, D.dname
FROM   patient-symptom P,disease-symptom D
WHERE [NOT] EXISTS
          (SELECT *
           FROM   patient-symptom p
           WHERE p.pname = P.pname AND
                      [NOT] EXISTS
                      (SELECT *
                       FROM   disease-symptom d
                       WHERE d.dname = D.dname and
                                  d.symptom = p.symptom))
```

```
SELECT P.pname, D.dname
FROM   patient-symptom P,disease-symptom D
WHERE P.symptom = D.symptom
GROUP BY P.pname, D.dname
HAVING count(P.symptom)
            = (SELECT count(P1.symptom)
                 FROM patient-symptom P1
                 WHERE P1.pname = P.pname)
```

Figure 2: Top: SQL formulation for the [`some | all | no | not all`] queries. Bottom: SQL formulation for the `all` query using GROUP BY and COUNT.

We now present a more real-world application for generalized quantifiers. Consider a scenario in a health-care database, where information is to be gathered on recipient-donor pairs for an organ transplant. An organ transplant from a donor to a recipient is said to have a high probability of success if the match between the donor's and recipient's *Human Leukocyte Antigen* types (*hla_types*) is high [3]. The query using the `at least 90%` quantifier "*Find the recipient-donor pairs* (`r`, `d`) *such that the recipient* `r` *has* `at least 90%` *of his/her hla_types match the hla_types of the donor* `d`." looks for a type match that implies a high probability of success for an organ transplant.

3 Quantified Query Processing in Relational Systems

The patient-disease quantified queries (`some`, `all`, `no`, and `not all` queries) can be translated into SQL using the `EXISTS` unary generalized quantifier. These four queries are expressed by various combinations of the presence or absence of the NOT operation in the SQL query in the top box in Figure 2.[6]

As we stated before, some quantified queries can also be expressed in SQL through the use of grouping and counting. The importance of this technique is that it allows the query optimizer to choose better plans than would be possible with the NOT EXISTS predicate. This strategy is also advocated by Hsu and Parker ([16]). The bottom box in Figure 2 shows the `all`-query expressed using GROUP BY and COUNT in SQL.[7]

[6]Note that in the case of the `some` query a direct join-predicate could be used instead of an EXISTS predicate as shown here.

[7]In practice, the COUNT clauses will require the DISTINCT keyword if duplicates are present. Our experimental setup

We ran these queries in the University-INGRES system and in two commercial relational database management systems using the experimental setup described in Appendix E.[8]

The performance of these systems was very poor for all queries other than the `some` query, and this was already the case for *very small* relations. Using the reformulation of the `all`-query with grouping and counting, the performance of the commercial relational systems improved dramatically.

Unfortunately, reformulating the `no`-query and the `not all`-query via grouping and counting mechanisms ([9]) will not result in a similar speed-up. The reason for this is simple; both these queries contain a hidden *complementation*, which means, for all practical purposes, that a *cartesian product* needs to be attempted to account for the negation.

4 Quantified Query Processing

In this section, we discuss the implementation issues for Q^{2}P, a `Quantified Query Processor`. The main thrust of the implementation is to support operators corresponding to the generalized quantifiers `some`, `all`, `no`, and `not all` directly.

An important criterion for Q^{2}P was that its data structures be implementable as run-time structures so as to facilitate Q^{2}P's integration into relational systems. Relational database literature is replete with data structures and algorithms. In general, these structures are targeted to support particular relational operators (selection, join) or query types (PSJ queries, range queries), and therefore are not directly applicable for our purposes. Our approach is - 1) to use novel variations of known structures, and 2) to integrate these variations into a general system for supporting an algebra on run-time structures for generalized quantifier operators. Furthermore, it is important to observe that our system is closed with respect to this algebra, i.e., expressions in the algebra map a run-time structure to another run-time structure in the system. Hence compositionality of operations is supported which removes the overhead of re-building intermediate structures from their corresponding relations.

We first describe the mapping mechanisms that form a basis for our structures. We then describe the implementation details of these data structures followed by the operator algorithms. Following this we show comparative performance results.

involved a dataset without duplicates.

[8]We are not publishing the performance figures for the commercial RDBMSs due to legal concerns. However, we fully describe the details of the experiments in [25].

4.1 Conceptualization

In relational query processors, hash-based join ($\equiv$ `some`) algorithms have been known to perform well [10]. They map relations at run-time to a 1-dimensional space by hashing on the join attribute values (each hash-value represents a point in this space). However, this scheme is inadequate for supporting the `all`, `no`, and `not all` operators efficiently.

The `all` operator can be supported as a composition of certain basic operators by adopting a general mapping scheme that could be exploited by all the basic operators. Mapping a k-ary relation to a k-dimensional space (i.e., every tuple $[a_{i_1}, a_{i_2}, \ldots, a_{i_k}]$ is mapped to a point $[i_1, i_2, \ldots, i_k]$, $i_j \geq 1$, $1 \leq j \leq k$) provides a general scheme not biased towards any particular operator. The `no` and `not all` operators can be expressed again in terms of simpler operators as `no` $\equiv$ `complement(some)` and `not all` $\equiv$ `complement(all)`. The `complement` operation can be efficiently performed in a 2-dimensional space. The 2-dimensional mapping is achieved by partitioning each k-dimensional point $[i_1, i_2, \ldots, i_k]$ (from the earlier k-dimensional mapping) into two components by grouping particular attributes together to form two composite-attribute groupings which is mapped to a point $[j_1, j_2]$ in a 2-dimensional space.

We use a multi-attribute structure (similar to those in [19, 20]) for providing the mapping to a multidimensional space. We use a boolean matrix structure for providing the mapping to a 2-dimensional space and therefore to support the `complement` operator. The boolean matrix lends itself well to computations that take place in a cartesian-product space, i.e., when the boolean matrix tends to be dense (which is the case since either the input or output matrix for a complement is going to be dense).[9] Other operators are also built in to transform one structural representation to another.

4.2 Storage Structures

Certain auxiliary storage structures are required to aid in mapping a relation to a boolean matrix or to a multidimensional structure, before performing query operations, and vice-versa after performing the operations.[10] These structures also provide information on relation and attribute domain cardinalities which are required at run-time. For the rest of the implementation discussion, assume R is a relation over domains A, B, and C (i.e., $R \subseteq A \times B \times C$) and S is a relation over domains C and D ($S \subseteq C \times D$). For simplicity, all attributes belonging to a common domain in the

[9]Otherwise the high sparsity will prove expensive in terms of disk space usage and I/O operations.

[10]Run-time hash table structures can be used for providing this mapping information [14]. This is an acceptable alternative if these structures can be constructed efficiently with minimal overhead. Otherwise, the incremental cost of maintaining auxiliary storage structures can be amortized in the long run.

database are named after the domain.

The base relations R and S are stored as file structures with tuples $[a_i, b_j, c_k]$ in R and $[c_l, d_m]$ in S $(i, j, k, l, m \geq 1)$. Each one of the attribute domains A, B, C, and D is supported by a domain index (the one for C is shared by both R and S).[11] Each domain index maintains a mapping that associates domain values to points (unique non-negative integers serving as *surrogate* values). For instance, a value $a_i \in A$ is mapped to $i \in [1 \ldots |A|]$. Each attribute domain also has an associated file structure that helps map the surrogate values back to the original attribute domain values. The domain file structure for, say A, holds each value a_i at an offset that is a multiple of i. R' and S' are *surrogate relations* of R and S respectively that are stored as flat file structures containing surrogate tuples (i.e., tuples $[i, j, k]$ corresponding to the original tuples $[a_i, b_j, c_k]$).

Now given a tuple $[a_i, b_j, c_k]$ for R, it is mapped using the domain indexes of A, B, and C to a surrogate tuple $[i, j, k]$ in the surrogate relation R'. R' is later used to construct the run-time structures. After performing query operations, each surrogate tuple $[i, j, k]$ from the output structure is mapped back to a tuple like $[a_i, b_j, c_k]$ using the domain file structures.

We have not discussed the steps involved in insert-operations or the cost of such operations here. This is because Q^{2}P is designed for systems with complex query work-loads as in Decision Support systems. These database systems undergo batch-updates with data from an OLTP environment. Conventional techniques like data-partitioning and pre-sorting would be extremely useful in efficiently populating the storage structures we maintain.

4.3 Run-time Structures

At run-time, the surrogate relation for each relation queried upon is projected on all the attributes required in the query and is used to construct the multidimensional structure. This structure is constructed by partitioning the multidimensional space represented by the projected surrogate relation. The structure is represented by a set of flat-file partitions and a flat-file directory providing pointers to these partitions. Figure 3 shows the multidimensional structure constructed for relation R using its surrogate relation R'. The directory has four columns A', B', C' (corresponding to A, B, and C which are the attributes from R required in the query) and *pointer*. The *pointer* column stores pointers to the partitions in the structure. Each partition has three columns A'', B'', and C''.[12]

[11]Every attribute domain is supported by an index because any attribute can potentially be queried upon.

[12]A relation with k non-join attributes and l join attributes is represented by a $(k + l)$-dimensional structure with a directory having $k + l + 1$ columns having pointers to partitions that have $k + l$ columns.

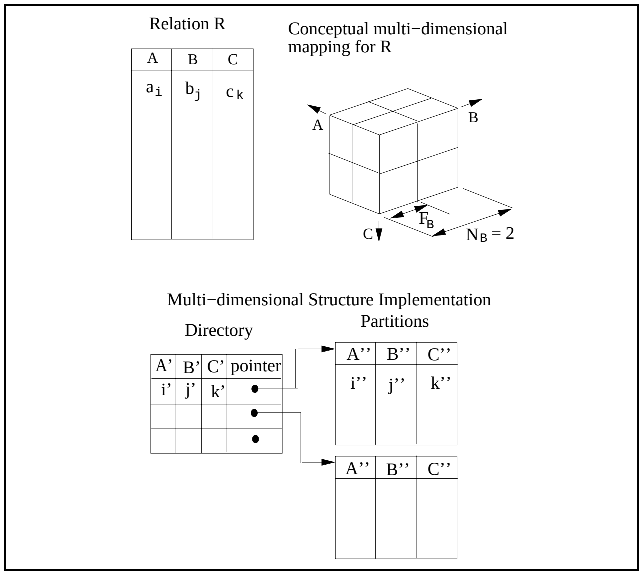


Figure 3: A ternary relation R implemented as a 3-dimensional structure.

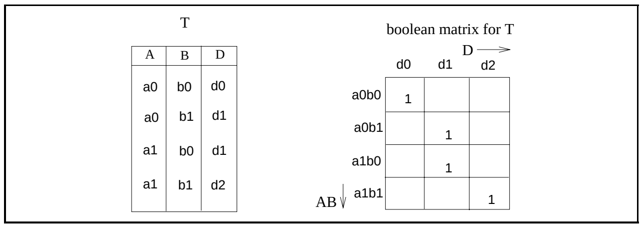
The partitioning is done so as to satisfy the following constraints - 1) each partition should have an adequate number of points mapped from the relation, 2) each partition should fit into main memory, and 3) the directory should fit into main memory.

This multidimensional structure has several advantages in an ad-hoc querying environment. Since it represents the query-time component of the relation in its entirety, it can be used directly for all operators except non-equijoin operators.[13] It presents a partitioning of the relation which can be exploited regardless of the attributes queried upon. Moreover, the closure property on the operators creates output structures that inherit the partitioning.

We next describe the partitioning scheme keeping the constraints in view.

In Figure 3, $N_{\mathcal{A}}$ refers to the total number of divisions along the dimension corresponding to attribute domain $\mathcal{A}$ while $F_{\mathcal{A}}$, the cluster-factor, refers to the unit-length on that dimension. The partitioning is done by applying a simple hash function to each surrogate tuple $[i, j, k]$ so as to split each co-ordinate value into two components, a directory component and a partition component (for instance $i \equiv (i', i'') \equiv (i \ div \ F_A, i \ mod \ F_A)$). The directory tuple $[i', j', k']$ $(= [i \ div \ F_A, j \ div \ F_B, k \ div \ F_C])$ is used to compute the index value $(i' * N_B + j') * N_C + k'$ of the partition

[13]Join-based operators require a normalization step that facilitates bringing matching partitions together. Selection operators (not discussed in this paper) would use the domain indexes and the multidimensional structure for exact match queries and domain indexes (based on sorted order) for range-queries. Projection operators would project out some of the dimensions.



T

A	B	D
a0	b0	d0
a0	b1	d1
a1	b0	d1
a1	b1	d2

boolean matrix for T

AB ↓ / D →	d0	d1	d2
a0b0	1		
a0b1		1	
a1b0		1	
a1b1			1

Figure 4: Relation $T = R$ `some` S represented as a boolean matrix for a subsequent `complement` operation. $\{A, B, C\}$, $\{C, D\}$, and $\{A, B, D\}$ are the schemas of R, S, and T respectively.

to which the tuple belongs. This partition index value serves as the offset for an entry in the directory for the structure for R. The directory tuple is stored at that entry in the directory. The partition tuple $[i'', j'', k'']$ ($= [i \ mod \ F_A, j \ mod \ F_B, k \ mod \ F_C]$) is stored in the partition determined by the computed index value. The surrogate tuple can be easily computed back again as $[i, j, k] = [(i' * F_A + i''), (j' * F_B + j''), (k' * F_C + k'')]$.

The cluster-factors for the joining attributes of the structures participating in a join-based operator are normalized. In Appendix A, we present the steps used to compute the cluster-factor values and for subsequent normalization.

A boolean matrix is constructed only if a complementation is necessary during the computation of a query. For example, for a `no` operation on R and S with C as the join attribute a `some` operation is performed first between R and S, resulting in a multidimensional output structure for the resultant relation T. The next step involves a `complement` operation on T. To achieve this, the boolean matrix for T is constructed with one dimension representing AB formed by grouping the attributes A and B derived from R and the other dimension representing D derived from S (see Figure 4 - refer Appendix B for an intuitive proof on the correctness of this complementation). To construct the boolean matrix structure, we first map the surrogate tuples (multidimensional points) derived from T's multidimensional structure to 2-dimensional points corresponding to $\{AB, D\}$. The 2-dimensional points are then used to set bits in the boolean matrix. The boolean matrix structure is also implemented as a directory providing pointers to partitions which are of the size of a buffer page. The cluster-factor values are made the same for both the dimensions and are determined by the buffer page size and the number of bits that fit into a page.

4.4 Algorithms

The basic operators currently built into the system are the quantifier operator `some` and the operators `group` [14], `complement`, `rel2multi`, `multi2bool`, `bool2multi`, `multi2rel`, and `bool2rel`.[15] The `all`, `no`, and `not all` quantifier operators are implemented in terms of the other operators. The algorithms for the quantifier operators are given in Figures 5 through 8 and the algorithms for the remaining operators are given in Figures 9 through 10 in Appendix C. [16] We use the arrow ($\rightarrow$) notation in the algorithms to refer to substructures or related structures; for e.g., "$multi_0 \rightarrow dir$" refers to the directory of the multidimensional structure $multi_0$.

[14]The `group` operator was built in for two reasons - 1) the `all` operators and other variations like `at least k [%]`, etc., can use a general parameterized algorithm which uses counting mechanisms and 2) for later support for aggregate queries.

```
some(input: multidim_str $multi_0$, multidim_str $multi_1$,
    output: multidim_str $multi$)
    sort_on_join_attributes($multi_0 \rightarrow dir$)
    sort_on_join_attributes($multi_1 \rightarrow dir$)
    $multi \rightarrow dir$ = create_dir($multi_0 \rightarrow dir, multi_1 \rightarrow dir$)
    allocate_memory_for_partitions($multi$)
    for each partition $part_0$ in $multi_0 \rightarrow dir$
        /* Entries matching on the join-attributes are
           looked for.*/
        for each matching partition $part_1$ in $multi_1 \rightarrow dir$
            $hash\_tbl$ = build_hash_tables($part_0$)
            hash_and_probe($part_1, hash\_tbl$)
            /*For successful probes, hash_and_probe saves
              output tuples in an in-memory partition. When
              an in-memory partition is full, it is appended
              to its disk partition and $multi \rightarrow dir$ updated.*/
    return multi
```

Figure 5: `some` operator algorithm

The `rel2multi` operator (see Figure 9 in Appendix C) is used to construct a multidimensional structure given a surrogate relation. The `multi2bool` operator (Figure 11 in Appendix C) is used to transform a multidimensional structure to a boolean matrix. Each tuple from the input multidimensional structure is split into two parts corresponding to the two dimensions and mapped to a pair of non-negative integers. The mappings for the left and right components of all the pairs are stored in two flat composite-domain structures (to be used later - for instance, for relation T in Figure 4 a composite-domain structure is created for AB, whereas D doesn't require one being a single attribute). These pairs are used to set bits in the output boolean matrix structure. The `bool2multi` operator (not shown in any figure) transforms a boolean matrix to a multidimensional structure.

Figure 5 describes the algorithm for the `some` operator. The idea behind `some` is to perform a conventional join. Here, the directories of the two operand

[15]These latter operators permit a migration of a relation between multidimensional and boolean matrix representations depending on whether the computation is in a cartesian-product space or not. We have run experiments on the `some` and `all` operator algorithms for boolean matrices which show good characteristics [24].

[16]Every operator has access to information on the `join`, `non-join`, and `grouping` attributes of the input structures. These aspects are not shown in the algorithms to keep the algorithm description simple.

```
all(input: multidim_str $multi_0$, multidim_str $multi_1$,
    output: multidim_str $multi$)
    $tmp\_multi_{01}$ = some($multi_0$, $multi_1$)
    update_group_parameters($tmp\_multi_0$)
    $tmp\_multi_{01}$ = group($tmp\_multi_{01}$)
    update_group_parameters($multi_0$)
    $tmp\_multi_0$ = group($multi_0$)
    update_join_parameters($tmp\_multi_{01}$)
    update_join_parameters($tmp\_multi_0$)
    return some($tmp\_multi_0$, $tmp\_multi_{01}$)
```

Figure 6: `all` operator algorithm

```
group(input: multidim_str $multi_{in}$,
    output: multidim_str $multi_{out}$)
  $multi_{out} \rightarrow dir$ = create_dir($multi_{in} \rightarrow dir$)
  for each partition $part_{in}$ in $multi_{in} \rightarrow dir$
    allocate_memory_for_partition($multi_{out}$,$part_{in}$)
    $hash\_tbl$ = build_hash_tbl_for_group($part_{in}$)
    /*build_hash_tbl_for_group hashes each tuple in $part_{in}$
      on the grouping attributes and inserts into in-
      memory hash tables. If a duplicate is found it
      increments the count field.*/
    save_hash_buckets_to_partitions($multi_{out}$,$hash\_tbl$)
  return $multi_{out}$
```

Figure 7: `group` operator algorithm

multidimensional structures are first sorted on the join-attributes to bring together matching partitions. Then a join is performed on the matching partition pairs using in-memory hash tables. The `group` operator (described in Figure 7) also uses conventional in-memory hash-tables to group and perform a count. The structure here is upgraded to hold an additional *count* field in the partitions. The `complement` operator (not shown in any figure) retrieves every partition of the input boolean matrix, performs a bit-wise complement operation and places the partition in the output boolean matrix.

The `all` operator (described in Figure 6) uses `some` and `group` operator applications. The `all` quantifier semantics (for any two relations R and S) is captured by the expression

$$[R \texttt{ all } S] = \{x, y \mid \{z|R(x,z) \wedge S(z,y)\} = \{z|R(x,z)\}\}$$

Briefly, the algorithm uses the same semantics as given in the expression below

$$[R \texttt{ all } S] = \{x, y \mid count(\{z|R(x,z)\}) = count(\{z|R(x,z) \wedge S(z,y)\})\}$$

A join is first performed on the two operands. The next grouping operation groups the result on the non-join attributes and keeps a count for each unique tuple. This gives the number of join-attribute values on which the two operands matched to produce duplicate output tuples. The subsequent `group` operation groups the left operand on the non-join attributes coming from the left operand and then performs a count. The resulting

```
[no | not_all](input: multidim_str $multi_0$,
                      multidim_str $multi_1$,
               output: bool_matrix $bool$)
    $tmp\_multi$ = [some | all]($multi_0$, $multi_1$)
    $tmp\_bool$ = multi2bool($tmp\_multi$)
    return complement($tmp\_bool$)
```

Figure 8: [`no | not_all`] operator algorithm

relations from the two grouping operations are then joined on the non-join attributes of the left operand and the *count* field values to produce the result. Note that this algorithm can be extended as a parameterized algorithm to compute other generalized quantifiers like `at least k%`, `at most k%`, `exactly k%`, etc., as follows

$$[R \odot S] = \{x, y \mid (\tfrac{k}{100} * count(\{z|R(x,z)\})) \;\theta\; count(\{z|R(x,z) \wedge S(z,y)\})\}$$

given the generalized quantifier operator $\odot$, the comparision operator θ, and the percentage value k. The `no` and `not all` operator algorithms (see Figure 8) are self-explanatory.

The `multi2rel` (see Figure 10 in Appendix C) and `bool2rel` operators are used to transform a multidimensional and a boolean matrix structure respectively to a flat-file structure and to output the result. These operators basically use the domain file structures to map the surrogate values back to the original attribute values to produce the output tuples.

The complexity analysis of the quantifier operator algorithms is covered in [25]. The I/O time complexity for the `some`, `group`, and `all` operators is $O(n)$, where n is the size of the input relations, unless the output relations dominate the cost. The `multi2bool` operator is $O(n^2)$ for sparse matrices since the output matrix would then be $O(n^2)$ in size relative to the input structure. Otherwise, it is $O(n)$. The `bool2multi` operator has similar characteristics. The `complement` operator, by itself, is linear in the size of the input matrix. Hence, the `no` and `not all` operators are $O(n^2)$ for sparse inputs. During query evaluation, the total space utilized is linear in the size of the relations represented. This is because the multidimensional structures are compact and represent un-complemented relations while boolean matrix structures represent complemented relations. As a consequence of the former, the `rel2multi` operator is $O(n)$. An assumption made for the `bool2rel` and `multi2rel` operators is that the number of domain file structures required for a query at the output stage are very few ([18]) and can almost all be cached into memory for the kind of queries we are targeting.

Q^2P was implemented on top of the EXODUS storage manager [11].

PS	DS	`some`	`all`	`not all`	`no`
2k	2k	1.26	2.77	5.32	4.36
4k	2.1k	1.47	4.60	7.84	6.23
8k	2.2k	2.17	6.36	14.32	11.56
16k	2.3k	3.63	11.67	24.94	22.40
32k	2.4k	7.09	22.55	49.57	51.85
64k	2.5k	13.46	47.94	103.20	111.8
128k	2.6k	26.43	88.65	206.65	280.5
256k	2.8k	62.83	193.6	468.42	988.7
512k	2.9k	157.2	410.0	1045.9	2904.2

Table 1: This table gives performance figures for our Q^2P system on the `patient-disease` queries. The PS and DS columns refer to the size (in tuples) of the `patient-symptom` and `disease-symptom` relations respectively. All timing figures are in seconds.

Pat-Inv	`all`	`not all`
2k	4.93	5.41
4k	6.64	9.63
8k	12.41	16.43
16k	23.94	28.10
32k	46.32	57.59
64k	90.50	116.9
128k	189.9	225.4
256k	355.3	502.5

Table 2: The table gives performance figures for our Q^2P system on queries using a ternary relation. `Pat-Inv` is the Patient-Investigation relation with schema {patient,doctor,investigation}. `Lab-Investigation` is the second input relation with schema {lab,investigation}. The query used for the experiment was: *List the (patient,doctor,lab) triplets where the patient had* [`all`|`not all`] *of their investigations (as prescribed by the doctor) done in the same lab.* The `Lab-Investigation` relation had 5k tuples for all the experiments. All timing figures are in seconds.

4.5 Performance Results

We ran the queries described in Section 3 on Q^2P for comparison. The experimental setup is described in Appendix E. The queries were run by composing the appropriate operators on the system. We compared the results obtained on these experiments for the commercial relational database systems with those of our Q^2P system. Table 1 gives all the performance figures for Q^2P (all timing figures include the time required for building the run-time structures and transforming them appropriately). To demonstrate the generality of the system, in Table 2 we show results for the `all` and `not all` queries run on a ternary relation.

The performance behavior of Q^2P for `some`, `all`, and `not all` are linear all the way upto half a million tuples. The non-linear behavior for the `no` query, however, is evidence that we are working in a cartesian-product space. Using boolean matrices for `no` and `not all` queries pushes the region of non-linear performance behavior beyond relations several times larger than is possible with current relational systems. With the inclusion of selection operations in queries, as would be the case with most queries in practice, the region of non-linear behavior gets pushed further by a factor determined by the selectivity of the restriction.

Our comparative experiments indicated the following:

- `some` ($\equiv$ join) query – Q^2P was better than the relational database systems by a factor that was not very significant, i.e., relational database systems were quite efficient on joins.
- `all` query (GROUP BY formulation) – Q^2P was better than relational database systems by a significant factor, indicating that relational database systems can be improved much further.
- `all`, `no`, and `not all` queries (NOT EXISTS formulation) – Q^2P was better than the relational database systems by several orders of magnitude, which indicates that relational database systems can benefit a good deal by employing techniques from Q^2P.

Our figures in the table give an idea of the kind of performance that can be obtained on quantified queries with structures suited for quantifier operations.

5 Related Work

The research that applies to the problem of efficiently supporting quantified and other complex queries has been done in at least three areas: extended query languages, extended algebraic primitives and new implementations.

As for **extended query languages**, there has been work on extending relational languages with primitives that can manipulate sets. An early example is Ozsoyoglu and Wang's work [21], where the operations on sets are limited. Sets have also been incorporated into logical query languages (for a recent example, see [23]). However, the idea of using the general notion of quantifier and explicitly making sets part of the language does not seem to appear until [16] and [1].

As seen in the examples, Hsu and Parker try to integrate their system on top of SQL, and propose a translation of their extension back into SQL2. While this has a clear implementation advantage, we have seen that the queries resulting from the translation process are highly inefficient, and most relational systems do not support them very well.

As for **extended algebraic primitives**, there has been considerable work at the algebraic level to support more sophisticated primitives. Carlis presents a division operator that allows a wide variety of set operations to be considered [4]. Dadashzadeh [7] presents an extended division operator (GCD) that also allows more flexibility on the set operations and relates dividend and divisor values. Dayal presented generalized join and aggregate primitives for nested queries with quantifiers [9]. However, there is no discussion on how these operators perform in practice or on queries with more than one nesting level. The approach in [1] tries to be more general, by considering the quantification as an explicit operation on the sets, and several ways to construct the sets involved. As a result, the kind of universal quantification queries captured using relational division are a special case of the `all` quantifier.

As for **efficient implementations**, there is work on new structures with algorithms, and query optimization. With respect to the former, researchers and database system vendors have always looked at alternative structures for performance based on the requirements of their areas of application or on the kind of queries targeted. Scientific and statistical databases have used multidimensional structures for performance on statistical queries [26]. Multidimensional database products like Essbase from Arbor Software, Express from Oracle and so on are available in the market that target aggregate queries in an on-line analytical processing environment. Multidimensional structures have been suggested for support for range-queries for relational databases [19, 20]. Sybase, a major relational database vendor, has in recent times offered a product, Sybase IQ, which uses bit-vector indexes for better performance on queries with multiple restriction and join predicates [27]. Bit-mapped join indexes have also been proposed for multi-way joins in relational systems [15]. As far as we know, though, no structure is specially devoted to support quantified queries.

Graefe presented four algorithms for relational division with performance experiments [14]. Relational division, unlike our `all` operator, is an asymmetric operator in that no part of the result can come from the divisor. Observe also that other attributes in the divisor cannot be asked to match the grouping attribute of the dividend, as it happens on Dadashzadeh's GCD operator. Finally, observe that both Dadashzadeh and Carlis propose a limited number of set operators, while our approach for `all` allows the implementation of many different operators. Our general algorithm for `all` uses hash-based aggregation techniques similar to those of Graefe's, but with different structures.

With respect to optimization, there have been many papers on query transformations and optimizations for performance benefits [9, 13, 22]. As Pirahesh et. al. point out in [22], there has not been any implementation for performance testing before theirs. Even their paper does not discuss performance benefits for the NOT EXISTS and EXCEPT predicates in SQL.

6 Conclusion

Our primary objective was to explore the feasibility of extending relational systems for supporting `quantified queries` efficiently. We used the generalized quantifier framework as the basis for supporting `quantified queries`. We implemented Q^2P, a system based on multidimensional and boolean matrix structures, with suitable algorithms for the generalized quantifier operators `some`, `all`, `no`, and `not all`. We also showed that quantifiers like `at least k%`, `at most k%`, `exactly k%`, etc., can be supported easily by a parameterized version of the `all` operator algorithm. The main contributions of the paper are:

- A single framework under which `quantified queries` using a wide range of quantifiers can be processed efficiently.
- Introducing an algebra on run-time structures for the potential benefits of lowering overheads of rebuilding such structures.
- Demonstrating that a combination of multidimensional and boolean matrix structures with associated algorithms exhibit a performance that is better by several orders of magnitude over similar SQL queries using NOT EXISTS, and by significant factors over SQL formulations that use GROUP BY and COUNT mechanisms.

Our implementation experience and comparative performance studies on `quantified queries` indicate that: 1) relational systems lack good support for such queries, 2) alternative structures like multidimensional and boolean matrix structures can provide significant performance benefits for such queries, and 3) relational systems can be more competitive in the decision support market by incorporating query sub-systems like Q^2P.

Acknowledgement

We would like to thank Edward Robertson, Manoj Jain, and Mehmet Dalkilic for reading the paper and providing useful feedback. Our work was partly influenced by some early work with Latha Colby and Lawrence Saxton.

References

[1] Badia, A., Gyssens, M., and Van Gucht, D. Query Languages with Generalized Quantifiers. In *Applications of Logic Databases*, R. Ramakrishnan, Ed. Kluwer Academic Publishers, 1995, pp. 235–258.

[2] Barwise, J., and Cooper, R. Generalized Quantifiers and Natural Language. In *Linguistic and Philosophy* (1981), pp. 159–219.

[3] Braunwald, E., J.Isselbacher, K., G.Petersdorf, R., Wilson, J. D., Martin, J. B., and S.Fauci, A., Eds. *Harrison's Principles of Internal Medicine*, 11 ed. McGraw Hill Book Company, 1987.

[4] Carlis, J. V. HAS: A Relational Algebra Operator, or Divide is Not Enough to Conquer. In *IEEE Data Engineering* (1986), p. 254.

[5] Cattell, R., Ed. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, 1994.

[6] Chaudhuri, S., and Shim, K. Including Group-By in Query Optimization. In *Proc. of the 20th Int'l Conf. on VLDB* (1994), pp. 354–366.

[7] Dadashzadez, M. An Improved Division Operator for Relational Algebra. *Information Systems 14*, 5 (1989), 431–437.

[8] Dayal, U. Processing Queries With Quantifiers: A Horticultural Approach. In *Proc. of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Atlanta* (1983).

[9] Dayal, U. Of Nests and Trees: A Unified Approach to Processing Queries that contain Nested Subqueries, Aggregates, and Quantifiers. In *Proc. of the 13th Int'l Conf. on VLDB* (1987), pp. 197–208.

[10] DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R., and Wood, D. Implementation Techniques for Main Memory Database Systems. In *Proc. of ACM-SIGMOD Int'l Conf. on Management of Data* (1984), p. 1.

[11] EXODUS. Using the EXODUS Storage Manager V3.0. unpublished,included in the EXODUS Storage Manager Software Release.

[12] *Database Research: Achievements and Opportunities Into the 21st Century.* Report of an NSF Workshop on the Future of Database Systems Research, May, 1995. Avi Silberschatz, Mike Stonebraker, Jeff Ullman, editors.

[13] Ganski, R. A., and Wong, H. K. T. Optimization of Nested SQL Queries Revisited. In *Proc. of ACM-SIGMOD Int'l Conf. on Management of Data* (1987), pp. 23–33.

[14] Graefe, G. Relation Division: Four Algorithms and Their Performance. In *Proc. of IEEE Int'l Conf. on Data Eng.* (February 1989), p. 94.

[15] Graefe, G. Multi-Table Joins Through Bitmapped Join Indices. *SIGMOD Record 24*, 3 (September 1995), 8–11.

[16] Hsu, P., and Parker, D. Improving SQL with Generalized Quantifiers. In *Proc. of the 10th Int'l Conf. on Data Engineering* (1995).

[17] International Organization of Standardization (ISO). *Database Language SQL.* Document ISO/IEC 9075:1992.

[18] Lu, H., Chan, H. C., and Wei, K. K. A Survey on Usage of SQL. *SIGMOD Record* (1993), 60–65.

[19] Nievergelt, A., and Hinterberger, H. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems* (1984), 38–71.

[20] Otoo, E. J. A Multidimensional Digital Hashing Scheme for Files. In *Proc. of ACM-SIGMOD* (1985), pp. 214–229.

[21] Özsoyoğlu, G., and Wang, H. A Relational Calculus with Set Operators, Its Safety, and Equivalent Graphical Languages. *IEEE Transactions on Software Engineering* (1989), 1038–1052.

[22] Pirahesh, H., Hellerstein, J. M., and Hasan, W. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proc. of ACM SIGMOD Int'l Conf. on Management of Data* (1992), pp. 39–48.

[23] Ramakrishnan, R., Seshadri, P., Srivastava, D., and Sudarshan, S. The CORAL User Manual: A Tutorial Introduction to CORAL. Computer Science Department, University of Wisconsin-Madison, available via anonymous ftp from `ftp.cs.wisc.edu` in the directory `coral/doc.`, Software Release.

[24] Rao, S. G., Badia, A., and Van Gucht, D. Processing Queries Containing Generalized Quantifiers. Technical Report 428, Indiana University, April 1995.

[25] Rao, S. G., Badia, A., and Van Gucht, D. Efficient Processing Support for Quantified Queries. Technical Report 452, Indiana University, February 1996.

[26] Shoshani, A., and Wong, H. K. T. Statistical and Scientific Database Issues. *IEEE Transactions on Software Engineering 11* (1985), 1040–1047.

[27] Sybase, Inc. *Interactive Query Accelerator - Too Much Data, Not Enough Information.* Sybase White Papers, available on the web as of March 26, 1996 at http://www.sybase.com/toc.html.

[28] Transaction Processing Council (TPC). *TPC Benchmark^{TM} D (Decision Support) Standard Specification*, May 1995. Revision 1.0.

[29] Ullman, J. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, 1989.

[30] Westerstahl, D. Quantifiers in Formal and Natural Languages. In *Handbook of Philosophical Logic*, D. Gabbay and F. Guenthner, Eds. D. Reidel Publishing Company, 1989, pp. 1–131.

Appendix

A Computing Cluster-factors

The cluster-factors and number of divisions (for each dimension) are computed by taking into account main memory capacity and the cardinalities of R, A, B, and C (see Figure 3). For any attribute domain $\mathcal{A}$, $F_{\mathcal{A}}$ and $N_{\mathcal{A}}$ are related by the equation $F_{\mathcal{A}} = \frac{|\mathcal{A}|}{N_{\mathcal{A}}}$ where $N_{\mathcal{A}} \geq 1$. Let P represent the allowable size of a partition considering main memory capacity. Let the *number of partitions in R be* $N = \frac{|R|}{P}$.

$$\begin{aligned} N = N_A * N_B * N_C &= \lceil \tfrac{|A|}{F_A} \rceil * \lceil \tfrac{|B|}{F_B} \rceil * \lceil \tfrac{|C|}{F_C} \rceil \\ &= \lceil \tfrac{|A|}{2^{E_A}} \rceil * \lceil \tfrac{|B|}{2^{E_B}} \rceil * \lceil \tfrac{|C|}{2^{E_C}} \rceil \end{aligned}$$

where $E_A, E_B, E_C \geq 0$. F_A, for instance, is calculated as follows:

$$E_A = \frac{|A| * \log_2(\frac{|A|*|B|*|C|}{N})}{|A| + |B| + |C|}, \; F_A = 2^{E_A}$$

```
rel2multi(input: surrogate_rel $rel$,
          output: multidim_str $multi$)
  $multi \rightarrow cluster\_factors$ = compute_factors($rel$)
  $multi \rightarrow partition\_num$ = determine_partitions($multi$)
  $multi \rightarrow dir$ = create_dir($multi \rightarrow dir$)
  allocate_memory_for_partitions($multi$)
  for each $tuple$ in $rel$
     $dir\_component$ = compute_dir_component($tuple$)
     $part\_component$ = compute_part_component($tuple$)
     insert_dir_component($multi \rightarrow dir, dir\_component$)
     insert_part_component($multi \rightarrow dir, part\_component$)
     /* Insert partition component uses the in-memory part-
        itions and copies it to the corresponding disk partition
        when any of the in-memory partition becomes full.*/
  return $multi$
```

Figure 9: `rel2multi` operator algorithm

```
multi2rel(input: multidim_str $multi$)
    for each attribute $attr_i$ associated with $multi$
       $dom\_file\_str[i] = attr_i \rightarrow domain\_file\_structure$
    for each directory tuple $dir\_tup$ in $multi \rightarrow dir$
       for each partition tuple $part\_tup$ in $dir\_tup \rightarrow partition$
          $surro\_tuple$ = surrogate_tuple($dir\_tup, part\_tup$)
          for each attribute $attr_i$ in $rel$
             $tuple[i] = dom\_file\_str[surro\_tuple[i]]$
          print_tuple($tuple$)
          /* Currently, this results in an output tuple stream.
             Alternatively, a relation with the original tuple
             can be constructed. */
```

Figure 10: `multi2rel` operator algorithm

This scheme partitions the tuple space with respect to an attribute domain based on its cardinality relative to the cardinalities of the other attribute domains. The above formulas can be extended to relations of any arity. For join-based operators (like `some` and `all`), the cluster-factors for join-attributes of the participating relations are normalized. When the structures are not already constructed the cluster-factor values are made the same, otherwise a structure normalization stage should be incorporated. The experiments described in this paper have not required this latter stage.

B Correctness of `complement` for `no` and `not all`

Let T be the result of a `some` or `all` operation on R and S (see Figure 4). The boolean matrix for T will have AB along one dimension and D along the other. The `complement` operation on a boolean matrix such as T is correct only when the functional dependencies or the derived relationships amongst the attributes are not broken. Note that in T the functional dependency or the relationship between A and B (from R) is not destroyed. After the binary operation between R and S, we would have a new derived relationship between AB and D and `complement` seeks to introduce all those (AB, D) pairs that are not present in T while continuing to hold the relationship between A and B intact. Thus, a `complement` after `some` or `all` will always be correct.

C Algorithms for Q^{2}P's Operators

Algorithms for `rel2multi`, `multi2rel`, and `multi2bool` are given in Figures 9, 10, and 11.

D Analysis of Algorithms

Let R_1 and R_2 be the two relations to be operated upon. Let n_i be the number of partitions in the structure for R_i. Since the multidimensional structure is compact, $n_i \propto |R_i|$. Let the domain for the composite values of the non-join attributes be represented by Dom_{nj_i} in R_i. Let the domain for the composite values of the join-attributes be represented by Dom_j. Let $|Dom_j|$ and

```
multi2bool(input: multidim_str $multi$,
           output: bool_matrix $bool$)
   $bool \rightarrow cluster\_factors$ = determine_factors()
   $bool \rightarrow dir$ = create_bool_dir($bool$)
   $bool \rightarrow dim_0$ = init_dimension0($multi \rightarrow attributes$)
   $bool \rightarrow dim_1$ = init_dimension1($multi \rightarrow attributes$)
   $rel$ = multi2rel($multi$)
   /* bld_composite_domain_str projects out $multi$ on
      attributes corresponding to the given dimension and
      sorts or hashes the projected relation to assign unique
      surrogate values and then constructs the composite
      domain file str.*/
   if composite_attr_domain_reqd($bool \rightarrow dim_0$)
     $bool \rightarrow dim_0 \rightarrow dom\_str$ = bld_composite_domain_str(
                                   $multi, bool \rightarrow dim_0$)
     $rel$ = map_dim0_attr_values_to_surrogates($rel$,
                                   $bool \rightarrow dim_0 \rightarrow dom\_str$)
   if composite_attr_domain_reqd($bool \rightarrow dim_1$)
     $bool \rightarrow dim_1 \rightarrow dom\_str$ = bld_composite_domain_str(
                                   $multi, bool \rightarrow dim_1$)
     $rel$ = map_dim1_attr_values_to_surrogates($rel$,
                                   $bool \rightarrow dim_1 \rightarrow dom\_str$)
   /* create a temporary 2-dimensional "non-boolean"
      structure $temp$ (to streamline I/O accesses). */
   $temp$ = rel2multi($rel$)
   for each partition $part$ in $temp$
      $bool\_part$ = allocate_space_for_partition($bool$)
      for each pair $(i, j)$ in $part$
         set_bit_ij($bool\_part$)
      save_partition($bool, bool\_part$)
   return $bool$
```

Figure 11: `multi2bool` operator algorithm

$|Dom_{nj_i}|$ be in terms of number of partitions. For the analysis, let $\frac{n_i}{|Dom_j|} = k_i$, $i = 1, 2$ i.e. every k_i directory entries in the structures agree on all the `join`-columns.

D.1 I/O Costs for the `some` operator

During the `join`, matching pairs of partitions from R_1 and R_2 are brought together with a merge-join on the directories of their structures. At each stage during the merge-join $k_1 * k_2$ matching pairs of partitions are brought together. The total number of matching pairs is then given by $\sum_{Dom_j} k_1 * k_2 = |Dom_j| * k_1 * k_2 = \frac{n_1 * n_2}{|Dom_j|}$. The total I/O cost is proportional to the total number of matching pairs of partitions. Now if k_i is constant, $i = 1, 2$, then the I/O cost is $O(n_j)$ ($j = 1, 2$ and $j \neq i$) in complexity. If in the worst case $|Dom_j|$ is constant, then the I/O cost is $O(n^2)$ in complexity.[17]

D.2 I/O Costs for the `complement` operator

Since either the output matrix or the input matrix is dense in this case the I/O cost is dominated by the dense matrix i.e. the complexity is $O(\frac{n_1 * n_2}{|Dom_j|})$. Note that a `complement` as in the case of `no` and `not all` operators is performed after an earlier `some` or `all` operation. Hence, the reference to n_1 and n_2.

D.3 I/O Costs for the `all` operator

The first `some` operation (refer Figure 6) costs $O(\frac{n_1 * n_2}{|Dom_j|})$ depending on whether $|Dom_j|$ is a constant or varies with n. The cost of the subsequent grouping and aggregation operations is of the same order of magnitude as the size of the input structures. The cost of the last `some` operation is dominated by the output cost of the first `some` operation.

E The Experimental Setup

The experimental setup is described below in terms of the platform configuration and the benchmark dataset.

E.1 System Configuration

Table 3 gives details of the platform used for the experiments. The buffer size used for the systems was 12.5 MB. For the commercial systems, this translates to configuring the system's resource requirements appropriately. In the case of Q^{2}P, the Exodus Storage Manager was configured to have a 12.5 MB buffer with approximately 1 MB used for the server cache and the balance for the client local cache. The client and server processes were run on the same machine for all systems to avoid network related variances in the performance.

[17]Note that $\frac{n_1 * n_2}{|Dom_j|}$ is the cost of generating the output relation by the cost model given in chapter 11 of [29]. This implies that conventional relational `join` algorithms also are subjected to this $O(n^2)$ behavior in such a case.

System Configuration / Parameter	Device / Value
Platform	SPARCstation 10
Operating System	SunOS (version 4.1.3_U1)
Main Memory	64 MB
Disk	Fujitsu M2266SA (1 GB)

Table 3: Parameters and configuration for the experimental platform.

E.2 The Benchmark Dataset

The relations used in the benchmark are as follows:

`patient-symptom`(pname,symptom)
`disease-symptom`(dname,symptom)

Each of the attributes was defined to be a string of size 16 chars. The data sets were randomly generated to satisfy certain constraints. Each patient had 5 symptoms on average. The initial domain sizes were as follows: $|pname| = 400$, $|dname| = 400$, and $|symptom| = 1000$. During the course of the experiments, the patient domain (*pname*) size grew proportionally with the size of the `patient-symptom` relation while the other domains and the `disease-symptom` relation grew more slowly to better simulate realistic situations. The `dname` and `symptom` domains, and the `disease-symptom` relation grew by a factor of 1.05 at each iteration of the experiment, for e.g., at iteration i $|dname| = (1.05)^i * 400$ and so on. The size of the `patient-symptom` relation was doubled at each iteration starting with $|$`patient-symptom`$| = 2000$.

A C++ program was used to generate the datasets for the `patient-symptom` and `disease-symptom` relations. The random number generator 'random()' from the C Standard Library was used to generate the attribute values as numbers. The datasets were saved in Unix files corresponding to each relation. The Unix utilities *sort* and *uniq* were used to eliminate duplicates from these files. The relations in each database management system were populated from the corresponding files using utilities provided by each system.

The C++ program that was used to generate the dataset is given next.

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>

/* This program creates a dataset each for the patient-symptom relation and
   for the disease-symptom relation. The dataset is stored in Unix files
   named as patient_symptom and disease_symptom. Each of the files contains
   data in two columns meant for the corresponding attributes in the
   relations. The program expects the size of the patient-symptom relation
   as a command-line argument (for e.g., to generate a dataset with 1000
   tuples for the patient-symptom relation the command
   ‘<executable-for-this-program> 1000’ is issued.

   The disease-symptom relation size and the domain sizes for disease and
   symptom are progressively increased by a factor of 1.05. This is
   achieved by initially storing a value of 1.0 in a file named ‘constant’.
   The program is run starting with 2k tuples for the patient-symptom
   relation. For each run of the program, the program multiplies the
   constant value in ‘constant’ by 1.05. This value is then used for the
   next run of the program.
*/
/*
   ‘symptom_num’ is the initial symptom domain size.
*/
const unsigned int symptom_num = 1000;
/*
   ‘disease_num’ is the initial disease domain size.
*/
const unsigned int disease_num = 400;
/*
   ‘initial_ds_num’ is the initial number of tuples in the disease-symptom
   relation.
*/
const unsigned initial_ds_num = 2000;
const float constant = 1.0;

int main(int argc, char *argv[])
{
        unsigned int i,ps_num;
        /* ps_num gives the new patient-symptom relation size while
           ds_num, dis_num, and symp_num give the new disease-symptom,
           disease-domain, and symptom-domain sizes respectively.
        */
        unsigned int dis_num, symp_num, ds_num,tmp;
        float cnst;
        /* Output file stream for the patient_symptom file. */
        ofstream patient_symptom("patient_symptom");
        /* Output file stream for the disease_symptom file. */
        ofstream disease_symptom("disease_symptom");
        /* Input file stream for the constant file. */
        ifstream in("constant");

        if(argc != 2)
        {
                cout << "Usage: <executable> <patient-symptom-size> " << endl;
                exit(0);
        }
        else ps_num = atoi(argv[1]);

        if(!in.eof())
                in >> cnst;
        else cnst = constant;

        symp_num = (unsigned int)(symptom_num * cnst);
        dis_num = (unsigned int)(disease_num * cnst);
        ds_num = (unsigned int)(initial_ds_num * cnst);
        srandom(1);

        /* Using ‘random()%(ps_num/5)’ for patient ensures that the
           number of patients does not exceed ‘ps_num/5’ and consequently
           every patient has on average 5 symptoms.
        */
        for(i=0;i< ds_num;i++)
        {
                patient_symptom << random()%(ps_num/5) << ' ';
                patient_symptom << (tmp=(random()%symp_num)) << endl;

                disease_symptom << tmp << ' ';
                disease_symptom << random()%dis_num << endl;
        }
        /* The first loop was to ensure that some symptoms across patients
           and diseases were common. This will ensure that the result for
           very complex queries is non-null for the most part.
        */
        for(i=ds_num;i< ps_num;i++)
        {
                patient_symptom << random()%(ps_num/5) << ' ';
                patient_symptom << random()%symp_num << endl;
        }
        /* Output file stream for the constant file for generating the next
           value of the constant. */
        ofstream nextval("constant");

        /* Store next value of the constant. */
        nextval << (float)(cnst*1.05);
        /* Display details of sizes of relations for this run. */
        cout << "patient-symptom: " << ps_num << " tuples " ;
        cout << "disease-symptom: " << ds_num << " tuples" << endl;
}
```