ACKNOWLEDGEMENTS

This work would not have been possible without the support and encouragement of Dr. Chengkai Li under whose supervision I chose this topic and began the thesis. Dr. Li, my thesis advisor, has been immensely helpful, and has assisted me in numerous ways, which included helping me familiarize myself with the art of research, learning new technologies and introducing me to some of the most exciting directions in the fields of databases and data management. I believe that by working with him I have become more detail-oriented and dedicated to quality.

I would also like to thank Dr. Bahram Khalili, my graduate advisor and professor, for his help, advice and guidance throughout my M.S.

I cannot end without thanking my parents, on whose constant support, encouragement and love I have relied throughout my time at the University of Texas at Arlington. I am grateful also to my brother, Saad Safiullah, my sister, Amina Khan and my brother-in-law Aminuddin Ali Khan for inspiring me to always work with conviction and diligence.

July 14, 2008
ABSTRACT

EFFICIENT PROCESSING OF SET QUERIES
USING BITMAP INDEXES

Muhammad Assad Safiullah, M.S

The University of Texas at Arlington, 2008

Supervising Professor: Chengkai Li

Growing complexity of enterprise-wide data and business processes necessitates the efficiency of complex decision support set queries. However, contemporary DBMS remain unsuccessful in handling set queries efficiently. In this thesis we propose efficient set query processing methods using bitmap index. The methods use bitmap vectors to represent attributes values in binary format. The methods test groups within a schema in a hierarchical fashion. Satisfying groups are bisected further and checked recursively while non-satisfying groups are pruned resulting in significant reduction in response times. In addition, our iterative implementation avoids the inefficiency that can be introduced by recursive implementation by reading the same bitmap vector for intersection many times. We also introduce pre-processing methods to reduce the complexity of the bitmap vectors, thus to improve the efficiency. Our implementation is based on FastBit, an open-source efficient compressed bitmap index framework. Experimental results on large datasets and comparison with results from PostgreSQL prove that our approach is superior owing to the fact that we are able to discard non-satisfying groups and capably optimize complex queries.
3.1.1 Limitation of Expression.......................................................................................7
3.1.2 Limitation of Interpretation ..................................................................................8

4. OUR APPROACH .......................................................................................................9
4.1. Overview of Our Methods.......................................................................................9
4.1.1 Description of Bitmap Index ...............................................................................9
4.1.2 Limitations of Bitmap Index ..............................................................................10
4.2. Our Methods for Single Condition ......................................................................11
4.2.1 Recursive Algorithm .........................................................................................11
4.2.2 Iterative Algorithm ............................................................................................12
4.3. Our Methods for Multiple Conditions ................................................................12
4.3.1 Method 1: using Iterative Method for Single Condition multiple times ..............13
4.3.2 Method 2 .............................................................................................................14
4.4. Preprocessing .......................................................................................................15

5. IMPLEMENTATION ................................................................................................16
5.1. FASTBIT .............................................................................................................16
5.1.1 Index Creation ..................................................................................................16
5.1.2 Bit-sliced Index Vectors ...................................................................................17
5.1.3 Intersection and Union ....................................................................................17
5.1.4 Zero Tests ........................................................................................................17
5.2. Data Structure ....................................................................................................17

6. EXPERIMENTAL RESULTS AND FINDINGS ......................................................18
6.1. Data Settings .......................................................................................................19
6.2. Effect of Buffering ..............................................................................................19
6.3. Recursive vs. Iterative Implementations .............................................................19
6.4. The CONTAINS-SINGLE and CONTAINS-SINGLE-P Algorithms ..................19
6.5. Effect of Increasing Number of Satisfying Groups ............................................20
6.6. Effect of Increasing Number of Groups .......................................................... 20

6.7. The CONTAINS-MULTIPLE Algorithm and Alternatives ............................... 20

7. FUTURE WORK .................................................................................................... 27

7.1. Shortening of Bit-Sliced Index on Grouping Attribute ................................. 27

7.1.1 Transformation of BSI to reduce length ..................................................... 27

7.1.2 Limitations of Bit-Sliced Index Shortening ............................................... 27

7.2. Experimentation with More than Two Constant Values for CONTAINS ....... 28

7.3. Implementation of Other Set Operations ..................................................... 28

7.4. Experimentation with Different Data ........................................................... 28
<table>
<thead>
<tr>
<th>Figure</th>
<th>Illustration Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Overview of Our Methods</td>
<td>3</td>
</tr>
<tr>
<td>4-1</td>
<td>Bitmap Index on Gender</td>
<td>10</td>
</tr>
<tr>
<td>4-2</td>
<td>Bit-sliced Index on Class</td>
<td>10</td>
</tr>
<tr>
<td>4-3</td>
<td>Recursive Algorithm for CONTAINS with single condition</td>
<td>11</td>
</tr>
<tr>
<td>4-4</td>
<td>Iterative Algorithm for CONTAINS with single condition</td>
<td>12</td>
</tr>
<tr>
<td>4-5</td>
<td>Method 1 Algorithm for CONTAINS with multiple conditions</td>
<td>13</td>
</tr>
<tr>
<td>4-6</td>
<td>Method 2 Algorithm for CONTAINS with multiple conditions</td>
<td>14</td>
</tr>
<tr>
<td>4-7</td>
<td>Method 1 Algorithm for CONTAINS with multiple conditions</td>
<td>15</td>
</tr>
<tr>
<td>6-1</td>
<td>Effect of increasing ratio of # Satisfying Groups / # of Groups</td>
<td>22</td>
</tr>
<tr>
<td>6-2</td>
<td>Effect of increasing number of Tuples</td>
<td>23</td>
</tr>
<tr>
<td>6-3</td>
<td>Effect of increasing number of Satisfying Groups</td>
<td>24</td>
</tr>
<tr>
<td>6-4</td>
<td>Effect of increasing number of Groups</td>
<td>25</td>
</tr>
<tr>
<td>6-5</td>
<td>Effect of increasing number of Tuples</td>
<td>26</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1 Student Grades</td>
<td>6</td>
</tr>
<tr>
<td>4-1 Personnel Information</td>
<td>10</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

1.1. Set Queries

Comparisons between groups of tuples within a database relation with other attributes and values require syntax and semantics that currently available Database Management Systems do not provide. When it comes to expression of queries that involve group level operations, the SQL language significantly lacks the syntax that can effectively articulate such procedures. Similarly, the interpretation mechanism of current DBMS also falls short in efficiency owing to the fact that all comparisons are made at the tuple level and not at the group level. As a result, SQL queries that might involve set operations are required to be translated into queries composed of tuple level operations making them harder to construct and counterintuitive.

1.1.1 Description of Set Queries

Set queries, motivated and defined in a research paper draft “Set Predicates for SQL: A Simple and Efficient Approach to Enable Set Level Comparisons” [1] from Bin He et al., provide a means of expressing queries that may involve set operations not native to the SQL language. Examples of such operations are CONTAINS, NOT CONTAINS, CONTAINED BY, EQUALS and NEGATION of these operations. Each of these individual operations functions on sets and confirm the boolean value of the predicates they form by examining membership of group elements.

1.1.2 The Need for Set Queries

In recently emerging classes of database applications set queries can be of utmost importance. At present, users have to go through the tedious process of translating set queries into tuple level operations which are the only ones provided by current SQL. In addition to that
they have to manually optimize them otherwise performance could seriously suffer. A concrete example of applications that can benefit from set queries is of Decision Support Systems. DSS are employed for supporting business and organizational decision-making and involve meticulous set-level comparisons e.g. comparing sales figures of one group of products to another. Even though the semantics of these queries suggest use of group level operations, SQL supports only tuple level comparisons.

1.2. Challenges

For us to be able to incorporate set queries into the typical SQL language there are several challenges that we had to confront.

1.2.1 Unavailability of Syntax to Express Set Queries

An intrinsic limitation of SQL and hence a challenge for us was the absence of appropriate syntax to express set operations. Because of this, set level queries need to be reconstructed using tuple-level operations which made them complex and error-prone.

This necessitated the introduction of suitable keywords to denote set operations.

1.2.2 Tuple-level Query Processing

Query processing within current DBMS also occurs at tuple level. Hence, even for queries involving group operations each tuple within a group is processed individually. These factors contribute to sizeable loss in performance.

For this reason, we designed and implemented several algorithms to process groups at the group-level in order to avoid scanning of tuples in a scalar fashion
1.3. Overview of our methods

1.3.1 High-level Intuition

Conceptually, our methods test groups in a hierarchical fashion (Figure 1-1). Initially the whole relation is treated as a single super-group. For as long as a super-group satisfies our condition it is bisected recursively into further sub-groups. Non-satisfying groups are pruned and do not participate in any further processing. This essentially means that they are not further bisected and all of their sub-groups are discarded.

1.3.2 Algorithms

We have implemented several algorithms that can efficiently process set queries. We targeted the set operation CONTAINS and designed various algorithms to process it each with different features in order to study which of them can be most promising. We believe that these
algorithms form the basis of other algorithms that can in effect implement other set operations like NOT CONTAINS, CONTAINED BY and EQUALS.

Implementation wise, our algorithms fall into two categories (a) Recursive (b) Iterative. Each of the algorithms was run with and without some preprocessing resulting in different results.

1.4. Summary of Results

Experimentation results allowed us to make several conclusions about our proposed methodologies. They can be summarized by the following points:

(i) Recursively implemented algorithms are not as efficient as iteratively implemented algorithms owing to unnecessary recalculations

(ii) Experiments for which buffering was available were able to perform better because of lesser disk accesses

(iii) By introducing a certain preprocessing stage before all algorithms we can significantly improve performance and thus response time of queries

(iv) All of our algorithms proved to be, in most cases, better than PostgreSQL in terms of efficiency

1.5. Outline of the Rest of the Thesis

In the rest of this thesis document, we present initially a formal definition of set queries. We advance with our discussion by analyzing currently available DBMS and highlighting their limitations in regard to set operations. Next we briefly discuss related work laying the stage for our approach. We then discuss our algorithms along with their main features and implementation characteristics. Eventually, we present experimentation results and provide comparisons of our different algorithms and also with PostgreSQL. We conclude our document with a discussion of the future work directions.
CHAPTER 2

FORMAL DEFINITION OF SET QUERY

2.1. The Syntax of Set Predicates

As a first step in enabling expression of set queries, introduced by Bin He et al., \[1\] we established that there was a requirement of new syntax in the SQL language. Since, set predicates essentially operate on groups the most appropriate position that they could take up with the basic skeleton of an SQL query was following the HAVING clause that applies restrictions on groups listed in front of the GROUP BY clause.

2.1.1 General Syntax

The general syntax of the SQL query with set predicates incorporated in its structure is:

SELECT ....
FROM R₁,....,Rₙ
WHERE ..... GROUP BY g₁,......gₘ
HAVING predicates

where predicates evaluates to a true or false value being a boolean expression comprised of aggregate or set predicates.

2.1.2 Set Predicates

The syntax of each set predicate is:

set_predicate ::= attribute_set set_operator constant_values

attribute_set ::= SET(col₁,....colₖ)

set_operator ::= [NOT | CONTAINS | CONTAINED BY | EQUALS]

constant_values ::= {( v₁¹,....,vₖ¹),....,(v₁ᵗ,....,vₖᵗ)}, where vₖ belongs to the set Domain(colᵢ), i.e. each vₖ can be a value including integer, floating point number, string, etc).
Table 2-1 Student Grades

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacob</td>
<td>CSE5311</td>
<td>A+</td>
</tr>
<tr>
<td>Jacob</td>
<td>CSE5213</td>
<td>A</td>
</tr>
<tr>
<td>Joshua</td>
<td>CSE5142</td>
<td>B</td>
</tr>
<tr>
<td>Jake</td>
<td>CSE5311</td>
<td>B</td>
</tr>
<tr>
<td>Muhammad</td>
<td>CSE6321</td>
<td>A+</td>
</tr>
<tr>
<td>Navathe</td>
<td>CSE5311</td>
<td>B</td>
</tr>
</tbody>
</table>

For example, to retrieve all students who took the course CSE 5311 (Table 2) and received an A or A+ grade the corresponding set query would be:

SELECT Student
FROM Student_Grades
GROUP BY Student
HAVING SET(Course) CONTAINS {'CSE5311'}
AND
SET(Grade) CONTAINED BY {'A+','A'}

And the result would be Jacob only.
CHAPTER 3
CONTEMPORARY DBMS

3.1. Limitations of Contemporary DBMS

As briefly explained in the previous sections, current DBMS do not support group or set-level operations. Even though the SQL language includes syntax to group tuples and apply restrictions on them, the syntax largely lacks expressiveness when it comes to set operations. In addition, the processing of these queries is at the scalar level no matter how the query is written.

3.1.1 Limitation of Expression

From the user’s point of view the queries become hard to construct and counter-intuitive when it comes to expressing set-queries. The fact that there are no operators defined in the SQL language for articulating the user’s request requires that all such queries that involve set operation be translated into tuple level operations.

Sometimes, such translation leads to queries that are hard to understand and error-prone. The set query:

\[
\text{SELECT TV SHOWS} \\
\text{FROM } R \\
\text{GROUP BY TV SHOWS} \\
\text{HAVING SET(RATING) EQUALS '5+'}
\]

when translated to traditional SQL becomes:

\[
(\text{SELECT TV SHOW} \\
\text{FROM } R \\
\text{WHERE RATING='5+'} \\
\text{GROUP BY TV SHOW})
\]
EXCEPT
(SELECT TV SHOW
FROM R
WHERE RATING<>‘5+’
GROUP BY TV SHOW)

3.1.2 Limitation of Interpretation

Current DBMS do not take treat groups as units and therefore even for group operations each tuple is analyzed independently. As a result for a certain number of tuples irrespective of the number of groups and satisfying groups the time taken to respond to a query is largely the same. The foremost reason for this behavior is the fact that to answer any query a full scan of the relation is required.

These limitations cause current DBMS to be largely insufficient when it comes to set operations.
CHAPTER 4
OUR APPROACH

In this chapter we present an overview of our methods, followed by the comparison between contemporary and efficient indexing mechanism utilized by our methods, and the illustration for each method.

4.1. Overview of Our Methods

Our methods process queries over input data in hierarchical manner. Initially the entire relation is passed as an input to our methods. During execution the input data to each pass is bisected into candidate groups which are then tested for validity on subsequent passes. Bit-sliced Index is used for bisection. The non-qualifying groups like zero-vectors and those which don’t satisfy conditions are pruned. This essentially means that they are not further bisected and all of their sub-groups are discarded. The candidate groups on last pass are the satisfying groups, which are classified and displayed as query processing results.

Our methods achieve efficiency through iterative implementation, pruning of non-satisfying groups at early stages and utilization of Bit-sliced Index in comparison to indexing mechanisms utilized by contemporary DBMS like Bitmap Indexing.

4.1.1 Description of Bitmap Index

Most of the contemporary databases use Bitmap Indexes for efficient data retrieval and manipulation. A bitmap index on an attribute requires a bitmap for each unique attribute value. This indexing mechanism is most appropriate for columns having low distinct values or low cardinality e.g. Gender from Table 4.1.
Two binary vectors are created for representing each class of Gender. Only those bits, which correspond to class value under consideration, are set to 1.

![Figure 4-1 Bitmap Index on Gender](image)

### 4.1.2 Limitations of Bitmap Index

Bitmap indexes incur storage and maintenance cost if the indexing attribute has many values. Many encoding schemes like binning and improvements like Bit-sliced Index have been studied for tackling this issue.

![Figure 4-2 Bit-sliced Index on Class](image)

Bit-sliced Index (BSI) stores the binary representation of attribute values. This approach minimizes the storage space utilized, which is more evident in case of columns with high
cardinality. For example, the BSI on Class column from Table 4.1 will require three vectors in contrast to the five vectors necessitated by Bitmap Indexing.

4.2. Our Methods for Single Condition

Following are the various methods to process set operation CONTAINS with single condition:

4.2.1 Recursive Algorithm

```
Algorithm: query(int i, vector v, int id)

Input:

i: the ith-most significant bit of BSI(g)
v: intermediate vector
id: the id of group being considered

begin
1    if zero-test(v)
2        return
3
4    if (i < 0)
5        output id
6
7    query(i-1, v(g, i) \land v, id*2+1)
8    query(i-1, \neg v(g, i) \land v, id*2)
end
```

Figure 4-3 Recursive Algorithm for CONTAINS with single condition

This algorithm is invoked by query(length(BSI(g))-1, v_{a=x}, 0), where length(BSI(g))-1 is the number of bits or vectors in BSI on g. Starting with v_{a=x}, we bisect a satisfying group into two vectors and recursively check the sub-groups (line 7 & 8). All non-satisfying groups are pruned (line 1). The discarded groups are not used for bisection and their sub-groups are also deleted on early stages.
4.2.2 Iterative Algorithm

```
Algorithm: query(int i, vectorset V)

Input:
i: the ith-most significant bit of BSI(g)
V: intermediate (vector, id) pair

begin
1   for each (v, id) ∈ V
2     if zero-test(v)
3       remove (v, id) from V
4     else if (i < 0)
5       output id
6
7   V' = ∅
8   for each (v, id) ∈ V
9       insert(v(g,i) ∧ v, id+2) into V'
10      insert(~v(g,i) ∧ v, id+2) into V'
11
12   if (i ≤ -1)
13      query(i-1, V')
end
```

Figure 4-4 Iterative Algorithm for CONTAINS with single condition

The iterative version of this algorithm is invoked by query(length(BSI(g))-1, \((v_{a=x}, 0))\), where length(BSI(g))-1 is the number of bits or vectors in BSI on g. Starting with \((v_{a=x}, 0))\), we bisect a satisfying group into two vectors and save them as candidate groups sub-groups (line 9 & 10). All non-satisfying groups are pruned (line 1). The discarded groups are not used for bisection and their sub-groups are also deleted on early stages.

The iterative version improves on recursive version by reading v(g,i) only once for intersection.

4.3. Our Methods for Multiple Conditions

Following are the various methods to process set operation CONTAINS with multiple conditions:
4.3.1 Method 1: using Iterative Method for Single Condition multiple times

Algorithm:

```
begin
1   for each x_i
2       G_i = query(length(BS_i(g)), (gamma_xi, 0))
3   G = \bigcap G_i
4
5   for each g \in G
6       output g
end
```

Figure 4-5 Method 1 Algorithm for CONTAINS with multiple conditions

Method 1 processes queries for multiple conditions by invoking Iterative Algorithm (4.2.2) for each condition. The results from each call are retained for intersection of groups. The set of intersected groups represents the qualifying groups which satisfy all conditions.
4.3.2 Method 2

Algorithm: query\(\text{int}\ i, \text{vectorset}\ \mathcal{V}\)

\textbf{Input:}

\(i,: \text{the } i\text{-th most significant bit of } \text{BSI}(g)\)

\(\mathcal{V}: \text{intermediate (vector, id) pair}\)

\begin{align*}
\text{begin} \\
\text{for each } (v, \text{id}) \in \mathcal{V} \\
\quad \text{for each } x_i \\
\quad \quad v' &= v \land \neg x_i \\
\quad \quad \text{if zero-test}(v') \\
\quad \quad \quad \text{remove } (v, \text{id}) \text{ from } \mathcal{V} \\
\text{if } (i < 0) \\
\quad \text{output id} \\
\text{return} \\
\text{end} \\
\end{align*}

\(\mathcal{V}' = \emptyset\)

\begin{align*}
\text{for each } (v, \text{id}) \in \mathcal{V} \\
\quad \text{insert}\(v(g, i) \land v, \text{id}'2+1\) \text{ into } \mathcal{V}' \\
\quad \text{insert}\(\neg v(g, i) \land v, \text{id}'2\) \text{ into } \mathcal{V}' \\
\text{if } (i \leq -1) \\
\quad \text{query}(i-1, \mathcal{V}')
\end{align*}

Figure 4-6 Method 2 Algorithm for CONTAINS with multiple conditions

Method 2 is a self-sufficient algorithm for solving multiple-conditions queries. This algorithm is invoked by query(length(\text{BSI}(g))-1, (v, 0)), where length(\text{BSI}(g))-1 is the number of bits or vectors in \text{BSI} on \(g\), and \(v\) in (\(v, 0\)) is all ones vector with id = 0. Starting with the initial group, if a group satisfies the condition, we bisect the candidate group into two subgroups and save them for checking on subsequent pass. Whenever a group fails in satisfying one of the conditions, it is pruned immediately without further checking any subgroups contained in it.
4.4. Preprocessing

Our methods can further be improved by introducing preprocessing. It involves transforming BSI vectors such that new BSI vectors are the intersection of individual vectors from BSI on g and condition vector $v_a=x$.

Preprocessing:
begin
1 for each $v(g, i)$
2 $v(g, i) = v(g, i) \land v_a=x_i$
3
4 query(length(BSI(g])-1, (v, 0))
end

Figure 4-7 Method 1 Algorithm for CONTAINS with multiple conditions

Following to transformation any algorithm can be invoked to perform single-condition or multiple-conditions query processing.
CHAPTER 5
IMPLEMENTATION

5.1. FASTBIT

The in-built bit-sliced indexing mechanism, compression of large bit vectors and easy-to-use bit vector operators provided by FastBit \cite{2} facilitated in efficient implementation of set query processing methods.

The C++ implementation for our method uses open source Bitmap Index framework FastBit for optimized operations. FastBit implements a number of different bitmap indexes compressed with Word-Aligned Hybrid code. One of the supported indexing mechanisms is Bit-sliced Index. The simplest Bit-sliced Index, known as Verbatim Index, has been used in our implementation for suggested methods.

5.1.1 Index Creation

Conceptually, FastBit divides the index creation process into three steps: Binning, Encoding and Compression. The basic idea of bitmap index is to generate one bitmap to represent the presence of each distinct value. This is generally considered as an efficient approach for attributes with low cardinalities. For attributes with high cardinalities, one way to reduce the number of bitmaps is to bin the values and produce one bitmap for each bin. After the values are divided into bins, the next step is to encode values as bitmaps. The simplest encoding scheme is to have one bitmap for each bin. In this case, a bit value of one indicates an entry is in a specified bin. After the bitmaps are constructed, compression may be applied. Currently, the bitmaps are compressed at creation time.

Our methods utilize create method provided by ibis::index with an index specification of slice type which uses binary encoding. The method itself creates a specific concrete index object. The input values, in order, are the column-to-be-indexed, index file name and the index
specification. If this function fails to read a specified index file, it attempts to create a new index based on the current data file and index specification. The new index is then written under the old name.

5.1.2 Bit-sliced Index Vectors

Fastbit's ibis::index class provides functions for accessing individual vectors from created index. The function takes the index $i$ as an input and returns a pointer to $i^{th}$ bitvector of Bit-sliced Index.

5.1.3 Intersection and Union

Fastbit also provides the intersection and union operators which work with binary vectors of ibis::bitvector type. The & and &= operators provide the intersection functionality, and the | and |= operators provide the union functionality.

5.1.4 Zero Tests

The ibis::bitvector class provides a function for determining the number of ones in a bitvector. Our methods use this function for determining if a bitvector is a zero vector and needs to be pruned.

5.2. Data Structure

Initially our implementation used a linked list to store intermediate and qualifying group vectors. A newly created vector was always inserted at the front of linked list and thus was an O(1) operation. However, in worst case, the deletion of non-qualifying vectors required complete traversal of linked list and thus was an O(n) operation, where $n$ represents the number of group vectors in a linked list.

Experimentation showed the significant overhead incurred due to inefficient deletion strategy. To tackle this challenge, an array of bit vectors indexed by corresponding group id was introduced to perform O(1) deletion operation.
CHAPTER 6
EXPERIMENTAL RESULTS AND FINDINGS

To evaluate and compare the performance of our approach with current systems we performed each of the algorithms whose design and implementation is described in the previous chapter on a Dell PowerEdge Server with RAID.

Our experiments can be broadly categorized into two categories:

- CONTAINS
- CONTAINS-MULTIPLE

For each of the two, we performed the experiments with different configurations and derived different conclusions about each of the configurations.

The following suffixes will be used throughout this section to differentiate between various techniques:

- R: recursive
- I: iterative
- P: preprocessed
- N: without buffering enabled
- B: with buffering enabled
6.1. Data Settings

Each of the individual experiment with unique configuration was performed on 46 unique cases with different combinations of number of tuples, number of groups and satisfying groups.

The data was generated using uniform distribution. Each individual data set was a relation of three attributes, including a unique identifier for each tuple, a condition attribute and a grouping attribute.

The number of tuples of each data set ranged from 10,000 to 1,000,000. The number of groups within each relation ranged from 10 to the total number of tuples in case for the case of CONTAINS. For CONTAINS-MULTIPLE however, the maximum number of groups was half of the number tuples due to the fact that each group was comprised of at least two tuples. The number of satisfying groups ranged from 10 to the total number of groups.

6.2. Effect of Buffering

Figure 6-1 highlights the effect of buffering i.e. storage of disk data in the RAM to reduce number of device accesses. For each of the algorithms, experiments with buffering enabled perform better than their counterparts with buffering disabled since device accesses are fairly expensive. This is also true for experiments with PostgreSQL.

6.3. Recursive vs. Iterative Implementations

Figure 6-1 also highlights the difference in performance between recursive and iterative implementations. As explained earlier, recursive algorithms bear the overhead of reading data that is already available several times and hence reducing performing. Experimentation shows that iterative implementations consistently perform better than recursive implementations.

6.4. The CONTAINS-SINGLE and CONTAINS-SINGLE-P Algorithms

Figure 6-2 exhibits how the performance of the CONTAINS-SINGLE and CONTAINS-SINGLE-P algorithm compares with PostgreSQL. For the case demonstrated the number of groups and satisfying groups is fixed and the number of tuples is varied against time. It can be
deduced from the figure that when the number of satisfying groups is relatively small as compared to the total number of groups, the performance of CONTAINS-SINGLE is superior to the performance of PostgreSQL. In addition, CONTAINS-SINGLE-P (with processing) further improves the response time.

6.5. Effect of Increasing Number of Satisfying Groups

Figure 6-3 exhibits how the performance of the CONTAINS-SINGLE algorithm varies when number of satisfying groups increases. For the case demonstrated the number of groups and the number of tuples is fixed. It can be deduced that for a small number of satisfying groups the performance of CONTAINS-SINGLE and CONTAINS-SINGLE-P is much better than PostgreSQL. However, as the number of satisfying groups equals the total number of groups the performance of our algorithms decreases so much as to go even below that of PostgreSQL. It can also be concluded that the only factor that causes the performance of PostgreSQL to change considerably is the number of tuples and it is largely immune to changes in the number of groups and the number of satisfying groups owing to the fact that it operates on the scalar level and for any kind of operation performs a full scan of the relation.

6.6. Effect of Increasing Number of Groups

Figure 6-4 shows the effect of increasing the number of groups on the performance of the CONTAINS-SINGLE algorithm while the number of tuples and satisfying groups remains fixed. Again, this change does not affect the performance of PostgreSQL. However, for CONTAINS-SINGLE and CONTAINS-SINGLE-P, the performance decreasing with increasing number of groups.

However, for all cases, preprocessing ensures that the performance of the algorithm is improved.

6.7. The CONTAINS-MULTIPLE Algorithm and Alternatives

Figure 6-5 exhibits the performance of the CONTAINS-MULTIPLE and CONTAINS-MULTIPLE-P algorithms for two constant values. It also shows the performance of the
alternative described in previous sections. The alternative does NOT involve calling the CONTAINS-SINGLE algorithm more than once.

Our case demonstrates that for two constant values, CONTAINS-MULTIPLE performs better than its alternative. However, since our contention is that CONTAINS-MULTIPLE takes time that is linear in the number of constants it might be the case that for a large number of constant values the performance of the alternative is better than CONTAINS-MULTIPLE. However, we would require further investigation to be able to make such a conclusion.
Figure 6-1 Effect of increasing ratio of # Satisfying Groups / # of Groups
Figure 6-2 Effect of increasing number of Tuples
### Figure 6-3 Effect of increasing number of Satisfying Groups

<table>
<thead>
<tr>
<th>Number of Satisfying Groups (100,000 tuples and 100,000 groups)</th>
<th>PostgreSQL</th>
<th>Contains-single</th>
<th>Contains-single-P</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>142.49</td>
<td>30.544</td>
<td>47.2121</td>
</tr>
<tr>
<td>100</td>
<td>188.877</td>
<td>33.1631</td>
<td>52.1698</td>
</tr>
<tr>
<td>1000</td>
<td>165.123</td>
<td>137.913</td>
<td>156.69</td>
</tr>
<tr>
<td>10000</td>
<td>215.232</td>
<td>1115.05</td>
<td>1242.264</td>
</tr>
<tr>
<td>1E+05</td>
<td>341.902</td>
<td>4456.75</td>
<td>4716.06</td>
</tr>
</tbody>
</table>

- PostgreSQL
- Contains-single
- Contains-single-P
Figure 6-4 Effect of increasing number of Groups

Time (ms)

<table>
<thead>
<tr>
<th>Number of Groups (1,000,000 tuples and 10 satisfying groups)</th>
<th>PostgreSQL</th>
<th>Contains-single</th>
<th>Contains-single-P</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>434.896</td>
<td>20.1449</td>
<td>0.920296</td>
</tr>
<tr>
<td>100</td>
<td>460.556</td>
<td>20.0522</td>
<td>1.45793</td>
</tr>
<tr>
<td>1000</td>
<td>510.27</td>
<td>20.4091</td>
<td>2.0988</td>
</tr>
<tr>
<td>10000</td>
<td>510.352</td>
<td>22.893</td>
<td>3.05176</td>
</tr>
<tr>
<td>1E+05</td>
<td>512.521</td>
<td>30.9319</td>
<td>3.44706</td>
</tr>
<tr>
<td>1E+06</td>
<td>579.933</td>
<td>34.64</td>
<td>3.46899</td>
</tr>
</tbody>
</table>
Figure 6-5 Effect of increasing number of Tuples
CHAPTER 7
FUTURE WORK

The work presented in this thesis lays the foundation for a more thorough study of set queries and how they can enhance the quality of both expression and interpretation of queries that are common to the newly emerging database applications.

In addition, we feel that several methodologies can be built on top of our algorithms to further improve efficiency.

7.1. Shortening of Bit-Sliced Index on Grouping Attribute

One of the significant directions of future work could be shortening of the Bit-sliced index on the grouping attribute described in Chapter 5 to reduce the number of stages of the algorithms performed. This could lead to less vector operations and hence considerable improvement in efficiency.

7.1.1 Transformation of BSI to reduce length

In the course of our work we made an attempt to transform the BSI and in the process shortening it. The goal was to reduce the length of each of the vectors within the BSI by saving only those bits within the vectors whose corresponding bits in the bitmap created on the condition column are set.

For example, if the BSI vector i is 0101001 and the bitmap on the condition attribute is 0100101, the only bits that we would save in the BSI vector would be the 2\(^{nd}\), 5\(^{th}\) and 7\(^{th}\) and hence the new BSI vector i would be 101.

7.1.2 Limitations of Bit-Sliced Index Shortening

Although experimentation results revealed that shortening of the BSI yielded much faster response time we discovered that the preprocessing stage was very expensive so much that the total response time was increased by sometimes more than 10 times.
The reason lay in the methodology we applied for the conversion. To be able to figure out which of the positions of the bitmap were set, we were forced to perform intersections of the bitmap with n vectors, with n being the length of the bitmap. Each of the n vectors had only one bit set in a unique position. If an AND operation with a certain vector yielded a result with more than zero bits set in it we made the deduction that the position where the vector had a bit set was one where the bitmap also had a bit set.

Clearly, the procedure turned out to be infeasible and therefore one of our intentions for future work involves design of an efficient method to perform this transformation without making it too expensive.

7.2. Experimentation with More than Two Constant Values for CONTAINS

Currently our results for CONTAINS encompass experiments with at most two constant values. Our contention is that CONTAINS-MULTIPLE i.e. running the CONTAINS algorithm several times to get results for more than one constants would result in a response time that is linear in the number of constants. Here, in the future, we intend to perform experiments for a greater number of constants in order to deduce if the alternative that we presented for CONTAINS-MULTIPLE performs better or not.

7.3. Implementation of Other Set Operations

Our work primarily focuses on the set operation CONTAINS for single and multiple columns. We believe that other set operations namely CONTAINED BY, EQUALS and NEGATION of the same can be built on the same lines and hence we enumerate this as another effort we wish to make as part of future work.

7.4. Experimentation with Different Data

All of our experiments were performed on data generated using uniform distribution. As part of future endeavors we shall perform the same on data generated using exponential distribution to derive conclusions about the performance on different types of data.
REFERENCES


BIOGRAPHICAL INFORMATION

Muhammad Assad Safiullah, is an MS-CS candidate in the University of Texas at Arlington. Safiullah received his B.S. in Computer Science from the National University of Computer and Emerging Sciences, Islamabad, Pakistan in 2005 and will receive his M.S. in Computer Science in 2008 from the University of Texas at Arlington. His research interests include Artificial Intelligence, Machine Learning and Data Mining. Before joining UTA as a graduate student, Safiullah worked with a leading Business Process Management Vendor ULTIMUS as a Support and Maintenance Specialist in Rawalpindi, Pakistan for a period of one year. In summer 07, Safiullah joined Siemens Corporate Research in Princeton, NJ as an Intern and developed a Power-Down Consistent File System for the S7-300 Siemens PLC. After completion of his internship in January 08, Safiullah became the Sun Microsystems Campus Ambassador for the University of Texas at Arlington and held the position for the entire semester of Spring 08. His M.S. thesis focused on Efficient Processing of Set Queries using Bitmap Indexes.