Efficient Implementation of Generalized Quantification in Relational Query Languages

Bin Cao
Teradata Corporation
El Segundo, CA 90245 USA
bin.cao@teradata.com

Antonio Badia
CECS Department
University of Louisville, Louisville, KY 40292 USA
abadia@louisville.edu

ABSTRACT

We present research aimed at improving our understanding of the use and implementation of quantification in relational query languages in general and SQL in particular. In order to make our results as general as possible, we use the framework of Generalized Quantification. Generalized Quantifiers (GQs) are high-level, declarative logical operators that in the past have been studied from a theoretical perspective. In this paper we focus on their practical use, showing how to incorporate a dynamic set of GQs in relational query languages, how to implement them efficiently and use them in the context of SQL. We present experimental evidence of the performance of the approach, showing that it improves over traditional (relational) approaches.

1. INTRODUCTION

It is well known that relational query languages (i.e. relational algebra, SQL) are variants of a restricted subset of first order logic. As such, they have the capability to express basic properties, including those that rely on *quantification* over the objects of a domain. At the same time, as practical languages (especially SQL), they must pay attention to pragmatic issues that do not arise in purely logical scenarios: for instance, issues of efficient evaluation are of the utmost importance for implementation efforts, in particular in environments like Decision Support and Data Mining, where the volumes of data under consideration are very large.

It is likely a combination of such factors that has led to the lack of a universal quantifier in Relational Algebra (henceforth $\mathcal{RA}$) and SQL: it is easy to see that certain queries with universal quantification, in interaction with other features, can led to high complexity ([12, 26]). However, there is no lack of expressive power in such languages: universal quantification can be expressed through negation (relative negation, or set difference, for safety reasons) and existential quantification. Unfortunately, this paraphrasing leads to queries that are somewhat cumbersome to express and difficult to optimize[1] ([5]). It would seem that adding universal quantification to SQL and $\mathcal{RA}$ is therefore a good idea, and this indeed has been studied in the past ([7]).

However, it is also possible to go beyond this punctual solution and consider adding *generalized quantifiers (GQs)* to the query language. Intuitively, GQs are generalizations of the idea of quantifier in first order logic, so they admit more operators than $\exists$ and $\forall$[2]. Again, this is especially relevant in environments that use complex queries. As an example, in a data warehouse with supplier information, one may be interested in analyzing orders where *all* the suppliers involved are from some specific region (say, Europe). Or, the condition can be relaxed to *at least two thirds* of the suppliers. Similarly, one may look for orders where *at most 10%* of the items are considered (by some measure) expensive. This typical Decision Support queries involve set-based conditions.

The idea of adding GQs to query languages has been studied ([15, 28]), but all past approaches worked with a *fixed, finite* set of such quantifiers. This limited the language expressiveness and usefulness. Also, implementation was either based on an inefficient translation back into SQL ([15]) or in a very efficient but customized approach valid only for a few quantifiers ([28]). A different approach is presented in [3], that uses a logic-based query language with GQs, called QLGQ, with the ability to add an *unbounded* number of quantifiers to the language.

In this paper, we extend the approach proposed in [3] with the goal of making it practical (by this we mean, in this context, applying it to relational query processing). The approach of [3] is not readily usable for query processing, since it relies on a tailored logic-based query language to express queries, and issues of query processing and optimization are barely addressed. Here address those shortcomings. In particular, our contributions here are as follows:

- we define an extension of $\mathcal{RA}$ that incorporates GQs. Our approach is inspired by the work in [19], where an extended relational algebra is developed to better support *rank* queries. We take a similar approach in extending $\mathcal{RA}$ to make it aware of set-based operations (subsection 3.1). We also study several properties of the resulting language that provide the basis for query processing and optimization. In particular, we describe how GQs behave in the presence of (traditional) relational operators (subsection 3.2).

Proceedings of the VLDB Endowment, Vol. 6, No. 4
Copyright 2013 VLDB Endowment 2150-8097/13/02... $ 10.00.

[1] It is interesting to note that more modern query languages like OQL and XQuery do include universal quantification.

[2] A formal definition is given in section 3.

- after discussing, for completeness, the implementation of GQs proposed in [3], we use this approach to provide an implementation for the extended $\mathcal{RA}$ of 3.1, and give a detailed account of how to incorporate such implementation in traditional query processing, including optimization strategies (section 5).

- we show results of an exhaustive experimental study that provides evidence that the proposed approach is indeed competitive with standard (relational) approaches for the type of queries considered (section 6).

The rest of this paper is organized as follows. In the next section we provide some motivation for our work and explain the issue that we are addressing. In section 3 we introduce the concept of Generalized Quantifier (GQ) and an extension of relational algebra that incorporates GQs. We also give some basic properties of the resulting language. In section 4 we summarize the approach of [3] to GQ implementation and show how it can be used to deal with our extended algebra, detailing the approach used in our experiments. In section 5 we introduce several optimization techniques that can be applied to the proposed implementation. In section 6 we present our experiments and discuss its results. In section 7 we discuss related work and close in section 8 with some conclusions and further work.

2. MOTIVATION

It is well known that SQL's syntax sometimes forces users to write queries in an awkward way. A well studied example is that of *universal quantification* ([12, 26]). SQL relies on paraphrases that make formulas complex for users and difficult for the optimizer too, since it has to work harder to process them in an efficient manner. As an example, assume a database with two relations: `student(sid,name)` and `teaches(pid,sid)`, which denote the professor `pid` is a teacher of student `sid`. Consider the question "find the professors teaching **all** students". Since SQL does not directly support the quantifier **all** (note that the keyword `ALL` in SQL has a different meaning), there are several ways to express this question, but all are indirect. The following query is one of these expressions, which is presented in most textbooks:

```
select pid
from   teaches t
where  not exists (select sid
                  from   student
                  where  sid not in
                         (select sid
                          from   teaches
                          where  pid=t.pid))
```

The query structure is complicated, and more importantly, the performance of such a query may not be efficient due to the correlated predicate ([5]). Adding universal quantification to SQL allows this query to be written in a simpler manner using the GQ **all**, which can be defined as a operator that takes two sets X and Y as arguments, and returns true if $X \subseteq Y$[3] Thus, one could express the query above as:

```
select pid
from teaches t
where all (select sid from students),
          (select sid from teaches where pid = t.tid)
```

(We rely on the reader's knowledge of SQL to see the intuitive meaning of this formula; we will provide exact syntax and semantics later. Again, note the difference between the GQ **all**, that takes two subqueries as arguments, and SQL's `ALL`, which compares an attribute with a subquery on a given comparison). Such query is not only easier to write than the standard approach; it could potentially be executed more efficiently if a query processor is equipped to handle such queries. One of the main insights about GQs is that by expressing *declaratively* what the user wants and developing appropriate techniques for query optimization, the system may be able to execute such queries more efficiently than in the traditional approach.

A second insight is that *the idea can be generalized in a principled and efficient manner.* Intuitively, English determiners and noun phrase modifiers (such as **some**, **all**, **half of**, **at least 3**,...) can be seen as quantifiers. For instance, the query "find the professors teaching **half** of the students" could be written as

```
select pid
from teaches t
where half (select sid from students),
           (select sid from teaches where pid = t.tid)
```

An even better approach would be to eliminate the need for correlation, although this would require further changes to SQL syntax. For instance, the query above can be written as

```
select T.pid
where all (select sid from students),
          (select sid from teaches T)
```

under the understanding that the second subquery provides values for the `select` clause by acting as a correlated subquery. That is, this query is intended to be equivalent to the previous one (note that `pid` is a foreign key in `teaches`), but it eliminates the need for a self-join on `teaches`. The main point is that *SQL has a mechanism to form sets (subqueries), but that such mechanisms are not fully exploited by the language* as only a few predicates (IN, etc.) are allowed to operate on subqueries.

In order to exploit this intuition, several challenges need to be addressed: first, we need to give a formal semantics to the new queries proposed; second, we need to incorporate GQs into relational query processing (ideally, with minimal disruption); third, we need to provide algorithmic support to show that queries with GQs can be efficiently processed. An additional challenge is deciding which specific GQs are to be supported: since each GQ represents a certain property, adding a fixed set of GQs is of limited utility, as it provides direct support only when those properties are needed. However, more general approaches may be difficult to implement. Thus, the goal (as usual in query languages) is to balance generality with complexity.

3. BACKGROUND

In this section we introduce the formalism that we will use in the rest of the paper. First, we introduce the concept of

[3] This corresponds to the typical use of $\forall$ in first-order formulas like $\forall\ z\ (X(z) \rightarrow Y(z))$.

Generalized Quantifier (GQ). Then, we define the extended algebra with GQs. We also provide some basic properties of GQs and of the algebra that will be used later for query processing and optimization.

3.1 Generalized Quantifiers

We start by fixing some notation. We will use variables X, Y, X_1, Y_1,..., to refer to *sets*, and variables A, B,..., to refer to relational attributes. Relations themselves (more in general, algebra expressions) will be denoted by E, E_1, E_2,.... We will reserve M, M', ..., for sets that act as universes (they will correspond to *active domains* in databases). We use $|X|$ to denote the cardinality of a set X. For $\mathcal{RA}$ expressions, we will denote projection by π, selection by σ, Cartesian product by $\times$, group-by by GB, aggregate function by AGG, join by $\bowtie$, left (right) outerjoin by $\sqsupset\!\bowtie$ ($\bowtie\!\sqsubset$), and left (right) semijoin by $\ltimes$ ($\rtimes$).

Given a non-empty set M, a GQ Q on M is a binary relationship on subsets of M, that is, a subset of $\mathcal{P}(M) \times \mathcal{P}(\mathcal{M})$. We write $Q_M(X, Y)$ to state that $X, Y \subseteq M$ stand in the relationship determined by Q. Some common GQs are shown in Figure 1[4].

$$
\begin{aligned}
\textbf{all} &= \{X, Y \subseteq M | X \subseteq Y\}\\
\textbf{some} &= \{X, Y \subseteq M | X \cap Y \neq \emptyset\}\\
\textbf{no} &= \{X, Y \subseteq M | X \cap Y = \emptyset\}\\
\textbf{not all} &= \{X, Y \subseteq M | X \not\subseteq Y\}\\
\textbf{all but n} &= \{X, Y \subseteq M || X - Y| = n\}\\
\textbf{at least n} &= \{X, Y \subseteq M || X \cap Y| \geq n\}\\
\textbf{at most n} &= \{X, Y \subseteq M || X \cap Y| \leq n\}\\
\textbf{half} &= \{X, Y \subseteq M || X \cap Y| \times 2 = |X|\}
\end{aligned}
$$

Figure 1: A few common Generalized Quantifiers

Note that we admit *parametric GQs*, which allow parameters in their definitions (and in their names). As an example, **at least n** is a parametric GQ, with parameter **n** taking as value a natural number.

Not all such relations are considered GQs. As with most logical operators, GQs are required to be closed under isomorphisms, which in this context implies that for any bijection $f : M \to M$, $Q_M(X, Y)$ iff $Q_M(f[X], f[Y])$. Moreover, it can also be argued that the behavior of a quantifier should be independent of the context. Hence, it is sometimes assumed (and we will assume it here) that for all $M' \supseteq M$, $Q_M(X, Y)$ iff $Q_{M'}(X, Y)$. In the context of database query languages, closure under isomorphism ensures that quantifiers are *generic* operations, while independence from contexts ensures they are *domain independent* ([2]).

The following definitions introduce some properties of GQs that will be used later in this paper.

Definition 3.1. *A GQ Q is* upward (downward) monotone on the first argument *if whenever $Q(X, Y)$ and $X \subseteq X'$ ($X \supset X'$), then $Q(X', Y)$, and likewise in the second argument. A GQ is upward (downward) monotone if it is upward (downward) monotone on either argument.*

As an example, **some** is upward monotone on both arguments; **all** is downward monotone on the first argument and upward monotone on the second; **at most n** is downward monotone on both argument; and **10%of** is neither upward nor downward monotone.

Note that the definition can be easily extended to make GQs operate on relations by treating tuples as atomic units: for instance, given binary relations R, S, $\textbf{all}(R, S)$ is true iff $R \subseteq S$ (that is, all *pairs* in R are also in S). Technically, this is called the *lifting* of the GQ ([30]). In the following, we assume that a GQ is automatically lifted to whatever is needed by its arguments, i.e. a binary lifting when working with binary relations, and so on.

3.2 GQs and Relational Algebra

Here we extend the standard relational algebra to incorporate GQs. In the following, if E is a relational algebra expression, $sch(E)$ denotes its *schema* and $param(E)$ its set of parameters. The extended algebra is defined by:

- if E is a standard $\mathcal{RA}$ expression composed of the selection, projection, join and union operators (i.e. an SPJU expression), then E is an expression in the extended algebra[5]. $sch(E)$ is defined as usual; $param(E) = \emptyset$.
- if E is an algebra expression, and $X \subset sch(E)$, $E[X]$ is an extended algebra expression, called a *parametric* expression; the elements of X are called the parameters. $sch(E[X]) = sch(E)$; $param(E[X]) = X$.
- If E_1, E_2 are two extended algebra expressions, and Q is a quantifier, $Q(E_1, E_2)$ is an extended algebra expression, called a *quantified* expression. $sch(Q(E_1, E_2)) = param(E_1) \cup param(E_2)$; $param(Q(E_1, E_2)) = \emptyset$.

The semantics of the extended algebra are simple: parametric expressions are really only used when quantified expressions are evaluated; otherwise they behave like regular expressions. In particular, $E[\emptyset] = E$. The following rules specify the semantics formally:

- Let c be an arbitrary condition, with all the attributes mentioned in c denoted by $attr(c)$. Assuming (as usual) that $attr(c) \subseteq sch(E)$, $\sigma_c(E[X]) = (\sigma_c(E))[X]$.
- Let $L \subseteq sch(E)$. Then $\pi_L(E[X]) = (\pi_L(E))[X \cap L]$.
- Let E_1, E_2 be arbitrary expressions, with $X \subseteq sch(E_1)$ and $Y \subseteq sch(E_2)$. Let c be a condition with $attr(c) \subseteq sch(E_1) \cup sch(E_2)$. Then $E_1[X] \bowtie_c E_2[Y] = (E_1 \bowtie_c E_2)[X \cup Y]$.
- Let E_1 and E_2 be schema compatible, and $X \subseteq sch(E_1)$ and $Y \subseteq sch(E_2)$ be schema compatible too; then $E_1[X] \cup E_2[Y] = (E_1 \cup E_2)[X]$.
- Let t, t' be variables over tuples. As usual, for X a set of attribute names, $t[X]$ is the tuple that results

[4] We emphasize that here we are using a limited definition of GQ; the general concept is much more powerful (the exact relation between our notion of GQ and the general notion is discussed in section 7).

[5] Set difference will be expressed with the quantifier **no** in the extended algebra, and universal quantification with **all**.

from limiting tuple t to the values of attributes in X[6]. Given parametric expression $E[X]$, let $Z = sch(E) - X$. Then for $t \in E[X]$, $E^{t,X} = \{t'[Z] \mid t' \in E \wedge t'[X] = t[X]\}$. Note that $E^{t,Z}$ is a set of tuples. A quantified expression $Q(E_1[X], E_2[Y])$ denotes a relation defined by:

$$\{(t[X]t'[Y]) \mid t \in E_1 \wedge t' \in E_2 \wedge Q(E_1^{t,X}, E_2^{t',Y})\}$$

where $(t[X]t'[Y])$ denotes the *tuple concatenation* of $t[X]$ and $t'[Y]$.

The purpose of introducing parametric expressions is to directly reflect correlated subqueries in SQL. In our SQL extension, the additional conditions are of the form $Q(S_1, S_2)$ where Q is a quantifier and S_1, S_2 are SQL subqueries. Our approach is to translate such expression *directly* into the extended algebra. The rational for doing so is to give the query processor the opportunity to deal with quantification *directly* by choosing among several possible implementations for quantifiers. To illustrate this idea, and the semantics just introduced, we revisit our previous examples. The extended SQL for query "Find the professors teaching **all** students" written in section 1 is translated to

$$all(\pi_{sid}(STUDENTS), (\pi_{sid,pid}(TEACHES))[pid])$$

Here $E_1 = \pi_{sid}(STUDENTS)$, with $sch(E_1) = \{sid\}$ and $param(E_1) = \emptyset$; $E_2 = \pi_{sid,pid}(TEACHES)[pid]$, with $sch(E_2) = \{sid, pid\}$ and $param(E_2) = \{pid\}$. This expression denotes a relation with schema $\{pid\}$. It is equivalent to the division operator of standard algebra, in that for a given value v of $\pi_{pid}(TEACHES)$, we consider the set $T_v = \pi_{sid}(\sigma_{pid=v}(TEACHES))$, and add v to the result whenever $all(\pi_{sid}(STUDENTS), T_v)$, that is, $\pi_{sid}(STUDENTS) \subseteq T_v$ (i.e. v is a professor teaching all students). Note that the query "Find the professors teaching **half** of the students" gets translated into a similar extended algebra expression, given by

$$half(\pi_{sid}(STUDENTS), (\pi_{sid,pid}(TEACHES))[pid])$$

and its semantics are similar: for v a value of $\pi_{pid}(TEACHES)$, v is in the result iff $half(STUDENTS, T_v)$. Likewise, queries like "Find the professors teaching **at least 3**|**no**| **10%** of the students" are translated into similar expressions, with only the GQ used changing.

3.3 Properties of the Extended Algebra

It is not difficult to see that the main properties of $\mathcal{RA}$ that provide the basis for query optimization (like selection push-down, commutativity and associativity of joins) also hold in the extended algebra. Here we study how GQs interact with other relational operators.

Selections In the following, we use R, S, T, as variables over relational expressions, and use c, c_1, ... as variables over conditions, with $attr(c)$ as before.

LEMMA 3.2. *Let R, S be arbitrary $\mathcal{RA}$ expressions, $X \subseteq sch(R)$, $Y \subseteq sch(S)$, and Q an arbitrary GQ. Then*

$$Q(\sigma_c(R[X]), S[Y]) = \sigma_c(Q(R[X], S[Y]))$$

whenever $attr(c) \subseteq X$.

This lemma allows us to move selections in and out of quantification when they only affect parameters[7]. The following one deals with selections attached to joins.

LEMMA 3.3. *Let R, S be arbitrary $\mathcal{RA}$ expressions, $X \subseteq sch(R)$, $Y \subseteq sch(S)$, $attr(c_1) \subseteq X$ and $attr(c_2) \subseteq Y$, and Q an arbitrary GQ. Then*

$$\sigma_{c_1\theta c_2}(Q(R[X], S[Y])) =$$
$$Q(\pi_{sch(R)}(R \bowtie_{c_1\theta c_2} S)[X], \pi_{sch(S)}(R \bowtie_{c_1\theta c_2} S)[Y])$$

Note that even though the lemma seems to introduce a more complex $\mathcal{RA}$ expression, the join of R and S can be compute once and then reused to compute the GQ.

Joins A join can cause a tuple in a relation to be duplicated (if it matches several tuples in the other relation) or to go away (if it does not match any tuple in the other relation). The first effect is irrelevant since GQs are *set* operators and therefore duplicate-insensitive. However, the second effect may change the result of applying a given GQ.

When the join is on attributes that are parameters, we have the following:

LEMMA 3.4. *Let R, S, T be arbitrary $\mathcal{RA}$ expressions, $X \subseteq sch(R)$, $Y \subseteq sch(S)$, and $Z \subseteq sch(T)$. Let $attr(c) \subseteq X \cup Y$.*

- *If Q is upward monotone on the first argument,*

  $$Q((R[X] \bowtie_c S[Y]), T[Z]) \subseteq$$
  $$Q(R[X], T_1[Z]) \cap Q(S[Y], T_2[Z])$$

  where T_i is the same expression as T but with additional projections to match only attributes in R (S).

- *if Q is downward monotone on the first argument,*

  $$Q((R[X] \bowtie_c S[Y]), T[Z]) \supseteq$$
  $$Q(R[X], T_1[Z]) \cap Q(S[Y], T_2[Z])$$

Note that if Q is downward (upward) monotone in the second argument, and the second set term contains a conjunction, the case is completely symmetric.

When the join is on attributes that are not parameters, we have a similar result:

LEMMA 3.5. *Let R, S, T be arbitrary $\mathcal{RA}$ expressions, $X \subseteq sch(R)$, $Y \subseteq sch(S)$, and $Z \subseteq sch(T)$. Let $attr(c) \subseteq sch(R) \cup sch(S) - (X \cup Y)$.*

1. *If Q is upward monotone on the first argument,*

   $$Q((R[X] \bowtie_c S[Y]), T[Z]) \subseteq$$
   $$Q(R[X], T_1[Z]) \times Q(S[Y], T_2[Z])$$

   where T_i is the same expression as T but with additional projections to match only attributes in R (S).

2. *if Q is downward monotone on the first argument,*

   $$Q((R[X] \bowtie_c S[Y]), T[Z]) \supseteq$$
   $$Q(R[X], T_1[Z]) \times Q(S[Y], T_2[Z])$$

[6] If $X = \emptyset$, $t[X]$ is the *null* tuple.

[7] Proof of this and the following lemmas are not included for lack of space.

Union The final operator to consider is union, that is, formulas like $Q(R[X] \cup S[Y], T[Z])$, where R and S are *union compatible* (that is, they have the same number and type of attributes). Here we also consider several cases:

1. when $X = Y = \emptyset$, we have the following:

   LEMMA 3.6. *(a) If Q is downward monotone in the first argument, then*

   $$Q(R[\emptyset] \cup S[\emptyset], T[Z]) \subseteq \\ Q(R[\emptyset], T[Z]) \cap Q(S[\emptyset], T[Z])$$

   (b) If Q is upward monotone in the first argument, then

   $$Q(R[\emptyset] \cup S[\emptyset], T[Z]) \supseteq \\ Q(R[\emptyset], T[Z]) \cap Q(S[\emptyset], T[Z])$$

   The proof is based on the fact that $S \subseteq S \cup T$ and $T \subseteq S \cup T$. Note that technically we consider empty sets for downward monotone quantifiers; if this is not the case, than the lemma does not hold. For such cases, union should be used instead of intersection.

2. whenever X or Y are not empty, lemma 3.6 still holds but with a small modification:

   - if $X \cap Y = \emptyset$, then Cartesian product takes the place of intersection.
   - if $X = Y$, union takes the place of intersection.
   - else ($X \cap Y \neq \emptyset$ but $X \neq Y$), then join (on attributes $X \cap Y$) takes the place of intersection.

Since we only use SPJU expressions, this covers all cases to consider.

4. IMPLEMENTING QUANTIFIERS

We now discuss how the extended algebra can be supported in the back end. We focus on quantified expressions (since other expressions are similar to standard relational expressions), and in particular on how GQs can be implemented in a relational framework.

We use the approach of [3], which is summarized here for completeness. We start by noticing that expressions of the form $Q(E_1[X], E_2[Y])$ are defined recursively; therefore E_1 or E_2 may contain quantified expressions in turn. However, this can be handled in the usual manner, by executing the query tree from the bottom up, since at some point the expressions E_1 and E_2 are simply SPJU expressions that the query processor can handle. Thus, we focus on implementing $Q(E_1[X], E_2[Y])$, assuming that both arguments to the quantifier have already been dealt with.

Since GQs are high-level, declarative operators, several approaches to their implementation are possible -this is one of the advantages of using GQs. One such approach is to provide algorithms that directly compute the set properties that GQs express, an approach used in [25, 21] for a related issue (set-based joins). The advantages of such approach are that it yields very efficient implementations, and that it can be integrated in a relational framework quite nicely. The main disadvantage is that in order to support more than a fixed, finite set of GQs, the algorithms would have to be parameterized, an interesting option that has not been addressed in the literature. The usual setup is to consider two set-valued attributes, A from relation R and B from relation S, and determine which tuples $t \in R$ and $t' \in S$ are such that $t[X] \cap t'[Y] \neq \emptyset$ (i.e. a set-based join). These algorithms provided in [25] and [21] could be adapted to compute $|t[X] \cap t'[Y]|$, so we could determine whether GQs like **some**, **at least n**, **at most n**, etc. hold. Also, the algorithms could be used to check **all** either by computing whether $t[X] \subseteq t'[Y]$ directly or by comparing $|t[X] \cap t'[Y]|$ with $|t[X]|$. Note that if we start with a *flat* relation R, making attribute A set-valued is achieved by *nesting* R, which is exactly what our parametric expressions denote -in this case, $R[A]$ and $S[B]$ would be the arguments to the algorithm for computing the set-based properties.

The approach of [3] enables supporting an *extendible* framework. This approach (called the *counter approach* in the rest of the paper) is explained in some detail since it is the one used in our experiments. Our target language is a $\mathcal{RA}$ without GQs, but with aggregation, group by and the ability to detect nulls (an `is null` predicate) and to count under certain simple conditions (that is, a `CASE` statement). This algebra is similar to that used by most query processors and optimizers.

Recall that a GQ Q is (in our context) a binary relation on subsets of a given domain M, and that such relation is closed under isomorphisms: for $f : M \to M$ bijective, $Q_M(X, Y)$ iff $Q_M(f[X], f[Y])$. It can be shown that such quantifiers can be completely characterized by a purely combinatorial approach, since isomorphism between sets depends uniquely on *cardinality* conditions. In particular, it can be shown that $|X - Y|$, $|Y - X|$ and $|X \cap Y|$ determine whether $Q_M(X, Y)$ ([30]). Moreover, elements outside the arguments (that is, elements in $M - (X \cup Y)$) do not matter, due to our assumption of *domain independence*. It follows that any quantifier is determined by a relation between the cardinalities of the three sets above, in the following sense (this concept is called *number-theoretic definability* in [30]).

DEFINITION 4.1. *Let Q be a quantifier, and X and Y sets. Let $p_1^{X,Y} = |X - Y|$; $p_2^{X,Y} = |Y - X|$ and $p_3^{X,Y} = |X \cap Y|$, and let $\varphi(x, y, z)$ be a formula in some arithmetic language with (at least) 3 free variables. We say that Q is* number-definable *when*

$$Q(X, Y) \text{ iff } \varphi(p_1^{X,Y}, p_2^{X,Y}, p_3^{X,Y})$$

Since quantifiers are determined, in this sense, by numerical properties, we define a *language on numbers* which allows us to generate an infinite number of formulas, and therefore, an infinite number of quantifiers. This framework allows us to define new GQs as needed, instead of having a finite, fixed set.

DEFINITION 4.2. *The* proportional language *(PL) is the set of formulas on natural numbers obtained under the following syntax:*

1. *Terms. We assume an infinite set $x, y, z, \ldots$ of domain variables.*

   (a) Numerals: $\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots$ and variables are terms.

   (b) If m, n are terms, $m + n$ and $m \times n$ are terms.

2. *Formulas*

(a) *If t_1 and t_2 are terms, $t_1\ \theta\ t_2$ is a (atomic) formula, where $\theta \in \{=, \neq, \leq, < \geq, >\}$.*

(b) *If φ, ψ are formulas, then $\varphi \wedge \psi$, $\varphi \vee \psi$ are formulas.*

(c) *If φ is a formula* where the symbol × does not appear*, and x is a variable, $\exists x \varphi(x)$ is a formula.*

We say that a quantifier Q is definable in PL iff it is number-definable, in the sense of definition 4.1, by a formula $\varphi(x, y, z)$, where φ is in PL[8]. Even though PL is a very simple language[9], the quantifiers definable in PL can express some important properties that are not first order logic-definable. For instance, PL can express, besides all standard first order quantifiers, the *comparison* quantifiers (like **I**, defined by the formula $p_1 = p_2$[10]; **more**, defined by the formula $p_1 < p_2$; **most**, defined by the formula $p_3 > p_1$), the **even** quantifier (since $X \cap Y$ *is even* can be expressed as $\exists n\ p_3 = \ n + n$)[11]; and the *proportional* quantifiers (like $\frac{n}{m}\ of\ (A, B) \ = \ p_3 \times \mathbf{m} = (p_1 + p_3) \times \mathbf{n}$, with the meaning $\frac{n}{m}$ *of As are Bs*[12]). Thus, PL quantifiers are quite useful, while admitting an efficient implementation.

From now on, we assume that for any quantifier Q appearing in a query, the system has a PL formula that defines Q. Note that there may be more than one formula for a given quantifier; this fact can be used for optimization purposes, since each definition of the quantifier can be used to generate a translation of the formula using the quantifier. The query plan generator then simply uses the PL formula associated with Q (or several, if several are available), and proceeds as follows: first, generate the expression $E_1 - E_2$ if p_1 is involved; $E_2 - E_1$ if p_2 is involved; and $E_1 \cap E_2$ if p_3 is involved (we can use join instead of intersection, and will do so in the following). Then, we use aggregation (with the parameters in either set term as the grouping attributes) and count to obtain a corresponding counter. Finally, a selection is added with a condition that reflects the PL formula. As an example, the query "find the professors teaching all students", expressed by

$$all(\pi_{sid}(STUDENTS), (\pi_{sid,pid}(TEACHES))[pid])$$

causes the interpreter to attempt to evaluate the formula with the **all** quantifier. Assume **all** has been defined as $p_1 = 0$ (that is, $X \subseteq Y$ iff $|X - Y| = 0$). This yields the formula:

$T1 = AGG_{count(distinct\ sid)\ as\ cnt2}(\texttt{student})$
$T2 = GB_{pid,count(distinct\ sid)\ as\ cnt1}(\texttt{student} \bowtie\!\!\sqsubset_{sid=sid} \texttt{teaches})$
$RESULT = \pi_{pid}(\sigma_{p_1=0}(\pi_{pid,p_1=cnt1-cnt2}(T1 \times T2)))$

Here, $T1$ computes $|X|$, $T2$ computes $|X \cap Y_v|$, where Y_v is the set that results for parameter v (since Y is a parametric expression); and the final expression computes $|X - Y_v|$ by computing $|X| - |X \cap Y_v|$, and then checks that it is equal to zero, in accordance to the PL formula. Note that an outerjoin is used for the intersection to avoid a *zero-count bug* ([11]). Note also that $T1$ is really a number, so the Cartesian product in the last line is trivial to compute. Alternatively, we could define **all** as $p_1 + p_3 = p_3$ (that is, $X \subseteq Y$ iff $|X| = |X \cap Y|$). This generates the alternative formula (with $T1$ as above)

$T2 = GB_{pid,count(distinct\ sid)\ as\ cnt1}(\texttt{student} \bowtie_{sid=sid} \texttt{teaches})$
$RESULT = \pi_{pid}(T1 \bowtie_{cnt1=cnt2} T2)$

Here as above, $T1$ computes $|X|$, $T2$ computes $|X \cap Y_v|$ (but a join is used this time), and the final expression checks the PL formula. Again, a cost-based optimizer would generate a cost estimate for each alternative and choose the one with the least cost. The details of an interpreter that takes in two extended $\mathcal{RA}$ expressions and a PL formula and outputs an algebra expression without quantifiers are given in [3], and are not repeated here for lack of space. More examples are shown in section 6.

4.1 GQs and SQL

Two issues need to be addressed if quantifiers are to be considered as practical operators: handling of duplicates and handling of nulls. In our exposition so far, we have stressed that we are working with *sets* of elements. We believe that this is what naturally fits with the quantifiers supported, i.e. **at least two** means *at least two different things.* To achieve set semantics, we use projection with duplicate removal and use the aggregate `count` with a `distinct` in its argument. In order to handle multisets, we can adopt multiset semantics for intersection, difference and (more importantly) for cardinality (so that multiset $\{a,\ a,\ b\}$, for instance, has cardinality 3). In practice, this can be achieved by not removing duplicates (in projection or in aggregation). This means that not only we can handle multiset semantics, but it is actually *more efficient* to do so than set semantics. The other practical issue is the nulls. This is a more delicate issue, in that null semantics is a big area of debate which is not dealt with very consistently in SQL itself. With respect to our approach, we can define set operations when nulls are allowed by copying SQL semantics as followed in the WHERE clause (since GQs are used in the WHERE clause). In this approach, two nulls are treated as different for computing intersection and difference. To support this, we need to use `COUNT(att)` for a given attribute `att`, or `COUNT(*)`, as needed by the semantics, as the former will ignore nulls and the latter count them.

Finally, we note that a database system could easily support PL; then a DBA could add GQs to the system by giving the quantifier a name and a PL formula (or several) that indicates the semantics of the GQ. Clearly, using such GQs in query processing can be done within the counters approach. This makes the approach *extendible*, in contrast with previous work, that deals with fixed sets of GQs.

5. OPTIMIZATION

In this section, we outline optimization techniques that apply to expressions in the extended $\mathcal{RA}$ used in the previous section. Hence, these can be applied before GQs are implemented.

5.1 Optimization using GQ Properties

There are several logical properties of quantifiers that can be exploited for optimization. One of them is that of *being a sieve* [4]. The intuition behind this idea is very simple:

[8] A very similar family of quantifiers is called the *properly proportional* quantifiers in [16].

[9] Note that we avoid multiplication within quantification in order to keep the language decidable.

[10] Since $|X - Y| = |Y - X|$ implies that $|X| = |Y|$

[11] This example shows that PL takes us beyond languages that have a 0-1 law ([2]).

[12] When $m \ = \ 100$, we have the *percent* family of quantifiers. This also allows PL to express *proportional* GQs, like **one half of, one third of...**. Note again that these definitions are parameterized by **m** and **n**.

GQ	Sieve	Type
at least n	$\lvert X\rvert \geq n$ and $\lvert Y\rvert \geq n$	constraint
at most n	$\lvert X\rvert \geq n$ and $\lvert Y\rvert \geq n$	implication
exactly n	$\lvert X\rvert \geq n$ and $\lvert Y\rvert \geq n$	constraint
all	$A \neq \emptyset$; $B \neq M$	implication
all but at least n	$\lvert X\rvert \geq n$	constraint
all but at most n	$\lvert X\rvert \geq n$	implication
all but exactly n	$\lvert X\rvert \geq n$	implication

Figure 2: Quantifiers, sieve conditions and their negations

given quantifier Q, it is expected that, in almost every domain M, some subsets of M will be in the relationships denoted by Q, and some will not. However, this does not always hold: sometimes the quantifier *degenerates*, in the sense that all subsets of a domain are in its relationship, or none is. By looking at different quantifiers' definitions, it is easy to identify conditions under which the quantifiers do not behave as sieves; such conditions are called *sieve conditions*. Note that, when such conditions are met, there is no point in processing the quantifier formula (all arguments will qualify, or none will); hence the value of this property for query optimization. The following definitions and examples formalize the idea and show how it contributes to query optimization.

Definition 5.1. *Let Q be a quantifier. Then $Q(X)_1 = \{Y \mid Q(Y,X)\}$ and $Q(X)_2 = \{Y \mid Q(X,Y)\}$.*

Definition 5.2. *Let Q be a quantifier. Q behaves as a sieve in its ith argument ($i = 1, 2$) whenever for all $X \subseteq M$, $Q(X)_i \neq \emptyset$ and $Q(X)_i \neq \mathcal{P}(M)$. Q behaves as a sieve when it behaves as a sieve in any argument.*

Definition 5.3. *Let $\varphi(X)$ be a PL formula on set X and Q a quantifier. φ is a* sieve condition *for Q if, whenever $\varphi(X)$ is true, Q behaves as a sieve, that is*
$\varphi(X) \to \neg\forall Y\ Q(X,Y)$ and $\varphi(X) \to \exists Y\ Q(X,Y)$ or
$\varphi(X) \to \neg\forall Y\ Q(Y,X)$ and $\varphi(X) \to \exists Y\ Q(Y,X)$.

Note that when a sieve condition fails, two things may happen: all sets qualify for a quantifier's definition, or none does. We distinguish both aspects because they tell us different things about how to proceed in query processing.

Definition 5.4. *Let $\varphi(X,Y)$ be a formula on sets X, Y and Q a standard quantifier. φ is an* implication *for Q if, whenever $\varphi(X,Y)$ is true, $Q(X,Y)$ is also true, that is, $\forall X,Y\ \varphi(X,Y) \to Q(X,Y)$. φ is a* constraint *for Q if, whenever $\varphi(X,Y)$ is true, $Q(X,Y)$ is false, that is, $\forall X,Y\ \varphi(X,Y) \to \neg Q(X,Y)$.*

The table in Figure 5.1 shows sieve conditions for some common quantifiers and their type (implication or constraint). As an example, consider the TPCH benchmark database ([1]) and the question "select the orders where **all but 10** suppliers are from Europe." We would compute this query by generating two subqueries: one (call it X) giving the orders and their suppliers, parametric on the orders; another (call it Y), listing the suppliers from Europe. The quantifier **all but (exactly) 10** can be computed by $p_1 = 10$. According to the procedure of section 4, an outer join between X and Y would be performed. But before carrying out this outer join, both X and Y can be checked for sieve conditions. Since only orders which have more than 10 suppliers may qualify (note that this is the last (bottom) case in table 5.1) we can, before outer join, add a group-by and an aggregation to count the number of suppliers for each order and remove tuples whose number is less than 10. This could reduce the input (and hence the cost) of the outer join[13] Note, though, that checking sieve conditions is a *heuristic* and is not guaranteed to improve performance. This is due to the fact that carrying out the checking has a cost (in this example, an extra grouping and aggregation, plus a selection, are introduced in the query plan[14]), while the expected benefit will vary from case to case, depending on features like data distribution (what is average number of suppliers per order? How are they distributed?) Therefore, the cost-based optimizer should consider the plans with and without sieve conditions, and choose the most efficient plan for a given query in a given database, using the information on data distribution available.

5.2 Optimization of $\mathcal{RA}$ Expressions

Some $\mathcal{RA}$ expressions produced by the interpreter in [3] involve outer joins. Since such outer joins are introduced to deal with GQs and are followed by selections that reflect the PL formula associated with the GQ, it is very easy to determine, by inspection, when the outer join can be simplified to a join, following the concepts and approach of [10]. Other traditional optimization techniques such as pushing down group-by (e.g., [13]) and exploiting *common subexpressions* (e.g., [32, 31]), can also be applied to the formulas produced by the interpreter. Pushing down group-by is very useful for the counter approach, since it needs to apply grouping to any parametric set term (correlated subquery); such push-down can reduce the cost of join or outer join. Also, when the first and the second $\mathcal{RA}$ expressions that are arguments to a GQ share common expressions (accessing the same relations, for instance), the shared expressions can be computed once and then reused by later processing. This technique is used repeatedly in our experiments (see section 6 for details).

Another special case which can be further optimized is the case where the two $\mathcal{RA}$ expressions that are arguments to a GQ are connected by a primary key/foreign key relationship. In this case, the join or outer join introduced by the formula may not be needed at all. For instance, in the previous example (end of section 4), $T1$ can be simplified, in either strategy, to
$GB_{pid,count(distinct\ sid)\ as\ cnt1}$(`teaches`)
since *sid* is a foreign key in `Teaches` and a primary key in `Student`.

5.3 Optimization of Multiple Quantifiers

Since SQL rules for subqueries are recursive, there can be cases where several nested GQs are used within a single query. Our previous expressions focused on single quantifier rules to simplify the presentation; but the approach can clearly be extended to handle several nested GQs. Focusing

[13] Note the relation of this reasoning with *semantic query optimization* techniques that identify *presuppositions* in a query.

[14] Although the grouping would be done anyways, so it is not an extra cost!

on this case, though, makes additional optimizations possible. Consider queries of the form

$$Q_1(Q_2(R[X], S[Y])[Z], T[W])$$

The process described previously can be used here, with Q_2 processed first and then Q_1 processed separately, using the previous result. At the algorithmic level there is an optimization strategy for this case: *pipelining.* In the counter approach, we have to join $R[X]$ and $S[Y]$, group the result by both XY and select according to the PL formula for Q_2, then join the result with $T[W]$, and group by WZ to do the final filtering according to the PL formula for Q_1. In this context, pipelining means that as soon as the selection for Q_2 produces some results, we can start joining with T. Thus, pipelining can also be used with set-based approaches, as far as we use non-blocking algorithms. But with GQs, this tactic presents another interesting possibility: to completely skip part of the work. Assume Q_2 is *upward monotone* in the first argument -that is, if $Q_2(R, S)$ and $R \subseteq R'$, then $Q_2(R', S)$. Then when we compute Q_2 we do not need to "finish" the computation: as soon as some value for XY satisfies the condition, we can pass it to the (outer)join with T. Further, if Q_1 is also upward monotone, as soon as a value associated with W has qualified for Q_1 we can stop further computation for that value. For instance, if Q_1 is "at least 2" and Q_2 is "at least half", for a given value $t[XY]$, we partition $R \bowtie S$ by this value, further partition this with values for $t[Z]$ and count the associated $R_{t,XY\cup Z}$. As soon as we get half of $R_{t,XY\cup Z}$ we can pass the associated value to the join with T. As soon as we have two such values that pass the test for Q_1, wen can move on to the next group for $R_{t,XY\cup Z}$. A symmetric idea can be exploited if both quantifiers are *downward monotone*; and if one GQ is downward and another upward monotone, similar strategies can be implemented (details are left out for lack of space). What is required is that the *flow of control* can be passed backwards to not only resume work, as in regular pipelining, but also to stop work -at least on certain groups- when a GQ signals that it has already "reached a decision" for a given parameter. Finally, note that this strategy can be combined with the use of sieve conditions. For instance, in the previous example, if for a certain value of Z Q_2 can only produce one associated XY value, then we know right away that the GQ "at least 2" will not succeed, so there is no need to join this value with T.

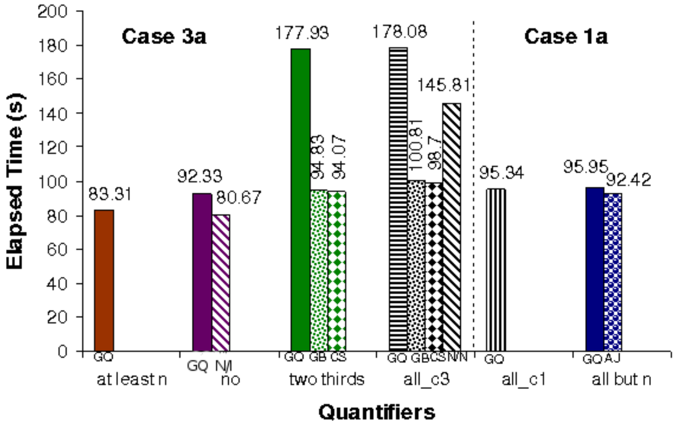

Figure 3: Query 1

6. EXPERIMENTS

To verify the efficiency of our approach, we simulated the implementation of SQL with GQs with the approach presented here by using a market-leading commercial RDBMS systems, which we call "System A" in the following. In our experiments, we created two copies of the TPC-H database [1], one with size 1GB and another one of size 10GB, both in System A, which was hosted on a server with an Intel Pentium 4 2.80GHz processor, two 36GB SCSI disks, and 1GB memory, running Red Hat Enterprise Linux WS release 3. We configured a buffer cache of size 32MB, and installed all data and indexes in a single disk. B+ tree indexes on the primary key of each base table are automatically built by System A, and indexes on all foreign keys were manually created. We marked as NOT NULL all the attributes used on the benchmarks queries. We carefully built a query set to be representative of the approach as a whole. Each query was written in plain SQL and submitted to System A, as a baseline. Then the same query was written in SQL with GQs, transformed into extended $\mathcal{RA}$, and then translated to traditional $\mathcal{RA}$ with the GQs in the query translated using the *counter* approach. Finally, this last expression was transformed back into SQL, and the resulting SQL submitted to System A. In the case of **all** and **no**, due to their importance in SQL and in our approach, we added several optimization techniques such as pushing down group-by (labeled GB) and reusing common subexpressions (labeled CS). Because the counter approach uses slightly different transformations to different types of queries, we used variations of our test queries to cover all cases. Cases are denoted as xy, where $x = 1, 2, 3$ (depending on whether p_1, p_2 or p_3 is used in the quantifier formula), and $y = a, b, c$, corresponding, respectively, to: only the first (second, both) argument to the GQ is parametric. In our test, we examine cases 1a to 1d and 3a to 3d (cases 1 and 2 are symmetric). For each query, we chose multiple quantifiers that could be expressed using p_1 or p_3 or a combination. Our goal was to test if different cases led to different performance, even for the same GQ. For **all** and **no**, we compared SQL with GQs to the plain SQL presented in most textbooks: using `NOT EXISTS` and `NOT IN` subqueries for **all** (labeled by N/N), and `NOT EXISTS` and `IN` for **no** (labeled by N/I). The performance metric used everywhere is the average elapsed time of multiple runs of a query. All the graphs of results plot elapsed time on the Y-axis and quantifiers with different approaches for each on the X-axis.

Experiment 1: Our first experiment was to test Case 1a and Case 3a on the following query (**Query 1**): "Select the orders (ordered between 1993-01-01 and 1993-03-01) where **Q** (**at least n** | **no** | **two thirds** | **all** | **all but n**) suppliers are from Europe." The extended SQL query is:

```
select O.o_orderkey
where Q  (select l_supkey
          from orders O, lineitem
          where o_orderkey = l_orderkey
   and l_date between 1993-01-01 and 1993-03-01)
         (select s_supkey
          from supplier, nation, region
          where s_nationkey = n_nationkey and
               n_regionkey = r_regionkey and
```

```
              r_regionname = 'EUROPE')
```

In Query 1, the parameter of the first subquery is `o_orderkey`; the second one has no parameters. This translates into our extended relational algebra as[15]:

$$Q\quad(\pi_{orderkey,supkey}(O \bowtie \sigma_{date\ between...}(L))[orderkey],$$
$$\pi_{supkey}(S \bowtie N \bowtie \sigma_{rname='Europe'}(R)))$$

We tested five quantifiers over Query 1, where **at least n** (n=3), **no**, and **two thirds** belong to Case 3a due to the formulas $p_3 >= 3$, $p_3 = 0$, and $|X_1| * 2/3 = p_3$ respectively; **all but n** (n=2) belongs to Case 1a because of $p_1 = 2$; and **all** belongs to both Case 1a and Case 3a because it can be computed either by $p_1 = 0$ or $|X_1| = p_3$. The results are shown in Figure 3. Although the original approaches using the formulas in Section 4 have different performance, when optimization techniques are applied all tested quantifiers have similar performance and reasonable cost. For Case 3a, we can see that **at least n** performs most efficiently compared to other quantifiers because it only needs join. The performance of **no** is slightly worse than **at least n** due to outer join. Both quantifiers perform faster than **two thirds** and **all**, which need an extra computation of $|X_1|$. For Case 1a, **all** and **all but n** perform comparably. For **all**, Case 1a performs much faster than Case 3a (even with optimization). Although no obvious common subexpressions are present in the query, quantifiers computed by $|X_1| * n = |X_1 \cap X_2|$ can be further optimized by either pushing down group-by (labeled by GB) or exploiting common subexpressions (labeled by CS). We apply the optimizations to **two thirds** and **all**; this results on performance about 40% faster than the original approaches. We also did an experiment on optimizing **all but n** when $n >= 1$ using antijoin (labeled by AJ) instead of outer join, and this improved performance over the original approach. We also compared **all** and **no** using our approach to the traditional SQL approach. Figure 3 shows that both SQL baseline queries no (N/I) and all (N/N) perform slightly faster than our approach for Case 3a, but all(N/N) performs worse than Case 3a with optimizations and Case 1a. The reason is that the `IN` or `NOT IN` subquery is a non-correlated subquery and is efficiently processed first by System A, and as a result the two subqueries are reduced to one. Then the `NOT EXISTS` is executed by antijoin. Here we have to point out that the ability of System A to use antijoin for `NOT EXISTS` depends on several factors: the NOT NULL definition on the involved attributes, the absence of multiple correlated subqueries, etc. Our original approach always uses outer join so as to work on a more general setting. By taking into account the NOT NULL, our approach can also be simplified to achieve better performance. Still, our optimized approach beats SQL in each but one case.

Experiment 2: To see how query structures affect performance, we modified Query 1 such that the `IN` or `NOT IN` subquery is a correlated subquery and the `NOT EXISTS` subquery is a non-correlated subquery. Accordingly, this leads to Case 1b and Case 3b. Our second experiment was performed on the following query (**Query 2**): "Select the orders where **Q** (**all** | **no**) suppliers from Europe (and with some customer complaints) are suppliers of that order." The extended relational algebra expression is:

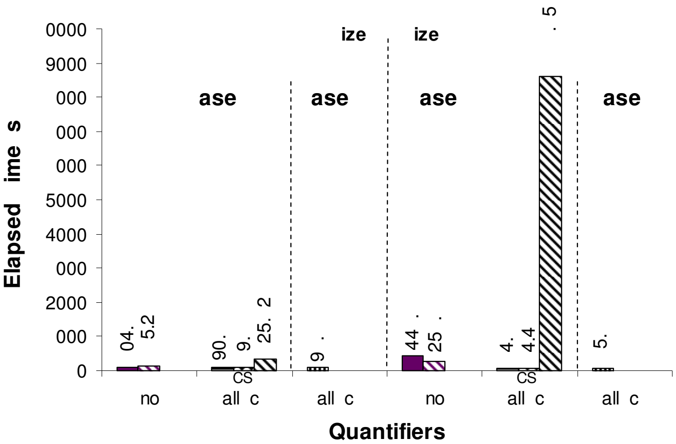

Figure 4: Query 2

$$Q($$
$$\pi_{supkey}(\sigma_{comment\ like\ '...'}(S) \bowtie N \bowtie \sigma_{rname='Europe'}(R)),$$
$$\pi_{supkey,orderkey}(\sigma_{date\ between\ ...}O \bowtie L)[orderkey])$$

We tested quantifiers **no** and **all**. Similar to the first experiment, **no** belongs to Case 3b, and **all** belongs to both Case 1b and Case 3b. We run queries with two different sizes (by changing the condition on `o_orderdate`). The results are shown in Figure 4. In both sizes, our approach performs significantly faster than the traditional SQL approach for **all**, especially in the larger one. But for **no**, in the larger size, our approach performs slightly worse because the cost of outer join is expensive for larger relations, particularly if the size of the intermediate join result is large. The traditional SQL approach for `no` has one `NOT EXISTS` non-correlated subquery and one `IN` correlated subquery, and uses the nested iteration method only for `IN`. Compared to outer join, the performance could be better if indexes are properly defined to speed up table access. Although our approach is not always better than the traditional approach, it is less sensitive to different sizes, while the traditional approach exhibits significant variations in performance, with its worst-case much worse than the GQ approach.

Experiment 3: Our third experiment was to test Case 1c and Case 3c. The test query (**Query 3**) is: "Select the common orders between the orders (with commit date earlier than receipt date and with the part size between 10 to 20) and the orders (with ship date earlier than commit date), and the latter has **Q** (**all** | **no**) parts of the former." The extended algebra expression is:

$$Q($$
$$\pi_{partkey,orderkey}(\sigma_{commitdate<receiptdate}(L))[orderkey],$$
$$\pi_{partkey}(\sigma_{shipdate<commitdate}(L)))$$

We tested two quantifiers **no** and **all**. Similarly, **no** belongs to Case 3c, and **all** belongs to both Case 1c and Case 3c. For **all**, Case 1c performs slightly worse than Case 3c because Case 1c needs a semijoin to obtain common orders first and then an outer join, while Case 3c can be done using join only due to p_3 always greater than zero, even though it needs to compute $|X_1|$. A further optimization can be

[15] To make the $\mathcal{RA}$ expressions fit into the text, we identify each relation by its first letter and shorten the conditions in selections.

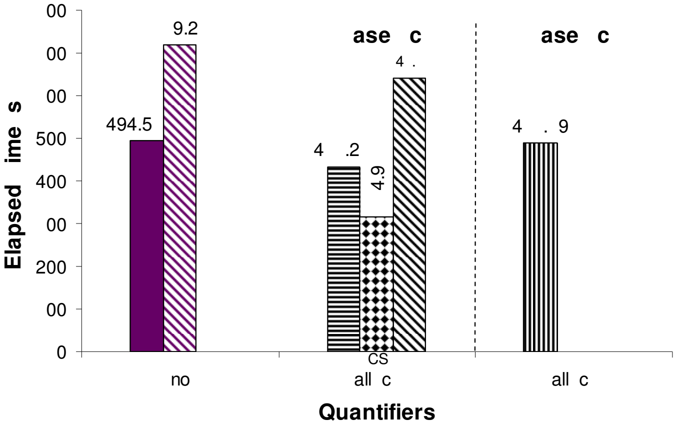

Figure 5: Query 3

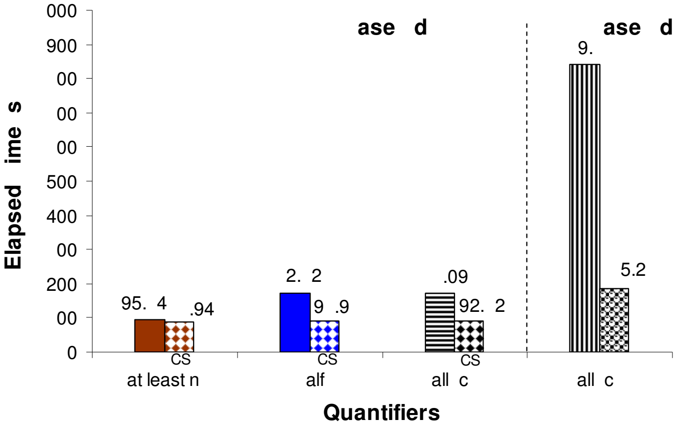

Figure 6: Query 4

made to Case 3c because there are common subexpressions when computing $|X_1| = p_3$, which only needs an outer join followed by computing two aggregations. The results shown in Figure 5 correspond to the above analysis. Similar to the previous experiments, we also run the query using the traditional SQL approach for **all** (N/N) and **no** (N/I); both perform worse than any of our approaches.

Experiment 4: Our fourth experiment was to test Case 1d and Case 3d on the following query (**Query 4**): "Select the pairs (supplier, customer) such that 'customer' is involved in **Q** (**at least n** | **half** | **all** |) of the orders (ordered between 1993-01-01 and 1993-02-01) from the 'supplier'." The extended algebra expression is:

$$Q(\\ \pi_{supkey,custkey,orderkey}((\sigma_{date...}O) \bowtie L)[supkey, custkey],\\ \pi_{orderkey}(\sigma_{date...}O))$$

We tested three quantifiers: **at least n** (n=2), **half**, and **all**, where **at least n** and **half** belong to Case 3d, and **all** belongs to both Case 3d and Case 1d. Since **at least n** is implemented by $p_3 >= 2$, **half** and **all** are implemented by $|X_1| * n = p_3$ ($n = 0.5$ for **half** and $n = 1$ for **all**), these three quantifiers can be implemented efficiently by join

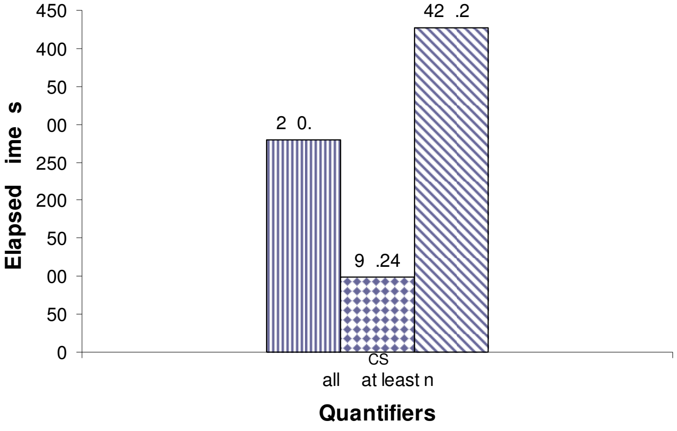

Figure 7: Query 5

operations like Case 3a. And, as common subexpressions are present between two set terms, the original approaches can be further optimized. Figure 6 shows the results. Using $p_1 = 0$ (Case 1d) to implement **all** is more expensive than Case 3d because a Cartesian product is required. The worst performance is the traditional SQL using `NOT EXISTS` and `NOT IN` for **all**, it gives results in about 7.5 hours (compared to a few minutes for our approach). The number is too big to be shown in Figure 6.

Experiment 5: Our fifth experiment was to test the proposed optimizations based on exploiting common subexpressions. We added one more quantifier to Query 1, obtaining the following (**Query 5**): "Select the orders (ordered between 1993-01-01 and 1993-03-01) where **all** suppliers are from Europe and **at least one** is from Germany." The extended algebra expression is:

$$\textbf{all}(\\ \pi_{orderkey,supkey}(O \bowtie \sigma_{date\ between\ ...}(L))[orderkey],\\ \pi_{supkey}(S \bowtie N \bowtie \sigma_{rname='Europe'}(R)))\cap\\ \textbf{at least one}(\\ \pi_{orderkey,supkey}(O \bowtie \sigma_{date\ between\ ...}(L))[orderkey],\\ \pi_{supkey}(S \bowtie \sigma_{nname='Germany'}(N)))$$

Query 5 has two quantifiers, **all** and **at least one**. It can be executed by intersection of the results of two separate queries, with each query following its own formula. Since these two queries are exactly the same except for the GQ used, the process can be optimized by executing the common subexpressions once and then computing two quantifiers simultaneously over the common subexpressions; this performs 65% faster than the original approach (see Figure 7). We also run the query in SQL using two subqueries `NOT EXISTS` and `NOT IN` (N/N)for **all**, and join operations for **at least one**, which performs 0.5 and 3.3 times slower than the original and the optimized approaches respectively.

Experiment 6: Our final experiment was to test optimizations based on sieve conditions (see subsection 5.1). We added sieve conditions to **at least n** (Case 3a) and **all but n** (Case 1a) on Query Q1. For both **at least n** ($n = 5$) and **all but n** ($n = 2$ and $n = 7$ respectively), we know $|X_1|$ must be greater than or equal to n. Thus, we added an aggregation to $E1$ which serves as a sieve condition and

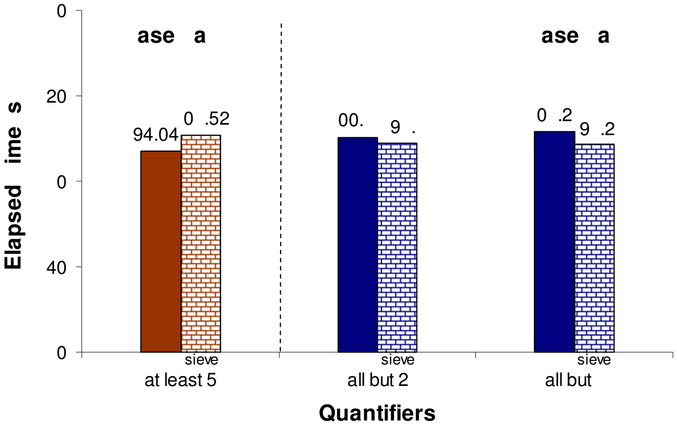

Figure 8: Query 6

discards unqualified tuples before the join (for **at least n**) or outer join (for **all but n**). The results are shown in Figure 8. For **at least n**, adding a sieve condition makes query performance worse because the original approach only needs join operations. For **all but n**, adding a sieve condition slightly improve query performance because the sieve condition makes the outer join operation less expensive and the result smaller by getting rid of useless tuples in advance. The difference is more significant for $n = 7$ than for $n = 2$.

More experiments were run, with results not shown for lack of space (we point out that we repeated a large part of our experiments in a 10GB TPC-H database.) Overall, the results obtained support the following conclusions: *first*, our approach handles a large class of queries in a uniform manner. This contrasts with SQL, where expression of different quantifiers requires different types of queries, some of them difficult to write and optimize. In this sense, it is interesting to note that upward monotonic GQs (like **some**, **at least n**) can be expressed easily in $\mathcal{RA}$/SQL; on the other hand, downward monotonic GQs (e.g., **all**, **no**) are difficult to express in $\mathcal{RA}$/SQL ([18] provides theoretical justification for this distinction). *Second*, our experiments show that our approach outperforms the traditional SQL approach (using `NOT IN`/`NOT EXISTS`) in complex queries involving downward monotonic operators, while doing as well as SQL with upward monotone ones. *Third*, our approach scales, both in the complexity of the query (see Query 5 above) and in the complexity of the data: when repeating experiments with larger databases, we saw a linear growth in all cases. On the other hand, SQL scalability is erratic, depending heavily on the type of query. *Finally*, our optimizations make a positive difference. Whenever used, they result in improvement, sometimes quite significant.

7. RELATED RESEARCH

GQs were introduced in [23] and refined in [20]. A large, theoretical body of work focuses on the complexity and expressive power that GQs bring to typical logics ([17, 14]). Due to lack of space, here we mention only a few directly relevant results.

As stated in section 3, the full concept of GQ is very powerful. A *type* is a finite sequence of natural numbers, $t = (k_1, \ldots, k_n)$; a GQ of type t on domain M is a relation on $M^{k_1} \times \ldots M^{k_n}$ that is closed under isomorphisms. In a fundamental result ([14]), Hella proved that the expressive power of quantifier classes goes up with the type: if we use t to denote first order logic extended with all GQs of type t, and $L < L'$ to denote that language L' has more expressive power than L, then
$(1) < (1,1) < (1,1,1) < \ldots < (2) < (2,1) < (2,1,1) < \ldots < (2,2) < \ldots < (3) < \ldots$

Quantifiers that deal only with sets (*monadic quantifiers*, of types (1), (1, 1), (1, 1, 1), . . .) have been studied extensively ([17, 30]). While such GQs are the most limited class in terms of expressive power, there are sound reasons to focus the present work on them. First, our approach is a practical one, and considerations of efficient implementation are paramount. Beyond monadic, complexity raises immediately and strongly: note that GQs of type (2) are, essentially, graph properties, so some of them represent NP-hard problems. Second, even simple set-based computations are not well supported at the language level; [18] shows that there is no efficient way to express most set operations in $\mathcal{RA}$. Third, type (1, 1) quantifiers cover reasonable practical uses. Research on formal linguistics has shown that many English sentences (as well as other Indo-European languages) can be formally analyzed using this type of GQ (see [29] for details). This research has also shown that quantifiers of higher types are rare in natural languages. Since querying is also a linguistic activity, we believe this gives some pragmatic support to our approach.

Work on quantification in query languages has a long history ([7, 8]). Most past work focuses exclusively on the universal quantifier, but [24] deals directly with set operations. There is an intuition behind most of this work that the one-tuple-at-a-time approach is not satisfactory to express set predicates. Efficient algorithms for direct implementation of universal quantification are given in [12]. Such algorithms were a precursor of modern day approaches to dealing with set-based computation ([25, 21]) that provide an alternative implementation for GQs. However, it remains unexplored how they would cope with an extendible (not fixed) list of GQs. Traditional approaches to universal quantification in the context of SQL have added outerjoins and grouping, which are difficult to deal with. This has motivated research to deal with this issue ([27]). Our naive approach is very similar to the SegmentApply operator, and hence our optimization follows some of the steps of [9]. When dealing with the universal operator, the plans generated by the definition of $\mathbf{all}(X, Y)$ as $|X| = |X \cap Y|$ are very similar to those of [6]. They implement their approach using a new operator called the Multidimensional join operator. While [26] focuses on universal quantification and gives several optimization choices based on $\mathcal{RA}$, for us universal quantification is but one case among many (although admittedly an important case). Note that we provide several options for computing universal quantification, and leave it up to the optimizer to decide which one to use for a given query. Subqueries and quantification in the context of XQuery are studied in [22], focusing on ordered structures. Incorporating order is an important idea in the context of XML, but outside the scope of our paper.

Finally, there are a few prior attempts to incorporate GQs in query languages ([15, 28]). The former argues for incorporating GQs into SQL in order to make queries easier to write. However, they only consider a finite, fixed set of GQ, and rely on rewriting an SQL query with GQs back into

SQL without GQs, even though it is acknowledged that this results in bad performance. On the other hand, [28] provides tailored algorithms to compute the quantifiers **some, all, not some, not all** very efficiently. The approach performs well and is highly scalable on the data, but not on the complexity of the query, and is very difficult to extend to other quantifiers. Again, we follow the approach of [3] because it can deal with an unbounded set of quantifiers while providing efficient implementation.

8. CONCLUSION AND FURTHER RESEARCH

We have introduced an extension of SQL and $\mathcal{RA}$ with Generalized Quantifiers. We have shown how GQs can be integrated in a standard relational framework. We have also given implementations and optimization techniques for the extended language, and have provided experimental evidence that the proposed approach provides significant performance improvements over standard SQL on certain queries. This approach is high-level and declarative, and therefore provides an excellent tool to study quantification in query languages. We are currently considering alternative implementations for GQs, as well as further optimization techniques. In the same vein, we are studying the use of GQs to express complex constraints and in other data models besides relational.

Acknowledgments: This research was sponsored by NSF under grant CAREER IIS-0347555. The second author wants to thank his program manager, Dr. Maria Zemankova, for her patience and support.

9. REFERENCES

[1] The TPCH benchmark, `www.tpc.org`.

[2] R. Abiteboul, S. Hull, , and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1994.

[3] A. Badia. *Generalized Quantifiers in Action*. Springer, 2010.

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

[5] B. Cao and A. Badia. A nested relational approach to processing sql subqueries. In *Proc. of SIGMOD*, pages 191–202, 2005.

[6] D. Chatziantoniou, M. O. Akinde, T. Johnson, and S. Kim. Md-join: an operator for complex olap. In *Proc. of ICDE*, pages 524–533, 2001.

[7] M. Dadashzadeh. An improved division operator for relational algebra. *Information Systems*, 14(5):431–437, 1989.

[8] C. Fragatelli. Technique for universal quantification in sql. *SIGMOD Record*, 20(3):16–24, 1991.

[9] C. Galindo-Legaria and M. Joshi. Orthogonal optimization of subqueries and aggregation. In *Proc. of SIGMOD*, pages 571–581, 2001.

[10] C. Galindo-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Trans. Database Syst.*, 22(1):43–74, 1997.

[11] R. A. Ganski and H. K. T. Wong. Optimization of nested sql queries revisited. In *Proc. of SIGMOD*, pages 23–33, 1987.

[12] G. Graefe and R. Cole. Fast algorithms for universal quantification in large databases. *ACM Trans. Database Syst.*, 20(2):187–236, 1995.

[13] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proc. of VLDB*, pages 358–369, 1995.

[14] L. Hella. Definability hierarchies of generalized quantifiers. *Annals of Pure and Applied Logic*, 43:235–271, 1989.

[15] P. Y. Hsu and D. S. Parker. Improving sql with generalized quantifiers. In *Proc. of ICDE*, pages 298–305, 1995.

[16] E. Keenan and L. Moss. Generalized quantifiers and the expressive power of natural language. In van Benthem and ter Meulen [29].

[17] P. Kolaitis and J. Väänänen. Generalized quantifiers and pebble games on finite structures. *Annals of pure and applied logic*, 74:23–75, 1995.

[18] D. Leinders and J. V. den Bussche. On the complexity of division and set joins in the relational algebra. In *Proc. of PODS*, pages 76–83, 2005.

[19] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. Ranksql: query algebra and optimization for relational top-k queries. In *Proc. of SIGMOD*, 2005.

[20] P. Lindstrom. First order predicate logic with generalized quantifiers. *Theoria*, 32:186–195, 1966.

[21] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *Proc. of SIGMOD*, pages 157–168, 2003.

[22] N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. In *Proc. of ICDE*, pages 239–249, 2004.

[23] A. Mostowski. On a generalization of quantifiers. *Fundamenta Mathematica*, 44:12–36, 1957.

[24] G. Ozsoyogly and H. Wang. A relational calculus with set operators, its safety, and equivalent graphical languages. *IEEE Trans. on Software Engineering*, 15(9):1038–1052, 1989.

[25] K. Ramasamy, J. Patel, J. Naughton, and R. Kaushik. Set containment joins: The good, the bad and the ugly. In *Proc. of VLDB*, pages 351–362, 2000.

[26] R. Rantzau and C. Mangold. Laws for rewriting queries containing division operators. In *Proc. of ICDE*, page 21, 2006.

[27] J. Rao, H. Pirahesh, and C. Zuzarte. Canonical abstraction for outerjoin optimization. In *Proc. of SIGMOD*, pages 671–682, 2004.

[28] S. Rao, D. Van Gucht, and A. Badia. Providing better support for a class of decision support queries. In *Proc. of SIGMOD*, pages 217–227, 1996.

[29] J. van Benthem and A. ter Meulen, editors. *Generalized Quantifiers in Natural Language*. Foris Publications, 1985.

[30] D. Westerstahl. *Handbook of Philosophical Logic*, volume IV, chapter Quantifiers in Formal and Natural Languages. Reidel Publishing Company, 1989.

[31] J. Zhou, P. Larson, J. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *Proc. of SIGMOD*, pages 533–544, 2007.

[32] C. Zuzarte, H. Pirahesh, W. Ma, Q. Cheng, L. Liu, and K. Wong. Winmagic: Subquery elimination using window aggregation. In *Proc. of SIGMOD*, pages 652–656, 2003.