University of Louisville
ThinkIR: The University of Louisville's Institutional Repository

Electronic Theses and Dissertations

12-2010

Generalized quantifiers in distributed databases.

Michael Dobbs
University of Louisville

Follow this and additional works at: https://ir.library.louisville.edu/etd

Recommended Citation

Dobbs, Michael, "Generalized quantifiers in distributed databases." (2010). *Electronic Theses and Dissertations.* Paper 360.
https://doi.org/10.18297/etd/360

This Master's Thesis is brought to you for free and open access by ThinkIR: The University of Louisville's Institutional Repository. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of ThinkIR: The University of Louisville's Institutional Repository. This title appears here courtesy of the author, who has retained all other copyrights. For more information, please contact thinkir@louisville.edu.

GENERALIZED QUANTIFIERS IN DISTRIBUTED DATABASES

By

Michael Dobbs
B.S., University of Louisville, 2009

A Thesis
Submitted to the Faculty of the
Graduate School of the University of Louisville
in Partial Fulfillment of the Requirements
for the Degree of

Master of Engineering

Department of CECS
University of Louisville
Louisville, Kentucky

December 2010

GENERALIZED QUANTIFIERS IN DISTRIBUTED DATABASES

By

Michael Dobbs
B.S., University of Louisville, 2009

A Thesis Approved On

____
Date

by the following Thesis Committee:

____
Thesis Director - Dr Antonio Badia

____
Dr Dar-jen Chang

____
Dr Tim Hardin

ACKNOWLEDGEMENTS

I am thankful to my advisor, Antonio Badia, whose guidance and support from the begining to the end enabled me to develop an understanding of the subject.

ABSTRACT

GENERALIZED QUANTIFIERS IN DISTRIBUTED DATABASES

Michael Dobbs

June 22, 2010

Optimizing queries in a distributed database is quite difficult. This work proposes defining new generalized quantifiers which operate on sets rather than tuples. These quantifiers would allow for easier optimization in a horizontally distributed database. These operators are scalable with respect to both the number of hosts in the environment and the size of the data used.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS iii
ABSTRACT iv
LIST OF TABLES viii
LIST OF FIGURES ix

CHAPTER

I INTRODUCTION 1

II BACKGROUND 3

III PROPOSED APPROACH 13

A Generalized Quantifiers 13
B A Query Language with Generalized Quantifiers 15
C Application of $QL(Q)$ to SQL 19
D Implementation 22
E Some 24
F At Least 25
G At Most 26
H Ratio Operators 26
I All But 26

IV Experiments 27

A Technical Setup 27
B Some 28
1 SQL 28

2 Naive Algorithm . . . 29
3 Set Operator Semijoin . . . 29
4 Comparison . . . 30
C At Least . . . 31
1 SQL . . . 32
2 Naive Algorithm . . . 33
3 Set Operator Semijoin . . . 33
4 Comparison . . . 34
D At Most . . . 37
1 SQL . . . 37
2 Naive Algorithm . . . 37
3 Set Operator Semijoin . . . 38
4 Comparison . . . 38
E Ratio Operators . . . 39
1 SQL . . . 40
2 Naive Algorithm . . . 41
3 Set Operator Semijoin . . . 41
4 Comparison . . . 42
F All . . . 44
1 SQL . . . 44
2 Naive Algorithm . . . 45
3 Set Operator Semijoin . . . 45
4 Comparison . . . 45
G AllBut . . . 47
1 SQL . . . 47
2 Naive . . . 47
3 Semijoin . . . 49
4 Comparison . . . 49

H Summary and Evaluation of Results 51

V CONCLUSION **53**

VI APPENDIX **54**

REFERENCES **73**

CURRICULUM VITAE **75**

LIST OF TABLES

TABLE **Page**

1 Dataset Sizes . . . 9
2 Dataset Sizes . . . 28
3 Some(x) SQL Run-times . . . 55
4 Some(x) Naive Run-times . . . 56
5 Some(x) Semijoin Run-times . . . 57
6 AtLeast(x) SQL Run-times . . . 58
7 AtLeast(x) Naive Run-times . . . 59
8 AtLeast(x) Semijoin Run-times . . . 60
9 AtMost(x) SQL Run-times . . . 61
10 AtMost(x) Naive Run-times . . . 62
11 AtMost(x) Semijoin Run-times . . . 63
12 Ratio(x) SQL Run-times . . . 64
13 Ratio(x) Naive Run-times . . . 65
14 Ratio(x) Semijoin Run-times . . . 66
15 All(x) SQL Run-times . . . 67
16 All(x) Naive Run-times . . . 68
17 All(x) Semijoin Run-times . . . 69
18 AllBut(x) SQL Run-times . . . 70
19 AllBut(x) Naive Run-times . . . 71
20 AllBut(x) Semijoin Run-times . . . 72

LIST OF FIGURES

figure		Page
1	TPCH Schema	6
2	Some on TPCH 1X dataset with 3 hosts	30
3	Some on TPCH 10X dataset with 3 hosts	31
4	Some on TPCH 100X dataset with 3 hosts	31
5	Some on TPCH 1X dataset with 3 hosts	31
6	Some on TPCH 10X dataset with 3 hosts	32
7	Some on TPCH 100X dataset with 3 hosts	32
8	At Least(x) on TPCH 1X dataset with 3 hosts	35
9	At Least(x) on TPCH 10X dataset with 3 hosts	35
10	At Least(x) on TPCH 100X dataset with 3 hosts	35
11	At Least(x) on TPCH 1X dataset with 5 hosts	36
12	At Least(x) on TPCH 10X dataset with 5 hosts	36
13	At Least(x) on TPCH 100X dataset with 5 hosts	36
14	At Most(x) on TPCH 1X dataset with 3 hosts	38
15	At Most(x) on TPCH 10X dataset with 3 hosts	38
16	At Most(x) on TPCH 100X dataset with 3 hosts	39
17	At Most(x) on TPCH 1X dataset with 5 hosts	39
18	At Most(x) on TPCH 10X dataset with 5 hosts	39
19	At Most(x) on TPCH 100X dataset with 5 hosts	40
20	Ratio operator on TPCH 1X dataset with 3 hosts	42
21	Ratio operator on TPCH 10X dataset with 3 hosts	43
22	Ratio operator on TPCH 100X dataset with 3 hosts	43
23	Ratio operator on TPCH 1X dataset with 5 hosts	43

24 Ratio operator on TPCH 10X dataset with 5 hosts 44
25 Ratio operator on TPCH 100X dataset with 5 hosts 44
26 All(x) on TPCH 1X dataset with 3 hosts 45
27 All(x) on TPCH 10X dataset with 3 hosts 46
28 All(x) on TPCH 100X dataset with 3 hosts 46
29 All(x) on TPCH 1X dataset with 5 hosts 46
30 All(x) on TPCH 10X dataset with 5 hosts 46
31 All(x) on TPCH 100X dataset with 5 hosts 47
32 AllBut(x) on TPCH 1X dataset with 3 hosts 50
33 AllBut(x) on TPCH 10X dataset with 3 hosts 50
34 AllBut(x) on TPCH 100X dataset with 3 hosts 50
35 AllBut(x) on TPCH 1X dataset with 5 hosts 50
36 AllBut(x) on TPCH 10X dataset with 5 hosts 51
37 AllBut(x) on TPCH 100X dataset with 5 hosts 51

CHAPTER I

INTRODUCTION

Databases have classically been housed in large specialized hardware. However, due to rapidly decreasing commodity hardware prices, it has become increasingly desirable to distribute databases between hosts. Horizontal scalability, distributing processing between more processors, allows processing capacity to scale with the costs of ever cheapening commodity hardware.

Distributed processing presents its own unique set of challenges and problems. Specialized algorithms must be designed to be run on distributed hosts. The general concept behind distributed computing is that work is divided into units and these work units are sent to hosts for processing. Each host then performs the work and returns the results. Communication commonly occurs over commodity network hardware. For that reason, communication is usually considered an expensive resource and should be judiciously managed.

When applied to relational databases, the data is often distributed between the hosts of the system. Each host is responsible for a subset of the total volume of data. Querying distributed data presents a unique sent of challenges and opportunities. Challenges include minimizing network traffic and producing correct results. Opportunities include exploiting parallelism for performance gains and increased data capacity.

Currently, modern commercial databases do not do much to optimize queries with subqueries in a distributed setting. This work aims to extend the SQL language in such a way that some common queries can be rewritten in a manner more suitable for distributed optimization. This extension, inspired by generalized quantifiers, treat subqueries as sets. New operators are defined which express the degree to which two sets are related. For

instance, an operator could gauge how much one set may be contained within another. This allows the system to work on processing distributed set operations, rather than arbitrary tuple-level operations as in SQL. Consequently, query optimization can be carried out more easily.

In this thesis, generalized quantifiers will be described in more detail, a query language will be proposed based upon generalized quantifiers, and a collection of specific quantifiers will be implemented and benchmarked. Chapter II provides background information about SQL. Chapter III presents and explains an approach and the algorithms to implement it. Chapter IV describes how the algorithms were tested and the results obtained. Chapter V presents a conclusion.

CHAPTER II

BACKGROUND

A relational database is a database in which records are stored in the relational model. The relational model was first proposed by Edgar F Codd in 1970. Almost all commercial databases implement this design today. In the relational model a database is represented by one or more relations. A relation is described by a table. That table has rows and columns. The columns describe aspects of the relation. The rows are elements of data, which are combinations of the descriptions provided by the columns.

One special type of column is called the primary key. The primary key is composed of one or more columns and serves as a unique identifier for that row. Because it is unique, no two rows in a table can share the same primary key. The uniqueness of the primary key allows the database to refer to a particular row in other places of the system by only the primary key.

One place where it is common to refer to other rows is in a foreign key relationship. A foreign key is a special kind of column or columns. This key refers to the primary key of another relation. The system will verify that any record mentioned in the foreign key indeed exists in the referenced table. This is a useful method of maintaining data integrity and enforcing sound database design. One key aspect of relational database design is to remove duplication anywhere in the database. When information is duplicated it must be maintained in each location. Thus, if any instance of the data is not updated the database will no longer be consistent. Additionally, duplicated data wastes space. For instance, if a catalog has products and each product has a sales representative the products and sales representatives should be stored in two different tables. A foreign key should exist in products pointing to the appropriate sales representative. Data would be duplicated if

every product also stored all of the information about every sales representative.

Each row of a table is written to disk sequentially in a file. Files are composed of one or more disk pages. A disk page is the smallest unit of data that a disk can store. Typically, a disk page is 4 kilobytes. In some implementations, records are written to disk so that they never span multiple pages. The other alternative is for records to be written so that they span pages. Non-spanning implementations can be considered faster because they do not need to go through the code to split the record. However, they are less efficient in terms of disk space used. The converse is true of spanning systems. They are more space efficient but less time efficient.

In order to find a record in the system, the file must be read from the beginning until that record is found. If the record is not found, then the entire file will have to be read. This method becomes increasingly laborious as the file size grows. Indices are a specialized mechanism to make record retrieval more efficient.

Index structures include but are not limited to B-trees and hash tables. B-trees are more flexible because they can handle greater than or less than operators by simply looking at either the left or right children of a tree node respectively. Hash tables are theoretically faster for single element access. However, hash tables generally only support comparing for strict equality. Indices can aid in retrieval time; unfortunately, they also come at some cost. Indices must be maintained whenever a record is inserted, updated, or deleted. That maintenance requires time and thus indices will slow the aforementioned operations.

In SQL, each table can be thought of as a set of elements of a given format. That format is specified by the table's schema. The schema is the database's formal definition of the structure and interaction of the database's tables. Real world data often cannot be adequately represented with a single table. For that reason, SQL provides for relations between tables. One of the most fundamental ways of working with these relations is to join them.

One important feature of SQL is that it is a declarative language. This means that SQL queries only specify what information is requested. How to retrieve that information

is left to the system's implementation. Thus, different implementations may perform the retrieval differently. As long as they both supply the correct results, both are considered valid SQL databases. This creates room for database vendors to design systems to specific factors. Common factors include security, stability, and scalability.

Various benchmarks have been created to compare the performance of databases. TPCH is one such benchmark. TPCH is a product from the Transaction Processing Performance Council. It aims to benchmark the decision support capabilities of a database. Decision support generally consists of answering queries that analyze large amounts of legacy data to direct business decisions. The TPCH schema is diagrammed in Figure 1.

Queries are composed of different operators. Queries can be classified in to different types depending on which operators are used. A Select-Project (SP) query is the simplest type. An SP query only involves one table. Below is an example of an SP query in the TPCH schema:

```
SELECT o_orderkey
FROM orders
WHERE o_custkey = 1
```

This SP query selects the orderkey from all of the orders that were ordered by a specific customer. Another, more complicated, query is the Select-Project-Join (SPJ) query. The following is an example of an SPJ query:

```
SELECT l_partkey
FROM orders, lineitem
WHERE o_custkey = 1
AND l_orderkey = o_orderkey
```

The SPJ query is similar to the SP query except it has a join, hence the "J" in the description. In this example, the join is between *orders* and *lineitem*. Joins create the cross product of the two sets involved. In order to restrict ourselves to only the relevant combinations, a condition was added in the "WHERE" block so that the orderkeys must

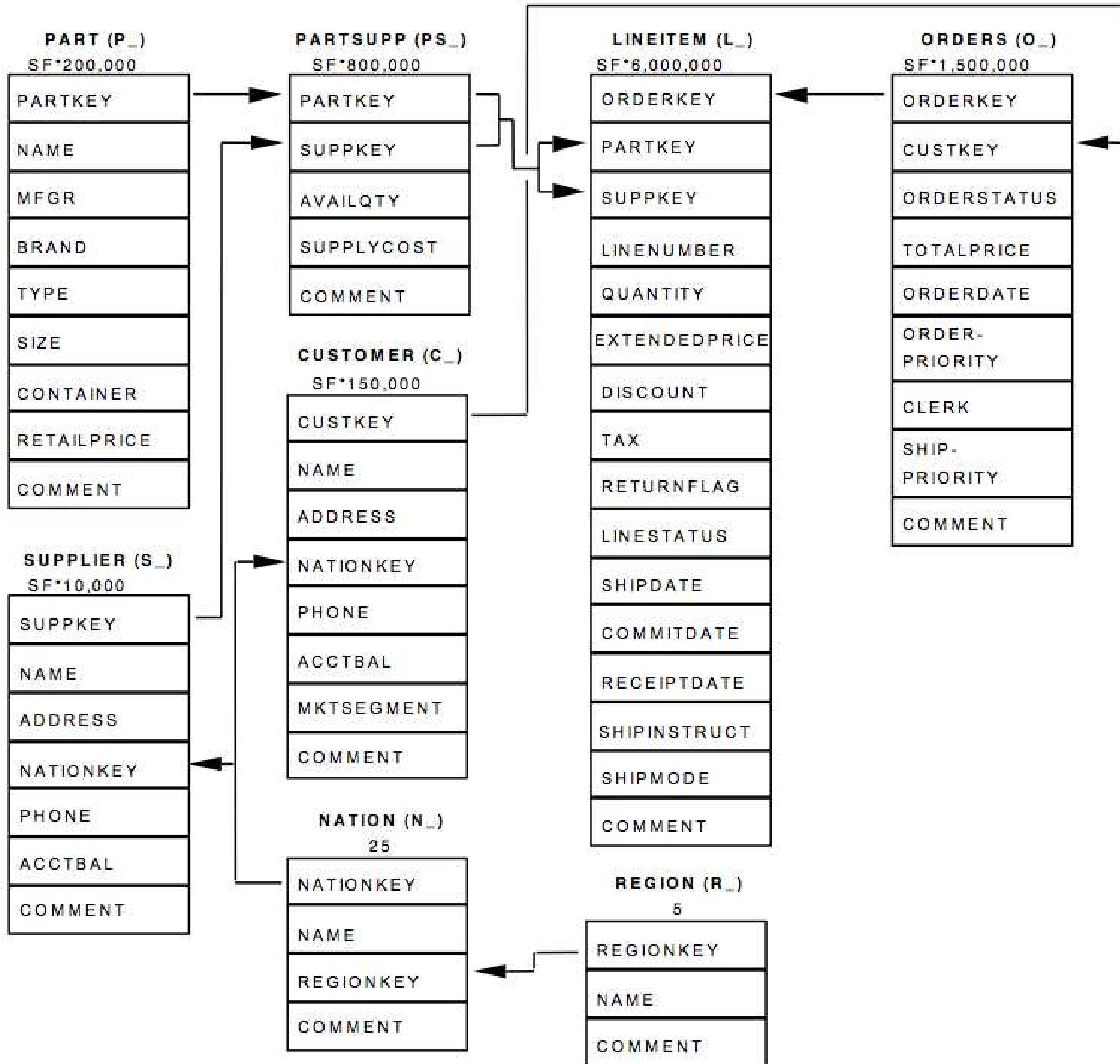

Figure 1. The TPC-H Schema. Source: TPC BENCHMARK H (Decision Support) Standard Specification

match. Group-By is an operator that can be added to an SP or SPJ query to make an SPG or SPJG query respectively. Below is an example of an SPJG query:

```
SELECT o_orderkey, count(l_partkey)
FROM orders, lineitem
WHERE o_custkey = 1
AND l_orderkey = o_orderkey
GROUP BY o_orderkey
```

This SPJG query lists all of the orders and how many parts were ordered by customer 1. The Group-By operator creates tuples that are not necessarily in the underlying tables. Notice the returned tuples are orderkeys and a count. None of the underlying tables, lineitem and orders, have the count column. Another useful operator is the Having operator. Having restricts results like the Where operator but it acts on the tuples returned by the Group-By rather than the tuples in the underlying tables. Queries using the Having operator must use the Group-By operator. Queries with a Having clause can be symbolized with an H at the end. Below is an SPJGH query:

```
SELECT o_orderkey, count(l_partkey)
FROM orders, lineitem
WHERE o_custkey = 1
AND l_orderkey = o_orderkey
GROUP BY o_orderkey
HAVING count(l_partkey) > 3
```

This query is like the previous example except that it adds the restriction that the number of partkeys must be greater than 3.

In order to answer a query, the system will generally have multiple options for orders of joining tables, when to apply projections, etc. The wrong choice can require orders of magnitude more work than more efficient alternatives. This would bring the system to a crawl. Thus, it is important to try to select the optimal query plan. Unfortunately,

evaluating every query plan and picking the optimal one is an NP-complete problem in general. This level of optimization is, for practical purposes, too time consuming to be solved in real-time. Modern commercial systems use a combination of heuristics and proprietary techniques to produce query plans of sufficient efficiency. Because there is no freely published 'perfect' solution, there is fierce competition between vendors to produce query optimizers that are both fast and produce efficient query plans.

Historically, databases have been housed on large centralized servers. In this architecture, all of the data is stored on one server. A distributed database stores data across multiple servers. The majority of query processing optimization has been focused on non-distributed databases. Optimizing queries in distributed databases is difficult because it requires peeling back some of the assumptions used in centralized databases. One important assumption used in centralized databases is the closed world assumption, abbreviated CWA [1]. The CWA states that anything not known to be true is false. Thus, any and all information is held within the database. To illustrate this point, an example is provided below:

Suppose a large company uses multiple databases DB_1, DB_2, DB_3. The company processes lots of orders for the parts it sells. In order to accommodate the workload, orders are stored in whichever database as the lowest work load at the time. Thus, orders are partitioned across the hosts horizontally. Horizontal partitioning divides tuples between hosts. Vertical partitioning divides columns between hosts. Relational technology is used to implement the databases. Assume each database has the relation *Lineitem* from the TPCH schema described above, where *orderkey* is the order's identifier and *partkey* is the part's identifier, but each database only sees its own part of the world and not all of it (i.e *Lineitem* can be considered as *horizontally partitioned* among databases). The term *system* refers to all of the databases together, and the term *node* refers to an individual database. When the system receives a query, the user expects back an answer that takes into account all the information in the system, that is, that is correct with respect to the totality of the data. Since we do not want to make a commitment to a particular architecture, we assume

that the node that receives the query can somehow communicate with all other nodes, and in particular send requests to them for any information needed.

TABLE 1

An example of a horizontally partitioned database

DB1		**DB2**		**DB3**	
Lineitem		Lineitem		Lineitem	
orderkey	partkey	orderkey	partkey	orderkey	partkey
O1	P1	O1	P2	O2	P1
O2	P2	O2	P3	O4	P2
O3	P1	O3	P1	O5	P3
O4	P1				

Table 1 shows the data at a certain point in time. Assume now that the following queries are issued by a user:

- Query 1: *Which orders ordered part P1?*. Clearly, the answer is the set {O1, O2, O3, O4}. In SQL, the query could be written as

  ```
  SELECT l.orderkey
  FROM lineitem l
  WHERE l.partkey = 'P1'
  ```

  This is a Select-Project query that can be simply answered by sending a copy of the query *as is* to each database and taking the union of the answers.

- Query 2: *Which orders share a part with O1?*. This requires a self-join of *Lineitem* with itself. In SQL, the query could be written as

  ```
  SELECT l2.orderkey
  FROM Lineitem l1, Lineitem l2
  WHERE l1.orderkey= 'O1' and l1.partkey = l1.partkey
  ```

This is a Select-Project-Join (SPJ) query. If we send this query as is to each database, DB_1 will return the answer {O3, O4}, and DB_2 and DB_3 will return the empty answer. However, the real answer over the system is {O3, O4, O2}. It can be seen that O2 and O1 are both ordered P2 but since the tuples specifying so are in different databases, they are never part of any local join; therefore O2 is not retrieved. There is no obvious way to decompose the given query in order to obtain the correct answer.

- Query 3: *Which orders share at least two parts with order O1?.* This query can be expressed as three self-joins of $WorksOn$ with itself, or as one self join followed by a group-by and a count, as follows:

```
SELECT l1.orderkey
FROM Lineitem l1, Lineitem l2
WHERE l1.partkey = l2.partkey and
      l2.orderkey = 'O2'
GROUP BY l1.orderkey
HAVING count(l1.orderkey) > 2
```

  Note that this is a Select-Project-Join-Groupby (SPJG) query on this formulation, but it could be written as an SPJ query too. In either case, it is easy to see that if we send copies of the query as is, all databases will return the empty answer, missing O2 (O1 and O2 both ordered P1 and P2). Again, there is no clear way to break down the query (note that it could be written in two very different ways).

- Query 4: *Which orders do not share any parts with O1?* This query can be formulated as the negation of Query 2; in SQL, it could be written in a number of ways: using subqueries and NOT IN, correlated subqueries and NOT EXISTS, or simply set difference. For instance, the second approach yields

```
SELECT l1.orderkey
FROM lineitem l1
```

```
WHERE NOT EXISTS (SELECT l2.orderkey
                  FROM Lineitem l2
                  WHERE l2.orderkey = 'O2'
                        and l1.partkey = l2.partkey)
```

In any case, this is not an SPJG query. As before, sending the query as it is does not provide the right answer; DB_1 yields {O2}, DB_2 yields {O2,O3} and DB_3 yields {O2,O3,O5}. However, the correct answer is {O5}. It is not obvious how to combine the partial answers into the correct one, and it is not obvious how to break down the original query to retrieve the information (and recall that it could be expressed in several different ways).

- Query 5: *List the orders that share all the parts with O1.* It is well known that universal quantification is not directly supported in SQL; as before, we have several options, all involving subqueries (with NOT IN, or NOT EXISTS, or EXCEPT; or even subqueries with grouping and count). One typical approach is to write:

```
SELECT l1.orderkey
FROM Lineitem l1
WHERE NOT EXISTS
      (SELECT l3.partkey
       FROM Lineitem l3
       WHERE l3.orderkey = 'O1'
             and l3.partkey NOT IN
             (SELECT l2.partkey
              FROM Lineitem l2
              WHERE l2.orderkey = l1.orderkey))
```

Again, this is not an SPJG query. And again, sending the query as is will not retrieve the right answer, but there is no obvious way to handle the query in all its possible variations.

Thus, only one of the above five queries is answered correctly by the system. These incorrect answers are caused by each database using the CWA with nothing but the data it has available. A person could analyze each case, craft appropriate SQL queries to send to each node, and combine the information returned into a correct answer. This reliance on manual labor underscores a shortcoming of distributed query processing that can hopefully be reduced, if not eliminated, with the approach proposed in the follow chapter.

To generalize the above examples, it is straightforward to express set containment requests for some elements or all elements from one set contained in another. Expressing more complex operators like Half, 25%, All But 3, At Least 3, etc requires subqueries. Support for query optimization with subqueries in a distributed setting is currently lacking from most commercial systems.

In addition to the semantic gap in distributed queries the costs of implementation are also not trivial. Sending messages from one host to another costs network resources. These resources must be managed diligently in any suitable solution.

In order to incorporate such operators in a commercial grade database, the underlying algorithms must be at least as scalable as existing database algorithms with respect to the amount of data processed. For reference, the 'order by' clause in SQL sorts the results in O(n lg n) time given enough memory. A suitable algorithm should also be horizontally scalable as well. Horizontal scalability refers to the algorithm being scalable across multiple hosts. The rest of this work will focus on demonstrating both the data size and horizontal scalability of the algorithms used to implement a more robust set of set containment operators.

CHAPTER III

PROPOSED APPROACH

The following section will explore some general algorithms for set comparison. Details about specific operators and their analysis and optimization will be described in subsequent sections. These algorithms assume that the sets are horizontally split between multiple hosts and that duplicate records do not exist. The algorithms to be described will exploit some properties of generalized quantifiers. As such, a description of generalized quantifiers and some of their properties will be provided. The properties of these quantifiers will then be used to construct a query language

A Generalized Quantifiers

Historically, Generalized Quantifiers(GQs) were mainly studied by logicians. Mostowski explored the fundamentals ([10]). The original goal was to be able to express properties that are not first order logic-definable. Lindstrom ([12]) refined the concept and studied logics with GQs in a general setting. The following will use a restricted version of the general concept. Specifically, the following work will be constricted to using generalized quantifiers of type $[1,1]$, GQs with two parameters. Both parameters are sets. GQs of this type have been found to be the most common type of GQ used in query languages ([2, 3, 4]).

In the interest of disambiguation, $Q, Q_1, ...$ will represent variables over GQs. $Q(A, B)$ indicates that sets A, B are related by Q's denotation. $|A|$ represents the cardinality of set A. Below are a few natural examples, particular quantifiers are in boldface.

most is given here the sense of "more than half"; other meanings are possible, but

$$
\begin{array}{lcl}
\textbf{all} & = & \{A, B \subseteq M | A \subseteq B\} \\
\textbf{some} & = & \{A, B \subseteq M | A \cap B \neq \emptyset\} \\
\textbf{no} & = & \{A, B \subseteq M | A \cap B = \emptyset\} \\
\textbf{not all} & = & \{A, B \subseteq M | A \not\subseteq B\} \\
\textbf{at least n} & = & \{A, B \subseteq M || A \cap B| \geq n\} \\
\textbf{at most n} & = & \{A, B \subseteq M || A \cap B| \leq n\} \\
\textbf{(exactly) n} & = & \{A, B \subseteq M || A \cap B| = n\} \\
\textbf{more} & = & \{A, B \subseteq M || A| > |B|\} \\
\textbf{most} & = & \{A, B \subseteq M || A \cap B| > |A - B|\} \\
\textbf{H} & = & \{A, B \subseteq M | \ | A | = | B |\} \\
\textbf{n\%of} & = & \{A, B \subseteq M \mid |A \cap B| \times \frac{100}{n} = |B|\}
\end{array}
\tag{1}
$$

some require the introduction of a metric in the domain. Traditional first-order quantifiers $\exists$ and $\forall$ bind only one variable in one formula. It can be seen that **all**(A, B) corresponds to the first order formula $\forall\ x\ (A(x) \rightarrow B(x))$, while **some**$(A, B)$ corresponds to $\exists\ x\ (A(x) \wedge\ B(x))$. These are the *guarded* forms of quantification often used in queries; the reason is that they make explicit the *domain of discourse*, that is, the group of objects that the query is considering, and that should be used to evaluate the quantifier. This usage of type $[1, 1]$ quantifiers makes them particularly useful in a distributed context.

Sometimes, other constraints have been added to the definition of a GQ. Some definitions imply that the behavior of a quantifier is independent of the context, as is the case for the usual logic constants. This is a desirable property. There is an axiom that makes context independence a characteristic:

Definition A.1 *(EXT) Quantifier Q follows EXT if for all M, M', all A, B such that $A, B \subseteq M \subseteq M'$, $Q_M(A, B)$ iff $Q_{M'}(A, B)$*[1].

The importance of this concept is that it makes GQs behave like *domain independent* operators, thus satisfying the basic conditions usually attributed to query language operators ([1]). While not all GQs are domain independent, all the ones used in this paper do have this property.

These definitions are from [14]. Quantifiers with this property are called *monotone*

[1] *EXT stands for* extensionality.

in [7] and *monotonic* in [8], where upward monotone quantifiers are called increasing and downward monotone decreasing. A monotonically increasing quantifier will only increase or not change. A monotonically decreasing quantifier will only decrease or not change. An important property to be exploited later is the following: since monadic quantifiers deal with sets, their definition can be completely specified by *cardinality properties*. In particular, for Q of type $[1,1]$, any sets A, B, $Q(A,B)$ is completely determined by four numbers, $|A \cap B|$, $|A - B|$, $|B - A|$, and $|M - (A \cup B)|$. For quantifiers that follow EXT, or in finite models, only the first three are needed. This property is already exploited in other research ([2, 5, 14, 13]).

B A Query Language with Generalized Quantifiers

This section will propose a query language that exploits Generalized Quantifiers. The language will be composed of variables, constants, predicate, and $[1,1]$ quantifier symbols. The following assumes a set $\mathcal{V}$ of variables, a set $\mathcal{C}$ of constants, a set $\mathcal{R}$ of relation names and a set $\mathcal{Q}$ of quantifier names are specified.

Definition B.1 The *quantified language* $QL(\mathcal{Q})$ is made up of set terms and formulas. A set term or formula φ has associated a set of free variables (in symbols, $Fvar(\varphi)$). Set terms, formulas, and their free variables are defined as follows:

1. *Basic terms*

   (a) If $x \in \mathcal{V}$ then x is a basic term. $Fvar(x) = \{x\}$

   (b) If $c \in \mathcal{C}$ then c is a basic term. $Fvar(c) = \emptyset$

   (c) '_' is a term, called an *anonymous* term. $Fvar(_) = \emptyset$

2. *Set terms*

   (a) If φ is a formula[2], $\{x_1, \ldots, x_m\} \subseteq Fvar(\varphi)$ with $m \geq 1$, and $\{y_1, \ldots, y_r\} \subseteq Fvar(\varphi)$, with $r \geq 0$, then $\{x_1, \ldots, x_m(y_1, \ldots, y_r) \mid \varphi\}$ is a set

[2]Set terms and formulas are mutually recursive in $QL(\mathcal{Q})$. The definition of formula is below.

term.

$Fvar(\{x_1, \ldots, x_m \mid \varphi\}) \;=\; Fvar(\varphi) \;-\; (\{x_1, \ldots, x_m\} \cup \{y_1, \ldots, y_r\})$.

(b) If S_1, S_2 are set terms, $S_1 \cup S_2$ is a set term, provided that

$Fvar(S_1) = Fvar(S_2)$.

$Fvar(S_1 \cup S_2) = Fvar(S_1) \cup Fvar(S_2)$;

3. Formulas:

   (a) If $R \in \mathcal{R}$ and $t_1, \ldots, t_{arity(R)}$ are basic or anonymous terms, then $R(t_1, \ldots, t_{arity(R)})$ is a (**basic**) formula.

   $Fvar(R(t_1, \ldots, t_{arity(R)})) \;=\; Fvar(t_1) \cup \ldots \cup Fvar(t_{arity(R)})$.

4. If φ_1 is a formula and φ_2 is a formula, then $\varphi_1 \wedge \varphi_2$ is a formula.

   $Fvar(\varphi_1 \wedge \varphi_2) = Fvar(\varphi_1) \cup \; Fvar(\varphi_2)$.

   Also, if φ_1 is basic formula and φ_2 is a basic formula, then $\varphi_1 \wedge \varphi_2$ is a basic formula.

5. If φ is a formula, then $\varphi \wedge t_1\theta\ t_2$ (with θ one of $=, \leq, \geq, <, >$ and t_i is a basic term $(i = 1, 2)$) is a formula provided that

   (a) If θ is $=$, then $Fvar(t_1 \;=\; t_2) \cap \; Fvar(\varphi) \; \neq \emptyset$.

   (b) If θ one of $\leq, \geq, <, >$, then $Fvar(t_1\theta\ t_2) \; \subseteq Fvar(\varphi)$.

   $Fvar(\varphi \wedge t_1\theta\ t_2) = Fvar(\varphi) \cup Fvar(t_1) \cup Fvar(t_2)$. Also, if φ is basic, then $\varphi \wedge t_1\theta t_2$ is also basic.

6. If $Q \in \mathcal{Q}$ is a quantifier name and S_1, S_2 are set terms, then $Q(S_1, S_2)$ is a formula.

   $Fvar(Q(S_1, S_2)) = Fvar(S_1) \cup Fvar(S_2)$.

   $Q(S_1, S_2)$ is called a **quantified** formula.

A *query* is a set term S such that $Fvar(S) = \emptyset$, that is, a set term with no free variables. A *sentence* is a formula φ such that $Fvar(\varphi) = \emptyset$, that is, a formula with no free variables. Call a query $q = \{\vec{x} \mid \varphi(\vec{x})\}$ *basic* if φ is a basic formula, and *quantified* if φ is a quantified formula. Note that all formulas in the query language are either basic or quantified.

To give a feel for the language, and to introduce the semantics in an intuitive way, we give an example. Recall the TPCH schema described in Figure 1. Specifically, recall the relations *Lineitem*, *Orders*, and *Part*. Basic relations can be translated into relational algebra as demonstrated by the following:

- $\{x \mid \text{Lineitem}(x, \textbf{Order1})\}$ denotes the set of parts (identified by partkey) ordered in **Order1**;
- $\{x, y \mid \text{Order}(x, y, _) \wedge y > 50,000\}$ retrieves the orderkeys and prices with prices over $50,000$.

Note that $\{x \mid Order(x)\}$ can be seen as a subquery which denotes the *Order* relation. Likewise $\{y \mid \text{Lineitem}(w, y)\}$ can be seen as a *correlated* subquery, one which returns a particular relation for each value of the correlated variable w. A set term like $\{y(w) \mid \text{Lineitem}(w, y)\}$ is not correlated; it is equivalent to $\{y \mid \text{Lineitem}(_, y)\}$, that is, it corresponds simply to a projection (variable y is bound but discarded). The following queries are quantified:

- – *"Find the parts that were ordered in some order"*
  – $\{z \mid \textbf{some } \{x \mid \text{Orders}(x)\} \ \{x \mid \text{Lineitem}(z, x)\}\}$
- – *"Find the parts ordered by all orders"*
  – $\{z \mid \textbf{all } \{x \mid \text{Orders}(x)\} \ \{x \mid \text{Lineitem}(z, x)\}\}$
- – *"Find the parts no one ordered"*
  – $\{z \mid \textbf{no } \{x \mid \text{Orders}(x)\} \ \{x \mid \text{Lineitem}(z, x)\}\}$
- – *"Find the parts ordered by at least three orders"*
  – $\{z \mid \textbf{at least 3 } \{x \mid \text{Orders}(x)\} \ \{x \mid \text{Lineitem}(z, x)\}\}$
- – *"Find the parts ordered in half of the orders"*
  – $\{z \mid \textbf{half } (\{x \mid Orders(x)\}, \ \{y \mid \text{Lineitem}(z, y)\})\}$

The queries above give the values a such that the set $\{x \mid Orders(x)\}$ (the set of all orders) and the set $\{x \mid \text{Lineitem}(x, \mathbf{a})\}$ (the set of all parts in order a) are in the relationships established by the quantifier: non-empty intersection in the first case[3], subset in the second (so that a was in all the orders), disjointness in the third (so that a is not currently in any order), and so on. How exactly such computation is achieved is discussed in the next section.

In the context of generalized quantifiers, queries are nothing but set terms and thus blend naturally with the approach. Furthermore, since set terms can be nested, complex queries can be expressed by combining different generalized quantifiers.

We point out that we have not really defined a language, but a *family* of languages. The expressive power of QL depends on the set $\mathcal{Q}$ used. When we need to emphasize which set is used, we will write a particular language as $QL(\mathcal{Q})$. Also, the language has no disjunction or negation of basic formulas. Such operators are potentially unsafe; set union and the quantifier **no** provide safe counterparts. In particular, it is not difficult to see that the language $QL(\{\mathbf{some}, \mathbf{all}, \mathbf{no}\})$ is equivalent to Relational Algebra; that is, to the safe and domain independent fragment of FOL ([2]). A consequence of this is that basic set terms (those without quantifiers) correspond to SPJ formulas when conjunction is used; and to SP formulas when no conjunction is used. All quantified queries are expressed in our language through the use of a GQs. That is, we have the following facts:

- If q is a basic QL query, then there is an RA query R that is equivalent to q and R is an SP query.
- If q is a QL(**some**) query, q is either a basic query or a quantified formula with quantifier **some**. Hence, there is an RA query R that is equivalent to q and R is an

[3] This query is basically equivalent to a join, that is, it is equivalent to

$$\pi_{ename}(Orders \bowtie_{orderkey} Lineitem)$$

In QL, queries with a join can be written with the quantifier **some** or with standard logical notation, using the same variable:

$$\{z, (x) \mid Orders(x) \wedge Lineitem(x, z)\}$$

SPJ query.

- Queries that use quantifiers other than **some**, like **no**, **all**, **half**, **10%**, etc. require subqueries or group by and having when expressed in SQL.

C Application of $QL(\mathcal{Q})$ to SQL

The SQL standard could be extended to accommodate $QL(\mathcal{Q})$ by allowing operators to represent these quantifiers. Such operators would be in the $WHERE$ clause. These operators would take two parameters. Each parameter would be a set and the operator would return a boolean value. These boolean values returned could then be used like any boolean SQL operator. The operator syntax could look like:

```
OPERATOR
(
     SQL QUERY
) IN (
     SQL QUERY
)
```

In this structure, $OPERATOR$ could be *Some*, *All*, *AtLeast*, etc. Each $SQLQUERY$ is simply a nested SQL query. These queries would have all of the properties of existing nested SQL queries. With the proposed extensions, these nested queries could also include their own instances of these new operators.

In order to illustrate these operators in use, we will revisit some of the previous needs for data and the SQL query to fill that request.

- Query 1: *Which orders ordered part P1?*. The current SQL approach worked when this query was distributed to each hosts, therefore; the existing query is already easily optimized in a distributed database.

- Query 2: *Which orders share a part with O1?*.

```
SELECT o.orderkey
FROM orders o
WHERE
SOME (
     SELECT partkey
     FROM lineitem l
     WHERE l.orderkey = o.orderkey
) IN (
     SELECT partkey
     FROM lineitem l
     WHERE l.orderkey = O1
)
```

This query enumerates a list of orders, then looks at that order's parts. If one of those parts is in $O1$ then the WHERE clause is satisfied and that order is returned. Notice, that each of the nested queries are SP queries. Query 1 shows that SP queries can be easily distributed. Thus, Query 2 should be optimized far more easily than the previous SQL approach discussed previously.

- Query 3: *Which orders share at least two parts with order O1?*.

```
SELECT o.orderkey
FROM orders o
WHERE
AT LEAST 2 (
     SELECT partkey
     FROM lineitem l
     WHERE l.orderkey = o.orderkey
) IN (
     SELECT partkey
```

```
    FROM lineitem l
    WHERE l.orderkey = O1
)
```

This query looks similar to Query 2. The only difference is the operator $SOME$ was replaced with $ATLEAST2$. This query also has the benefits of the nested SP queries being more easily distributed as discussed in Query 2.

- Query 4: *Which orders do not share any parts with O1?*

```
SELECT o.orderkey
FROM orders o
WHERE
NO (
    SELECT partkey
    FROM lineitem l
    WHERE l.orderkey = o.orderkey
) IN (
    SELECT partkey
    FROM lineitem l
    WHERE l.orderkey = O1
)
```

This query uses the NO operator. This query could also be expressed with $ATMOST0$.

- Query 5: *List the orders that share all the parts with O1.*

```
SELECT o.orderkey
FROM orders o
WHERE
```

```
ALL (
     SELECT partkey
     FROM lineitem l
     WHERE l.orderkey = o.orderkey
) IN (
     SELECT partkey
     FROM lineitem l
     WHERE l.orderkey = O1
)
```

Much like Query 2 and Query 3, the nested queries are simple SP queries.

These queries illustrate that requests for information can be encoded using QL(Q) in queries that are more readily optimized in a distributed setting. Specific algorithms for how to implement these algorithms are discussed in the following sections.

D Implementation

The following section will explore some algorithms to implement the described GQs. Details about specific operators and their analysis and optimization will be described in subsequent sections. These algorithms assume that the sets are horizontally split between multiple hosts and that duplicates do not exist. Let h_i denote the host in the set of all hosts H and A_i the part of A stored on H_i with $A = \bigcup_i A_i$.

In order to make a reference point for algorithm performance a naive algorithm is proposed. This algorithm is described in Algorithm 1.

The naive algorithm can be adapted to accommodate all of the specific operators proposed. The computer running this algorithm shall be called the processing host, h_0. Each of the code blocks contained inside of a 'for each host' loop can be executed in parallel because, there should be no interaction between hosts in this scope. Each host is free to independently answer the part of the larger query it is given. The naive algorithm

Algorithm 1 Naive Set Operation Algorithm

$B \leftarrow \emptyset$
for all $h_i \in Hosts$ **do**
  $B_i \leftarrow$ select all elements from h that belong to set B
  send B_i to h_0
end for
(on h_0) $B \leftarrow B_0 \cup B_1 \cup ..B_n$
for all $h_i \in Hosts$ **do**
  $A_i \leftarrow$ select all elements from h that belong to set A
  send A_i to h_0
end for
(on h_0) $A \leftarrow A_0 \cup A_1 \cup ..A_n$
compute $Q(A, B)$

then sends all of the elements of set A and set B back to the processing host. Therefore, with respect to network traffic the algorithm would be considered $O(|A| + |B|)$. The query executed on each host must be at most polynomial computational complexity because it is expressed in standard SQL. If the query is an SP query it would only be linear. Building a hash table to represent set A and looking up each element in A in the hash table would be of complexity $O(|A| + |B|)$ assuming enough memory. In the case of large datasets, the network transfer could potentially be quite expensive. For that reason, an alternative algorithm is proposed.

This algorithm distributes the set B to each of the hosts. Each host then returns the minimal amount of information required to the processing host in a similar fashion to a semijoin. Thus, this method will be referred to as the Set Operation Semijoin. The Set Operation Semijoin is describe in Algorithm 2. The definition of 'minimal amount of information required' will vary depending on the set operator in use.

The network traffic required to construct and distribute a complete instance of set B would be $(N + 1) * |B|$, where N is the number of hosts in the network, since the sum of all contributions to the central instance of set B would be $|B|$. Distributing set B to N hosts would then cost $N * |B|$. Thus, the total network cost to construct and distribute set B would be $(N + 1) * |B|$. Each host then processes set B with a SQL statement issued by the processing host. The amount of information retrieved would then be determined based on

Algorithm 2 Generic Set Operation Semijoin Algorithm

```
for all h_i ∈ Hosts do
    B_i ← select all elements from h that belong to set B
    send B_i to h_0
end for
(on h_0) B ← B_0 ∪ B_1 ∪ ..B_n
send B to each of the hosts
for all h_i ∈ Hosts do
    A_i ← select all elements from h that belong to A_i order by A_i
    send Q(A_i, B) to h_0
end for
(on h_0) compute Q(A, B) from results from each host
```

the specific operator.

Thus far, the assumption has always been made that set B would always be the distributed set. It is possible that set A could be the set that gets collected and distributed. The logic executed on each host would only need to be trivially altered to maintain the 'direction' of the set operator. It is also possible that one may want to evaluate multiple set A's.

E Some

The first operator analyzed in SQL was **Some**. **Some** is straightforward in SQL because it can be implemented by joining the two sets. The **Some** operator can be expressed as $|A_{h_i} \cap B| \neq 0$. This gives **Some** the property of being monotonically increasing; therefore, once the **Some** operator has been satisfied by some set no other data can negate that satisfaction.

In the case of the Naive Set Operation Algorithm, this property means that once an element of A was found in B, the rest of the elements in A can be consumed without inspection. In the case of the Set Operator Semijoin, this means that the only information that needs to be returned to the processing host is a token to identify that set A satisfies the condition. In the case of multiple set A's in a Set Operator Semijoin being evaluated only a series of identifiers to represent those sets needs to be returned to the processing

host.

The Naive Set Operation Algorithm would compute **Some** by retrieving all of the elements of set B and storing them in a hashmap. Because hashmaps store key-value pairs, a token that represents the tuple will be chosen for the key and an arbitrary indicator will be chosen for the value. After each element of set B is stored in the hashmap, all of the elements in set A are fetched. Each element in set A is checked in the hashmap. If the hashmap contains the arbitrary value for the key, an identifier based on the element, then set B contains that element. That would mean that the element from set A is contained in set B and the quantifier would return true. Otherwise, it would return false.

F At Least

The **At least** operator functions similarly to **Some**. $AtLeast(n)$ can be written formally as $|A \cap B| >= n$. Thus, the number of common elements between A and B must be greater than some specified number, n. Based on this description, the algorithm can be shown to be monotonically increasing in the same way as the **Some** operator. One important distinction between **Some** and **At least** is that **At least** needs to not only return the token to identify the set as having common elements but also the number of these common elements. This is trivial to implement in SQL using a group by on the A set identifier and a count on the number of rows, common elements. The count must be returned because the total number n may not be achieved on any single host so the results must be aggregated at the processing host.

One may observe that if a single host contains n elements in common between A and B then that host has enough information to identify the set A as satisfying the operator and based on the monotonically increasing property the messages returned from all other hosts can be safely ignored. Theoretically it should be possible for one host to alert the others that set A has been identified as being successful and there is no need to return it to the processing host. Unfortunately, alerting all of the other hosts would require N-1 messages and it would best case only save N-1 results being sent to the processing

host. All things being equal, it would therefore not be advantageous to alert other nodes that a candidate set qualifies.

G At Most

At Most can be formally defined as $|A \cap B| <= n$. Therefore, it does not share the monotonically increasing property with **Some** and **At Least** because additions to $|A \cap B|$ could make it larger than n. But it is downward monotone, this can be used to eliminate sets from consideration symmetrically to **At Least**. $AtMost(n)$ can be implemented using the Set Operator Semijoin by returning a candidate set identifier and the number of elements shared between set A and set B.

H Ratio Operators

The Ratio Operators are a set of operators that include **Half**,**Percent**, and **All**. These operators require that $|A \cap B| \, / \, |A| = n$, where n is defined by the specific algorithm. **Half** could be expressed as **50 Percent**. In order to evaluate the **Percent** operator, $|A \cap B|$ and $|A|$ must both be calculated. **All** could be implemented by looking for all elements that satisfy $\exists\, x\ (A(x) \wedge \neg B(x))$ and then removing all of those elements from a list of all possible A's.

I All But

The **All But** Operator is slightly different from the Ratio Operators. It is defined as $|A \cap B| + n = |A \cup B|$ where n is a specified constant. The similarity between All But and the Ratio Operators is enough to allow all of the steps to be identical between the two except those performed on the processing host.

CHAPTER IV

Experiments

A Technical Setup

A leading commercial relational database system was installed and configured on commodity Linux desktop computers. Trials were conducted with a cluster of 3 and 5 computers. Each database was loaded with the TPCH database. Trials were conducted using the TPCH database with a factor of 1, 10, and 100. The TPCH utility 'dbgen' was used to generate data to populate these tables. Dbgen created several text files that represented the data in the various tables in the TPCH database.

Different database users were created to represent each of the trial configurations. For example 5 hosts had a user called 'tpch5x10' which represented the TPCH dataset with a factor of 10 divided between 5 hosts. The TPCH lineitem data of the respective scale was divided randomly between the number of hosts for the trial and loaded to the appropriate user. This was to simulate a horizontally distributed database.

The database system's built in utility for querying remote databases was used to communicate between hosts. For each division factor and data scale, a view was created on one host which contained all of the data unioned together. For instance the 3x10 view was created using the following query:

```
CREATE VIEW ALL_LINEITEMS AS
SELECT * FROM LINEITEM@DBLABB
UNION ALL
SELECT * FROM LINEITEM@DBLABC
UNION ALL
```

```
SELECT * FROM LINEITEM@DBLABD
```

This view combines data from 3 hosts in a single view. Querying this view will then utilize the distributed features of the database. Queries were written such that they would handle small, medium, and large amounts of data. The dataset sizes are displayed in Table 2. The TPCH data generator did not create orders with large numbers of products so ranges of orders were used for the medium and large datasets. As such, there were some orders in the range which shared the same product. These duplicates were removed in the respective algorithms.

TABLE 2

Queries were formulated with sets of the following sizes.

Name	Number Unique of Elements	Total Number of Elements
Small	4	4
Medium	9723	9965
Large	198679	1000048

B Some

1 SQL

Some was implemented using a SQL query as follows:

```
SELECT DISTINCT ITEMS.L_ORDERKEY
FROM ALL_LINEITEMS ITEMS
WHERE ITEMS.L_PARTKEY IN
(
        SELECT DISTINCT L_PARTKEY
        FROM ALL_LINEITEMS
```

```
        WHERE L_ORDERKEY  = X
)
```

2 Naive Algorithm

The generic naive algorithm as described previously was adapted to Algorithm 3. The generic algorithm builds a hashmap to represent set B. It then compares each element in A against B's hashmap. If the element is found it consumes the rest of the elements in set A without inspection and then returns set A's identifier.

Algorithm 3 Naive Some Algorithm

$B \leftarrow \emptyset$
for all $h_i \in Hosts$ **do**
  $B_i \leftarrow$ select all elements from h that belong to set B
  send B_i to h_0
end for
(on h_0) $B \leftarrow B_0 \cup B_1 \cup ..B_n$
for all $h_i \in Hosts$ **do**
  $A_i \leftarrow$ select A_j, all elements from h that belong to A_j order by A_j
  send A_i to h_0
end for
for all $A_j \in A_{0..n}$ **do**
  for all $e \in A_j$ **do**
    if $e \in B$ **then**
      return A_j
    end if
  end for
end for

3 Set Operator Semijoin

The Set Operator Semijoin was adapted to Algorithm 4. As can be seen, the other hosts in the network only need to send $A_i \cap B$ back to the processing host. This feature should improve performance where large data sets are involved.

Algorithm 4 Some Set Operator Semijoin Algorithm

for all $h_i \in Hosts$ **do**
  $B_i \leftarrow$ select all elements from h that belong to set B
  send B_i to h_0
end for
(on h_0) $B \leftarrow B_0 \cup B_1 \cup ..B_n$
send B to each of the hosts
for all $h_i \in Hosts$ **do**
  $A_i \leftarrow$ select all elements from h that belong to A_i and B order by A_i
  send $|A_i \cap B|$ to h_0
end for
for all $A_i \in$ results from hosts **do**
  for all $e \in A_j$ **do**
    if $e \in B$ **then**
      return A_j
    end if
  end for
end for

4 Comparison

Figures 2, 3, and 4 display the absolute run-times for the three implementations in a linear scale for the 1, 10, and 100X datasets respectively. The performance for the small query in the 1X set is pretty equal. However, as the query size and the data size increases, the semijoin outperforms the SQL and naive algorithms.

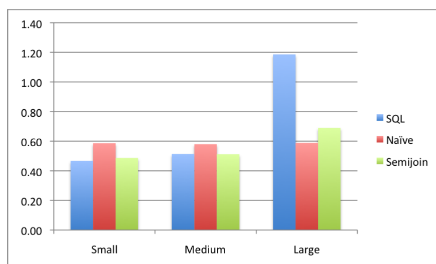

Figure 2. The absolute run-times of the implementations of Some for the 1X TPCH dataset split between 3 hosts.

Figures 5, 6, and 7 display the absolute run-times for the three implementations in a linear scale. As was noted for the 3 host trials, the relative performance of the semijoin approach gets better as the query size and dataset size increased. One particularly glaring

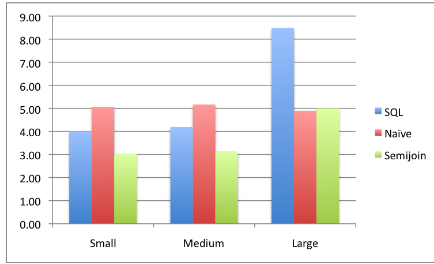

Figure 3. The absolute run-times of the implementations of Some for the 10X TPCH dataset split between 3 hosts.

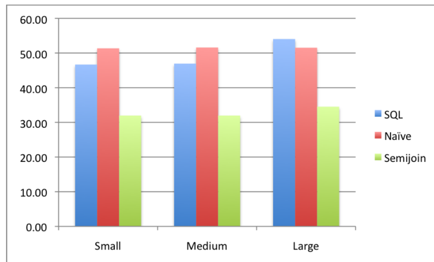

Figure 4. The absolute run-times of the implementations of Some for the 100X TPCH dataset split between 3 hosts.

example is the 5 host 100X large trial. The SQL approach on this trial took several times more time than the semijoin approach.

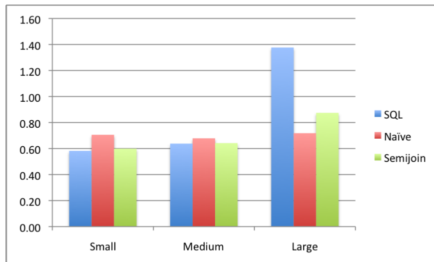

Figure 5. The absolute run-times of the implementations of Some for the 1X TPCH dataset split between 5 hosts.

C At Least

$Some(x)$ can be considered a more specific implementation of $AtLeast(x)$. $Some(x)$ is equivalent to $AtLeast1(x)$.

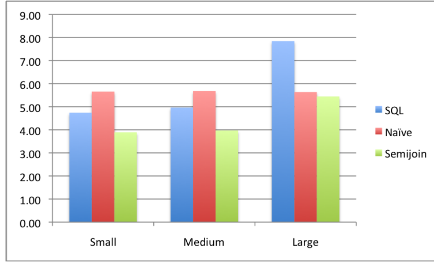

Figure 6. The absolute run-times of the implementations of Some for the 10X TPCH dataset split between 5 hosts.

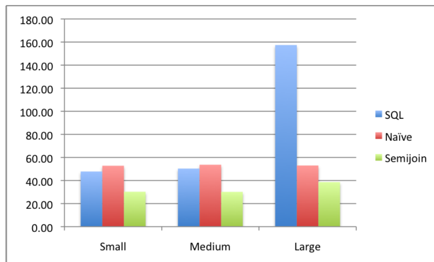

Figure 7. The absolute run-times of the implementations of Some for the 100X TPCH dataset split between 5 hosts.

1 SQL

Although there are many different possible ways to implement $AtLeast(x)$ in SQL, the following query was used for testing:

```
select l1.l_orderkey
from all_lineitems l1,
 (
 select distinct l_partkey from all_lineitems where l_orderkey  = X
 ) l2
 where l1.l_partkey = l2.l_partkey
 group by l1.l_orderkey
having count(l1.l_orderkey)>=2
```

This query builds a set of all the parts in the specified range of orders. It then joins these parts with all_lineitems on partkey. The results are then grouped by the orderkey.

Only the orders with 2 or more parts shared with the specified order range will be returned based on the having clause.

2 Naive Algorithm

The generic naive algorithm as described previously was adapted in Algorithm 5. The generic algorithm builds a hash map to represent set B. It then compares each element in A against B's hash map. If the element is found it increments a counter. Once the threshold for the counter is reached, it will consume the rest of the elements in set A without inspection and return set A's identifier.

Algorithm 5 Naive At Least Algorithm

$B \leftarrow \emptyset$
for all $h_i \in Hosts$ **do**
  $B_i \leftarrow$ select all elements from h that belong to set B
  send B_i to h_0
end for
(on h_0) $B \leftarrow B_0 \cup B_1 \cup ..B_n$
for all $h_i \in Hosts$ **do**
  $A_i \leftarrow$ select A_j, all elements from h that belong to A_j order by A_j
  send A_i to h_0
end for
for all $A_j \in A_{0..n}$ **do**
  $matches \leftarrow 0$
  for all $e \in A_j$ **do**
    if $e \in B$ **then**
      $matches \leftarrow matches + 1$
    end if
    if $matches \geq N$ **then**
      return A_j
    end if
  end for
end for

3 Set Operator Semijoin

The Set Operator Semijoin was adapted in Algorithm 6. The algorithm starts by

Algorithm 6 At Least Set Operator Semijoin Algorithm

for all $h_i \in Hosts$ **do**
  $B_i \leftarrow$ select all elements from h that belong to set B
  send B_i to h_0
end for
(on h_0) $B \leftarrow B_0 \cup B_1 \cup ..B_n$
send B to each of the hosts
for all $h_i \in Hosts$ **do**
  $A_i \leftarrow$ select all elements from h that belong to A_i order by A_i
  send $|A_i \cap B|$ to h_0
end for
for all $A \in$ results from hosts **do**
  $sum \leftarrow 0$
  for all $A_i \in A$ **do**
    $sum \leftarrow sum + |A_i \cap B|$
  end for
  if $sum \geq N$ **then**
    return A
  end if
end for

building set B and storing it in a table T. Each host then joins table T with its own lineitem table. The query is grouped by l_orderkey and counts the number of rows per order. The processing host will sum the number of matches found per host and then decide if it satisfies the $AtLeast(x)$ operator.

4 Comparison

Figures 8, 9, and 10 display the absolute run-times for the three implementations in a linear scale for the 1, 10, and 100X datasets respectively. The three implementations appear evenly matched in the TPCH 1X dataset. However, in the larger databases the Naive and Semijoin implementations outperform the SQL implementation.

Figures 11, 12, and 13 display the absolute run-times for the three implementations in a linear scale. The relative performance of the algorithms on the 5 host cluster appears to closely resemble the performance on the 3 host cluster. In the TPCH 1X dataset, the algorithms appear neck and neck. In the 10X and 100X databases, the Semijoin

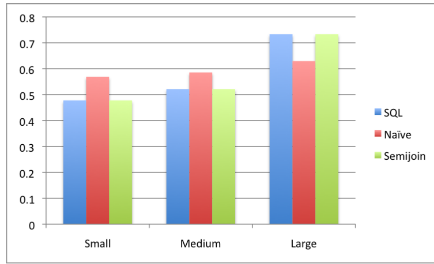

Figure 8. The absolute run-times of the implementations of at least for the 1X TPCH dataset split between 3 hosts.

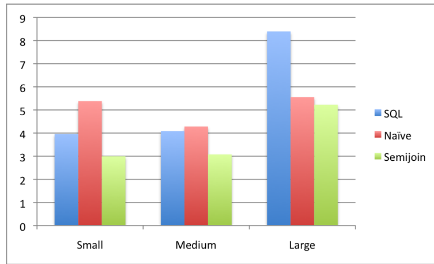

Figure 9. The absolute run-times of the implementations of at least for the 10X TPCH dataset split between 3 hosts.

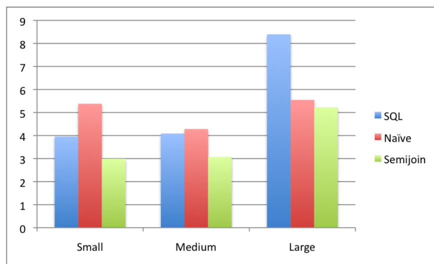

Figure 10. The absolute run-times of the implementations of at least for the 100X TPCH dataset split between 3 hosts.

implementation pulls ahead of SQL implementation. The Naive implementation maintains a steady time regardless of the query size. In the small and medium queries, the Naive implementation was the slowest. However, in the large queries, the naive implementation was the fastest once and second fastest twice.

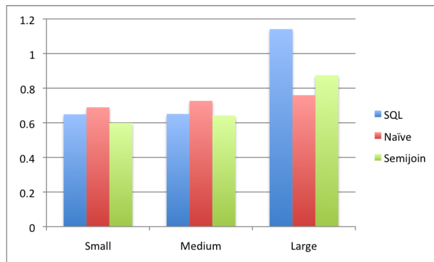

Figure 11. The absolute run-times of the implementations of at least for the 1X TPCH dataset split between 5 hosts.

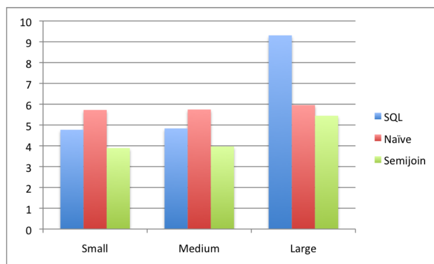

Figure 12. The absolute run-times of the implementations of at least for the 10X TPCH dataset split between 5 hosts.

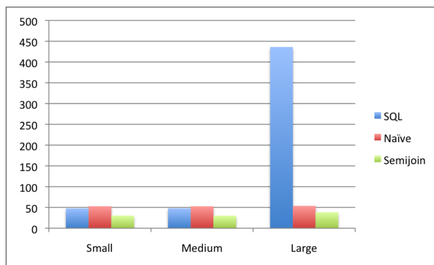

Figure 13. The absolute run-times of the implementations of at least for the 100X TPCH dataset split between 5 hosts.

D At Most

Fundamentally $AtMost(x)$ is similar to $AtLeast(x)$. The key difference between the two is the direction of the greater than/less than operator. For that reason, the implementation of this algorithm was almost identical to $AtLeast(x)$ other than the greater than/less than operator being switched.

1 SQL

The following SQL query was used to test $AtMost(x)$:

```
select l1.l_orderkey
 from all_lineitems l1,
 (
 select distinct l_partkey from all_lineitems where l_orderkey = X
 ) l2
 where l1.l_partkey = l2.l_partkey
 group by l1.l_orderkey
 having count(l1.l_orderkey)<=2
 order by l1.l_orderkey
```

The query works very similarly to the SQL query for $AtLeast(x)$. The next query in the some builds a set of parts in the specified range of orders. That set is joined with with the total set of lineitems. The results are grouped by order and counted. Orders with a count of no more than 2 were returned.

2 Naive Algorithm

$AtMost(x)$ was implement in the same fashion as $AtLeast(x)$. The only difference was that the line "if $matches \geq n$" was replaced by "if $matches \leq n$".

3 Set Operator Semijoin

$AtMost(x)$ was implement in the same fashion as $AtLeast(x)$. The only difference was that the line "if $sum \geq n$" was replaced by "if $sum \leq n$"

4 Comparison

Figures 14, 15, and 16 display the absolute run-times for the three implementations in a linear scale for the 1, 10, and 100X datasets respectively. The performance of the implementations of $AtMost(x)$ appear to be consistent with the performance of $AtLeast(x)$. The implementations of the two are almost identical and so this is not surprising.

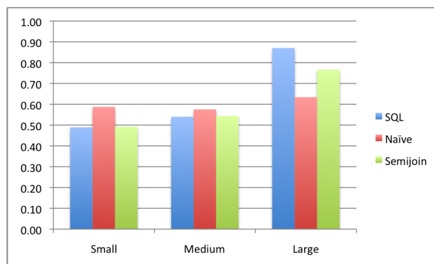

Figure 14. The absolute run-times of the implementations of at most for the 1X TPCH dataset split between 3 hosts.

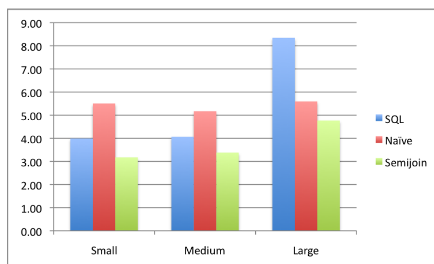

Figure 15. The absolute run-times of the implementations of at most for the 10X TPCH dataset split between 3 hosts.

Figures 17, 18, and 19 display the absolute run-times for the three implementations in a linear scale. The performance of the algorithms in a 5 host cluster is consistent with that of the 3 host cluster. The only notable difference between the 3 and 5 host clusters is

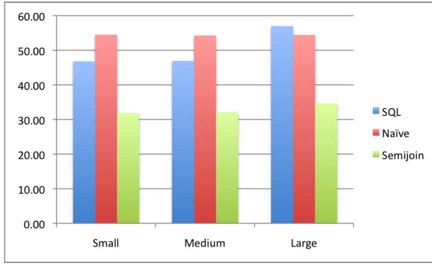

Figure 16. The absolute run-times of the implementations of at most for the 100X TPCH dataset split between 3 hosts.

that the gap between the performance of SQL and the other two algorithms is larger in the larger queries and database sizes.

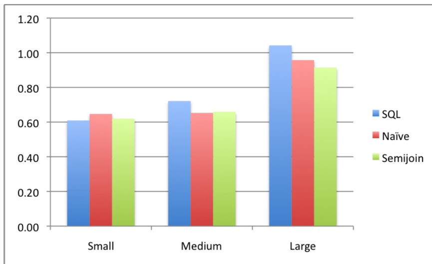

Figure 17. The absolute run-times of the implementations of at most for the 1X TPCH dataset split between 5 hosts.

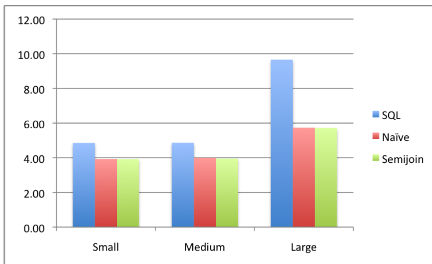

Figure 18. The absolute run-times of the implementations of at most for the 10X TPCH dataset split between 5 hosts.

E Ratio Operators

As previously discussed, the half operator is a special case of the percent operator. For that reason, 50 Percent/Half were tested to measure the performance of ratio

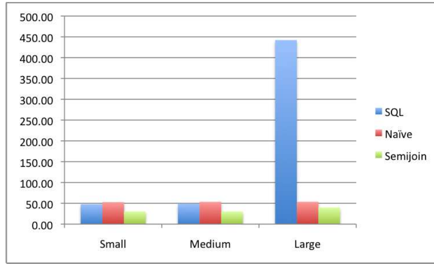

Figure 19. The absolute run-times of the implementations of at most for the 100X TPCH dataset split between 5 hosts.

operators. All is a special case of the ratio operator where that ratio is 1:1. Due to that unique constraint, All was tested in its own section later in this chapter. The following sections describe the results for 50 Percent/Half:

1 SQL

```
SELECT l1.l_orderkey
FROM all_lineitems l1
LEFT JOIN
(
SELECT DISTINCT l_partkey FROM all_lineitems WHERE l_orderkey = X
 ) l2
ON l1.l_partkey = l2.l_partkey
GROUP BY l1.l_orderkey
HAVING COUNT(l2.l_partkey)/COUNT(l1.l_partkey) >= 1/2
ORDER BY l1.l_orderkey
```

The SQL query above works almost identically to the $AtLeast(x)$ and $AtMost(x)$ SQL queries. There are two differences between the queries. The Ratio query left joins the set of parts from the set of relevant orders on the set of all lineitems. $AtLeast(x)$ and $AtMost(x)$ naturally joined the selected part set with all lineitems. The second difference is the having clause. The having clause states that the number of parts from the range of orders divided by the number of parts from all lineitems per order must be greater than

1/2. 1/2 can be substituted with any percentage or fraction.

2 Naive Algorithm

The naive algorithm was adapted for $Percent(x)$ and $Half(x)$ in Algorithm 7.

Algorithm 7 Naive Ratio Algorithm

```
results ← ∅
B ← ∅
for all h_i ∈ Hosts do
  B_i ← select A_j, all elements from h that belong to A_j order by A_j
  send B_i to h_0
end for
for all h_i ∈ Hosts do
  A_i ← select l_orderkey, l_partkey from lineitem order by l_orderkey
  send A_i to h_0
end for
for all orderkey ∈ A_{0..n} do
  matches ← 0
  total ← 0
  for all partkey ∈ A_{iorderkey} do
    total ← total + 1
    if partkey ∈ B then
      matches ← matches + 1
    end if
  end for
  if matches/total ≈ N then
    results ← results ∪ orderkey
    break
  end if
end for
return results
```

This algorithm calculates the ratio of the number of items of set A in B compared to the number of items in set A. If the ratio is as specified by the algorithm, greater than or equal to N, then the set is considered a match.

3 Set Operator Semijoin

Algorithm 8 Ratio Set Operator Semijoin Algorithm

for all $h_i \in Hosts$ **do**
  $B_i \leftarrow$ select all elements from h that belong to set B
  send B_i to h_0
end for
$B \leftarrow B_0 \cup B_1 \cup ..B_n$
send B to each of the hosts
for all $h_i \in Hosts$ **do**
  calculate and send $|A_i \cap B|$ and $|A_i|$ to h_0
end for
for all $A \in$ results from hosts **do**
  $matches \leftarrow 0$
  $total \leftarrow 0$
  for all $A_i \in A$ **do**
    $matches \leftarrow matches + |A_i \cap B|$
    $total \leftarrow total + |A_i|$
  end for
  if $matches/total \geq N$ **then**
    return A
  end if
end for

4 Comparison

Figures 20, 21, and 22 display the absolute run-times for the three implementations in a linear scale for the 1, 10, and 100X datasets respectively. The Naive implementation performed admirably well in this instance. The Naive implementation's times for the Small, Medium, and Large trials were almost level for the 1X, 10X, and 100X trials. The Naive implementation was either the fastest or a close second in all of the configurations tested.

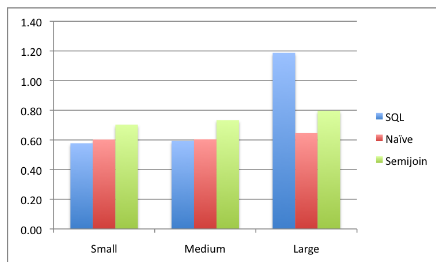

Figure 20. The absolute run-times of the implementations of ratio for the 1X TPCH dataset split between 3 hosts.

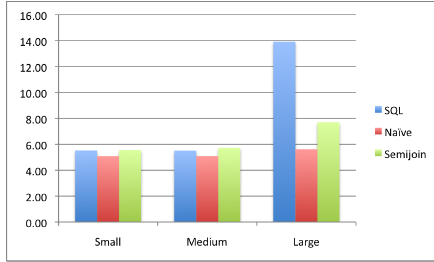

Figure 21. The absolute run-times of the implementations of ratio for the 10X TPCH dataset split between 3 hosts.

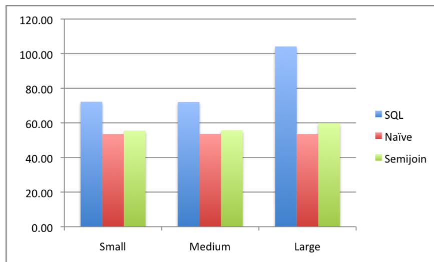

Figure 22. The absolute run-times of the implementations of ratio for the 100X TPCH dataset split between 3 hosts.

Figures 23, 24, and 25 display the absolute run-times for the three implementations in a linear scale. The results for the 5 host cluster were consistent with the 3 host cluster. The SQL implementation particularly under performed on the TPCH 100X Large query. It took about 4.5 times as long as the Naive implementation.

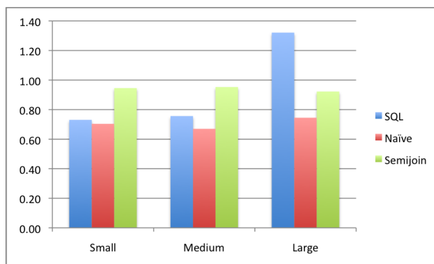

Figure 23. The absolute run-times of the implementations of ratio for the 1X TPCH dataset split between 5 hosts.

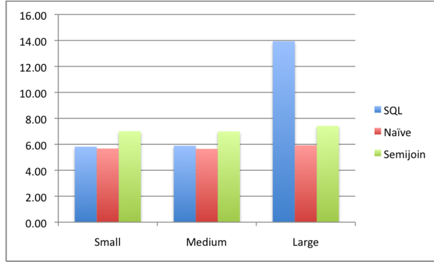

Figure 24. The absolute run-times of the implementations of ratio for the 10X TPCH dataset split between 5 hosts.

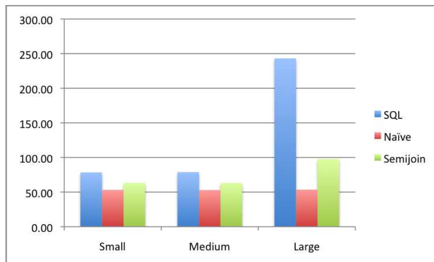

Figure 25. The absolute run-times of the implementations of ratio for the 100X TPCH dataset split between 5 hosts.

F All

As described previously, $All(x)$ is true if $A \subseteq B$. The following sections describe the results from the different implementations tested.

1 SQL

All was implemented in SQL with the following query:

```
SELECT DISTINCT L_ORDERKEY
FROM ALL_LINEITEMS
WHERE L_ORDERKEY NOT IN(
         SELECT L.L_ORDERKEY
         FROM ALL_LINEITEMS L
         WHERE L_PARTKEY NOT IN
         (
```

```
                    SELECT L_PARTKEY
                    FROM ALL_LINEITEMS
                    WHERE L_ORDERKEY  = X
          )
)  ORDER BY L_ORDERKEY
```

2 Naive Algorithm

$All(x)$ was implemented using the ratio operator described in the previous section with $N = 1.0$.

3 Set Operator Semijoin

$All(x)$ was implemented using the ratio operator described in the previous section with $N = 1.0$.

4 Comparison

Figures 26, 27, and 28 display the absolute run-times for the three implementations in a linear scale for the 1, 10, and 100X datasets respectively. The results for $All(x)$ on 3 hosts were comparable to the results for the Ratio operator tested in the previous section.

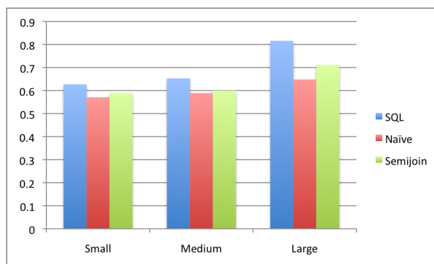

Figure 26. The absolute run-times of the implementations of all for the 1X TPCH dataset split between 3 hosts.

Figures 29, 30, and 31 display the absolute run-times for the three implementations in a linear scale. As was noted for the 3 host trials, the relative performance of the algorithms is comparable to the Ratio operator in the previous section.

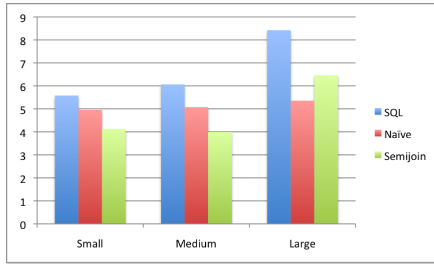

Figure 27. The absolute run-times of the implementations of all for the 10X TPCH dataset split between 3 hosts.

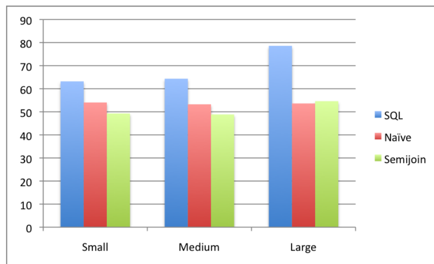

Figure 28. The absolute run-times of the implementations of all for the 100X TPCH dataset split between 3 hosts.

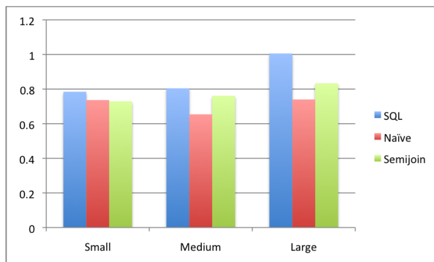

Figure 29. The absolute run-times of the implementations of all for the 1X TPCH dataset split between 5 hosts.

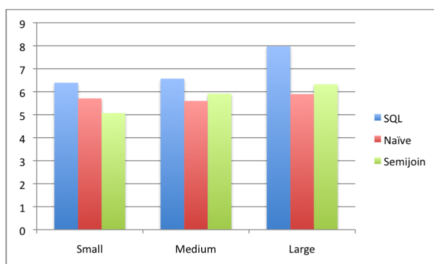

Figure 30. The absolute run-times of the implementations of all for the 10X TPCH dataset split between 5 hosts.

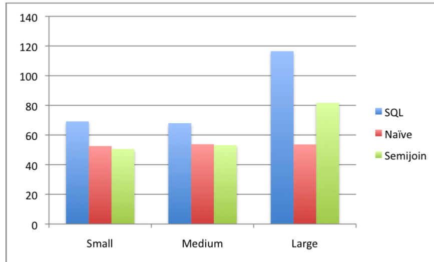

Figure 31. The absolute run-times of the implementations of all for the 100X TPCH dataset split between 5 hosts.

G AllBut

All But can be expressed as $|A| - |A \cap B| = N$. Based on this $|A|$ and $|A \cap B|$ must both be calculated.

1 SQL

```
SELECT l1.l_orderkey
FROM all_lineitems l1
LEFT JOIN
(
SELECT DISTINCT l_partkey
FROM all_lineitems
WHERE l_orderkey = X
 ) l2
ON l1.l_partkey = l2.l_partkey
GROUP BY l1.l_orderkey
HAVING COUNT(l2.l_partkey)  > 0
AND COUNT(l2.l_partkey) + 2 = COUNT(l1.l_partkey)
ORDER BY l1.l_orderkey
```

2 Naive

Algorithm 9 Naive All But Algorithm

```
results ← ∅
b ← ∅
for all h_i ∈ Hosts do
  B_i ← select A_j, all elements from h that belong to A_j order by A_j
  send B_i to h_0
end for
for all h_i ∈ Hosts do
  A_i ← select l_orderkey, l_partkey from lineitem order by l_orderkey
  send A_i to h_0
end for
for all orderkey ∈ A_{0..n} do
  matches ← 0
  total ← 0
  for all partkey ∈ A_{i orderkey} do
    total ← total + 1
    if partkey ∈ B then
      matches ← matches + 1
    end if
  end for
  if total − matches = N then
    results ← results ∪ orderkey
    break
  end if
end for
return results
```

3 Semijoin

Algorithm 10 All But Set Operator Semijoin Algorithm

```
for all h_i ∈ Hosts do
  B_i ← select all elements from h that belong to set B
  send B_i to h_0
end for
B ← B_0 ∪ B_1 ∪ ..B_n
send B to each of the hosts
for all h_i ∈ Hosts do
  calculate and send |A_i ∩ B| and |A_i| to h_0
end for
for all A ∈ results from hosts do
  matches ← 0
  total ← 0
  for all A_i ∈ A do
    matches ← matches + |A_i ∩ B|
    total ← total + |A_i|
  end for
  if total − matches = N then
    return A
  end if
end for
```

4 Comparison

Figures 32, 33, and 34 display the absolute run-times for the three implementations in a linear scale for the 1, 10, and 100X datasets respectively. The Naive algorithm was the fastest algorithm in every configuration tested. The Naive algorithm was level across the Small, Medium, Large trials for the 1, 10, and 100X databases. The SQL implementation distinctly ramped up as the query size went from Small to Large.

Figures 35, 36, and 37 display the absolute run-times for the three implementations in a linear scale. The results were consistent with the 3 host trials.

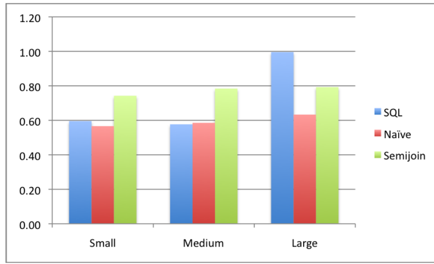

Figure 32. The absolute run-times of the implementations of all but for the 1X TPCH dataset split between 3 hosts.

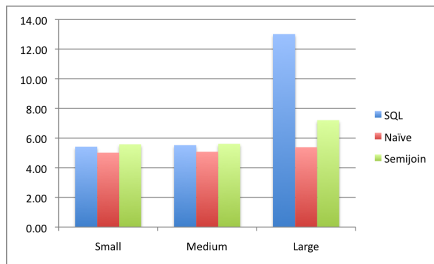

Figure 33. The absolute run-times of the implementations of all but for the 10X TPCH dataset split between 3 hosts.

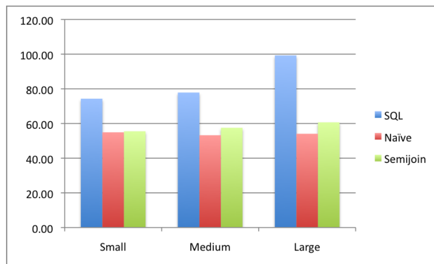

Figure 34. The absolute run-times of the implementations of all but for the 100X TPCH dataset split between 3 hosts.

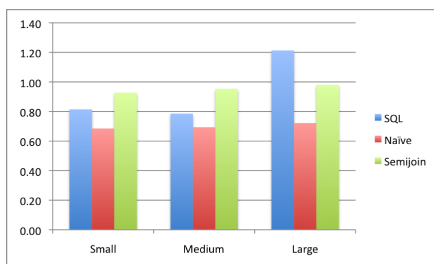

Figure 35. The absolute run-times of the implementations of all but for the 1X TPCH dataset split between 5 hosts.

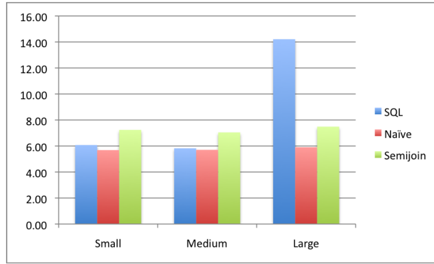

Figure 36. The absolute run-times of the implementations of all but for the 10X TPCH dataset split between 5 hosts.

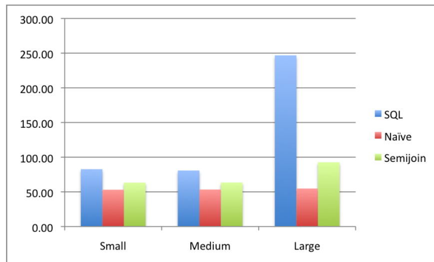

Figure 37. The absolute run-times of the implementations of all but for the 100X TPCH dataset split between 5 hosts.

H Summary and Evaluation of Results

In general, the most significant variable in an algorithms run time is the scale of the TPCH data. Each of the trials tested TPCH scales of 1X, 10X, and 100X. These scales were factors of the original dataset where the TPCH 1X set was 1 gigabyte. The 100X set was 100 times larger.

For the 1X dataset, different implementations outperformed others and there was no clear best option. However, as the dataset size increased from 1X to 100X the SQL approach became less favorable. At the 100X size, the SQL implementation was never the fastest algorithm. The Semijoin implementation performed the best overall for Some, AtLeast and AtMost. For the Ratio operator, All, and AllBut, the Naive implementation was the best overall performer.

The Semijoin and Naive algorithms were not without their own experimental shortcomings. They were written in Java and queries were issued through the vendors JDBC interface. This extra layer intuitively introduced some extra latency in the

experimental run times for these two algorithms.

One shortcoming of the naive algorithm was that it assumed that all objects would fit in memory. The hashmap that gets constructed fit into memory for all of the tests in this experiment. Had there been too many items in the hashmap to fit into memory, swap space would have been used. Intuition would dictate that the naive algorithms performance would suffer detrimentally if parts of the hashmap had to be retrieved from swap space.

With these limitations in mind it still appears that there is some merit to utilizing these generalized quantifiers in an extension to the SQL standard.

CHAPTER V

CONCLUSION

To conclude, the current implementation of SQL makes optimizing distributed queries difficult. Improvements can be made by leveraging some of the discussed properties of generalized quantifiers. The proposed query language's implementation appears plausible. The plausibility was measured against existing database technology. In the cases tested, these implementations tended to outperform the existing methods where the dataset was large and/or the size of the set in the query was large. These tests were intended to be sufficiently general so that it would not favor a specific implementation.

Implementing this system could be useful for academic and commercial interests. This query language could be used in data mining. These Generalized Quantifiers could be utilized to find association rules.

Future work should include testing on systems where assumptions are not as closely followed. For the trials conducted, it was assumed that all hosts in the system were operating with equivalent processing power, equal load, and an equal sized partition of data. Loosening any of those constraints should intuitively change the performance of the system. Thus, it should be investigated what happens when these assumptions are degraded.

Another area worth investigating would be in the fault tolerance of the system. The algorithms assume that all hosts will be up all the time. This is not completely accurate for a real word scenario.

CHAPTER VI

APPENDIX

This appendix lists all of the results underlying the experimental chapter. Every list is the average of five runs. Results are grouped by quantifier and implementation.

TABLE 3

Some(x) SQL

Number of Hosts	TPCH Scale	Query Size	Runtime
3	1X	Small	0:00:28
3	1X	Med	0:00:31
3	1X	Large	0:01:11
3	10X	Small	0:04:00
3	10X	Med	0:04:12
3	10X	Large	0:08:29
3	100X	Small	0:46:40
3	100X	Med	0:46:56
3	100X	Large	0:54:02
5	1X	Small	0:00:35
5	1X	Med	0:00:38
5	1X	Large	0:01:23
5	10X	Small	0:04:44
5	10X	Med	0:04:58
5	10X	Large	0:07:51
5	100X	Small	0:47:47
5	100X	Med	0:50:20
5	100X	Large	2:37:24

TABLE 4

Some(x) Naive

Number of Hosts	TPCH Scale	Query Size	Runtime
3	1X	Small	0:00:35
3	1X	Med	0:00:35
3	1X	Large	0:00:35
3	10X	Small	0:05:04
3	10X	Med	0:05:10
3	10X	Large	0:04:54
3	100X	Small	0:51:20
3	100X	Med	0:51:35
3	100X	Large	0:51:31
5	1X	Small	0:00:42
5	1X	Med	0:00:41
5	1X	Large	0:00:43
5	10X	Small	0:05:40
5	10X	Med	0:05:41
5	10X	Large	0:05:38
5	100X	Small	0:52:47
5	100X	Med	0:53:41
5	100X	Large	0:53:01

TABLE 5

Some(x) Semijoin

Number of Hosts	TPCH Scale	Query Size	Runtime
3	1X	Small	0:00:29
3	1X	Med	0:00:31
3	1X	Large	0:00:41
3	10X	Small	0:03:01
3	10X	Med	0:03:08
3	10X	Large	0:05:00
3	100X	Small	0:31:58
3	100X	Med	0:31:58
3	100X	Large	0:34:31
5	1X	Small	0:00:36
5	1X	Med	0:00:39
5	1X	Large	0:00:52
5	10X	Small	0:03:53
5	10X	Med	0:03:58
5	10X	Large	0:05:27
5	100X	Small	0:30:16
5	100X	Med	0:30:09
5	100X	Large	0:38:37

TABLE 6

AtLeast(x) SQL

Number of Hosts	TPCH Scale	Query Size	Runtime
3	1X	Small	0:00:28
3	1X	Med	0:00:30
3	1X	Large	0:00:59
3	10X	Small	0:03:57
3	10X	Med	0:04:05
3	10X	Large	0:08:24
3	100X	Small	0:46:47
3	100X	Med	0:50:06
3	100X	Large	0:57:44
5	1X	Small	0:00:39
5	1X	Med	0:00:39
5	1X	Large	0:01:08
5	10X	Small	0:04:46
5	10X	Med	0:04:50
5	10X	Large	0:09:18
5	100X	Small	0:47:45
5	100X	Med	0:47:55
5	100X	Large	7:16:08

TABLE 7

AtLeast(x) Naive

Number of Hosts	TPCH Scale	Query Size	Runtime
3	1X	Small	0:00:34
3	1X	Med	0:00:35
3	1X	Large	0:00:38
3	10X	Small	0:05:23
3	10X	Med	0:04:17
3	10X	Large	0:05:33
3	100X	Small	0:51:08
3	100X	Med	0:51:08
3	100X	Large	0:51:13
5	1X	Small	0:00:41
5	1X	Med	0:00:44
5	1X	Large	0:00:46
5	10X	Small	0:05:43
5	10X	Med	0:05:45
5	10X	Large	0:05:58
5	100X	Small	0:52:57
5	100X	Med	0:52:52
5	100X	Large	0:53:49

TABLE 8

AtLeast(x) Semijoin

Number of Hosts	TPCH Scale	Query Size	Runtime
3	1X	Small	0:00:29
3	1X	Med	0:00:31
3	1X	Large	0:00:44
3	10X	Small	0:02:59
3	10X	Med	0:03:05
3	10X	Large	0:05:14
3	100X	Small	0:31:50
3	100X	Med	0:31:53
3	100X	Large	0:34:40
5	1X	Small	0:00:37
5	1X	Med	0:00:40
5	1X	Large	0:00:54
5	10X	Small	0:04:00
5	10X	Med	0:03:56
5	10X	Large	0:05:40
5	100X	Small	0:30:31
5	100X	Med	0:30:24
5	100X	Large	0:39:37

TABLE 9

AtMost(x) SQL

Number of Hosts	TPCH Scale	Query Size	Runtime
3	1X	Small	0:00:29
3	1X	Med	0:00:32
3	1X	Large	0:00:52
3	10X	Small	0:04:00
3	10X	Med	0:04:04
3	10X	Large	0:08:21
3	100X	Small	0:46:48
3	100X	Med	0:46:57
3	100X	Large	0:57:00
5	1X	Small	0:00:37
5	1X	Med	0:00:43
5	1X	Large	0:01:03
5	10X	Small	0:04:51
5	10X	Med	0:04:52
5	10X	Large	0:09:39
5	100X	Small	0:47:55
5	100X	Med	0:48:52
5	100X	Large	7:22:17

TABLE 10

AtMost(x) Naive

Number of Hosts	TPCH Scale	Query Size	Runtime
3	1X	Small	0:00:35
3	1X	Med	0:00:35
3	1X	Large	0:00:38
3	10X	Small	0:05:30
3	10X	Med	0:05:10
3	10X	Large	0:05:36
3	100X	Small	0:54:29
3	100X	Med	0:54:16
3	100X	Large	0:54:25
5	1X	Small	0:00:39
5	1X	Med	0:00:39
5	1X	Large	0:00:57
5	10X	Small	0:03:56
5	10X	Med	0:04:00
5	10X	Large	0:05:44
5	100X	Small	0:52:38
5	100X	Med	0:53:26
5	100X	Large	0:53:33

TABLE 11

AtMost(x) Semijoin

Number of Hosts	TPCH Scale	Query Size	Runtime
3	1X	Small	0:00:30
3	1X	Med	0:00:33
3	1X	Large	0:00:46
3	10X	Small	0:03:11
3	10X	Med	0:03:23
3	10X	Large	0:04:46
3	100X	Small	0:31:53
3	100X	Med	0:32:06
3	100X	Large	0:34:43
5	1X	Small	0:00:37
5	1X	Med	0:00:40
5	1X	Large	0:00:55
5	10X	Small	0:03:56
5	10X	Med	0:03:57
5	10X	Large	0:05:43
5	100X	Small	0:30:23
5	100X	Med	0:30:13
5	100X	Large	0:39:43

TABLE 12

Ratio(x) SQL

Number of Hosts	TPCH Scale	Query Size	Runtime
3	1X	Small	0:00:35
3	1X	Med	0:00:36
3	1X	Large	0:01:11
3	10X	Small	0:05:32
3	10X	Med	0:05:31
3	10X	Large	0:13:56
3	100X	Small	1:12:12
3	100X	Med	1:12:03
3	100X	Large	1:44:11
5	1X	Small	0:00:44
5	1X	Med	0:00:45
5	1X	Large	0:01:19
5	10X	Small	0:05:49
5	10X	Med	0:05:53
5	10X	Large	0:13:56
5	100X	Small	1:18:26
5	100X	Med	1:18:56
5	100X	Large	4:03:06

TABLE 13

Ratio(x) Naive

Number of Hosts	TPCH Scale	Query Size	Runtime
3	1X	Small	0:00:36
3	1X	Med	0:00:36
3	1X	Large	0:00:39
3	10X	Small	0:05:05
3	10X	Med	0:05:05
3	10X	Large	0:05:37
3	100X	Small	0:53:35
3	100X	Med	0:53:43
3	100X	Large	0:53:40
5	1X	Small	0:00:42
5	1X	Med	0:00:40
5	1X	Large	0:00:45
5	10X	Small	0:05:41
5	10X	Med	0:05:39
5	10X	Large	0:05:55
5	100X	Small	0:53:09
5	100X	Med	0:52:52
5	100X	Large	0:53:29

TABLE 14

Ratio(x) Semijoin

Number of Hosts	TPCH Scale	Query Size	Runtime
3	1X	Small	0:00:42
3	1X	Med	0:00:44
3	1X	Large	0:00:48
3	10X	Small	0:05:33
3	10X	Med	0:05:44
3	10X	Large	0:07:42
3	100X	Small	0:55:32
3	100X	Med	0:55:44
3	100X	Large	0:59:52
5	1X	Small	0:00:57
5	1X	Med	0:00:57
5	1X	Large	0:00:55
5	10X	Small	0:07:00
5	10X	Med	0:07:00
5	10X	Large	0:07:25
5	100X	Small	1:03:14
5	100X	Med	1:03:11
5	100X	Large	1:37:53

TABLE 15

All(x) SQL

Number of Hosts	TPCH Scale	Query Size	Runtime
3	1X	Small	0:00:38
3	1X	Med	0:00:39
3	1X	Large	0:00:49
3	10X	Small	0:05:35
3	10X	Med	0:06:04
3	10X	Large	0:08:25
3	100X	Small	1:03:12
3	100X	Med	1:04:21
3	100X	Large	1:18:35
5	1X	Small	0:00:47
5	1X	Med	0:00:48
5	1X	Large	0:01:00
5	10X	Small	0:06:23
5	10X	Med	0:06:34
5	10X	Large	0:07:59
5	100X	Small	1:09:11
5	100X	Med	1:07:57
5	100X	Large	1:56:29

TABLE 16

All(x) Naive

Number of Hosts	TPCH Scale	Query Size	Runtime
3	1X	Small	0:00:34
3	1X	Med	0:00:35
3	1X	Large	0:00:39
3	10X	Small	0:04:57
3	10X	Med	0:05:04
3	10X	Large	0:05:22
3	100X	Small	0:54:03
3	100X	Med	0:53:15
3	100X	Large	0:53:38
5	1X	Small	0:00:44
5	1X	Med	0:00:39
5	1X	Large	0:00:44
5	10X	Small	0:05:42
5	10X	Med	0:05:36
5	10X	Large	0:05:54
5	100X	Small	0:52:33
5	100X	Med	0:53:47
5	100X	Large	0:53:40

TABLE 17

All(x) Semijoin

Number of Hosts	TPCH Scale	Query Size	Runtime
3	1X	Small	0:00:35
3	1X	Med	0:00:36
3	1X	Large	0:00:43
3	10X	Small	0:04:08
3	10X	Med	0:04:00
3	10X	Large	0:06:28
3	100X	Small	0:49:17
3	100X	Med	0:48:52
3	100X	Large	0:54:37
5	1X	Small	0:00:44
5	1X	Med	0:00:46
5	1X	Large	0:00:50
5	10X	Small	0:05:04
5	10X	Med	0:05:55
5	10X	Large	0:06:19
5	100X	Small	0:50:33
5	100X	Med	0:53:09
5	100X	Large	1:21:38

TABLE 18

AllBut(x) SQL

Number of Hosts	TPCH Scale	Query Size	Runtime
3	1X	Small	0:00:36
3	1X	Med	0:00:35
3	1X	Large	0:01:00
3	10X	Small	0:05:25
3	10X	Med	0:05:32
3	10X	Large	0:13:01
3	100X	Small	1:14:17
3	100X	Med	1:17:51
3	100X	Large	1:39:16
5	1X	Small	0:00:49
5	1X	Med	0:00:47
5	1X	Large	0:01:13
5	10X	Small	0:06:04
5	10X	Med	0:05:49
5	10X	Large	0:14:13
5	100X	Small	1:22:42
5	100X	Med	1:20:47
5	100X	Large	4:06:47

TABLE 19

AllBut(x) Naive

Number of Hosts	TPCH Scale	Query Size	Runtime
3	1X	Small	0:00:34
3	1X	Med	0:00:35
3	1X	Large	0:00:38
3	10X	Small	0:05:01
3	10X	Med	0:05:05
3	10X	Large	0:05:23
3	100X	Small	0:54:54
3	100X	Med	0:53:13
3	100X	Large	0:54:05
5	1X	Small	0:00:41
5	1X	Med	0:00:42
5	1X	Large	0:00:43
5	10X	Small	0:05:41
5	10X	Med	0:05:42
5	10X	Large	0:05:54
5	100X	Small	0:53:02
5	100X	Med	0:53:20
5	100X	Large	0:54:44

TABLE 20

AllBut(x) Semijoin

Number of Hosts	TPCH Scale	Query Size	Runtime
3	1X	Small	0:00:45
3	1X	Med	0:01:34
3	1X	Large	0:01:35
3	10X	Small	0:11:10
3	10X	Med	0:11:14
3	10X	Large	0:14:25
3	100X	Small	1:51:02
3	100X	Med	1:55:02
3	100X	Large	2:01:22
5	1X	Small	0:01:51
5	1X	Med	0:01:54
5	1X	Large	0:01:58
5	10X	Small	0:14:29
5	10X	Med	0:14:05
5	10X	Large	0:14:58
5	100X	Small	2:06:38
5	100X	Med	2:06:50
5	100X	Large	3:05:22

REFERENCES

[1] Abiteboul, S., Hull, R. and Vianu, V., *Foundations of Databases*, Addison-Wesley, 1995.

[2] Badia, A. *Generalized Quantifiers in Action*, Springer, 2009.

[3] Gyssens, M. and Van Gucht, D. and Badia, A., *Query Languages with Generalized Quantifiers*, in *Application of Logic Databases*, Ramakrishnan, Ragu, editor, Kluwer Academic Publishers, 1995.

[4] Hsu, P. Y. and Parker, D. S., *Improving SQL with Generalized Quantifiers*, in Proc. of ICDE, 1995.

[5] Keenan, E. and Moss, L., *Generalized Quantifiers and the Expressive Power of Natural Language*, in [11].

[6] Kossman, D. *The State of the Art in Distributed Query Processing*, ACM Computing Surveys, 32(4), December 2000.

[7] Kolaitis, P. and Väänänen, J., *Generalized Quantifiers and Pebble Games on Finite Structures*, Annals of Pure and Applied Logic, Elsevier, v. 74, 1995, pages 23-75.

[8] Keenan, E. and Westerstahl, D., *Generalized Quantifiers in Linguistics and Logic*, in [11].

[9] Lindstrom, P., *First Order Predicate Logic with Generalized Quantifiers*, Theoria, vol. 32, 1966.

[10] Mostowski, A., *On a Generalization of Quantifiers*, Fundamenta Mathematica, vol. 44, 1957.

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

[12] Transaction Processing Performance Council, *TPC BENCHMARK H (Decision Support) Standard Specification*, http://www.tpc.org/tpch/spec/tpch2.13.0.pdf , 2010.

[13] Vaananen, J. *Unary Quantifiers on Finite Models*, Journal of Logic, Language and Information, v. 6, 1997, pages 275-304.

[14] Westerstahl, D., *Quantifiers in Formal and Natural Languages*, Handbook of Philosophical Logic, Gabbay, D. and Guenther, F., eds., Reidel Publishing Company, 1989, vol. IV, ch. 4.

[15] Ramasamy, K. and Patel, J. and Naughton, J. and Kaushik, R., *Set Containment Joins: The Good, The Bad and The Ugly*, Proc. of VLDB, 2000.

[16] Mamoulis, N., *Efficient Processing of Joins on Set-valued Attributes*, Proc. of SIGMOD, 2003.

CURRICULUM VITAE

NAME: Michael Dobbs

ADDRESS: Computer Engineering Computer Science
University of Louisville
Louisville, KY 40292

EDUCATION: B.S. CECS
University of Louisville
2009