Processing Queries Containing Generalized Quantifiers

Sudhir Rao, Antonio Badia, and Dirk Van Gucht
Indiana University [*]

Abstract

We considered the problem of processing queries that contain *generalized quantifiers.* 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 propose a boolean matrix approach which establishes the feasibility of building systems that can process queries with generalized quantifiers efficiently, and we provide insights into the intrinsic difficulties associated with processing such queries.

1 Introduction

Numerous existing query languages (SQL [25], OQL [6], CORAL [24], RC/S [23] etc.) allow queries with embedded sub-queries as well as sub-query comparison statements.[1] It is often argued that these features enhance the declarativeness of the query language. In two recent papers, Hsu and Parker [17] and, independently, Badia, Van Gucht, and Gyssens [2], validated this argument by establishing a link between the phenomenon of sub-query syntax in query languages and the theory of *generalized quantifiers* as it was introduced by Barwise and Cooper [3] in linguistics.[2]

Taking as established that generalized quantifiers are crucial in query languages, it becomes vital to demonstrate that queries containing generalized quantifiers (GQ-queries) can be supported effectively. The present paper takes a step in this direction. As will become clear, however, it will be necessary to augment conventional languages, such as SQL, with appropriate mechanisms to allow the formulation of GQ-queries, and, it will be necessary to augment existing file organizations and access methods with appropriate data structures and algorithms to support critical operations necessary in the evaluation of GQ-queries.

> *We are of the opinion that, as long as query languages and query processors continue to offer limited support for GQ-queries, the database community can not claim to have solved the query processing problem in the* decision support *area.*

The paper is organized as follows. In Section 2, we provide an introduction to generalized quantifiers. In Section 3, we demonstrate that GQ-queries are inadequately supported in current relational database management systems. There are

[*]CS Department, Bloomington, IN 47405. E-mail: {srao, abadia, vgucht}@cs.indiana.edu.

[1]Several recent papers [7, 20] have pointed out that users commonly use these features.

[2]The authors of [2] took this validation a step further and postulated the *conjunctive formulation thesis.* This thesis states that real-world queries can be formulated most naturally as a conjunction of first-order-predicate statements and generalized-quantifier statements over sub-queries. [2] also introduced a query language, called $\mathcal{QLGQ}$, which was designed in accordance with this thesis. (It is clear from Hsu and Parker's paper that SQL is designed in accordance with the conjunctive formulation thesis. The problem with SQL is that it stops short of taking full advantage of its inherently good design. In fact, original-SQL (SEQUEL/2) [5] was better than SQL in this regard.)

two essential problems: 1) SQL's syntax is too restricted to formulate GQ-queries, and 2) SQL queries that simulate GQ-queries are frequently ill-supported by existing relational query processor. In Section 4, we describe a system, called `GQ-BMM`[3], to process GQ-queries, and we compare the performance of our system with that of relational systems. We also point out the strengths and weaknesses of our approach. In Section 5 we give a list of requirements for query languages and database systems that we believe are fundamental to adequate GQ-queries processing.

2 Generalized quantifiers

Generalized quantifiers were introduced by mathematical logicians in the late 1950s who wanted to study logical properties of quantifiers that could not be expressed within first-order logic [19, 22].[4] In the early 1980s, they became prominent in linguistics. Building on Montague's work [21], Barwise and Cooper [3] advocated the adoption of generalized quantifiers in natural language formalization. We illustrate their insights through an example. Consider, as part of an health-care study, the relations

`patient(pname, age-group)`	`symptom(description)`
`disease(dname, dtype)`	
`patient-symptom(pname, symptom)`	`disease-symptom(dname, symptom)`

Next, consider the following natural language sentences.

(1) Some of John's symptoms are hepatitis-A symptoms.
(2) All of John's symptoms are hepatitis-A symptoms.
(3) Not all of John's symptoms are liver-disease type symptoms.
(4) At least 10% of middle-aged patients' disease symptoms are heart-disease symptoms.

The grammatical structure of each of these sentences is of the form

$$\langle\text{noun phrase}\rangle\langle\text{verb phrase}\rangle.$$

For example, the grammatical structure of the third sentence is

$$\underbrace{\text{Not all of John's symptoms}}_{\text{noun phrase}}\ \underbrace{\text{are liver-disease type symptoms}}_{\text{verb phrase}}.$$

We invite the reader to formulate these sentences in FO-logic[5] to convince him or herself that there is an *impedance mismatch* between natural language and FO-logic. To remove this mismatch, Barwise and Cooper pursued a drastically different route to formalize natural language. They observed that sentences, such as the above have the following substructure

$$\underbrace{\langle\text{quantifier}\rangle\langle\text{set-expression}_1\rangle}_{\text{noun phrase}}\ \underbrace{\langle\text{set expression}_2\rangle}_{\text{verb phrase}}.$$

For example, sentence (4) can be broken down as follows.

$$\underbrace{\text{At least 10\%}}_{\text{quantifier}}\ \underbrace{\text{Middle-aged patients' disease symptoms}}_{\text{set expression}_1}\ \underbrace{\text{Heart-disease symptoms}}_{\text{set expression}_2}$$

Observe that each set expression that occurs in the example sentences can be formalized using the standard set abstraction mechanism from logic, wherein $\widehat{x}\,[\varphi(x)]$, with $\varphi(x)$ some well-formed formula, represents the set $\{x \mid \varphi(x)\}$. For example,

[3]`GQ-BMM` is a boolean-matrix file system implemented on top of the EXODUS storage manager [14]

[4]The body of literature on generalized quantifiers in mathematical logic—and, more recently, in finite model theory (e.g., [12])—is extensive.

[5]The fourth sentence is technically not expressible in FO-logic.

"John's symptoms"	=	$\widehat{\mathtt{s}}[\mathtt{patient\text{-}symptom}(\text{`}\mathtt{John}\text{'},\mathtt{s})]$
"Hepatitis-A symptoms"	=	$\widehat{\mathtt{s}}[\mathtt{disease\text{-}symptom}(\text{`}\mathtt{Hepatitis\text{-}A}\text{'},\mathtt{s})]$
"Heart-disease symptoms"	=	$\widehat{\mathtt{s}}[\exists \mathtt{d}\ \mathtt{disease\text{-}symptom}(\mathtt{d},\mathtt{s}) \wedge \mathtt{disease}(\mathtt{d},\text{`}\mathtt{heart\text{-}disease}\text{'})]$
etc.		

Given the sentence structure exhibited by Barwise and Cooper, it is natural to view quantifiers as binary *set predicates*. For example, the quantifier `some` is such that "`some`(S,T)" is true if and only if "$S \cap T \neq \emptyset$", the quantifier `(not) all` is such that "`(not) all`(S,T)" is true if and only if "$(\neg)S \subseteq T$", and for the quantifier `atleast 10%`, "`atleast 10%`(S,T)" is true if and only 10% of the elements of S are also in T. Following these insights, the example sentences can be formalized as follows

(1) $\mathtt{some}(\widehat{\mathtt{s}}[\mathtt{patient\text{-}symptom}(\text{`}\mathtt{John}\text{'},\mathtt{s})],\widehat{\mathtt{s}}[\mathtt{disease\text{-}symptom}(\text{`}\mathtt{hepatitis\text{-}A}\text{'},\mathtt{s})])$

(2) $\mathtt{all}(\widehat{\mathtt{s}}[\mathtt{patient\text{-}symptom}(\text{`}\mathtt{John}\text{'},\mathtt{s})],\widehat{\mathtt{s}}[\mathtt{disease\text{-}symptom}(\text{`}\mathtt{hepatitis\text{-}A}\text{'},\mathtt{s})])$

etc.

Remark 2.1 All the generalized quantifiers in the example sentences are *binary*. There are, however, natural *unary*, as well as higher arity, generalized quantifiers [18, 27]. 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 $S \neq \emptyset$.

Remark 2.2 (**GQ-queries**) All the examples we have discussed so far are *sentences* (i.e., true-false formulas). It is straightforward to extend our discussion to *queries*. For example, consider the queries "*Find the patient-disease pairs* $(\mathtt{p},\mathtt{d})$ *such that patient* `p` *has* [`some`|`all`|`most`|...] *of the symptoms associated with disease* `d`." These queries can be formalized as follows.

$$\mathtt{some}(\widehat{\mathtt{s}}[\mathtt{patient\text{-}symptom}(\mathtt{p},\mathtt{s})],\widehat{\mathtt{s}}[\mathtt{disease\text{-}symptom}(\mathtt{d},\mathtt{s})])$$
$$\mathtt{all}(\widehat{\mathtt{s}}[\mathtt{patient\text{-}symptom}(\mathtt{p},\mathtt{s})],\widehat{\mathtt{s}}[\mathtt{disease\text{-}symptom}(\mathtt{d},\mathtt{s})])$$
$$\mathtt{most}(\widehat{\mathtt{s}}[\mathtt{patient\text{-}symptom}(\mathtt{p},\mathtt{s})],\widehat{\mathtt{s}}[\mathtt{disease\text{-}symptom}(\mathtt{d},\mathtt{s})])$$
etc.

Observe that in these formulas the variables `p` and `d` are *free*. An answer to such a query is the set of patient-disease tuples which make the formula true. Formulas like these, with free variables and containing generalized quantifiers, will be called *GQ-queries*.

In two recent papers, Hsu and Parker [17], and independently, Badia, Van Gucht and Gyssens [2] introduced query languages that naturally incorporate the ideas proposed by Barwise and Cooper. Hsu and Parker's language is a natural generalization of SQL. The language in [2], called $\mathcal{QLGQ}$, is developed in the style of the domain relational calculus [8, 9]. The GQ-queries given in Remark 2.2 are specified in $\mathcal{QLGQ}$ syntax. In Section 3, we will formulate queries such as these in Hsu and Parker's extended SQL.

3 Processing GQ-Queries in Relational Systems

Hsu and Parker [17] 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 conventional SQL (actually SQL2 [25]). In this section we consider this approach. To focus the discussion, we will only consider GQ-queries containing the generalized quantifiers `some`, `all`, `no`, and `not all`.

Reconsider the health-care example and consider the following queries (see also Remark 2.2): "*Find the patient-disease pairs* $(\mathtt{p},\mathtt{d})$ *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 follows:

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

These queries can be translated into SQL using the EXISTS unary generalized quantifier as follows.

```
SELECT P.pname, D.dname
FROM   patient-symptom P, disease-symptom D
WHERE  [NOT] EXISTS
       (SELECT *
        FROM   symptom S1
        WHERE  P.symptom = S1.description AND
               [NOT] EXISTS
               (SELECT *
                FROM   symptom S2
                WHERE  D.symptom = S2.description AND
                       S1.description = S2.description))
```

We ran these SQL queries in the University-INGRES system[6] and in two (well-known) relational database management systems. The queries were run on a SPARC-10 workstation. Both server and client processes were run on the same workstation for all the systems. The data sets were prepared as follows.

1. The `patient-symptom` relation was populated by randomly constructing patient-symptom tuples from a domain of patients `P` and a domain of symptoms `S`. This random process was controlled so that a patient was related to an average of 5 symptoms. The relation `disease-symptom` was similarly populated (of course with a suitable disease domain `D`). (The size of each tuple was 20 bytes.)
2. Initially, the domain sizes were as follows $|\texttt{P}| = 500$, $|\texttt{D}| = 400$, and $S = |1000|$, respectively. Through the experiments, we let the domains grow as follows. The domain `P` doubled at each step, and the domains `D` and `S` grew with a factor of 1.05. For the health-care database, this kind of a data set we believe, is realistic.
3. For simplicity, we assumed that the (unary) relation `symptom` consisted of the symptoms that occur in the relations `patient-symptom` or `disease-symptom`.
4. We decided not to create indexes on the relations because, in general, it can not be assumed that indexes are available, especially on intermediate relations in complex queries. Moreover, as we show in the appendix B using indexes makes no appreciable difference for the queries mentioned.

In Figure 1, we show the results of our experiments. As can be observed, except for the `some`-query, the performance of each relational system on each of the queries

[6]The corresponding QUEL queries are specified in Appendix A.

is very poor, and this is already these case on *very small* relations. Fortunately, it is possible to reformulate at least one other query, the all-query, to improve performance. In particular, the all-query can be reformulated with SQL's GROUP BY and COUNT mechanisms.[7]

```
CREATE VIEW PatDisGrp (pname, dname, cnt)
AS
  SELECT P.pname, D.dname, COUNT(distinct P.symptom)
  FROM   patient-symptom P, disease-symptom D
  WHERE  P.symptom = D.symptom
  GROUP BY P.pname,D.dname

CREATE VIEW PatGrp (pname, cnt)
AS
  SELECT P.pname, COUNT(distinct P.symp)
  FROM   patient-symptom P
  GROUP BY P.pname

SELECT  D.dname, P.pname
FROM   PatDisGrp D, PatGrp P
WHERE D.pname = P.pname AND D.cnt = P.cnt
```

Using this formulation of the all-query, the performance of the commercial relational systems improves dramatically (see Figure 1). (Observe however the large discrepancy between the two commercial systems.)

Unfortunately, reformulating the no-query and the not all-query via grouping and counting mechanisms (as advocated by Hsu and Parker [17], see also [13]) does 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. In particular, the count information necessary to solve these queries needs to be stored in relations that approximate the size of the cartesian product of the patient and disease domains.[8] We did several experiments with the various relational systems to create cartesian products. The results were as expected abysmal. In the next section, we suggest alternative storage mechanisms which allow such queries to be performed as efficiently as their positive counter parts.

We may conclude from these simple experiments that in general, relational systems ill-support GQ-queries. First, it is often difficult to formulate GQ-queries in SQL. Second, the SQL queries simulating the GQ-queries are frequently ill-supported by existing relational query processors.

4 GQ Processing as Boolean Matrix Multiplication

We start by conceptualizing "generalized quantifier processing" as boolean matrix multiplication. We then describe a system, called GQ-BMM, that implements this conceptualization and we give its performance on the queries specified in Section 3. We conclude by discussing these results and by pointing out strengths and weaknesses of GQ-BMM.

[7]This strategy has been suggested on numerous occasions (see for example [11, 13]) and is also advocated in Hsu and Parker [17].

[8]In contrast, in the case of the all query, one is fortunate because the count information necessary to solve the query is not of the order of the cartesian product but rather that of the join result.

PS	DS	some Query		
		Timings(seconds)		
		Univ. Ingres	Comm. 1	Comm. 2
2k	2k	9.40	6.11	4.27
4k	2.1k	12.61	9.49	6.93
8k	2.2k	35.60	13.70	14.16
16k	2.3k	36.96	25.57	27.80
32k	2.4k	148.47	50.15	56.10
64k	2.5k	176.95	98.20	114.48
128k	2.6k	372.84	189.82	249.64

PS	DS	all Query (NOT EXISTS)		all Query (with grouping)		
		Timings(seconds)		Timings(seconds)		
		Comm.1	Comm.2	Univ.Ingres	Comm.1	Comm.2
500	500	564.9	4649.3	119.6	-	-
2k	2k	9973.6	41239+	1623+	8.0	20.2
4k	2.1k	-	-	-	12.4	53.8
8k	2.2k	-	-	-	22.0	174.7
16k	2.3k	-	-	-	39.2	605.7
32k	2.4k	-	-	-	76.7	2052.9
64k	2.5k	-	-	-	162.7	7885.8

PS	DS	no Query			not all Query		
		Timings(seconds)			Timings(seconds)		
		Univ.Ingres	Comm.1	Comm.2	Univ.Ingres	Comm.1	Comm.2
500	500	117.6	737+	13529+	139.3	566.0	2228+
2k	2k	805+	16838.8	41236+	602+	10053.2	-
4k	2.1k	-	-	-	-	-	-

Figure 1: The top table gives the performance of three relational systems on the `some`-query (the left most column in this table, as well as the following tables, gives the number of tuples in the `patient-symptom` table (`PS`) and in the `disease-symptom` table (`DS`)); the performance on this query is quite balanced across the three systems. The middle top tables give the performance of the three systems on the `all` query. (A "-" indicates that the query could not be run on the system either because the times required to wait for the answer were getting too long or it was a trivial case. A "+" indicates the abortion of the experiment at the specified time.) The left-middle table gives the performance for the `NOT EXISTS` formulation of the query; the performance is clearly very poor. The right-middle table gives the performance for the `GROUP BY` formulation; the performance improves dramatically, but notice the significant difference in performance between commercial 1 and commercial 2. The bottom two tables give the performance of the various systems on the `no`-query and the `not all`-query; the performance is extremely poor.

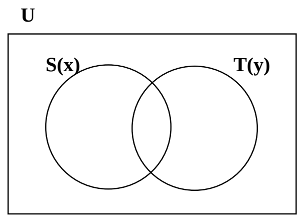

Figure 2: Parameterized sets $S(x)$ and $T(y)$ occurring in the context of a domain U.

4.1 Conceptualization

In $\mathcal{QLGQ}$ and in Hsu and Parker's extended-SQL, a (basic) formula involving a binary generalized quantifier is, in general, of the form

$$\texttt{quantifier}(S(x), T(y))$$

wherein "`quantifier`" is the generalized quantifier of interest, x and y are vectors of free variables, and $S(x)$ and $T(y)$ are parameterized set terms. The internal structures of the set terms $S(x)$ and $T(y)$ are in general of the form

$$S(x) \equiv \widehat{z}[P(x,z)] \text{ and } T(y) \equiv \widehat{z}[Q(z,y)]$$

wherein P and Q are formulas denoting relations. (We deliberately choose the same bounded variable z in these formulas to indicate that the elements in $S(x)$ and $T(y)$ are assumed to come from a common domain, say U. This situation is depicted in Figure 2.)

Now assume that $a_1, \ldots, a_k$ are the x-values, $b_1, \ldots, b_l$ are the y-values, and $c_1, \ldots, c_m$ are the z-values that occur in P **or** Q. Then we can view P as a boolean $[k,m]$-matrix, wherein $P(i,j) = 1$ if $P(a_i, c_j)$ and $P(i,j) = 0$ if $\neg P(a_i, c_j)$. Similarly, we can view Q as a boolean $[m,l]$-matrix.[9] The relation corresponding to the formula $\texttt{quantifier}(S(x), T(y))$ can then be obtained by "multiplying" the matrices P and Q. The particular multiplication algorithm will of course depend on the semantics attached to the generalized quantifier "`quantifier`". For example, in the case of the `some` quantifier, the multiplication algorithm is the conventional boolean matrix multiplication algorithm; i.e., if we call $P \times_{\texttt{some}} Q$ the result of multiplying P and Q according to the generalized quantifier `some`, then

$$P \times_{\texttt{some}} Q(i,j) = \bigvee_{z=1}^{m} P(i,z) \wedge Q(z,j).$$

In the case of the `all`, `no`, and `not all` generalized quantifiers, the multiplication formulas are respectively

$$P \times_{\texttt{all}} Q(i,j) = \bigwedge_{z=1}^{m} P(i,z) \rightarrow Q(z,j),$$

$$P \times_{\texttt{no}} Q(i,j) = \neg \bigvee_{z=1}^{m} P(i,z) \wedge Q(z,j), \text{ and}$$

$$P \times_{\texttt{notall}} Q(i,j) = \neg \bigwedge_{z=1}^{m} P(i,z) \rightarrow Q(z,j).$$

[9]The idea of associating bit vectors to domain elements was advocated by Graefe [16] to efficiently support *relational division*.

pname	symptom
p_1	s_1
p_2	s_1
p_2	s_3
p_3	s_1
p_4	s_2
p_4	s_3
p_5	s_2

$$\begin{array}{c|ccc} & s_1 & s_2 & s_3 \\ \hline p_1 & 1 & 0 & 0 \\ p_2 & 1 & 0 & 1 \\ p_3 & 1 & 0 & 0 \\ p_4 & 0 & 1 & 1 \\ p_5 & 0 & 1 & 0 \end{array}$$

dname	symptom
d_1	s_2
d_1	s_3
d_2	s_1
d_3	s_1
d_3	s_3

$$\begin{array}{c|ccc} & d_1 & d_2 & d_3 \\ \hline s_1 & 0 & 1 & 1 \\ s_2 & 1 & 0 & 0 \\ s_3 & 1 & 0 & 1 \end{array}$$

Figure 3: The `patient-symptom` and `disease-symptom` relations represented as a boolean matrices. (We actually show the transpose of the boolean matrix corresponding to the `disease-symptom` relation.)

$\mathtt{P} \times_{\mathtt{some}} \mathtt{D}$

$$\begin{array}{c|ccc} & d_1 & d_2 & d_3 \\ \hline p_1 & 0 & 1 & 1 \\ p_2 & 1 & 1 & 1 \\ p_3 & 0 & 1 & 1 \\ p_4 & 1 & 0 & 1 \\ p_5 & 1 & 0 & 0 \end{array}$$

$\mathtt{P} \times_{\mathtt{all}} \mathtt{D}$

$$\begin{array}{c|ccc} & d_1 & d_2 & d_3 \\ \hline p_1 & 0 & 1 & 1 \\ p_2 & 0 & 0 & 1 \\ p_3 & 0 & 1 & 1 \\ p_4 & 1 & 0 & 0 \\ p_5 & 1 & 0 & 0 \end{array}$$

$\mathtt{P} \times_{\mathtt{no}} \mathtt{D}$

$$\begin{array}{c|ccc} & d_1 & d_2 & d_3 \\ \hline p_1 & 1 & 0 & 0 \\ p_2 & 0 & 0 & 0 \\ p_3 & 1 & 0 & 0 \\ p_4 & 0 & 1 & 0 \\ p_5 & 0 & 1 & 1 \end{array}$$

$\mathtt{P} \times_{\mathtt{not\ all}} \mathtt{D}$

$$\begin{array}{c|ccc} & d_1 & d_2 & d_3 \\ \hline p_1 & 1 & 0 & 0 \\ p_2 & 1 & 1 & 0 \\ p_3 & 1 & 0 & 0 \\ p_4 & 0 & 1 & 1 \\ p_5 & 0 & 1 & 1 \end{array}$$

Figure 4: The matrices $\mathtt{P} \times_{\mathtt{some}} \mathtt{D}$, $\mathtt{P} \times_{\mathtt{all}} \mathtt{D}$, $\mathtt{P} \times_{\mathtt{no}} \mathtt{D}$, and $\mathtt{P} \times_{\mathtt{not\ all}} \mathtt{D}$.

And, in the case of the generalized quantifier `atleast 10%`, we have

$$P \times_{\mathtt{atleast10\%}} Q(i,j) = (count(\{z|P(i,z) \wedge P(z,j)\}) \geq count(\{z|P(i,z)\})/10.0)$$

Example 4.1 Consider the `patient-symptom` (`P`) and `disease-symptom` (`D`) relations shown in Figure 3 alongside their respective boolean matrix representation (actually the boolean-matrix alongside `D` corresponds to the *inverse* of `D`). In Figure 4, we show the results of the matrix multiplications $\mathtt{P} \times_{\mathtt{some}} \mathtt{D}$, $\mathtt{P} \times_{\mathtt{all}} \mathtt{D}$, $\mathtt{P} \times_{\mathtt{no}} \mathtt{D}$, and $\mathtt{P} \times_{\mathtt{not\ all}} \mathtt{D}$. Observe, that the matrices $\mathtt{P} \times_{\mathtt{some}} \mathtt{D}$ and $\mathtt{P} \times_{\mathtt{no}} \mathtt{D}$ are each others boolean complement. A similar fact is true for the matrices $\mathtt{P} \times_{\mathtt{all}} \mathtt{D}$ and $\mathtt{P} \times_{\mathtt{not\ all}} \mathtt{D}$.

4.2 Implementation

To validate the above conceptualization, we implemented a system called `GQ-BMM` (`G`eneralized `Q`uantifier Processing as `B`oolean `M`atrix `M`ultiplication) on top of the University of Wisconsin EXODUS storage manager [14] (Currently, `GQ-BMM` handles only binary relations.) We next describe the data structures and algorithms implemented in `GQ-BMM`.

Data Structures

In `GQ-BMM`, each (binary) relation is implemented as a boolean matrix. Assume that `R` is a relation over domains `A` and `B`, i.e., $\mathtt{R} \subseteq \mathtt{A} \times \mathtt{B}$. If $|\mathtt{A}| = k$ and $|\mathtt{B}| = l$, then `R` is stored as a $k \times l$ bit-matrix $m(\mathtt{R})$ indexed via pairs (i,j), where $i \in [1 \dots k]$ and $j \in [1 \dots l]$. In addition, `GQ-BMM` maintains a *row mapping* which associates the elements in `A` with the row indexes in $[1 \dots k]$, and a *column mapping* which associates the elements in B with the column indexes in $[1 \ldots l]$, respectively. The matrix $m(\mathtt{R})$ is laid-out in secondary memory by fragmenting it into equal-sized sub-matrices in such a way that each sub-matrix can fit on a disk page. Thus the space occupied by R in GQ-BMM is kl-bits plus $O(k+l)$.

Algorithms

GQ-BMM currently supports the `some`, `all`, `no` and `not all` generalized quantifiers. It is straightforward, however, to extend the system to support other generalized quantifiers. An important property of GQ-BMM is that it guarantees *compositionality*; in particular the result of multiplying two boolean matrices is again a boolean matrix maintained according to the data structure described above.

Remark 4.2 *It is worthwhile to mention that* GQ-BMM *supports, besides the various matrix multiplication operations, other standard operations on boolean matrices, i.e.,* complementation, transposition*, and* addition *[4, 10]. In combination with the various matrix multiplication operations, these operations make* GQ-BMM *a system in which many complex queries can be processed. Due to space limitations, we can not go into more details regarding this aspect of our system.*

We now briefly describe the various multiplication algorithms implemented in GQ-BMM.

- `some`: For the `some` generalized quantifier, we experimented with two algorithms, called **`Kronrod`** and **`Witness`**, respectively. **`Kronrod`** is an adaptation, to secondary memory, of Kronrod's well-known boolean matrix multiplication algorithm [1][10]. **`Witness`** is an algorithm that uses certain *witnessing* techniques [15][11] found in most join algorithms [16]. In particular, in the **`Witness`** algorithm, a fragment of a boolean matrix is, before the multiplication, first transformed into a list of sets, each set corresponding to a witness. The multiplication is then done via local cartesian products, again guided by the witnesses. Finally, the result is mapped back into a bit matrix.

  Our experience with these algorithms reveals that `Kronrod` performs best when the matrices are non-sparse (this is typically not the case in practice, except when complementation is required), and `Witness` performs best on sparse matrices.[12]

- `all`: The `all`-algorithm implements matrix multiplication in accordance with the multiplication formula

$$P \times_{\mathtt{all}} Q(i,j) = \bigwedge_{z=1}^{m} P(i,z) \rightarrow Q(z,j).$$

  The `all`-algorithm has time complexity $O(k \times l \times m)$ when P and Q are $[k,m]$ and $[m,l]$ matrices, respectively. However, our implementation takes advantage of the fact that to make the formula $\bigwedge_{z=1}^{m} P(i,z) \rightarrow Q(z,j)$ false, it suffices that just one of its factors is false. In particular, the `all`-algorithm is implemented to break out of the $\bigwedge_{z=1}^{m}$-loop when a z is discovered such that $P(i,z) \rightarrow Q(z,j)$ is false. It turns out that, in practice such a z is found early in the loop. The complexity of the `all`-algorithm is therefore $O(k \times l \times C)$, where C denotes the average time to discover a z which results in a false-factor.

[10] To multiply two $n \times n$ matrices, the complexity of Kronrod's algorithm is $O(n^3/logn)$.

[11] Given matrices P and Q, and given i and j, a *witness* for i and j is a value z such that $P(i,z)$ and $Q(z,j)$ are true. So the value z "witnesses" that P and Q "join" on the (i,j) combination.

[12] It is interesting to point out that the join algorithms reported in the literature are assumed to work on relations corresponding to **very** sparse matrices. When these algorithms are applied to relations corresponding to matrices of even modest sparsity, their behavior deteriorates rapidly. The most significant factor in this deterioration can be attributed to excessive duplicate generation.

PS	DS	Commercial DBMS 1				GQ-BMM			
		Timings (seconds)				Timings (seconds)			
		`some`	`all`	`no`	`notall`	`some`	`all`	`no`	`notall`
2k	2k	6.11	8.01	16838.8	10053.2	0.57	1.53	0.85	1.59
4k	2.1k	9.49	12.42	-	-	1.02	2.32	1.44	2.44
8k	2.2k	13.70	22.03	-	-	2.00	3.74	2.02	4.14
16k	2.3k	25.57	39.25	-	-	3.92	12.13	4.38	14.60
32k	2.4k	50.15	76.76	-	-	6.61	28.64	12.22	44.18
64k	2.5k	98.20	162.74	-	-	21.21	69.55	37.69	77.22
128k	2.6k	189.82	322.19	-	-	49.50	130.13	135.46	152.49

Figure 5: The left-table gives the performance of commercial system 1 and the right-table gives the performance of GQ-BMM.

- `no` and `not all`: Because `GQ-BMM` supports boolean-matrix complementation (Remark 4.2), the multiplication algorithms corresponding for these generalized quantifiers were implemented as `no` $\equiv$ *complement*(`some`) and `notall` $\equiv$ *complement*(`all`), respectively.

4.3 Performance Results

We ran the queries described in Section 3 on the same relations. Of course, these relations were represented as boolean matrices in `GQ-BMM`. As in the case of the relational system experiments, the client and server process were run on a SPARC 10. The main-memory buffers for both the client and server put together were 10 MB (which was the same as that configured for the relational systems). The results of our experiments are shown in Figure 5.

Discussion

Foremost, our results indicate that processing queries with generalized quantifiers is a feasible proposition.

Secondly, when we compare our results with those obtained for relational systems, we observe that relational systems do not live-up to this feasibility, and this is so already on very small relations. It is therefore safe to argue that relational systems have *not* yet solved the query processing problem in the *decision support area.*

On the other hand, we realize that our current approach also does not provide the "ultimate" solution. To make this clear, we conclude this section by pointing out some of the pros and cons of `GQ-BMM`.

- **Pros**
  1. We did get some good results; on all the problems discussed in the paper `GQ-BMM` performed better than the various relation systems. And, we did establish that in principle GQ-query processing can be done effectively.
  2. The conceptualization of GQ-query processing as matrix multiplication is natural, as we demonstrated at the begining of this section.
  3. `GQ-BMM` does not suffer from the duplicate elimination problem that exists in relational systems.
  4. Problems involving complementation or problems involving "dense" relations pose no problem for `GQ-BMM`, but they are an almost insurmountable barrier for current relational query processing.
- **Cons**

1. On very "sparse" relations, the `GQ-BMM` pays a storage size overhead that does not exist in relational systems. This storage overhead (which also affects the computation time) can become a burden; in such cases, however, there is always the option to consider a hybrid-system approach. Tracking the nature of the relations (sparse or dense) can be done using statistical information as is common in most database systems.
2. Currently, `GQ-BMM` only handles binary relations. Of course, it will be necessary to augment the system to also work with n-ary relations; we have designed plans to extend `GQ-BMM` in this way.

5 Conclusion

In this paper, we considered the problem of processing queries that contain generalized quantifiers. We demonstrated that current relational systems are ill-equipped, both at the language and at the query processing level, to deal with such queries. We subsequently proposed a boolean matrix approach which established the feasibility of building systems that can process GQ-queries efficiently.

We like to conclude by stipulating some of the key requirements that need to be met by a system to make it a good candidate for effective GQ-query processing.

1. *The system needs to offer a better query language than SQL to enable natural GQ-query formulation.* Languages such as reported in [17] and [2] should guide the design of such languages.
2. *The system must, at the storage level, be able to deal with relations that have a matrix representation which can range in sparsity from very sparse to very dense.* Current relational systems are designed to only deal with relations that have a very sparse matrix representation; relational systems will need to be extended to also deal with relations of modest to very high sparsity.
3. *The algorithms to manipulate relations must be sensitive to the sparsity conditions of these relations.* For example, current join algorithms are designed to work optimally on very sparse matrices; they therefore should be used in such circumstances instead of, say boolean matrix multiplication. However, when the matrices become, even marginally dense, these same join algorithms deteriorate rapidly in performance; they then need to be replaced by other matrix multiplication algorithm such as, for example, those available in `GQ-BMM`.

We believe that what is needed is a hybrid relational-matrix system that is set up to take advantage of the good properties of current relational systems when the situation is called for, but that is also flexible enough to switch to a "matrix" approach when necessary. We think that such systems can be designed and built. *Such systems, unlike current relational systems, will be in a better position to claim to have solved the query processing problem in the* decision support area.

References

[1] Baase, S., *Computer Algorithms*, Addison Wesley, 1988.

[2] Badia, A., Gyssens, M. and D. Van Gucht. "Query languages with generalized quantifiers," in *Applications of Logic Databases*, edited by R. Ramakrishnan, Kluwer Academic Publishers, 1995, pp. 235–258.

[3] Barwise, J. and Cooper, R., "Generalized quantifiers and natural language," *Linguistic and Philosophy*, 4, 1981, pp. 159–219.

[4] Brink, C., "Boolean Circulants, Groups, and Relation Algebras", *American Mathematical Monthly*, 1992.

[5] Chamberlin, D., *et.al.*, "SEQUEL/2: A unified approach to data definition, manipulation, and control," *IBM Journal of R&D*, 20, 6, 1976.

[6] Cattell, R., *ed*, *The Object Database Standard: ODMG-93*, Release 1.1, Morgan Kaufmann, 1994.

[7] Chaudhuri, S. and Shim, K. "Including group-by in query optimization," *Proceedings of the Twentieth International Conference on Very Large Databases*, 1994, pp. 354–366".

[8] Codd, E. F., "Further normalizations of the relational model," in *Data Base Systems*, R. Rustin, Ed., Prentice-Hall, Englewood Cliffs, N.J., 1972, pp. 33–64.

[9] Codd, E. F., "Relational completeness of database sublanguages," in *Data Base Systems*, R. Rustin, Ed., Prentice-Hall, Englewood Cliffs, N.J., 1972, pp. 65–98.

[10] Copilowish, Irving M., Matrix Development of the Calculus of Relations, Journal of Symbolic Logic, Dec.,1948, Vol. 13, No. 4.

[11] Date, C., *An Introduction to Database Systems*, Sixth Edition, Addison-Wesley Publishing Company, 1994.

[12] Dawar, H. and Hella, L., "The expressive power of finitely many quantifiers,", *Proc. of the 9th IEEE Symposium on Logic in Computer Science*, pp. 20–29, 1994.

[13] Dayal, U., "Of nests and trees: a unified approach to processing queries that contain nested subqueries, aggregates and quantifiers," in *VLDB 1987*, pp. 197–208.

[14] EXODUS, "Using the EXODUS Storage Manager V3.0", unpublished, included in the EXODUS Storage Manager Software Release, available via anonymous ftp from `ftp.cs.wisconsin.edu`.

[15] Galil, Z. and Margalit, O., "Witnesses for Boolean Matrix Multiplication and for Transitive Closure", Journal of Complexity, 1993, vol. 9, pp.201-221.

[16] Graefe, G., "Query evaluation techniques for large databases," *ACM computing Surveys*, 25, 2, 1993.

[17] Hsu, P.Y. and Parker, D.S., "Improving SQL with Generalized Quantifiers", *Proceedings of the Tenth International Conference on Data Engineering*, 1995.

[18] Keenan, E.L. and Stavi, J., "A semantic characterization of natural language determiners", *Linguistics and Philosophy*, 9, 1986, pp. 253-326.

[19] Lindström, P., "First order predicate logic with generalized quantifies," *Theoria*, 32, pp. 186–195, 1966.

[20] Lu, H., Chan, H.C., and Wei, K.K., "A survey on usage of SQL," *SIGMOD Record*, 22, 1993, pp. 60–65.

[21] Montague, R., *Formal Philosophy: Selected Papers*, R. H. Thomason, Ed., Yale University Press, New Haven, Conn., 1974.

[22] Mostowski, A., "On a generalization of quantifiers," *Fundamenta Mathematica*, 44, 1957, pp. 12–36.

[23] Özsoyoğlu, G. and Wang, H., "A relational calculus with set operators, its safety, and equivalent graphical languages," *IEEE Transactions on Software Engineering*, 15, 1989, pp. 1038–1052.

[24] 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`.

[25] International Organization for Standardization (ISO), *Database Language SQL*. Document ISO/IEC 9075:1992.

[26] van Benthem, J., “Questions about quantifiers,” *Journal of Symbolic Logic*, 49, 1984, pp. 443–466.

[27] Westerstahl, D., ”Quantifiers in Formal and Natural Languages”, in *Handbook of Philosophical Logic*, D. Gabbay and F. Guenthner, eds., vol. IV, D. Reidel Publishing Company, 1989, pp. 1–131.

Appendix

A Queries for Generalized Quantifiers in QUEL

- `some` query:

```
range of P is patient
range of D is disease
retrieve  unique (i1=P.pname,i2=D.dname)
where P.symptom=D.symptom
```

- `no` query:

```
range of P is patient
range of D is disease
retrieve into temp1(i1=P.pname , i2=D.dname)
where any(P.symptom by P.pname ,D.dname
  where P.symptom = D.symptom) = 0
```

- `all` query:

```
range of P is patient
range of D is disease
retrieve into temp1(i1=P.pname , i2=D.dname)
where count(P.symptom by P.pname ,D.dname
  where P.symptom = D.symptom)
  =
  count(P.symptom by P.pname)
```

- `not all` query:

```
range of P is patient
range of D is disease
retrieve into temp1(i1=P.pname , i2=D.dname)
where not(count(P.symptom by P.pname ,D.dname
  where P.symptom = D.symptom)
  =
  count(P.symptom by P.pname))
```

B Impact of Indexes on Commercial System 2

Figure 6 compares the performance of commercial system 2 with and without indexes for the `all` and `some queries`. For the `all` query, clustered indexes were set up for the `pname` and `dname` attributes in the `patient-symptom` and `disease-symptom` relations. We also built secondary indexes on the `symptom` attribute for both the tables. As is evident there is no appreciable difference in the performance (Figure 6, left-table). For the `some` query however , clustered (and later dense) indexes were set-up for the `symptom` attribute on both the relations while the other attributes were

PS	DS	`all` Query		`some` Query		
		Timings(seconds)		Timings(seconds)		
		Comm.2	Comm.2(indexed)	Comm.2	Comm.2(clust.)	Comm.2(dense)
2k	2k	20.2	19.3	4.2	4.0	5.2
4k	2.1k	53.8	48.6	6.9	5.8	7.7
8k	2.2k	174.7	155.9	14.2	11.7	12.8
16k	2.3k	605.7	545.2	27.8	21.6	25.1
32k	2.4k	2052.9	2064.6	56.1	42.7	50.1
64k	2.5k	-	-	114.5	84.9	105.6
128k	2.6k	-	-	249.6	182.9	291.8

Figure 6: The left-table compares the impact of indexing on the `all` query for commercial system 2. The right-table compares the impact of `clustered` (clust.) and `dense` indexing with that of the un-indexed situation. As before, PS is the patient-symptom relation while DS is the disease-symptom relation.

not indexed. In the clustered-index case a marginal improvement can be observed while in the dense case there was no difference (Figure 6, right-table). Our point here is to show that indexes do not make a significant impact on generalized-quantifier processing.