Set Predicates in SQL: Enabling Set-Level Comparisons for Dynamically Formed Groups

Chengkai Li[1], Ning Yan[1], Muhammad Safiullah[2]*, Rakesh Ramegowda[1], Bin He[3]

[1]*cli@uta.edu,{ning.yan,rakesh.ramegowda}@mavs.uta.edu, University of Texas at Arlington*
[2]*muhammadassad.safiullah@mavs.uta.edu, Microsoft, Seattle, WA*
[3]*binhe@us.ibm.com, IBM Almaden Research Center*

ABSTRACT

The current SQL language supports the selection of tuples using conditions on each tuple (i.e., the predicates in WHERE) and a group of tuples (i.e., the predicates in HAVING). However, the comparison condition in a predicate, even for a group of tuples, is always performed at the *scalar level*. That is, for each comparison, only one value extracted or aggregated from one column is compared to another. Such *scalar-level* predicates become increasingly inadequate to support a new class of operations that require *set-level* comparison semantics, i.e., comparing a group of tuples with multiple values. Complex SQL queries composed by scalar-level operations are often formed to obtain even very simple set-level semantics. Such queries are not only difficult to write but also challenging for the database engine to optimize, thus can result in costly evaluation. To address this problem, this paper proposes to augment SQL with a new type of predicates, *set predicates*, to bring out the otherwise obscure semantics and thus facilitate the direct and efficient support. We proposed three approaches to answer queries with set predicates: a query rewriting method, an aggregate function-based method and a bitmap index-based method. We conducted comprehensive experiments over synthetic data and TPC-H data. The results show that the bitmap index-based approach can significantly outperform the other two alternatives.

1. INTRODUCTION

With real-world database applications becoming more sophisticated, there is a high demand of querying data with the semantics of *set-level comparisons*. For instance, a company may search their resume database for job candidates with a set of mandatory skills; in business decision making, an executive may want to find the departments whose monthly average ratings for customer service in this year have been both good and bad, *i.e.*, the set of ratings subsumes {excellent, poor}; in online advertisement, a marketing strategist may want to find all the Websites that publish ads for ING and Emigrant but not HSBC yet. Since these banks have similar business, HSBC can be a potential advertiser for those publishers.

In most database design, without the explicit support of set-valued attributes, such attribute is often modeled as a separate table. For example, a table Resume_Skills (id, skill) can be used to connect a set of skills to job candidates in another table. The GROUP BY clause can then be used to dynamically group the tuples by id, with the values on attribute skill in the same group forming a set. The problem is that the current GROUP BY clause can only do scalar value comparison by an accompanying HAVING clause. For instance, the SUM/ COUNT/ AVG/ MAX function will produce a single numeric value which is compared to a constant or another single aggregate value. A set-level comparison such as the set of skills subsuming {'Java programming', 'Web services'} cannot be accommodated without complex SQL queries. (Section 3.1 demonstrates such SQL queries.)

The data modeling literature has long recognized that set-valued attributes provide a concise and natural way to model complex data concept such as set [13, 16]. Many DBMSs (e.g., *nested table* in Oracle and SET data type in MySQL) nowadays support the definition of attributes involving a set of values. For example, the "skill" attribute in the resume table can be defined as a set data type. Set operations can thus be natively supported on such attributes. Previous research has extensively studied query processing on set-valued attributes and set containment joins [4, 14, 8, 7, 9, 17]. A set containment join is a join operation between the set-valued attributes of two relations, where the join condition is specified by set containment relationship. Although set-valued attributes together with set containment joins can be used to support set-level comparisons, this solution has inherent limitations: 1) They have to be pre-defined at the schema definition time; 2) A set can only cover one attribute and cannot be across multiple attributes.

In real application scenarios, especially OLAP and decision making, the groups and the corresponding sets are often dynamically formed according to the query needs. For instance, in the aforementioned customer service rating example, the sets are formed by departments, i.e., the monthly ratings of each department form a set. In a different query, the sets may be formed by individual employees. Set-valued attributes, being pre-defined, cannot support such dynamic set formation.

Moreover, the definition of a set may be over multiple attributes. Take the example of online advertisement. The strategist may want to discover the Websites that publish ads for ING with good profit returns, because those Websites are already proven to be highly profitable with bank ads. In this case, the set is defined over both the advertiser and the profit attribute. In many systems a set-valued attribute is only defined on a single attribute, thus cannot capture the cross-attribute association in the example. Implementations such as nested table in Oracle allow sets over multiple attributes but do not easily support set-level comparisons on such attributes.

Observing the demand for complex and dynamic set-level comparisons in databases, and the limitations of the current SQL lan-

*Work performed while the author was a student at UTA.

guage and set-leveled attribute, we propose to tackle the challenge with the introduction of a new concept of *set predicate* in the HAVING clause. While we elaborate its syntax in detail in Section 2, below we present several examples of queries with set predicates.

To find the candidates with skills "Java programming" and "Web services", the query can be as follow:

```
SELECT id    FROM Resume_Skills    GROUP BY id
HAVING SET(skill) CONTAIN {'Java','Web services'}
```

Given the above query, after the grouping, a dynamic set of values on the attribute skill is formed for each unique id, and the groups whose corresponding SET(skill) contain both "Java programming" and "Web services" are returned as query answers.

For the decision making example, suppose we have a table Ratings(department, avg_rating, month, year). The following query finds the departments whose monthly average ratings in 2009 have always been poor (assuming the rating is from 1 to 5):

```
SELECT department  FROM Ratings  WHERE year = 2009
GROUP BY department
HAVING SET(avg_rating) CONTAINED BY {1,2}
```

Here CONTAINED BY is used to capture the set-level condition.

Set predicates can cover multiple attributes. Consider the online advertisement example, suppose the table is Site_Statistics(website, advertiser, CTR). To find the Websites that publish ads for ING with more than 1% click-through rate (CTR) and do not publish ads for HSBC yet, the query can be:

```
SELECT website   FROM Site_Statistics
GROUP BY website
HAVING SET(advertiser,CTR) CONTAIN {('ING',(0.01,+))}
AND    NOT (SET(advertiser) CONTAIN {'HSBC'})
```

In this example, the first set predicate involves two attributes and the second set predicate has the negation of CONTAIN. Note that we use `(0.01,+)` to represent the partial condition CTR>0.01.

Without the explicit notion of set predicates, the query semantics can be captured by using sub-queries connected by SQL set operations (UNION, INTERSECT, EXCEPT), in coordination with join and GROUP BY. As Section 3.1 will show, such queries can be quite complex for users to formulate and difficult to optimize for a database engine, thus can result in unnecessarily costly evaluation. On the contrary, the proposed construct of set predicate explicitly enables set-level comparisons. The concise syntax not only makes query formulation simple but also facilitates the native efficient support of such queries in a query engine.

Our work is mostly related to set-valued attributes and set containment joins [4, 14, 8, 7, 9, 17]. In comparison, the proposed solution of set predicates has several critical advantages: (1) Unlike set-valued attributes, which bring significant hassles in redesigning the database storage for the special data type of set, set predicate requires no change in data representation and storage engine, therefore can be directly incorporated into any standard relational databases; (2) Using set predicates, users can dynamically form set-level comparisons based on their query needs with no limitation caused by the database schema. On the contrary, in set-valued attributes, sets are statically pre-defined in the schema design phase. Therefore set-level comparisons can only be possible on set-valued attributes, and non-set query conditions could not be easily supported on set-valued attributes. (3) It allows cross-attribute set-level comparison, which is not supported by set-valued attributes.

We developed three approaches to implement set predicates: (1) The query rewriting approach rewrites a SQL query with set predicates into standard SQL statements. It can be directly deployed by extending the query parser, without changing the query processing engine. However, it can be hard to optimize the complex rewritten queries and thus the query performance can suffer. The resulting query plans could involve multiple subqueries with grouping and set operations. (2) The aggregate function-based approach handles a set predicate like a user-defined aggregate function [3]. The deployment again would be simple. Instead of the wasteful evaluation of subqueries, only one pass of table scan could be sufficient. (3) The bitmap index-based approach evaluates queries with set predicates by exploiting bitmap index, which is known for its efficient support aggregate queries in OLAP and data warehousing. This approach can significantly outperform the aggregate function-based approach because it focuses on only tuples from the groups that satisfy the conditions and only the bitmaps for relevant columns.

In summary, this paper makes the following contributions:

- We proposed to extend SQL with the new concept of set predicate for expressing an important class of analytical queries, which otherwise would be difficult to write and optimize (Section 2).
- We developed three query evaluation approaches for set predicates. The simple query rewriting approach and aggregate function-based approach form the baseline (Section 3). The bitmap index-based approach leverages the advantages of bitmap indexes in efficient support of aggregation and analytical queries (Section 4).
- We conducted extensive experiments to evaluate and compare the three approaches, over both benchmark and synthetic data. We also investigated the effects of various query parameters and data characteristics on these approaches (Section 5).

2. THE SYNTAX OF SET PREDICATES

We extend the SQL syntax to support set predicates. Since a set predicate compares a group of tuples to a set of values, it fits well into the GROUP BY and HAVING clauses. Specifically in the HAVING clause there is a Boolean expression over multiple regular aggregate predicates and set predicates, connected by logical operators ANDs, ORs, and NOTs. The syntax of a set predicate is:

set_predicate ::= **attribute_set set_operator constant_values**
attribute_set ::= SET(v_1, ..., v_m)
set_operator ::= CONTAIN | CONTAINED BY | EQUAL
constant_values ::= $\{(v_1^1, ..., v_m^1), ..., (v_1^n, ..., v_m^n)\}$,

where $v_i^j \in Dom(v_i)$, *i.e.*, each v_i^j can be a value (integer, floating point number, string, etc.) in the domain of attribute v_i. Succinctly we denote a set predicate by $(v_1, ..., v_m)$ **op** $\{(v_1^1, ..., v_m^1), ..., (v_1^n, ..., v_m^n)\}$, where **op** can be $\supseteq$, $\subseteq$, and $=$, corresponding to the set operator CONTAIN, CONTAINED BY, and EQUAL, respectively.

We further extend relational algebra to include set predicates for concise representation of queries. Given relation R, the grouping and aggregation can be represented by the following operator [1]:

$$\gamma_{\mathcal{G},\mathcal{A}}\mathcal{C}(R),$$

where $\mathcal{G}$ is a set of grouping attributes, $\mathcal{A}$ is a set of aggregates (e.g., COUNT(*)), and $\mathcal{C}$ is a Boolean expression over set predicates and conditions on aggregates (e.g., AVG($grade$)>3), which may overlap with the aggregates in $\mathcal{A}$. Note that the relation R can be the result of an arbitrary query.

We now provide examples of SQL queries with set predicates, over the classic student-course table (Figure 1).

Single-Predicate Set Operation: The following query Q1: $\gamma_{student}$ $course \supseteq$ {'CS101', 'CS102'} (SC)[2] identifies the students who have taken both CS101 and CS102. The results are Mary and John. The keyword CONTAIN represents a superset relationship, *i.e.*, the set variable SET($course$) is a superset of {'CS101', 'CS102'}.

```
Q1: SELECT student   FROM SC   GROUP BY student
    HAVING SET(course) CONTAIN {'CS101', 'CS102'}
```

[1]Note that there is no agreed-upon notation for grouping and aggregation in relational algebra.

[2]To be rigorous, it should be ($course$) $\supseteq$ {('CS101'), ('CS102')}, based on the aforementioned syntax.

Table: SC

student	course	grade
Mary	CS101	4
Mary	CS102	2
Tom	CS102	4
Tom	CS103	3
John	CS101	4
John	CS102	4
John	CS103	3

Figure 1: A classic student and course example.

A more general query can include the regular WHERE clause and conditions on aggregate functions in HAVING. In the following Q2: $\gamma_{student,COUNT(*)}$ $course \supseteq$ {'CS101', 'CS102'} && AVG($grade$) > 3.5 ($\sigma_{sem='Fall09'}$(SC)), we look for students with average grade higher than 3.5 in FALL09 and have taken both CS101 and CS102 in that semester. (Assuming there is a column sem in the table SC.) It also returns the number of courses they took in that semester.

```
Q2: SELECT student, COUNT(*)    FROM SC
    WHERE  sem = 'Fall09'       GROUP BY student
    HAVING SET(course) CONTAIN {'CS101', 'CS102'}
    AND    AVG(grade)>3.5
```

We use CONTAINED BY for the reverse of CONTAIN, *i.e.*, the subset relationship. The following query selects all the students whose grades are never below 3. The results are Tom and John. Note that set predicates follow set semantics instead of bag semantics, therefore John's grades, {4,4,3}, is subsumed by {4,3}.

```
Q3: SELECT student   FROM SC   GROUP BY student
    HAVING SET(grade) CONTAINED BY {4, 3}
```

To select all the students that have and only have taken CS101 and CS102, the following query uses EQUAL to represent the equal relationship in set theory. Its result contains only Mary.

```
Q4: SELECT student   FROM SC   GROUP BY student
    HAVING SET(course) EQUAL {'CS101', 'CS102'}
```

Multi-Predicate Set Operation: A query with multiple set predicates can be supported by using Boolean operators, *i.e.*, AND, OR, and NOT. For instance, to identify all the students who did not take both CS101 and CS102 and whose grades are never below 3, we can use `NOT (SET(course) CONTAIN {'CS101', 'CS102'}) AND SET(grade) CONTAINED BY {4,3}`.

Multi-Attribute Set Operation: The query syntax also allows cross-attribute comparisons, e.g., we use `SET(course,grade) CONTAIN {('CS101',4),('CS102',2)}` for all the students who received grade 4 in CS101 but 2 in CS102.

Constant Values from Subqueries and in Ranges: The constant values in the set predicate can be from the result of a subquery, e.g., `SET(course) CONTAIN (SELECT course FROM basic_course)`. Moreover, for data types such as numeric attributes and dates, the constant value can be a range. One example is the online advertisement query in Section 1.

3. TWO BASELINE APPROACHES FOR EVALUATING SET PREDICATES

3.1 Query Rewriting Approach

One easy-to-deploy approach is to rewrite queries with set predicates into standard SQL queries. The rewriting is rather simple, thus we give the details in Appendix A and only provide a summary here. The basic idea is to rewrite a set predicate by using SQL set operations. A CONTAIN predicate with m constant values can be rewritten using m-1 INTERSECT operations. A CONTAINED

```
SELECT  sum(a)
FROM  R,
 ( SELECT   g
   FROM     R
   EXCEPT
   ( SELECT  g
     FROM    R
     WHERE   v <> 1
     AND     v <> 2
     AND     v <> 3)
 ) AS TMP
WHERE  R.g = TMP.g
GROUP BY  R.g
```

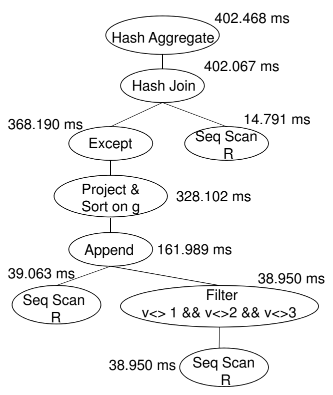


Figure 2: *Rewrt* Query Plan for CONTAINED BY, 100K tuples, 1K groups, 10 satisfying groups.

BY predicate can be rewritten by using EXCEPT. The rewriting of EQUAL predicates naturally combines that of CONTAIN and CONTAINED BY. To rewrite more complex queries with multiple predicates, we use the rewriting of individual predicates as the building blocks and connect them together by their logical relationships. If the SELECT clause in the query contains aggregate values, the rewritten query needs to be joined with the original table on the grouping attributes.

The advantage of the rewriting approach is that no change is necessary in the query processing engine. However, the rewritten query can be complex thus difficult to optimize for a database engine, resulting in over-costly evaluation plan. While more details on our experiments with this approach are in Section 5, below we give an example to just get a sense of the issues.

Figure 2 illustrates a PostgreSQL query plan for the rewritten query of $\gamma_{g,SUM(a)}$ $v \subseteq$ {1,2,3}(R) over a 100K-tuple table R with 1K groups on g, resulting in 10 qualifying groups. The plan is hand-picked and the most efficient among the ones we have investigated. It also shows the time spent on each operator (which recursively includes the time spent on all the operators in the sub-plan tree rooted at the given operator, due to the effect of the iterators' GetNext() interfaces.) The real PostgreSQL plan had more detailed operators. We combine them and give the combined operators more intuitive names, for the ease of presentation.

Figure 2 shows, given the simple original query, the rewritten query involves a set difference operation (Except) and a join. The set difference is between R itself (100K tuples) and a subset of R (98998 tuples that do not have 1, 2, or 3 on attribute v). Both sets are large, making the Except operator cost much more than a simple sequential scan.

3.2 Aggregate Function-Based Approach

The example at the end of Section 3.1 indicates that the rewritten query obscures the semantics of set predicate, making it very difficult for the query optimizer to optimize. With the new syntax in Section 2 bringing forward the semantics explicitly, a set predicate-aware query plan could potentially be much more efficient by just scanning the table and processing the tuples sequentially. The key to such a direct approach seems to be processing the grouping and the set-level comparison together, through a one-pass iteration of the tuples, instead of using set operations and joins. In light of this idea, we design another baseline approach, the aggregate function-based method, that handles set predicates as user-defined aggregate functions [3]. The outline of the method is shown in Algorithm 1.

For the ease of presentation, we first assume the simplest query, $\gamma_{g,\oplus a}$ v ***op*** $\{v^1, ..., v^n\}(R)$, i.e., a query with one grouping attribute (g), one aggregate for output ($\oplus a$), and one set predicate defined by

Algorithm 1 Aggregate Function-Based Approach

```
Input:  Table $R(g, a, v)$, Query $Q = \gamma_{g,\oplus a}\ v$ op $\{v^1, ..., v^n\}(R)$
Output:  Qualifying groups $g$ and their aggregate values $\oplus a$
    /* $g$: grouping attribute; $a$: aggregate attribute; $v$: attribute
    in set predicate; $A$: hash table for aggregate values; $M$: hash
    table for constant value masks; $G$: hash table for Boolean in-
    dicators of qualifying groups */
 1: while r($g$,$a$,$v$)$\Leftarrow$GetNext( ) != End of Table do
 2:     if group $g$ is not in hash table $A$,$M$,$G$ then
 3:         Initialize $A[g]$; $M[g] \Leftarrow 0$; $G[g] \Leftarrow (op \in \{\subseteq,=\})$
 4:     /* Aggregate the value $a$ for group $g$. */
 5:     $A[g] \Leftarrow A[g] \oplus a$
 6:     /* In $M[g]$, set the mask for the $j$th constant value. */
 7:     if ($M[g]$ != $2^n-1$)&&((op is $\supseteq$)||((op is $=$)&&$G[g]$)) then
 8:         if $v$ == $v^j$ for some $j$ then
 9:             $M[g] \Leftarrow M[g] \mid 2^{j-1}$
10:     /* For $\subseteq$, $=$, if $v \notin \{v^1, ..., v^n\}$, $g$ does not qualify. */
11:     if $G[g]$ && (op $\in \{\subseteq,=\}$) && ($v \neq v^j$ for any $j$) then
12:         $G[g] \Leftarrow False$
13: /* Output qualifying groups and their aggregates. */
14: for every group $g$ in hash table $M$ do
15:     if (op is $=$) && ($M[g]$ != $2^n - 1$) then $G[g] \Leftarrow False$
16:     if $G[g]$ then print ($g$, $A[g]$)
```

a set operator ($\supseteq$, $\subseteq$, or $=$) over a single attribute (v). Algorithm 1 uses the standard iterator interface GetNext() to go thorough the tuples in R, may it be from a sequential scan over table R or the sub-plan over sub-query R. Follow the common implementation of user-defined aggregate functions in database systems, e.g., PostgreSQL, a set predicate is treated as an aggregate function, defined by an initial state (line 3), a state transition function (line 4-12), and a final calculation function (line 15-16). The hash table M maintains a mask value for each unique group. The bits in the binary representation of a mask value indicate which of the constant values v^1, ..., v^n in the set predicate are contained in the corresponding group. (The $|$ in line 9 is the bitwise OR operator instead of logical OR $||$.) If the mask value equals $2^n - 1$, where n is the number of the constant values, the corresponding group contains all the constant values. The algorithm sketch covers the three kinds of set operators, CONTAIN, CONTAINED BY, and EQUAL.

The above algorithm can be conveniently extended for general queries with set predicates. Such queries can be denoted by $\gamma_{\mathcal{G},\mathcal{A}}\mathcal{C}(R)$, as introduced in Section 2. The aggregates that are in $\mathcal{A}$ (to appear in the SELECT clause) or in the conditions in $\mathcal{C}$ (to appear in HAVING) can be evaluated simultaneously with set predicates. The aggregate $\oplus a$ in Algorithm 1 already demonstrates that. [3]

We have assumed that one-pass algorithm is sufficient, i.e., memory is available for storing the hash tables for all the groups. If the required memory size is too large to be affordable, we can adopt the standard two-pass hash-based or sort-based aggregation method in DBMSs. The table is sorted or partitioned by a hash function so that the tuples in the same group can be loaded into memory and aggregations over different groups can be handled independently.

4. BITMAP INDEX-BASED APPRAOCH

Our third approach is based on bitmap index [10, 11, 12]. Given a bitmap index on an attribute, there exists a bitmap (a vector of bits) for each unique attribute value. The length of the vector equals the number of tuples in the indexed relation. In the vector for value x of attribute v, its ith bit is set to 1, when and only when the value of v on the ith tuple is x, otherwise 0. With bitmap indices, complex selection queries can be efficiently answered by bitwise operations (AND (&), OR(|), XOR(^), and NOT(~)) over the bit vectors. Moreover, bitmap indices enable efficient computation of aggregates such as SUM and COUNT [12].

The idea of using bitmap index is in line with the aforementioned intuition of processing set-level comparison by a one-pass iteration of the tuples (in this case, their corresponding bits in the bit vectors). On this aspect, it shares the same advantage as the aggregate function-based approach. However, this method also leverages the distinguishing characteristics of bitmap index, making it potentially more efficient than the aggregate function-based approach. Specifically, using bitmap index to process set predicates brings several advantages. (1) We only need to access the bitmap indexes on columns involved in the query, therefore the query performance is independent of the width of the underlying table. (2) The data structure of bit vector is efficient for basic operations such as membership checking (for matching constant values in set predicates). Bitmap index gives us the ability to skip tuples when they are irrelevant. Given a bit vector, if there are chunks of 0s, they can be skipped together due to effective encoding of the vector. (3) The simple data format and bitmap operations make it convenient to integrate the various operations in a query, including the dynamic grouping of tuples and the set-level comparisons of values. It also enables efficient and seamless integration with conventional operations such as selections, joins, and aggregates. (4) It allows straightforward extensions to handle features that could be complicated, such as multi-attribute set predicates, multi-attribute grouping, and multiple set predicates.

As an efficient index for decision support queries, bitmap index has gained broad interests. The state-of-the-art developments of bitmap compression methods [22, 1, 5] and encoding strategies [2, 24, 12] have substantially broaden the applicability of bitmap index. Today's bitmap index can be applied on all types of attributes (e.g., high-cardinality categorical attributes [21, 22], numeric attributes [21, 12] and text attributes [18, 15]). Studies show that compressed bitmap index occupies less space than the raw data [22, 1] and provides better query performance for equality query [22], range query [21], aggregation [12], and keyword query [18]. Nowadays, bitmap index is supported in most commercial database systems (e.g, Oracle, DB2, SQL Server), and its often the default (or only) index option in column-oriented database systems (e.g., Vertica, C-Store [19], LucidDB), especially for applications with read-mostly or append-only data, such as OLAP and data warehouses.

One disadvantage of this approach is the requirement of bitmap index which, while widely adopted in DBMSs, arguably is not as standard as some other index structures such as B+ tree. We argue that with the effective encoding schemes and small size, it is affordable to build bitmap indexes on many attributes. Index selection based on query workload could allow a system to selectively create indexes on attributes that are more likely to be used in queries. These issues warrant further investigation.

4.1 Basic Set Predicate Queries

In presenting the bitmap index-based approach, we first focus on the simplest case– a query with only one set predicate, $\gamma_{g,\oplus a}\ v$ ***op*** $\{v^1, ..., v^n\}(R)$, where ***op*** can be any of the set operators $\supseteq$, $\subseteq$, and $=$. Section 4.2 discusses the extension to more general queries.

One particular type of bitmap index that we will use is *bit-sliced index* (BSI) [15]. Given a numeric attribute on integers or floating-point numbers, BSI directly captures the binary representations of attribute values. To be more specific, the tuples' values on an at-

[3]Note that line 5 of Algorithm 1 only shows the state transition of $\oplus$. The initialization and final calculation steps are omitted.

Algorithm 2 Basic Set Operation ($\supseteq$, $\subseteq$, $=$)

```
Input:  Table $R(g, a, v)$ with $t$ tuples;
        Query $Q = \gamma_{g,\oplus a}$ $v$ **op** $\{v^1, ..., v^n\}(R)$;
        bit-sliced index BSI($g$), BSI($a$), and bitmap index BI($v$).
Output: Qualifying groups $g$ and their aggregate values $\oplus a$
    /* $gID$: array of size $t$, storing the group ID of each tuple */
    /* $A$: hash table for aggregate values */
    /* $M$: hash table for constant value masks */
    /* $G$: hash table for Boolean indicators of qualifying groups */
    /* Step 1. get the vector for each constant in the predicate */
 1: for each $v^j$ do
 2:     $vec_{v^j} \Leftarrow$ QUERYBI (BI($v$), $v^j$)
    /* Step 2. get the group ID for each tuple */
 3: Initialize $gID$ to all zero
 4: for each bit slice $B_i$ in BSI($g$), $i$ from 0 to $s$-1 do
 5:     for each set bit $b_k$ in bit vector $B_i$ do
 6:         $gID[k] \Leftarrow gID[k] + 2^{i-1}$
 7: for each $k$ from 0 to $t$-1 do
 8:     if group $gID[k]$ is not in hash table $A$,$M$,$G$ then
 9:         Initialize $A[gID[k]]$;$M[gID[k]] \Leftarrow 0$;
10:         $G[gID[k]] \Leftarrow$ (op is $\subseteq$)
    /* Step 3. find qualifying groups */
11: if op $\in \{\supseteq, =\}$ then
12:     for each bit vector $vec_{v^j}$ do
13:         for each set bit $b_k$ in $vec_{v^j}$ do
14:             $M[gID[k]] \Leftarrow M[gID[k]] \mid 2^{j-1}$
15:     for each group $g$ in hash table $M$ do
16:         if $M[g] == 2^n - 1$ then $G[g] \Leftarrow True$
17: if op $\in \{\subseteq, =\}$ then
18:     for each set bit $b_k$ in $\sim(vec_{v^1} \mid ... \mid vec_{v^n})$ do
19:         $G[gID[k]] \Leftarrow False$
    /* Step 4. aggregate the values of $a$ for qualifying groups */
20: for each $k$ from 0 to $t$-1 do
21:     if $G[gID[k]]$ then
22:         $agg \Leftarrow 0$
23:         for each slice $B_i$ in BSI($a$) do
24:             if $b_k$ is set in bit vector $B_i$ then $agg \Leftarrow agg + 2^{i-1}$
25:         $A[gID[k]] \Leftarrow A[gID[k]] \oplus agg$
26: for every group $g$ in hash table $M$ do
27:     if $G[g]$ then print ($g$, $A[g]$)
```

student		course			grade		
B_1	B_0	B_{CS101}	B_{CS102}	B_{CS103}	B_2	B_1	B_0
0	0	1	0	0	1	0	0
0	0	0	1	0	0	1	0
0	1	0	1	0	1	0	0
0	1	0	0	1	0	1	1
1	0	1	0	0	1	0	0
1	0	0	1	0	1	0	0
1	0	0	0	1	0	1	1

Mapping from values in *course* to numbers:
Mary => 0; Tom => 1; John => 2

Figure 3: Bitmap indexes for the data in Figure 1.

tributes are represented in binary format and kept in s bit vectors (i.e., *slices*), which represent 2^s different values. Categorial attributes can be also indexed by BSI, with a mapping from unique categorical values to integer numbers.

The approach requires that bit-sliced indexes are built on g (BSI(g)) and a (BSI(a)), respectively, and there is a bitmap index on v (BI(v)), which can be a BSI or any regular bitmap index. Note that the algorithm presented below will also work if we have other types of bitmap indexes on g and v, with modifications that we omit. The advantage of BSI is that it indexes high-cardinality attributes with small number of bit vectors, thus improves the query performance if the grouping or aggregation is on such high-cardinality attribute.

Example 1: Given the data in Figure 1 and query $\gamma_{student,AVG(grade)}$ *course* ***op*** {'CS101','CS102'}(SC), Figure 3 shows the bitmap indexes on g (*student*), v(*course*), and a(*grade*). BSI(*student*) has two slices. For instance, the fourth bits in B_1 and B_0 of BSI(*student*) are 0 and 1, respectively. Thus the fourth tuple has binary value 1 on attribute *student*, which represents 'Tom' according to the mapping from the original values to numbers. There is also a BSI(*grade*) on *grade*. The bitmap index on *course* is not a BSI, but a regular one where each unique attribute value has a corresponding bit vector. For instance, the bit vector B_{CS101} is 1000100, indicating that the 1st and the 5th tuples have 'CS101' as the value of *course*. ■

The outline of the algorithm for processing the query is in Algorithm 2. It takes four steps. Step 1 is to get the tuples having values v^1, ..., v^n on attribute v. Given value v^j, the function $QueryBI$ in line 2 queries the bitmap index on v, BI(v), and obtains a bit vector vec_{v^j}, where the kth bit is set (i.e., having value 1) if the kth tuple of R has value v^j on attribute v. Note that this is the basic functionality supported by every bitmap index.

Step 2 is to get the group IDs, i.e., the values of attribute g, for the tuples in R, by querying BSI(g). The group IDs are calculated by iterating through the slices of BSI(g) and summing up the corresponding values for tuples with bits set in these vectors.

Step 3 gets the groups that satisfy the set predicate, based on the vectors from Step 1. The algorithm outline is applicable to any of the three set operators, although the details differ in Step 3. Therefore below we will introduce Step 3 for different operators.

Step 4 gets the aggregates for the qualifying groups from Step 3 by using BSI(a). It is pretty similar to how Step 2 and Step 3 are accomplished. It aggregates the value of attribute a from each tuple into its corresponding group if it is a qualifying group. The value of attribute a for the kth tuple is obtained by assembling the values 2^{i-1} from the slices B_i when their kth bits are set. The algorithm outline checks all the bit slices for each k. Note that this can be efficiently implemented by iterating through the bits (k from 0 to $t-1$) of the slices simultaneously. We do not show such implementation details in the algorithm outline, for simplicity.

CONTAIN ($\supseteq$): In Step 3, a hash table M maintains a mask value $M[g]$ for each group g. The mask value has n bits, corresponding to the n constant values (v^1, ..., v^n) in the set predicate. A bit is set to 1 if there exists at least one tuple in group g that has the corresponding constant value on attribute v. Therefore for each set bit b_k in vector vec_{v^j}, the jth bit of $M[gID[k]]$ will be set (line 14 of Algorithm 2). (The | in line 14 is the bitwise OR operator instead of logical OR ||.) Therefore if $M[g]$ equals $2^n - 1$ at the end, i.e., all the n bits are set, the group g contains tuples matching every constant value in the set predicate, thus satisfies the query (line 16). We use another hash table G to record the Boolean indicators of qualifying groups. The values in G were initialized to $True$ for every group if the set operator is $\subseteq$ (line 10), otherwise $False$.

CONTAINED BY ($\subseteq$): If the set operator is $\subseteq$, Step 3 will not use hash table M. Instead, for a set bit b_k in $\sim(vec_{v^1} \mid ... \mid vec_{v^n})$ (i.e., the kth tuple does not match any constant v^j), the corresponding group $gID[k]$ could not satisfy the query condition and thus will immediately be disqualified (line 19).

EQUAL ($=$): The Step 3 for EQUAL ($=$) is naturally a combination of that for $\supseteq$ and $\subseteq$, since $X{=}Y$ means $X{\supseteq}Y$ and $X{\subseteq}Y$. It first marks a group as qualifying if $\supseteq$ is satisfied (line 16), then disqualifies group $gID[k]$ if $\subseteq$ is satisfied (line 19).

Example 2: Suppose the query is $\gamma_{student,AVG(grade)}$ $course = \{$'CS101','CS102'$\}(SC)$, using the bitmap indexes in Figure 3. After the 1st bit of vec_{CS101} (i.e., v^1='CS101') is encountered in line 13, M[0]=2^0=1 since the 1st tuple in SC belongs to group 0 (Mary). After the 2nd bit of vec_{CS102} (the 1st set bit) is encountered, M[0]=1|2^1=3. Therefore G[0] becomes $True$. Similarly G[2] (for John) becomes $True$ after the 6th bit of vec_{CS102} is encountered. However, since $\sim$(vec_{CS101} | vec_{CS102}) is 0001001, G[2] becomes $False$ after the last bit of 0001001 is encountered in line 18 (i.e., John has an extra course 'CS103'). ■

4.2 General Queries with Set Predicates

A general query $\gamma_{\mathcal{G},\mathcal{A}}\mathcal{C}(R)$ may use multiple grouping attributes and multiple set predicates, and each predicate may be defined on multiple attributes. Below we discuss how to extend the method in Section 4.1 for such general queries. A query may also return multiple aggregates, e.g., $\gamma_{g,\oplus_1 a,\oplus_2 b}$ v ***op*** $\{v^1, ..., v^n\}(R)$, which can be simply handled by repeating the Step 4 of Algorithm 2 for multiple aggregates. Thus such case will not be further discussed.

Multi-Attribute Grouping: Given a query with multiple grouping attributes, $\gamma_{g_1,...g_l,\mathcal{A}}$ $\mathcal{C}(R)$, we can treat the grouping attributes as a single combined attribute g. That is, the concatenation of the bit slices of BSI(g_1), ..., BSI(g_l) becomes the bit slices of BSI(g). For example, given Figure 3, if the grouping condition is `GROUP BY student,grade`, the BSI of the conceptual combined attribute g has 5 slices, which are B_1(*student*), B_0(*student*), B_2(*grade*), B_1(*grade*), and B_0(*grade*). Thus the binary value of the combined group g of the first tuple is 00100.

Multi-Attribute Set Predicates: For a query with multiple attributes in defining the set predicate, $\gamma_{\mathcal{G},\mathcal{A}}$ $(v_1, ..., v_m)$ ***op*** $\{(v_1^1, ..., v_m^1), ..., (v_1^n, ..., v_m^n)\}(R)$, we again can view the concatenation of v_1, ..., v_m as a combined attribute v. To replace the Step 1 of Algorithm 2, we first obtain the vectors $vec_{v_1^j}$, ..., $vec_{v_m^j}$ by querying BI(v_1), ..., BI(v_m). Then the intersection (bitwise AND) of these vectors, $vec_{v^j} = vec_{v_1^j}$ & ... & $vec_{v_m^j}$, gives us the tuples that match the multi-attribute constant $(v_1^j, ..., v_m^j)$.

Multi-Predicate Set Operation: Given a query with multiple set predicates, the straightforward approach is to evaluate the individual predicates independently to obtain the qualifying groups for each predicate. Then we can follow the logic operations between the predicates (AND, OR, NOT) to perform the intersection, union, and difference operations over the qualifying groups. Different orders of evaluating the multiple predicates can potentially lead to different query performance, due to their different "selectivity" (i.e., the number of qualifying groups). We leave related query optimization issues to future study.

In a general query, the HAVING clause may also have conditions over regular aggregate expressions, e.g., AVG($grade$)>3.5 in Q2. Following the same way as the aggregates in SELECT clause are calculated, in Step 4 of Algorithm 2, we can obtain the multiple aggregates for each group and remove a group from the result if the condition over an aggregate is not satisfied.

Integration and Interaction with Conventional SQL Operations: In a general query $\gamma_{\mathcal{G},\mathcal{A}}\mathcal{C}(R)$, the relation R could be the result of other operations such as selections and joins. Logical bit vector operations easily allow us to integrate the bitmap index-based method for set predicates with the bitmap index-based solutions for Boolean selection conditions [2, 24, 12, 23] and join queries [11]. The details are in Appendix B. Note that the techniques applied here are similar to those in [6], which uses bitmap index to incorporate clustering with ranking, selections, and joins.

In Section 5.2 we present the experimental results over such queries on the TPC-H benchmark database [20].

parameter	meaning	values
O	set operators	$\supseteq, \subseteq, =$
C	number of constant values in set predicate	1, 2, ..., 10, 20, ..., 100
T	number of tuples	10K,100K, 1M
G	number of groups	10,100,...,T
S	number of qualifying groups	1,10,...,G

Table 1: Configuration Parameters.

5. EXPERIMENTS

5.1 Experimental Settings

We conducted experiments to compare the three methods discussed in previous sections under various query scenarios. Moreover, we investigated how the methods are affected by important factors with a variety of configurations. The experiments were conducted on a Dell PowerEdge 2900 III server running Linux kernel 2.6.27, with dual quad-core Xeon 2.0 GHz processors, 2x4MB cache, 8GB RAM, and three 146GB 10K-RPM SCSI hard drivers in RAID5. The reported results are the averages of 10 runs.

Methods: The compared methods are:

Rewrt: We translated the queries by the approach in Section 3.1. We used PostgreSQL 8.3.7 to store the data and execute the translated queries. We set the memory buffer size of *PostgreSQL* to 24MB, which is sufficient for *Agg* and *Bitmap* to hold the internal data structures. PostgreSQL provides several table scan and join methods. We manually investigated the alternative plans, by turning on and off various physical query operators, and made our best attempt to obtain the most efficient plan for each query. Below we only report the numbers obtained by using these hand-picked plans.

Agg: This is the aggregation function-based method in Section 3.2. We did not experiment with the scenario when the number of groups exceeds what the memory can accommodate, therefore it was unnecessary to use two-pass hashing-based or sorting-based methods for aggregation. It is implemented in C++.

Bitmap: This is the bitmap index-based method in Section 4. The algorithms are implemented in C++. The bit-sliced index implementation that we used is FastBit. [4] Its compression scheme, Word-Aligned Hybrid (WAH) code, makes the compressed bitmap indices efficient even for high-cardinality attributes [22].

While *Rewrt* uses a full-fledged database engine PostgreSQL, both *Agg* and *Bitmap* are implemented externally. We acknowledge that *Rewrt* would incur extra overhead from the query optimizer, tuple formatting, and so on. Our expectation is to show the clear disadvantages of *Rewrt* that very unlikely come from the extra overheads. (At least one order of magnitude slower than *Bitmap*, as verified below.) The results could encourage the database vendors to incorporate the proposed *Bitmap* method into a database engine.

Queries: We evaluated the aforementioned methods under various combinations of configuration parameters, which are summarized in Table 1. O can be one of the 3 set operators ($\supseteq$, $\subseteq$, $=$), and C is the number of constant values in a set predicate, varying from 1 to 10 and then 10 to 100. Therefore we have totally 3×19 combinations of (O, C) pairs. Each pair thus corresponds to a unique query with a single set predicate. The constant values always start from 1, increment by 1. Therefore $(\supseteq, 2)$ corresponds to Q=$\gamma_{g,SUM(a)}v \supseteq \{1, 2\}(R)$. Note that we assume SUM as the aggregate function since its evaluation is not our focus and Algorithm 1 and 2 process all the aggregate functions in the same way.

Data Tables: For each of the 3×19 single-predicate queries, we generated 61 data tables, each corresponding to a different com-

[4]https://sdm.lbl.gov/fastbit.

Query	*Rewrt*+Join	*Agg*+Join	*Bitmap*
TPCH-1	10.87+43.50 sec.	2.65+43.50 sec.	0.83 sec.
TPCH-2	4.33+3.22 sec.	0.77+3.22 sec.	0.18 sec.

Table 2: Results on TPC-H Data.

bination of (T, G, S) values. Given query (O, C) and data statistics (T, G, S), we generated the corresponding table that satisfies the statistics for the query. The table has schema $R(a, v, g)$, for query $\gamma_{g,SUM(a)} v\ O\ \{1, ..., C\}(R)$. Each column is a 4-byte integer. The details of how the column values were generated are in Appendix C.

5.2 Experimental Results

(A) Comparing the three methods:
We measure the wall-clock execution times of the three methods, *Rewrt*, *Agg*, *Bitmap*, over the 61 configurations (data tables) for each of the 3×19 queries. The comparison of the three methods under different queries are pretty similar. Therefore we only show the result for one query in Figure 4: $\gamma_{g,SUM(a)} v \subseteq \{1, \ldots, 10\}(R)$. It demonstrates that *Bitmap* is often several times more efficient than *Agg* and is usually one order of magnitude faster than *Rewrt*.

The low efficiency of *Rewrt* is due to the awkwardness of the translated queries and the incompetence of the current database engine in processing such queries, even though the semantics being processed is fairly simple. The performance advantage of *Agg* over *Rewrt* shows that even a simple modification of the query processor could improve the efficiency significantly. The shown advantage of *Bitmap* over *Agg* is due to the fast bit-wise operations and the skipping of chunks of 0s enabled by bitmap index, compared to the verbatim comparisons used by *Agg*.

(B) Experiments on TPC-H Data:
Two of the advantages of *Bitmap* mentioned in Section 4.2 could not be demonstrated by the above experiment. First, it only needs to process the necessary columns, while *Agg* (and *Rewrt*) have to scan the full table before irrelevant columns can be projected out. The tables used in the experiments for Figure 4 have the schema $R(a, v, g)$ which does not include other columns. We can expect the costs of *Rewrt* and *Agg* to increase by the width of the table, while *Bitmap* will stay unaffected. Second, the *Bitmap* method also enables convenient and efficient integration with selections and joins, while the above experiment is on a single table.

We thus designed queries on the TPC-H benchmark database [20] and compared the performances of the three methods. Two of the test queries are described in Appendix D and their results are in Table 2. In these queries, the grouping and set predicates are defined over the join result of multiple tables. Note that the joins are star-joins between keys and foreign keys. For *Rewrt* and *Agg*, we first pre-joined the tables to generate a single joined table, and then executed the algorithms over the joined table. Therefore in Table 2 we report the costs of *Rewrt* and *Agg* together with the costs of the pre-join. Given foreign key joins, we can create bitmap join index [11]. For example, we index the tuples in table `LINEITEM` on the values of attribute `N_NAME` which is from a different table `NATION`. With such a bitmap join index, given query TPCH-1 (and similarly TPCH-2), the *Bitmap* method works in the same way as for a single table, without pre-joining the tables.

Table 2 shows that, even with the tables pre-joined for *Rewrt* and *Agg*, *Bitmap* is still 3-4 times faster than *Agg* and more than 10 times faster than *Rewrt*. If we consider the cost of join, the performance gain is even more significant.

(C) Detailed breakdown and the effect of parameters:
To better understand the performance difference between the three methods, we looked at the detailed breakdown of their execution time. We further investigated the effect of various configuration parameters. In these experiments, we measured the execution times under various groups of configurations by the value combinations of four relevant parameters, C, T, G, and S. In each group of experiments, we varied the value of one parameter and fixed the values of the remaining three. We then used all the three methods, for all the 3 operators, and studied the detailed breakdown of execution time as well as how their performances are affected as the value of the varying parameter changed. In general the trend of the curves remains fairly similar when we use different values for the three fixed parameters and vary the values of the fourth parameter. Therefore we only report the result from four representative configuration groups, varying one parameter in each group. Due to space limitation, here we only report the results for *Bitmap* in Figure 5. The detailed breakdown for *Rewrt* and *Agg* are reported in Appendix E.

Figure 5 shows the results of four configuration groups for *Bitmap*. For each group, the upper and lower figures show the execution time and its detailed breakdown. Each vertical bar represents the execution time given one particular query (O,C) and the staked components of the bar represent the percentages of the costs of all the individual steps. *Bitmap* has four major steps, as shown in Algorithm 2: Step 1– obtain the vectors for set predicate constants; Step 2– obtain the group IDs of each tuple; Step 3– find qualifying groups; Step 4– get the aggregate values for qualifying groups. The figures show that no single component dominates the query execution cost. The breakdown varies as the configuration parameters change. Therefore we shall analyze it in detail while we investigate the effect of the configuration parameters below.

Figure 5(a) shows that the execution time of *Bitmap* increases linearly with C (number of constant values in set predicate). (Note that the values of C increase by 1 initially and by 10 later, therefore the curves appear to have an abrupt change at C=10, although they are linear.) This observation can be explained by the detailed breakdown in Figure 5(a). The costs of Step 1 and Step 3 increase as C increases, thus get higher and higher percentages in the breakdown. This is because the method needs to obtain the corresponding vector for each constant and find the qualifying groups by considering all the constants. On the other hand, the costs of Step 2 and 4 do not have much to do with C, thus get decreasing percentages.

Figure 5(b) shows the execution time of *Bitmap* increases slowly with S (number of qualifying groups). With more and more groups satisfying the query condition, the costs of Step 1-3 increase only moderately since C and G do not change, while Step 4 becomes dominating because the method has to calculate the aggregates for more and more qualifying groups.

As Figure 5(c) shows, the cost of *Bitmap* does not change significantly with G (number of groups). However, the curves do show that the method is least efficient when there are very many or very few groups. When G increases, with the number of satisfying groups (S) unchanged, the number of tuples matching the constants becomes smaller, resulting in cheaper cost of bit vector operations in Step 1. The number of tuples per group decreases, thus the cost of Step 4 also decreases. These two factors make the overall cost decreasing, although the cost of Step 2 increases due to more vectors in BSI(g). When G reaches a large value such as $100,000$, the cost of Step 2 becomes dominating, making the overall cost higher again.

Figure 5(d) shows that, naturally, *Bitmap* scales linearly with the number of tuples in the data (T). One interesting observation is, when T increases, Step 1 becomes less dominating and Step 4 becomes more and more significant.

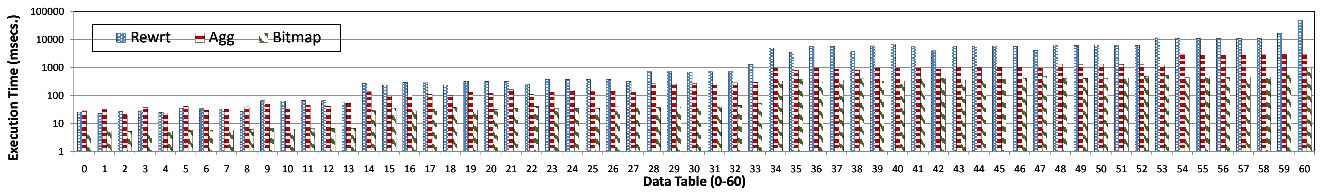
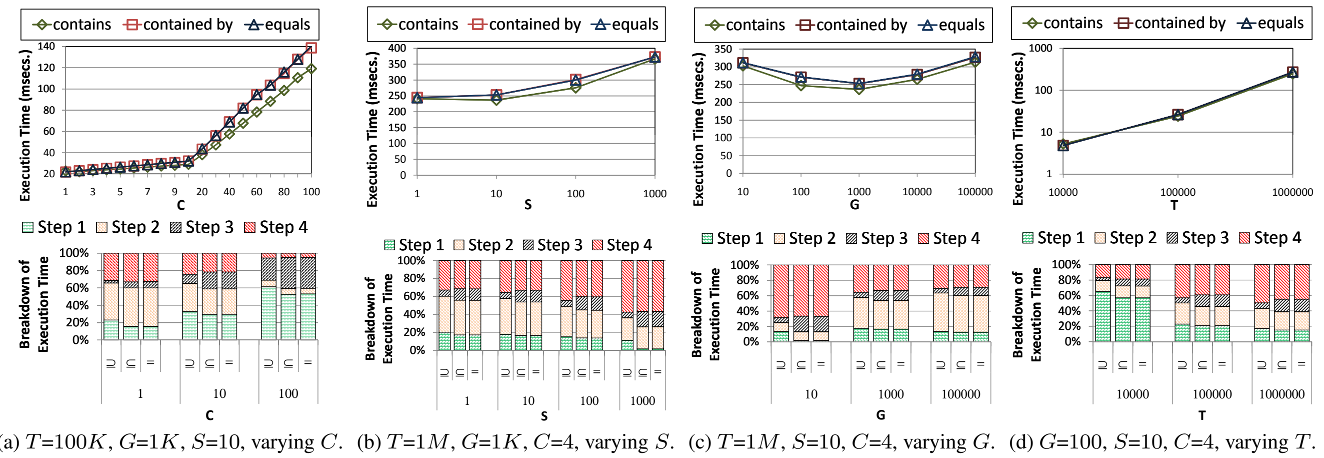


Figure 4: Overall comparison of the three methods, O=$\subseteq$, C=10.



(a) T=100K, G=1K, S=10, varying C. (b) T=1M, G=1K, C=4, varying S. (c) T=1M, S=10, C=4, varying G. (d) G=100, S=10, C=4, varying T.

Figure 5: Execution time of *Bitmap* and its breakdown.

6. CONCLUSION AND FUTURE WORK

We propose to extend SQL to support set-level comparisons by a new type of set predicates. Such predicates, combined with grouping, allow the selection of dynamically formed groups by the comparison between a group and a set of values. The semantics of set predicates, although can be expressed by SQL set operations, leads to over-complex queries that are not only challenging to write but also difficult for the database engine to optimize, resulting in costly evaluation plans. The proposed set predicate brings out the obscure semantics and thus facilitates the direct and efficient support. We presented three evaluation methods for queries with set predicates, including a query rewriting approach, an aggregate function-based approach, and a bitmap index-based approach. We conducted comprehensive experiments using synthetic data and TPC-H benchmark, to compare and investigate the three methods. The results show that the bitmap index-based approach can be several times to orders of magnitude more efficient than the alternatives.

7. REFERENCES

[1] G. Antoshenkov. Byte-aligned bitmap compression. In *Proceedings of the Conference on Data Compression*, 1995.

[2] C. Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *SIGMOD*, 1999.

[3] S. Cohen. User-defined aggregate functions: bridging theory and practice. In *SIGMOD*, pages 49–60, 2006.

[4] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *VLDB*, pages 386–395, 1996.

[5] T. Johnson. Performance measurements of compressed bitmap indices. In *VLDB*, pages 278–289, 1999.

[6] C. Li, M. Wang, L. Lim, H. Wang, and K. C.-C. Chang. Supporting ranking and clustering as generalized order-by and group-by. In *SIGMOD*, 2007.

[7] N. Mamoulis. Efficient processing of joins on setvalued attributes. In *SIGMOD Conference*, 2003.

[8] S. Melnik and H. Garcia-Molina. Divide-and-conquer algorithm for computing set containment joins. In *EDBT*, pages 427–444, 2002.

[9] S. Melnik and H. Garcia-Molina. Adaptive algorithms for set containment joins. *ACM Transactions on Database Systems*, 28(1):56–99, 2003.

[10] P. E. O'Neil. Model 204 architecture and performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, September 1987.

[11] P. E. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, 1995.

[12] P. E. O'Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD*, pages 38–49, 1997.

[13] G. Özsoyoğlu, Z. M. Özsoyoğlu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Trans. Database Syst.*, 12(4):566–592, 1987.

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

[15] D. Rinfret, P. O'Neil, and E. O'Neil. Bit-sliced index arithmetic. In *SIGMOD*, pages 47–57, 2001.

[16] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Trans. Database Syst.*, 13(4):389–417, 1988.

[17] D. Shaporenkov. Efficient main-memory algorithms for set containment join using inverted lists. In *ADBIS*, 2005.

[18] K. Stockinger, J. Cieslewicz, K. Wu, D. Rotem, and A. Shoshani. Using bitmap index for joint queries on structured and text data. In *Annals of Information Systems*, 2008.

[19] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column oriented dbms. In *VLDB*, 2005.

[20] Transaction Processing Performance Council. TPC benchmark H (decision support) standard specification. 2009.

[21] K. Wu, E. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *VLDB*, 2004.

[22] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31(1):1–38, 2006.

[23] M.-C. Wu. Query optimization for selections using bitmaps. In *SIGMOD*, pages 227–238, 1999.

[24] M.-C. Wu and A. P. Buchmann. Encoded bitmap indexing for data warehouses. In *ICDE*, pages 220–230, 1998.

APPENDIX

A. DETAILS OF THE QUERY REWRITING APPROACH

Due to space limitation, we will not present the detailed rewriting algorithm. Instead, we use examples to show what the rewriting could be for typical cases. There could be multiple ways in rewriting a query. We will not enumerate all of them.

Rewriting CONTAIN: Consider the query Q1 in Section 2, which has a CONTAIN predicate. It can be rewritten using INTERSECT, as shown in the following Q1'. In general, a CONTAIN predicate with m constant values can be rewritten using m-1 INTERSECT operations. Note that INTERSECT, UNION, and EXCEPT in SQL operate by set semantics instead of bag semantics, unless they are followed by ALL.

```
Q1': SELECT student FROM SC WHERE course = 'CS101'
     INTERSECT
     SELECT student FROM SC WHERE course = 'CS102'
```

If the SELECT clause in the query contains aggregate values, the rewritten query needs to be joined with the original table on the grouping attributes. For instance, suppose the SELECT clause in Q1 is `SELECT student,COUNT(*)`, i.e., we want to identify the qualifying students and the number of courses that they have taken, the rewritten query will be:

```
SELECT student, COUNT(*)
FROM SC,
    (SELECT student FROM SC WHERE course = 'CS101'
     INTERSECT
     SELECT student FROM SC WHERE course = 'CS102'
    ) as TMP
WHERE SC.student = TMP.student
GROUP BY student
```

Alternatively a subquery instead of join can be used to obtain the aggregate values:

```
SELECT student, COUNT(*)
FROM SC
WHERE student IN
    (SELECT student FROM SC WHERE course = 'CS101'
     INTERSECT
     SELECT student FROM SC WHERE course = 'CS102'
    ) as TMP
GROUP BY student
```

Rewriting CONTAINED BY: The CONTAINED BY predicate can be rewritten by using EXCEPT. For instance, the rewritten query for Q3 in Section 2 is:

```
Q3': SELECT student FROM SC
     EXCEPT
     SELECT student FROM SC
     WHERE grade <> 4 AND grade <> 3
```

Rewriting EQUAL: The rewriting of EQUAL predicates naturally combines that of CONTAIN and CONTAINED BY, since two sets $S_1 = S_2$ if and only if $S_1 \subseteq S_2$ and $S_1 \supseteq S_2$. For instance, Q4 in Section 2 can be rewritten as:

```
Q4':(SELECT student FROM SC WHERE course = 'CS101'
     INTERSECT
     SELECT student FROM SC WHERE course = 'CS102')
    EXCEPT
    (SELECT student FROM SC
     WHERE course <> 'CS101' AND course <> 'CS102')
```

Rewriting General Queries: To rewrite more complex queries with multiple predicates, we use the rewriting of individual predicates as the building blocks and connect them together by their logical relationships. For instance, given the following query:

```
SELECT student,AVG(grade)  FROM SC  GROUP BY student
HAVING MAX(grade) = 4
OR     SET(course) CONTAIN {'CS101', 'CS102'}
OR     SET(course) CONTAIN {'CS101', 'CS103'}
```

The rewritten query is:

```
SELECT student, AVG(grade)
FROM SC,
     ((SELECT student   FROM SC   GROUP BY student
       HAVING MAX(grade) = 4)
UNION (SELECT student FROM SC WHERE course='CS101'
       INTERSECT
       SELECT student FROM SC WHERE course='CS102')
UNION (SELECT student FROM SC WHERE course='CS101'
       INTERSECT
       SELECT student FROM SC WHERE course='CS103')
     ) as TMP
WHERE SC.student = TMP.student
GROUP BY student
```

Moreover, as mentioned in Section 2, the grouping and set predicates can be defined over a relation R that is the result of a subquery. Even though our examples only use a single table, the general applicability is straightforward.

B. INTEGRATION AND INTERACTION WITH CONVENTIONAL SQL OPERATIONS

For *selection* condition, suppose our query has a set of conjunctive/disjunctive selection conditions $c_1,\ldots,c_k$, where each c_i can be either a point condition a_i=b_i or a range condition $l_i \leq a_i \leq u_i$. We first obtain a vector $vec_{\mathcal{R}}$ that represents the result of the selection conditions. If a tuple does not belong to relation R, we must set its corresponding bit in $vec_{\mathcal{R}}$ to 0. After querying the bitmap indexes to obtain the vectors vec_{vj} for constant values in the set predicates (Step 1 of Algorithm 2), the vectors can be intersected with $vec_{\mathcal{R}}$ before they are further used in the later stages of the algorithm.

There are many works in the literature (e.g., [2, 24, 12, 23]) on answering selection queries using bitmap index, i.e., getting $vec_{\mathcal{R}}$. The essence is to compute one vector vec_{c_i} for each condition c_i, such that vec_{c_i} contains the bits for tuples satisfying the condition. After bitwise AND/OR operations on the vectors of all the conditions, the resulting vector is $vec_{\mathcal{R}}$. The bit vector vec_{c_i} is computed using bitmap operations over the bitmap index on the attribute a_i in condition c_i. In order to get vec_{c_i} with a small number of bitmap operations, it may be necessary to build the bitmap index using some encoding schemes (*e.g.*, [2, 24]). For instance, some scheme requires only one bitmap operation for any one-side range selection condition and some requires only two operations for any two-side condition.

When *join conditions* exist in the query, we assume the tables have a snowflake-schema, consisting of one fact table (e.g., $lineitem$ in TPC-H [20]), and multiple dimension tables (e.g., $orders$, $nation$, $customer$, $supplier$). Each dimension is described by a hierarchy, with one dimension table for each node on the hierarchy. The fact table is connected to the dimensions by foreign keys. The tables on each dimension are also connected by keys and foreign keys. As a special case of snowflake-schema, star-schema has only one table on every dimension, thus no hierarchy. (The join queries under star-schema are so-called "star-joins".)

Under such assumption, our technique can be easily extended to handle join queries. Consider a simple case with only two tables, where S is the fact table and R is the dimension table, $j1$ is a key of R and $j2$ is the corresponding foreign key in S. Due to the foreign key constraint, there exists one and only one tuple in R joining with each and every tuple $s \in S$. Therefore for a join condition $R.j1$=$S.j2$, virtually all the join results are in S, with some attributes stored in S and some other attributes in R. Therefore, for each attribute a in the schema of R except $j1$ (since $R.j1$=$S.j2$ and we already have $j2$ in S), we can construct a bitmap index on a for the tuples in S, even though a is not an attribute of S. In general, we can follow this way to construct bitmap index for the tuples in the single fact table, on all relevant attributes in the dimension tables. Thus the selection conditions involving these attributes can be viewed as being applied on the fact table only. A join query can then be processed like a single table query. More details about such idea of building bitmap join index are in [11].

C. COLUMN VALUE GENERATION

For our synthetic tables $R(a,v,g)$ (Section 5.1), the values of column a are randomly generated. The values in column g are generated by following a uniform distribution, to make sure there are G groups, *i.e.*, there are about T/G tuples in each group. We randomly choose S out of the G groups to be qualifying groups. For the tuples in each qualifying group, we generate their values on column v in a way such that the group satisfies the set predicate. The v values for the G-S unsatisfying groups are similarly generated, by making sure the groups cannot satisfy the set predicate while generating the tuples. For example, if the query is $\gamma_{g,SUM(a)} v \supseteq \{1,2\}(R)$, for a qualified group, we randomly select 2 tuples and set their v values to 1 and 2, respectively. The v values for the remaining tuples in the group are generated randomly. Given a group to be disqualified, we randomly decide if 1, 2, or both should be missing from the group, and then generate the values randomly from a pool of numbers excluding the missing values.

D. QUERIES ON TPC-H BENCHMARK

(TPCH-1) Get the total sales of each brand that has business in both USA and Canada. We use view but it can also be written in a single query:

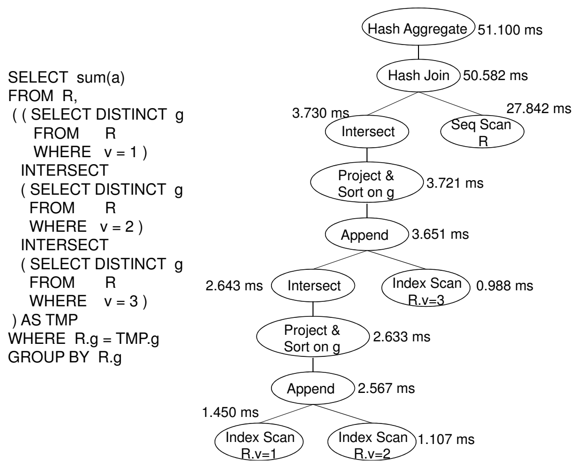
```
CREATE VIEW R1 AS
select P_BRAND, L_QUANTITY, N_NAME
FROM LINEITEM, ORDERS, CUSTOMER, PART, NATION
WHERE L_ORDERKEY=O_ORDERKEY
    AND O_CUSTKEY=C_CUSTKEY
    AND C_NATIONKEY=N_NATIONKEY
    AND L_PARTKEY=P_PARTKEY;

SELECT P_BRAND, SUM(L_QUANTITY)
FROM R1
GROUP BY P_BRAND
HAVING SET(N_NAME) CONTAIN {'United States','Canada'}
```

(TPCH-2) Get the available quantity for each part that is only available from suppliers in member nations of G8:

```
CREATE VIEW R2 AS
SELECT P_PARTKEY, PS_AVAILQTY, N_NAME
FROM PARTSUPP, SUPPLIER, PART, NATION
WHERE PS_PARTKEY=P_PARTKEY
    AND PS_SUPPKEY=S_SUPPKEY
    AND S_NATIONKEY=N_NATIONKEY;

SELECT PS_PARTKEY, SUM(PS_AVAILQTY)
FROM R2
GROUP BY PS_PARTKEY
HAVING SET(N_NAME) CONTAINED BY {'France, Germany',
       'Japan', 'United Kingdom', 'United States',
       'Canada', 'Russia', 'Italy'}
```



```
SELECT  sum(a)
FROM  R,
 ( ( SELECT DISTINCT  g
    FROM      R
    WHERE   v = 1 )
  INTERSECT
  ( SELECT DISTINCT  g
    FROM      R
    WHERE   v = 2 )
  INTERSECT
  ( SELECT DISTINCT  g
    FROM      R
    WHERE   v = 3 )
 ) AS TMP
WHERE  R.g = TMP.g
GROUP BY  R.g
```

Hash Aggregate 51.100 ms
Hash Join 50.582 ms
3.730 ms Intersect
Seq Scan R 27.842 ms
Project & Sort on g 3.721 ms
Append 3.651 ms
2.643 ms Intersect
Index Scan R.v=3 0.988 ms
Project & Sort on g 2.633 ms
Append 2.567 ms
1.450 ms Index Scan R.v=1
Index Scan R.v=2 1.107 ms

Figure 8: *Rewrt* Query Plan for CONTAIN, 100K tuples, 1K groups, 10 satisfying groups.

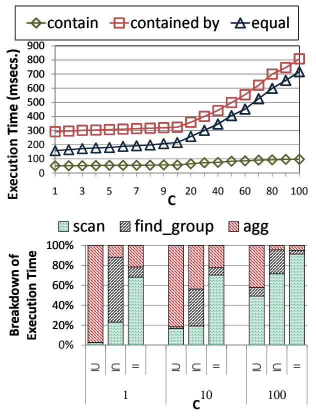
E. BREAKDOWN OF REWRT AND AGG

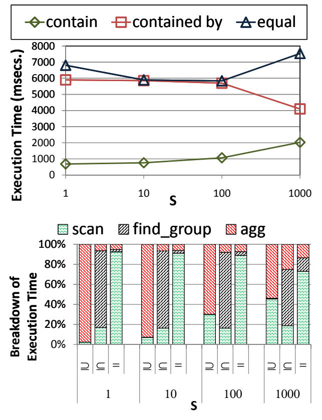
Similar to Figure 5 for *Bitmap*, the execution time and its detailed breakdown for *Rewrt* and *Agg* are shown in Figure 6 and 7, respectively, for different configuration groups. For the *Rewrt* method, the PostgreSQL plans can be roughly divided into three major steps: Step 1– scan the table (several times); Step 2– find qualifying groups that satisfy the query conditions; Step 3– calculate the required aggregates for each qualifying group. For *Agg*, we divide the cost into table scan and the rest. Similar to what we observed for Figure 5, no single component in the breakdown of *Rewrt* and *Agg* dominates the query execution cost. The breakdown varies as the configuration parameters change.

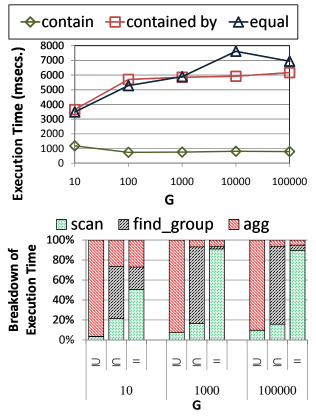
Figure 6(a) shows that the execution time of *Rewrt*, similar to that of *Bitmap*, also increases linearly with C (number of constant values in set predicate). The breakdown also changes by C. For example, the Step 3 (calculate aggregates) for CONTAIN ($\supseteq$) gets smaller and smaller percentage. We can understand this by analyzing the plan for the rewritten query of $\gamma_{g,SUM(a)}\ v \supseteq \{1,2,3\}(R)$, shown in Figure 8. It performs multiple index scans in Step 1 and intersects the scan results in Step 2. Therefore these two steps become more costly as C increases. The last step, calculating aggregates, does the same amount of work, since the aggregating is independent of C. Figure 6(a) also shows large performance difference between different operators when C increases. Given T=100K, G=1K, and S=10, the number of tuples with $R.v$ being 1, 2, ..., or C is small, in order to have only 10 out of $1,000$ groups satisfying the query condition. Therefore for CONTAIN, the set intersect operator in Figure 8 involves small cost. However for CONTAINED BY, the filter operator in Figure 2 produces much more result tuples, making the set difference (Except) operator more costly.

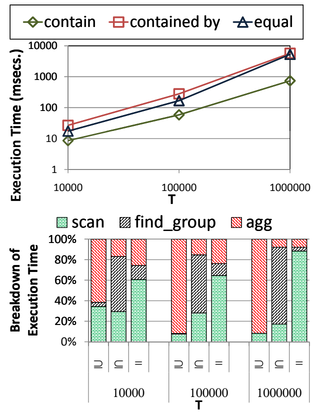
Figure 7(a) shows that the execution time of *Agg*, different from that of *Rewrt* and *Bitmap*, is not affected by C. As the outline in Algorithm 1 shows, the only cost component related to C is the checking of a tuple's v value against the given C constants (line 8 and 11), which is much cheaper than other components.

Figure 7(b) shows that the execution time of *Agg* increases with S (number of satisfying groups) under some operators and decreases under others, and the changes are only slight in both cases. Figure 6(b) shows that the behavior of *Rewrt* is similar to *Agg* with regard to S in that the execution time increases under some operators and decreases under others. The variations are small for some operators and larger for others. For example, the cost of CONTAINED BY decreases by S. Use Figure 2 to explain again. When



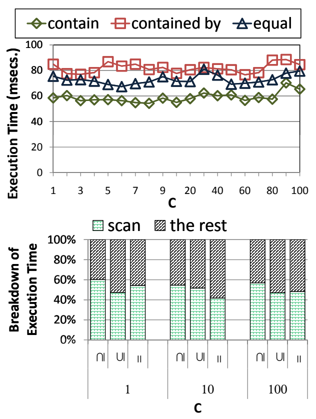


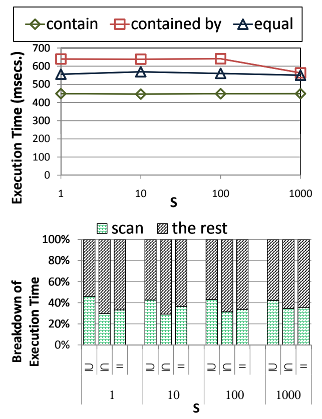


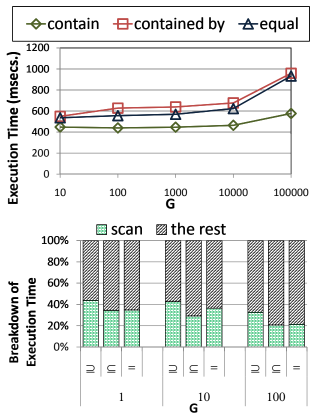


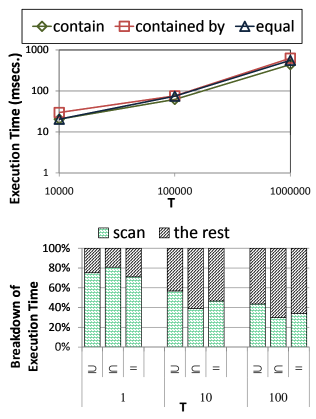
(a) T=100K, G=1K, S=10, varying C. (b) T=1M, G=1K, C=4, varying S. (c) T=1M, S=10, C=4, varying G. (d) G=100, S=10, C=4, varying T.

Figure 6: Execution time of *Rewrt* and its breakdown.









(a) T=100K, G=1K, S=10, varying C. (b) T=1M, G=1K, C=4, varying S. (c) T=1M, S=10, C=4, varying G. (d) G=100, S=10, C=4, varying T.

Figure 7: Execution time of *Agg* and its breakdown.

more groups satisfy the query condition, the output cardinality of operator Filter becomes smaller, resulting in smaller cost of set difference (Except) operation.

Figure 6(c) shows that, as G (number of groups) in the data increases, the execution time of *Rewrt* increases substantially for CONTAINED BY and EQUAL, but does not change much for CONTAIN. Use CONTAINED BY as an example. When G increases, with the number of satisfying groups unchanged, the number of tuples matching the constants becomes smaller, resulting in more result tuples from the Filter operator in Figure 2, making the set difference (Except) operator more costly. Figure 7(c) shows that the execution time of *Agg* increases slowly with G, for the reason similar to why the execution time is not affected much by C.

Figure 6(d) and Figure 7(d) show that, naturally, *Rewrt* and *Agg* scale linearly with T (number of tuples in the data table), similar to *Bitmap*.