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

Chengkai Li, *Member, IEEE,* Bin He, Ning Yan, Muhammad Assad Safiullah

Abstract—In data warehousing and OLAP applications, *scalar-level* predicates in SQL become increasingly inadequate to support a class of operations that require *set-level* comparison semantics, i.e., comparing a group of tuples with multiple values. Currently, 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 a database engine to optimize, thus can result in costly evaluation. This paper proposes to augment SQL with *set predicate*, to bring out otherwise obscured set-level semantics. We studied two approaches to processing set predicates– an aggregate function-based approach and a bitmap index-based approach. Moreover, we designed a histogram-based probabilistic method of set predicate selectivity estimation, for optimizing queries with multiple predicates. The experiments verified its accuracy and effectiveness in optimizing queries.

Index Terms—Set Predicates, Grouping, Data Warehousing, OLAP, Querying Processing and Optimization

1 Introduction

With data warehousing and OLAP 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 its resume database for job candidates with a set of mandatory skills. Here the skills of each candidate, as a set of values, are compared against the mandatory skills. Such sets are often dynamically formed. For example, suppose a table Resume_Skills (id, skill) connects skills to job candidates. A GROUP BY clause dynamically groups the tuples in it by id, with the values on attribute skill in each 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, aggregate functions SUM/ COUNT/ AVG/ MAX produce a single numeric value which is compared to a literal or another single aggregate value.

Observing the demand for complex and dynamic set-level comparisons in databases, we propose a concept of *set predicate*. Below are several example queries with set predicates.

Example 1: To find those candidates with skills “Java” and “Web services”, our query can be as follows. After grouping, a dynamic set of values on attribute skill is formed for each unique id, and groups whose corresponding SET(skill) contains both “Java” and “Web services” are returned.

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

Example 2: In business decision making, an executive may want to find the departments whose monthly average ratings for customer service in 2009 have always been poor (assuming ratings are from 1 to 5). Suppose the table schema is Ratings(department, avg_rating, month, year). The following query uses CONTAINED BY for the set-level condition.

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

Example 3: Set predicates can be defined across multiple attributes. Consider an online advertisement example. Suppose the table schema is Site_Statistics(website, advertiser, CTR). A marketing strategist uses the following query to find Websites that publish ads for ING with more than 1% and less than 2% click-through rate (CTR) and do not publish ads for HSBC yet:

```
SELECT   website   FROM Site_Statistics
GROUP BY website   HAVING
    SET(advertiser,CTR) CONTAIN {('ING',[0.01,0.02])}
AND NOT (SET(advertiser) CONTAIN {'HSBC'})
```

In this example, the first set predicate involves two attributes and the second set predicate uses the negation of CONTAIN. Note that we use `[0.01,0.02]` to represent a range-based condition $0.01 \leq \text{CTR} \leq 0.02$.

The semantics of set-level comparisons in many cases can be expressed using current SQL syntax without the proposed extension. However, resulting queries would be more complex than necessary. One consequence is that complex queries are difficult for users to formulate. More importantly, such complex queries are difficult for DBMS to optimize, leading to unnecessarily costly evaluation. The resulting query plans could involve multiple subqueries with grouping and set operations. On the contrary, the proposed concise syntax of set predicates enables direct expression of set-level comparisons in SQL, which not only makes query formulation simple but also facilitates efficient support of such queries. We developed two approaches to process set predicates:

- *C. Li and N. Yan are with the Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, TX 76019. E-mail: cli@uta.edu, ning.yan@mavs.uta.edu*
- *B. He is with IBM Almaden Research Center, San Jose, CA 95120. E-mail: binhe@us.ibm.com*
- *M. A. Safiullah is with Microsoft, Seattle, WA. E-mail: assad.safiullah@live.com. The work was done while the author was a student at UT-Arlington.*

Aggregate function-based approach: This approach processes set predicates in a way similar to processing conventional aggregate functions. Given a query with set predicates, instead of decomposing the query into multiple subqueries, this approach only needs one pass of table scan.

Bitmap index-based approach: This approach processes set predicates by using bitmap indices on individual attributes. It is efficient because it can focus on only the tuples from those groups that satisfy query conditions and only the bitmaps for relevant columns. State-of-the-art bitmap compression methods [33], [2], [16] and encoding strategies [5], [34], [23] have made it affordable to build bitmap index on many attributes. This index structure is also applicable on many different types of attributes. The bitmap index-based approach processes general queries (with joins, selections, multi-attribute grouping and multiple set predicates) by utilizing single-attribute indices. Hence it does not require pre-computed index for join results or combination of attributes.

We further developed an optimization strategy to handle queries with multiple set predicates connected by logic operations (AND, OR, NOT). A useful optimization rule is to prune unnecessary set predicates during query evaluation. Given a query with n conjunctive set predicates, the predicates can be sequentially evaluated. If no group qualifies after m predicates are processed, we can terminate query evaluation without processing the remaining predicates. The number of “necessary” predicates before we can stop, m, depends on the evaluation order of predicates. We designed a method to select a good order of predicate evaluation, i.e, an order that results in small m, thus cheap evaluation cost. Our idea is to evaluate conjunctive (disjunctive) predicates in the ascending (descending) order of their selectivities, where the selectivity of a predicate is its number of qualified groups. We designed a probabilistic approach to estimating set predicate selectivity by database histograms.

In summary, this paper makes the following contributions:

- We proposed to extend SQL with set predicates for an important class of analytical queries, which otherwise would be difficult to write and optimize (Section 3).
- We designed two query evaluation approaches for set predicates, including an aggregate function-based approach (Section 5) and a bitmap index-based approach (Section 6).
- We developed a histogram-based probabilistic method to estimate the selectivity of a set predicate, for optimizing queries with multiple predicates (Section 8).
- We conducted extensive experiments to evaluate proposed approaches over both real and synthetic data (Section 9).

2 Related Work

Set-valued attributes provide a concise and natural way to model complex data concepts such as sets [24], [29]. Many DBMSs nowadays support the definition of attributes involving a set of values, e.g., *nested table* in Oracle and *SET* data type in MySQL. For example, the “skill” attribute in Example 1 can be defined as a set data type. Set operations can be natively supported on such attributes. Query processing on set-valued attributes and set containment joins have been extensively studied [14], [27], [18], [19]. Although set-valued attributes together with set containment joins can support set-level comparisons, set predicates have several critical advantages:

(1) Unlike set-valued attributes, which bring hassles in redesigning database storage for the special set data type, set predicates require no change in data representation and storage, and thus can be incorporated into standard RDBMS.

(2) In real-world applications, groups and corresponding sets are often dynamically formed according to query needs. For instance, in Example 2, the monthly ratings of each department form a set. In a different query, sets may be formed by ratings of individual employees. With set predicates, users can dynamically form set-level comparisons with no limitation caused by database schema. On the contrary, set-valued attributes cannot support dynamic set formation because they are pre-defined at schema definition phase and set-level comparisons can only be issued on such attributes.

(3) Set predicates allow cross-attribute set-level comparison. For instance, sets are defined over advertiser and CTR together in Example 3. On the contrary, a set-valued attribute can only be defined on a single attribute in many implementations, thus cannot capture cross-attribute associations. Implementations such as nested table in Oracle allow sets over multiple attributes but do not easily support set-level comparisons on such attributes.

Set predicate is also related to universal quantification and relational division [12], which are powerful for analyzing many-to-many relationships. An example universal quantification query is to find the students that have taken all computer science courses required to graduate. It is a special type of set predicates with CONTAIN operator over all the values of an attribute in a table, e.g., Courses. By contrast, the proposed set predicates allow sets to be dynamically formed through GROUP BY and support CONTAINED BY and EQUAL, in addition to CONTAIN.

The SEQUEL 2 language (an extension of the original SEQUEL) for SYSTEM R proposed a special SET function, for comparing a set of attribute values with the result of a subquery [4]. The proposed comparison operators include CONTAINS, =, and their negations. Furthermore, these operators can be used in comparing the results of two subqueries. The proposal was brief, by several example queries. The SET function is only on one attribute and does not allow range-based values or bag semantics. No suggestion for implementation techniques was made. The SET function and CONTAINS operator were later dropped from the SQL language, possibly because of the difficulty in implementing it efficiently [11].

In [8], [9], [7], [10] the concept of *grouping variable* and *associated set* was introduced as an SQL extension to allow comparisons of multiple aggregates over the same grouping condition. That line of work only considered regular aggregates such as SUM and COUNT. Combining the concepts of set predicate and grouping variable can allow simpler syntax for complex queries. We provide more detailed discussions of this in the supplemental materials to this paper.

This paper focuses on relational data model and architecture. Some data analytics systems today are built on top of massive parallel computing architecture. The query languages for such

Table: SC

semester	student	course	grade
Fall09	Mary	CS101	4
Fall09	Mary	CS102	2
Fall09	Tom	CS102	4
Spring10	Tom	CS103	3
Fall09	John	CS101	4
Fall09	John	CS102	4
Spring10	John	CS103	3

Fig. 1. A classic student and course example.

systems (e.g., Pig Latin [21], Dremel [20], Jaql [1]) deal with complex data models such as set-valued attributes, maps, and nested data. Due to the fundamental architectural difference, supporting set predicates in such systems, although a very interesting future topic, is beyond the scope of this paper.

3 SET PREDICATES

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

SET(v_1, ..., v_m)
CONTAIN | CONTAINED BY | EQUAL
$\{(v_1^1$, ..., $v_m^1)$, ..., $(v_1^n$, ..., $v_m^n)\}$,

where $v_i^j \in Dom(v_i)$, *i.e.*, each v_i^j is a literal value (integer, floating point number, 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 set operator CONTAIN, CONTAINED BY, and EQUAL, respectively.

The syntax can be extended to allow set-level comparison with not only literal values, but also another dynamically formed group or the result of a subquery. We focus on literal values in the following sections and discuss such extension in the supplemental materials.

We further use relational algebra to concisely represent queries with set predicates. Given a relation R, grouping and aggregation are represented by the following operator:

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

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). The aggregates in $\mathcal{A}$ and $\mathcal{C}$ may overlap.

We now provide example queries over the classic student-course table (Figure 1). We use full SQL for the first query as we did in Section 1. For remaining queries, we will show either only set predicates or succinct relational algebra expressions.

The following Q1: $\gamma_{student}$ $course$ $\supseteq$\{'CS101','CS102'\}(SC) identifies the students who took both CS101 and CS102. [1] 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'}
```

A query can include WHERE clause and regular aggregate functions in HAVING. In Q2: $\gamma_{student,COUNT(*)}$ $course \supseteq$ \{'CS101', 'CS102'\} $\bigwedge$ AVG($grade$) $>$ 3.5 ($\sigma_{semester='Fall09'}$(SC)), we look for those students that had average grade higher than 3.5 in FALL09 and took both CS101 and CS102 in that semester. It also returns the number of courses they took in that semester.

We use CONTAINED BY for the reverse of CONTAIN, *i.e.*, the subset relationship. Query Q3: $\gamma_{student}$ $grade$ $\subseteq$\{4,3\}(SC) selects all the students whose grades are never below 3. The results are Tom and John.

To select the students that have only taken CS101 and CS102, we use EQUAL to represent the equal relationship in set theory. The query is Q4: $\gamma_{student}$ $course$ $=$\{'CS101','CS102'\}(SC). Its result contains only Mary.

In above queries we assumed set predicates follow set semantics. Therefore John's grades, \{4,4,3\}, are subsumed by \{4,3\}. The syntax also allows bag semantics for set predicates, where $\gamma_{student}$ $course$ $\supseteq$\{'CS101','CS101','CS102'\}(SC) finds students who took CS101 twice and CS102 once, and $\gamma_{student}$ $grade$ $\subseteq$\{4,4,3\}(SC) selects students who have obtained grade 4 in at most 2 courses and grade 3 in at most 1 course and have no other grades in record.

Note that the set/bag semantics of set predicates are orthogonal to the set/bag semantics of regular SQL constructs. If set semantics is applied for a set predicate, only distinct values on the set predicate attribute from the tuples in a group are used in determining if the group satisfies the set predicate. However, if (the default) bag semantics is applied for regular SQL operations, all the tuples in the group are included in calculating aggregates. For example, $\gamma_{student,AVG(grade)}$ $grade$ $\supseteq$\{4,4,3\}(SC) calculates GPA for students with at least two 4s and one 3, and all their grades are included in GPA calculation.

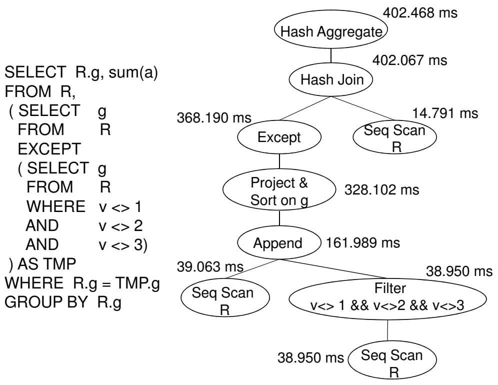
For simplicity of presentation, in the following sections we focus on the simplest query– $\gamma_{g,\oplus a}$ v ***op*** $\{v^1$, ..., $v^n\}(R)$, i.e., a query with one grouping attribute (g), one aggregate for output ($\oplus a$), and one set predicate defined by a set operator ***op*** ($\supseteq$, $\subseteq$, or $=$) over a single attribute (v). Moreover set semantics is assumed for set predicates. In Section 7 we discuss the syntax of expressing more general queries and the methods of processing general queries.

4 DRAWBACKS OF SET-LEVEL COMPARISONS BY REGULAR SQL

Without the proposed set predicate, we fall back to current SQL syntax in expressing set-level comparisons. Complex queries containing scalar-level operations are often formed to obtain even very simple set-level semantics. Such complex queries are difficult for users to formulate. A more severe consequence is that set-level semantics becomes obscure. Hence a DBMS may choose unnecessarily costly evaluation plans for such queries.

The semantics of set predicates can often be expressed by standard SQL queries. In fact, there can be multiple ways in

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



```
SELECT  R.g, 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
```

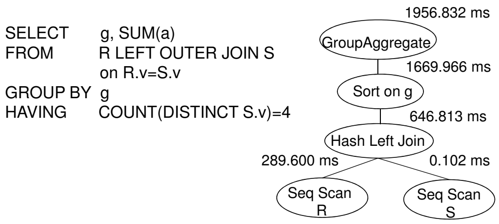
Fig. 2. SQL query and plan for $\gamma_{g,SUM(a)} v \subseteq \{1,2,3\}(R)$ over 100K tuples, 1K groups, and 10 qualified groups.

writing queries corresponding to even a single set predicate. For instance, for a CONTAIN predicate with m values, the query can use m-1 INTERSECT operations. Or, we can build a temporary table S containing the m values, left outer join R (the table being queried) with S, and process a sequence of duplicate elimination, grouping, and group selection by COUNT. As another example, a CONTAINED BY can be expressed by EXCEPT or CASE condition control. For multiple set predicates, we can use the queries for individual predicates as building blocks and connect them by logic relationships.

Our experience is that no matter how we express the semantics of a set predicate using regular SQL, the query often inevitably involves a combination of multiple operations such as join, union, intersection, set difference, duplicate elimination, grouping, etc. The performance of the resulting query is usually unsatisfactory. Although one cannot exhaust all possible queries in expressing a set predicate, the examples below illustrate this observation by using two different methods of writing queries. Section 9 empirically compares our proposed methods with the method of expressing set predicates by regular SQL queries. We discuss the details of such regular SQL queries in the supplemental materials to this paper.

Figure 2 shows a PostgreSQL query plan for a regular SQL query corresponding to $\gamma_{g,SUM(a)}\ v \subseteq \{1,2,3\}(R)$. The plan was executed over a 100K-tuple table $R(g,a,v)$ with 1K groups on g, resulting in 10 qualified groups. The plan was hand-picked and the most efficient one among the plans we investigated. Figure 2 also shows the time spent on each operator, which recursively includes the time spent on all operators in the sub-plan tree rooted at the given operator, due to the effect of iterators' GetNext() interfaces. The real PostgreSQL plan had more detailed operators. We combine them and give the combined operators more intuitive names, for simplicity of presentation. Figure 2 indicates that the query obscures the semantics of set-level comparison, as the query plan unnecessarily 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.

Figure 3 shows a plan for query $\gamma_{g,SUM(a)}\ v \supseteq \{1,2,3,4\}(R)$. The table R has 1M tuples. Among the 10K groups formed by attribute g, 10 groups satisfy the set predicate. The plan uses a temporary one-attribute table S that contains values 1, 2, 3, and 4. It performs a left outer join between R and S, followed by grouping on g. For each group, it checks if the group contains all four values by COUNT. The join and sorting operators are expensive. On the contrary, it is sufficient to use a one-pass grouping and aggregation, as Section 5 will show.



```
SELECT      g, SUM(a)
FROM        R LEFT OUTER JOIN S
            on R.v=S.v
GROUP BY  g
HAVING      COUNT(DISTINCT S.v)=4
```

Fig. 3. SQL query and plan for $\gamma_{g,SUM(a)} v \supseteq \{1,2,3,4\}(R)$ over 1M tuples, 10K groups, and 10 qualified groups.

We also executed the above two queries in a single-node installation of IBM DB2 V8. Interestingly DB2 chose essentially the same query plans, except that the EXCEPT operator in Figure 2 was replaced by a pair of grouping and filtering operators. This helps to show that PostgreSQL was not necessarily doing a bad job in optimizing the provided regular SQL queries.

5 Aggregate Function-Based Approach

With the new syntax in Section 3, which brings forward the semantics of set predicates, a set predicate-aware query plan could potentially be much more efficient by just scanning a table and processing its tuples sequentially. The key to such a direct approach is to perform grouping and set-level comparison together, through a one-pass iteration of tuples. The idea resembles how regular aggregate functions can be processed together with grouping. Hence we design a method that handles set predicates as aggregate functions.

The sketch of the method is in Algorithm 1. It covers all three kinds of set operators ($\supseteq$, $\subseteq$, $=$). It uses the standard iterator interface GetNext() to go through the tuples in R, may it be from a sequential scan over table R or the sub-plan over sub-query R. Following common implementation of aggregate functions in database systems, a set predicate is defined by an initial state (Line 3), a state transition stage (Line 4-14), and a final calculation stage (Line 16-18). A 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 values v^1, ..., v^n in the set predicate are contained in the corresponding group. If the mask value for a group equals $2^n{-}1$, the group contains all n values v^1, ..., v^n. A hash table G maintains a Boolean value for each group, indicating if it is a qualified group. The values in G were initialized to $True$ for every group if the set operator is $\subseteq$ or $=$ (Line 3), otherwise $False$. A hash table A maintains the aggregated values for the groups.

In detail, a tuple is skipped if the corresponding group is already disqualified and the operator is $\subseteq$ or $=$ (Line 4). It is also skipped if the group is already identified as qualified and the operator is $\supseteq$, except that we need to accumulate the aggregate (Line 6). For a non-skipped tuple, if its value on

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;$v$:set predicate attribute */
    /* $A$: hash table for aggregate values */
    /* $M$: hash table for value masks */
    /* $G$: hash table for Boolean indicators 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:         $M[g]$ $\Leftarrow$ 0; $G[g]$$\Leftarrow$(op $\in$ $\{\subseteq,=\}$); also initialize $A[g]$ according to the aggregate function.
 4:     if (op $\in$ $\{\subseteq,=\}$) $\wedge$ (! $G[g]$) then continue to next tuple
 5:     /* Aggregate the value $a$ for group $g$. */
 6:     $A[g] \Leftarrow A[g] \oplus a$
 7:     if (op is $\supseteq$) $\wedge$ ($G[g]$) then continue to next tuple
 8:     if $v$ == $v^j$ for some $j$ then
 9:         /* In $M[g]$, set the mask for $v^j$. */
10:         $M[g] \Leftarrow M[g]$ | $2^{j-1}$
11:         if ($M[g]$ == $2^n - 1$) $\wedge$ (op is $\supseteq$) then $G[g] \Leftarrow True$
12:     else
13:         /* For $\subseteq$, $=$, if $v \notin \{v^1$, ..., $v^n\}$, $g$ does not qualify. */
14:         if op $\in$ $\{\subseteq,=\}$ then $G[g] \Leftarrow False$
15: /* Output qualified groups and their aggregates. */
16: for every group $g$ in hash table $M$ do
17:     if (op is $=$) $\wedge$ ($M[g]$ != $2^n - 1$) then $G[g] \Leftarrow False$
18:     if $G[g]$ then output($g$, $A[g]$)
```

attribute v matches some v^j in v^1, ..., v^n, we set the jth-bit of the group's mask value in M to 1, indicating the existence of v^j in the group. This is done by the bitwise OR operation in Line 10. If the mask value becomes 2^n-1, we mark the group as qualified if the operator is $\supseteq$ (Line 11). On the other hand, if the tuple's v value does not match any such v^j, we mark the group as disqualified if the operator is $\subseteq$ or $=$ (Line 14). If the operator is $=$, we also check if the mask value equals 2^n-1 at the final calculation stage. If not, it means the group does not contain all the values v^1, ..., v^n. Therefore we mark the group as disqualified (Line 17).

Algorithm 1 is a one-pass algorithm where memory is available for storing the hash tables for all groups. We only implemented and experimented with such one-pass algorithm, given that the number of groups is seldom extremely large. Should the number of groups become so large that the hash tables cannot fit in memory, we can adopt standard two-pass hashing-based or sorting-based aggregation method in DBMSs. In the first pass the input table is sorted or partitioned by a hash function. In the second pass tuples in the same group are loaded into memory and aggregates over different groups are handled independently. Such two-pass method can be further improved by early aggregation strategies [17].

6 Bitmap Index-Based Approach

Our second approach is based on bitmap index [22], [23]. In a vanilla bitmap index on an attribute, there exists a bitmap (a vector of bits) for each unique attribute value. The vector length 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 if the ith tuple has value x on attribute v. Complex selection queries can be efficiently answered by bitwise operations (AND (&), OR(|), XOR(ˆ), and NOT(∼)) over bit vectors. Moreover, bitmap indices enable efficient computation of aggregates (e.g., SUM and COUNT) [23].

The idea of using bitmap index to process set predicates is in line with the aforementioned intuition of processing set-level comparison by a one-pass iteration of tuples (i.e., their corresponding bits in bit vectors). On this aspect, it is similar to the aggregate function-based approach. However, this method brings several advantages by leveraging the distinguishing characteristics of bitmap index. (1) We only need to access the bitmap indices on columns involved in a query. Hence the method's query performance is independent of the underlying table's width. (2) The data structure of bit vector is efficient for basic operations such as membership checking (for matching with values in set predicates). Bitmap index gives us the ability to skip irrelevant tuples. Chunks of 0s in a bit vector can be skipped together due to effective bitmap encoding. (3) The simple data format and bitmap operations make it convenient to integrate various operations in a query, including dynamic grouping of tuples and set-level comparisons. It also enables efficient and seamless integration with conventional selections, joins, and aggregations. (4) It allows straightforward extensions to handle otherwise complex features, such as multi-attribute set predicates and multiple set predicates.

As an efficient index for decision support queries, bitmap index has gained broad popularity. State-of-the-art bitmap compression methods [33], [2], [16] and encoding strategies [5], [34], [23] allow bitmap index to be applied on all types of attributes (e.g., high-cardinality categorical attributes [32], [33], numeric attributes [32], [23] and text attributes [28]). Bitmap index is now supported in major commercial database systems (e.g, Oracle, SQL Server), and it is often the default (or only) index option in column-oriented database systems (e.g., Vertica, C-Store [30], LucidDB). In applications with read-mostly or append-only data, such as OLAP and data warehouses, it is common that bitmap indices are created for many attributes. Moreover, index selection based on query workload allows a system to selectively create indices on attributes that are more likely to be used in queries.

The bitmap index-based approach only needs bitmap indices on individual attributes. Based on single-attribute indices, it copes with general queries, dynamic groups, joins, selection conditions, multi-attribute grouping and multiple set predicates. It does not require pre-computed index for join/selection results or combination of attributes. (Details in Section 7.)

Some systems (e.g., DB2, PostgreSQL) only build bitmap indices on the fly at query-time. We do not consider such scenario. We focus on bitmap indices built before query time.

The particular type of bitmap index we use is *bit-sliced index* (BSI) [28]. Given a numeric attribute on integers or floating-point numbers, BSI directly captures the binary representations of attribute values. The tuples' values on an attribute 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 distinct categorical values to integers.

The approach requires bit-sliced indices on g (BSI(g)) and a (BSI(a)) and a bitmap index on v (BI(v)), which can be a BSI

Algorithm 2 Bitmap Index-Based Approach

```
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: Qualified 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 value masks */
    /* $G$: hash table for Boolean indicators of qualified groups */
    /* Step 1. get the vector for each $v^j$ 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:         $gID[k] \Leftarrow gID[k] + 2^i$
 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:         $M[gID[k]]$⇐0; $G[gID[k]]$ $\Leftarrow$ $False$; also initialize
            $A[gID[k]]$ according to the aggregate function.
    /* Step 3. find qualified groups */
10: if op $\in$ {$\supseteq$, $=$} then
11:     for each bit vector $vec_{v^j}$ do
12:         for each set bit $b_k$ in $vec_{v^j}$ do
13:             $M[gID[k]] \Leftarrow M[gID[k]] \mid 2^{j-1}$
14:     for each group $g$ in hash table $M$ do
15:         $G[g] \Leftarrow (M[g] == 2^n - 1)$
16: if op $\in$ {$\subseteq$, $=$} then
17:     for each set bit $b_k$ in ∼($vec_{v^1}$ | ... | $vec_{v^n}$) do
18:         $G[gID[k]] \Leftarrow False$
    /* Step 4. aggregate the values of $a$ for qualified groups */
19: for each $k$ from 0 to $t$-1 do
20:     if $G[gID[k]]$ then
21:         $agg \Leftarrow 0$
22:         for each slice $B_i$ in BSI($a$) do
23:             if $b_k$ is set in bit vector $B_i$ then $agg \Leftarrow agg + 2^i$
24:         $A[gID[k]] \Leftarrow A[gID[k]] \oplus agg$
25: for every group $g$ in hash table $M$ do
26:     if $G[g]$ then output ($g$, $A[g]$)
```

or other type of bitmap index. Note that the algorithm below will also work if we have other types of bitmap indices on g and a, with modifications that we omit. The advantage of BSI is that it indexes high-cardinality attributes with small number of bit vectors, thus improves query performance if grouping or aggregation is on such high-cardinality attributes.

Example 4: Given the data in Figure 1 and query $\gamma_{student,AVG(grade)}$ $course$ ***op*** {‘CS101’,‘CS102’}(SC), Figure 4 shows the bitmap indices 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 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 distinct 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 this approach 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, 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. This is a basic bitmap index functionality.

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 *student* to numbers:

Mary=>0 (00)
Tom=>1 (01)
John=>2 (10)

Fig. 4. Bitmap indices for the data in Figure 1.

Step 2 gets group IDs, i.e., values of g, for 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. (See BSI(student) in Example 4.)

Step 3 gets the groups that satisfy the set predicate, based on the vectors from Step 1. Its logic is fairly similar to that of Algorithm 1. The algorithm outline covers all three set operators, although the details differ, as explained below.

Step 4 gets the aggregates for qualified groups from Step 3 by using BSI(a). It aggregates the value of attribute a from each tuple into the tuple's corresponding group if the group is qualified. The value of attribute a for the kth tuple is obtained by assembling the values 2^{i-1} from slices B_i when their kth bits are set. The algorithm outline checks all 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.

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 values (v^1, ..., v^n) in a set predicate. A bit is set to 1 if at least one tuple in group g has the corresponding 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 by a bitwise OR operation (Line 13 of Algorithm 2). 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 v^1, ..., v^n and thus satisfies the query (Line 15). We use another hash table G to record the Boolean indicators for qualified groups. The values in G were initialized to $False$ for every group (Line 9).

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 ∼(vec_{v^1} | ... | vec_{v^n}) (i.e., the kth tuple not matching any v^j), the corresponding group $gID[k]$ is disqualified (Line 18).

EQUAL ($=$): Step 3 for EQUAL ($=$) is naturally a combination of that for $\supseteq$ and $\subseteq$. It first marks a group as qualified if $\supseteq$ is satisfied (Line 15), then disqualifies a group if $\subseteq$ is not satisfied (Line 18).

Example 5: Suppose the query is $\gamma_{student,AVG(grade)}$ $course$ = {‘CS101’,‘CS102’}(SC). Use the bitmap indices in Figure 4. After the 1st bit of vec_{CS101} (i.e., v^1=‘CS101’) is encountered in Line 12, $M[0]$=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 ∼(vec_{CS101} | vec_{CS102}) is 0001001, G[2] becomes $False$ after the last bit of 0001001 is encountered in Line 17 (i.e., John has an extra course 'CS103').

7 General Set Predicate Queries

Our discussion so far has focused on simple queries that have one grouping attribute, one aggregate for output, and one single-attribute set predicate, under set semantics of set predicates. As introduced in Section 3, more general query is denoted by $\gamma_{\mathcal{G},\mathcal{A}}\mathcal{C}(R)$, where $\mathcal{G}$ is a set of grouping attributes (appear in GROUP BY clause), $\mathcal{A}$ is a set of regular aggregates for output (appear in SELECT), and $\mathcal{C}$ is a Boolean expression over set predicates and conditions on regular aggregates (appear in HAVING). In this section we discuss how to extend our algorithms for general queries.

(A) Multi-Attribute Grouping: Given a query with multiple grouping attributes, $\gamma_{g_1,\ldots,g_l,\mathcal{A}}$ $\mathcal{C}(R)$, 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 4, 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.

(B) Multi-Attribute Set Predicate: The query syntax also allows comparing sets defined on multiple attributes, e.g., `SET(course, grade) CONTAIN {('CS101',4), ('CS102',2)}` finds all the students who received grade 4 in CS101 and 2 in CS102. In general, for a query with a set predicate defined on multiple attributes, $\gamma_{\mathcal{G},\mathcal{A}}$ $(v_1, ..., v_m)$ ***op*** $\{(v_1^1, ..., v_m^1), ..., (v_1^n, ..., v_m^n)\}(R)$, we replace Step 1 of Algorithm 2 as follows. We first obtain vectors $vec_{v_1^j}$, ..., $vec_{v_m^j}$ by querying BI(v_1), ..., BI(v_m). Then their intersection (bitwise AND), vec_{v^j} = $vec_{v_1^j}$ & ... & $vec_{v_m^j}$, gives us the tuples that match the multi-attribute value (v_1^j, ..., v_m^j).

(C) 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 whose grades are never below 3, except those who took both CS101 and CS102, we can use query `SET(grade) CONTAINED BY {4,3} AND NOT (SET(course) CONTAIN {'CS101', 'CS102'})`.

With regard to the aggregation function-based method in Algorithm 1, during a one-pass scan of tuples, multiple set predicates are processed by simply repeating the same steps for each predicate. With regard to the bitmap index-based method, we defer the discussion of optimizing the evaluation of multiple set predicates to Section 8.

(D) Regular Aggregate Expression: A general query $\gamma_{\mathcal{G},\mathcal{A}}\mathcal{C}(R)$ may have multiple regular aggregate expressions in $\mathcal{A}$ (e.g., SUM(a) in Figure 2) and $\mathcal{C}$ (e.g., AVG($grade$)>3.5 in Q2). In the aggregation function-based method, all these aggregates are accumulated at Line 6 of Algorithm 1. [2] In the bitmap index-based method, they are handled by repeating Line 21-24 of Algorithm 2 for multiple aggregates. We remove a group from query result if a condition on a regular aggregate (e.g., AVG($grade$)>3.5) is not satisfied.

(E) Set Predicates under Bag Semantics: In Algorithm 1 and 2, in addition to hash table M, we maintain an extra hash table that stores arrays of integers. For each group, the corresponding array records how many times each v value has been encountered in the group. For CONTAIN/CONTAINED BY/EQUAL, the count of each value should be no less than/no more than/equal to the corresponding count in a set predicate, otherwise the group does not satisfy the predicate.

(F) Integration and Interaction with Conventional SQL Operations: In a general query $\gamma_{\mathcal{G},\mathcal{A}}\mathcal{C}(R)$, relation R could be the result of other operations such as selections and joins. Logical bit vector operations allow us to integrate the bitmap index-based method for set predicates with bitmap index-based solutions for selection conditions [5], [34], [23] and join queries [22]. This approach only requires bitmap indices on underlying tables instead of join and/or selection result.

With regard to *selection* conditions, 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 set its corresponding bit in $vec_{\mathcal{R}}$ to 0. After querying bitmap indices to obtain the vectors vec_{v^j} for the values in a set predicate (Step 1-2 of Algorithm 2), the vectors are intersected with $vec_{\mathcal{R}}$ before they are further used in later stages of the algorithm.

There is much previous work (e.g., [5], [34], [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 c_i. After bitwise AND/OR operations on the vectors of all conditions, the resulting vector is $vec_{\mathcal{R}}$. The bit vector vec_{c_i} is computed using bitmap operations over the bitmap index on attribute a_i in condition c_i.

With regard to *join conditions* in a query, our technique can be easily extended, by using bitmap join index [22]. Consider two tables S and T. Attribute $j1$ is a key of T and $j2$ is the corresponding foreign key in S. Due to foreign key constraint, there exists one and only one tuple in T joining with each and every tuple $s \in S$. Hence for a join condition $T.j1$=$S.j2$, virtually all join results are in S, with some attributes stored in S and other attributes in T. Therefore, for each attribute a in the schema of T except $j1$ (since $T.j1$=$S.j2$ 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 indices for tuples in a table S, on all relevant attributes in other tables referenced

[2] Note that Line 6 of Algorithm 1 only shows the state transition of $\oplus$. The initialization and final calculation steps are omitted.

through foreign keys in S. Thus selection conditions involving these attributes can be viewed as being applied on S only. A join query can then be processed like a single table query.

8 Optimizing Queries with Multiple Set Predicates: Selectivity Estimation by Histogram

Given a query with multiple set predicates, the straightforward approach is to evaluate individual predicates independently and follow the logic operations between predicates (AND, OR, NOT) to perform intersection, union, and difference operations over qualified groups. However, this approach can be an overkill. In this Section we present strategies to prune unnecessary set predicates.

If multiple predicates are defined on the same set of attributes, we can eliminate the evaluation of redundant or contradicting predicates based on set-containment or mutual-exclusion between the predicates' value sets. One example is query $\gamma_{g,\oplus a}$ $v \supseteq \{1\}(R)$ AND $v \supseteq \{1,2\}(R)$. The value set of the first predicate is a subset of the second value set. Evaluating the first predicate is unnecessary because its qualified groups always subsume the second predicate's qualified groups. Similarly the second predicate can be pruned if the query uses OR instead of AND. Another example is $\gamma_{g,\oplus a}$ $v \subseteq \{1\}(R)$ AND $v \supseteq \{2,3\}(R)$. The two value sets are disjoint. Without evaluating either predicate, we can report empty result. We do not elaborate on such logical optimization since query minimization and equivalence [6] is a well-known topic.

The above logical optimization is applied without evaluating the predicates because it is based on algebraic equivalences that are data-independent. A more general optimization is to prune unnecessary set predicates during query evaluation. The idea is as follows. Suppose a query has conjunctive set predicates $p_1, ..., p_n$. We evaluate the predicates sequentially, obtain the qualified groups for each predicate, and thus obtain the groups that satisfy all the evaluated predicates so far. If no satisfying group is left after $p_1, ..., p_m$ ($m<n$) are processed, we terminate query evaluation, without processing remaining predicates. Similarly, if the predicates are disjunctive, we stop the evaluation if all the groups satisfy at least one of $p_1, ..., p_m$. In general smaller m leads to cheaper evaluation cost. (We assume equal predicate cost for simplicity. Optimization by predicate-specific cost estimation warrants further study.)

The number of "necessary" predicates before we can stop, m, depends on predicate evaluation order. For instance, suppose a query has three conjunctive predicates p_1, p_2, p_3, which are satisfied by 10%, 50%, and 90% of all groups, respectively. Consider two different orders of predicate evaluation, $p_1p_2p_3$ and $p_3p_2p_1$. The former order may have a much larger chance than the latter order to terminate after 2 predicates, i.e., reaching zero qualified groups after p_1 and p_2 are evaluated. Hence different predicate evaluation orders can potentially result in much different costs. Given n predicates, by randomly selecting an order out of $n!$ possible orders, the chance of hitting an efficient one is slim. Our goal is to select a good order, i.e, an order that results in a small m.

Such good order hinges on the "selectivities" of predicates. Suppose a query has predicates p_1,...,p_m, which are in either conjunctive form (connected by AND) or disjunctive form (OR). Each predicate can have a preceding NOT.[3] Our optimization rule is to evaluate conjunctive (disjunctive) predicates in ascending (descending) order of selectivities, where the selectivity of a predicate is its number of qualified groups. Hence the key challenge in optimizing multi-predicate queries is to estimate predicate selectivity.

To optimize an SQL query with multiple selection predicates that have different selectivities and costs, the idea of *predicate migration* [13] is to evaluate the most selective and cheapest predicates first. The intuition of our method is similar. However, we focus on set predicates, instead of the tuple-wise selection predicates studied in [13]. Consequently the concept of "selectivity" in our setting stands for the number of qualified groups, instead of the typical definition based on the number of satisfying tuples.

Our method to estimating set predicate selectivity is a probabilistic approach that exploits histograms in databases. A histogram on an attribute partitions the attribute values from all tuples into disjoint sets called *buckets*. Different histograms vary by partitioning schemes. Some schemes partition by values. In an *equi-width* histogram the range of values in each bucket has equal length. In an *equi-depth* or *equi-height* histogram each bucket has the same number of tuples. Some other schemes partition by value frequencies. One example is *v-optimal* histogram [25].

The histogram on attribute x, $h(x)$, consists of a number of buckets $b_1(x), ..., b_s(x)$. For each bucket $b_i(x)$, the histogram provides its number of distinct values $w_i(x)$ and its depth $d_i(x)$, i.e., the number of tuples in the bucket. The frequency of each value is typically approximated by $\frac{d_i(x)}{w_i(x)}$, based on the *uniform distribution assumption* [15]. If the histogram partitions by frequency (e.g., v-optimal histogram), each bucket directly records $w_i(x)$ and all distinct values in it. If the histogram partitions by sortable values (e.g., equi-width or equi-depth histogram), the number of distinct values $w_i(x)$ is estimated as the width of bucket $b_i(x)$, based on the *continuous value assumption* [15]. That is, $w_i(x)$=$u_i(x)$-$l_i(x)$, where [$l_i(x)$,$u_i(x)$] is the value range of the bucket. When the attribute domain is an uncountably infinite set (e.g., real numbers), $w_i(x)$ can only mean the range size of bucket $b_i(x)$, instead of the number of distinct values in $b_i(x)$.

Given a query with multiple set predicates, we assume histograms are available on the grouping attributes, the set predicate attributes, and attributes involved in selection conditions (WHERE clause). Moreover, we also assume all attributes are independent of each other. For simplicity of discussion, from now on we assume single-attribute grouping and single-attribute set predicate and focus on selectivity estimation of groups. Selectivity estimation for tuples (i.e., selection conditions) can be incorporated by multiplying bucket sizes below by such selectivity. Multi-dimensional histograms, such as *MHIST* [26], can extend the techniques developed in

[3] Therefore our technique does not extend to queries that have both AND and OR in connecting the multiple set predicates.

this section to multi-attribute grouping and multi-attribute set predicate, as well as correlated attributes.

Suppose the grouping attribute is g. The selectivity of an individual set predicate p = v ***op*** $\{v^1, ...v^n\}$, i.e., the number of groups satisfying p, is estimated by the following formula:

$$sel(p) = \sum_{j=1}^{\#g} P(g_j) \tag{1}$$

where $\#g$ is the number of distinct groups, which is estimated by $\#g$=$\sum_i w_i(g)$. $P(g_j)$ is the probability of group g_j satisfying p, assuming the groups are independent of each other.

The histogram on v partitions the tuples in a group into disjoint subgroups. We use R_j to denote the tuples that belong to group g_j, i.e., R_j=$\{r|r{\in}R, r.g{=}g_j\}$. We use R_{ij} to denote the tuples in group g_j whose values on v fall into bucket $b_i(v)$, i.e., R_{ij}=$\{r|r{\in}R_j, r.v \in b_i(v)\}$. Similarly the histogram $h(v)$ divides the values V=$\{v^1, ..., v^n\}$ in predicate p into disjoint subsets $\{V_1, ..., V_s\}$, where V_i=$\{v'|v'{\in}b_i(v), v'{\in}V\}$.

A group g_j satisfies a set predicate on values $\{v^1, ..., v^n\}$ if and only if each R_{ij} satisfies the same set predicate on V_i. Thus we estimate $P(g_j)$, the probability that group g_j satisfies the predicate, by the following formula:

$$P(g_j) = \prod_{i=1}^{s} P_{\textbf{op}}(b_i(v), V_i, R_{ij}) \tag{2}$$

$P_{\textbf{op}}(b_i(v), V_i, R_{ij})$ is the probability that R_{ij} satisfies the same predicate on V_i, based on information in bucket $b_i(v)$. Specifically, $P_{\supseteq}(b_i(v), V_i, R_{ij})$, $P_{\subseteq}(b_i(v), V_i, R_{ij})$, and $P_{=}(b_i(v), V_i, R_{ij})$ are the probabilities that R_{ij} subsumes V_i, R_{ij} is contained by V_i, and R_{ij} equals V_i, respectively, by set semantics.

$P_{\textbf{op}}(b_i(v),\ V_i, R_{ij})$ is estimated based on the number of distinct values in $b_i(v)$, i.e., $w_i(v)$, according to the aforementioned continuous value assumption, the number of values in V_i, and the number of tuples in R_{ij}, i.e.,

$$P_{\textbf{op}}(b_i(v), V_i, R_{ij}) = P_{\textbf{op}}(w_i(v), |V_i|, |R_{ij}|) \tag{3}$$

For the above formula, $w_i(v)$ is stored in bucket $b_i(v)$ itself and $|V_i|$ is straightforward from V and $b_i(v)$. Based on the attribute independence assumption between g and v, the size of R_{ij} can be estimated by the following formula, where $d_k(g)$ and $w_k(g)$ are the depth and width of bucket $b_k(g)$ that contains value g_j:

$$|R_{ij}| = d_i(v) \times \frac{|R_j|}{|R|} = d_i(v) \times \frac{d_k(g)/w_k(g)}{|R|} \tag{4}$$

We do not need to literally calculate $P(g_j)$ for every group in formula (1). If two groups g_{j_1} and g_{j_2} are in the same bucket of g, R_{ij_1} and R_{ij_2} will be of equal size, and thus $P(g_{j_1})$=$P(g_{j_2})$.

We now describe how to estimate $P_{\textbf{op}}(N, M, T)$ (i.e., $w_i(v)$=N, $|V_i|$ = M, $|R_{ij}|$=T), for each operator. Apparently $P_{\textbf{op}}(N, M, 0)$=0. Moreover, $M{\le}N$, by the fashion V was partitioned into $\{V_1, ..., V_s\}$. Note that the estimation is only for set semantics of set predicates.

CONTAIN (*op* is $\supseteq$):

When $M{>}T$, i.e., the number of values in V_i is larger than the number of tuples in R_{ij}, $P(N, M, T)$=0.

When M=1, i.e., there is only one value in V_i, since there are N distinct values in bucket $b_i(v)$, each tuple in group g_j has probability $\frac{1}{N}$ to have that value on attribute v. With totally T tuples in group g_j, the probability that at least one tuple has that value is:

$$P_{\supseteq}(N, 1, T) = 1 - (1 - \frac{1}{N})^T \tag{5}$$

When $M{>}1$, i.e., there are at least two values in V_i, in group g_j the first tuple's value on attribute v has a probability of $\frac{M}{N}$ to be one of the values in V_i. If it indeed belongs to V_i, the problem becomes deriving the probability of $T{-}1$ tuples containing $M{-}1$ values. Otherwise, with probability $1{-}\frac{M}{N}$, the problem becomes deriving the probability of $T{-}1$ tuples containing M values. Hence:

$$\begin{aligned} P_{\supseteq}(N, M, T) = \frac{M}{N} P_{\supseteq}(N, M-1, T-1) \\ +(1 - \frac{M}{N}) P_{\supseteq}(N, M, T-1) \end{aligned} \tag{6}$$

By solving the above recursive formula, we get:

$$P_{\supseteq}(N, M, T) = \frac{1}{N^T} \sum_{r=0}^{M} (-1)^r \left(\begin{array}{c} M \\ r \end{array} \right) (N-r)^T \tag{7}$$

CONTAINED BY (*op* is $\subseteq$):

When $T = 1$, straightforwardly $P(N, M, 1)$ = $\frac{M}{N}$.

When $T > 1$, every tuple in R_{ij} must have one of the values in V_i on attribute v, for the group to satisfy the predicate. Each tuple has the probability of $\frac{M}{N}$ to have one such value on attribute v. Therefore we can derive the following formula:

$$P_{\subseteq}(N, M, T) = (\frac{M}{N})^T \tag{8}$$

EQUAL (*op* is $=$):

Straightforwardly $P(N, 1, T)$ = $\frac{1}{N^T}$ and $P(N, M, T)$=0 if $M{>}T$. For $1{<}M{\le}T$, we can drive the following equation:

$$\begin{aligned} P_{=}(N, M, T) = \frac{M}{N}[P_{=}(N, M, T-1) \\ +P_{=}(N, M-1, T-1)] \end{aligned} \tag{9}$$

That is, for the group to satisfy the predicate, if the first tuple in R_{ij} has one of the values in V_i on attribute v (with probability of $\frac{M}{N}$), the remaining $T{-}1$ tuples should contain either the M or the remaining $M{-}1$ values. Solving this equation, we get:

$$P_{=}(N, M, T) = \frac{1}{N^T} \sum_{r=0}^{M} (-1)^r \left(\begin{array}{c} M \\ r \end{array} \right) (M-r)^T \tag{10}$$

9 Experiments

9.1 Overview and Implementation Details

We conducted experiments on both query processing algorithms (Section 9.2 (A)-(C)) and query optimization techniques (Section 9.2 (D)). We compared the performance of three methods in evaluating set-level comparisons– the aggregate function-based method, the bitmap index-based method, and the method of using regular SQL queries. They are compared

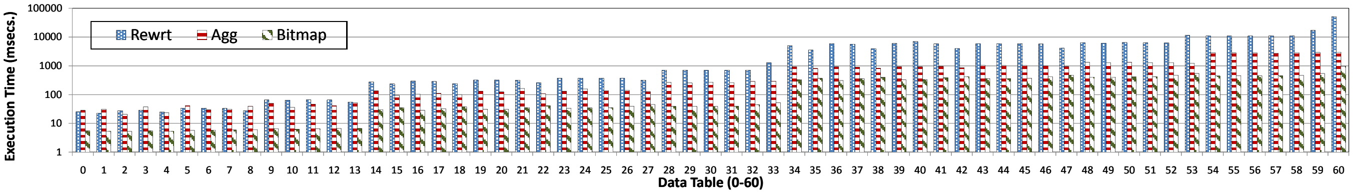


Fig. 5. Overall comparison of the methods, O=$\subseteq$, C=10. (Execution time is in logarithmic scale.)

on three different datasets– (1) Our own synthetic data (Section 9.2 (A)), for studying the effect of various parameters in the performance of these methods, including the number of tuples, the number of groups, the number of values in a set predicate, the number of qualified groups, and so on; (2) TPC-H benchmark database (Section 9.2 (B)), for studying the performance of these methods on general queries with join conditions and on benchmark data capturing the characteristics of decision support applications; (3) WorldCup98 dataset (Section 9.2 (C)), for evaluating the performance of the methods on real and big data.

The aggregate function-based method, denoted as *Agg*, is implemented in C++. The bitmap index-based method, denoted as *Bitmap*, is also implemented in C++ and leverages FastBit [4] for bit-sliced index implementation. The compression scheme of FastBit, Word-Aligned Hybrid (WAH) code, makes the compressed bitmap indices efficient even for high-cardinality attributes [33].

The method of using regular SQL to express set-level comparisons is denoted as *Rewrt*. We used PostgreSQL 8.3.7 to store data and execute regular SQL queries. In the supplemental materials to this paper, we describe how to rewrite queries with set predicate into regular SQL. It is not a complete enumeration of all possible query rewritings because in practice there will be infinite possible rewritings. We made our best effort to express each query by an appropriate regular SQL query and obtain an efficient query plan for the query. This was done by manually investigating alternative queries and plans and turning on/off various physical query operators. Below we report the numbers obtained by these hand-picked plans. Nevertheless, the queries we often used for CONTAINED BY are in the form of the rewriting in Figure 2. For a CONTAIN predicate with m values, we often used a query that intersects the results of m selection queries on the individual values. This rewriting approach can be found in the supplemental materials to this paper.

Note that *Rewrt* uses a full-fledged database engine PostgreSQL, while both *Agg* and *Bitmap* are implemented externally. Although *Rewrt* would incur extra overhead from query optimizer, tuple formatting, etc., we believe this comparison is still insightful. Our results show that *Rewrt* is often one or more orders of magnitude less efficient. It is unlikely that all the slowness comes from extra overheads. Moreover the query plans resulting from regular SQL queries discussed in Section 4 ultimately perform one-pass grouping and aggregation upon the results of (multiple) other upstream operations. Therefore the performance of *Agg*, which is also implemented externally, serves as a yardstick in comparison with the performance of *Bitmap*. Hence the results verify that using regular SQL queries obscures the semantics of set-level comparisons and leads to costly plans. The results could encourage vendors to incorporate the proposed approaches into a database engine.

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

TABLE 1
Configuration parameters of synthetic data experiments.

9.2 Results

The experiments were performed on a Dell PowerEdge 2900 III server with Linux kernel 2.6.27, dual quad-core Xeon 2.0GHz processors, 2x4MB cache, 8GB RAM, and three 146GB 10K-RPM SCSI hard drivers in RAID5. The reported results are the averages of 10 runs. All performance data were obtained with cold buffer.

(A) Comparison over synthetic data:

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

Data: For each of the 3×19 single-predicate queries, we generated 61 data tables, each corresponding to a different combination of (T, G, S) values in Table 1. Given query (O, C) and data statistics (T, G, S), we correspondingly generated a 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 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 disqualified

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

groups are similarly generated, by making sure the groups cannot satisfy the set predicate. 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 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 generate the values randomly from a pool of numbers excluding the missing values.

Results: We measured wall-clock execution time of *Rewrt*, *Agg*, and *Bitmap* over the aforementioned 61 data tables for each of the 3×19 queries. The comparison of these methods under different queries are fairly similar. Hence we only show the results for one query for data table 0-60 in Figure 5: $\gamma_{g,SUM(a)}v \subseteq \{1,\ldots,10\}(R)$. For instance, data table 54 in Figure 5 represents results of the three methods with T=1 million, G=100K, S=100K, under query O=$\subseteq$, C=10. Note that the purpose of the figure is not to compare the performance on different data tables. (Such detailed comparison is provided in Figure **??** in the supplemental materials to this paper.) It is rather to show the performance gap between several algorithms that is consistently observed in all data tables under various queries.

Figure 5 shows 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 expressing set-level comparisons by regular SQL and the difficulty in optimizing such queries. The performance advantage of *Agg* over *Rewrt* shows that the simple query algorithm could improve efficiency significantly. The shown advantage of *Bitmap* over *Agg* is due to fast bit-wise operations and skipping enabled by bitmap index, compared to the verbatim comparisons used by *Agg*.

Impact of the Skewness of Group Sizes: Since the grouping attribute values in the synthetic data were generated by uniform distributions, the groups in a table have about the same size (i.e., number of tuples). To further study the impact of skewness of group sizes on the several methods' performance, we generated more data tables. Given a combination of fixed values on the four configuration parameters C, T, G, and S, we generated 4 different data tables, by varying group size distribution– (1) *Uniform*, where the grouping attribute values follow a uniform distribution, thus the sizes of different groups tend to be equal. Note that this is the same as the data used in Figure 5. (2) *Random*, where the group size is a random variable in a given range. (3) *Exp1* and *Exp2*, where the sizes of groups follow an exponential distribution $(1/2)^n$, in which n is the variable for group size. A small constant is used when a group size generated by the distribution is smaller than 1. *Exp1* and *Exp2* are two opposite cases, in which the sizes of qualified groups are all large and small, respectively.

Figure 6 shows the results of this experiment under C=4, T=1M, G=1K, and S=10, for which the *Uniform* data table corresponds to data table 40 in Figure 5. We can make the following observations. (1) The skewness of group size had large impact on the performance of *Rwrt*, especially for CONTAINED BY and EQUAL operations. Consider CONTAINED BY. The execution time of *Rwrt* decreased when qualified groups are large (*Exp1*). Based on the way the skewed data was generated, the larger the qualified groups are, the more tuples matching the values in set predicates. Therefore the output cardinality of the Filter operation in Figure 2 was substantially reduced under *Exp1*. In contrast, when qualified groups are small (*Exp2*), the Filter operation produced large output, which increased the cost of the query method in this case. For EQUAL operation, the performance can also be analyzed similarly based on the query plan generated, which is omitted here. (2) The skewness of group size did not have much effect on the performance of *Agg*. This is because *Agg* always sequentially scans the whole table, regardless of the set operation and the skewness of group size. (3) The skewness of group size had some impact on the performance of *Bitmap*, although not as much as on *Rewrt*. Consider CONTAIN operation. Step 3 and Step 4 (and thus the whole *Bitmap* method) in Algorithm 2 are more expensive when there are more tuples matching the values in set predicates (*Exp1*). It becomes the opposite case when qualified groups are small (*Exp2*). For all the methods, the performance on *Random* is not much different from that on *Uniform*, because the numbers of tuples matching the values in set predicates only differ slightly in the two data tables.

(B) Experiments over TPC-H data:

Two of the advantages of *Bitmap* mentioned in Section 6 could not be demonstrated by the above experiment. First, it only needs to process 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 5 have schema $R(a,v,g)$ which does not include other columns. We can expect the costs of *Rewrt* and *Agg* to increase by table width, while *Bitmap* will stay unaffected. Second, *Bitmap* enables seamless integration with selections and joins, while the above experiment is on a single table.

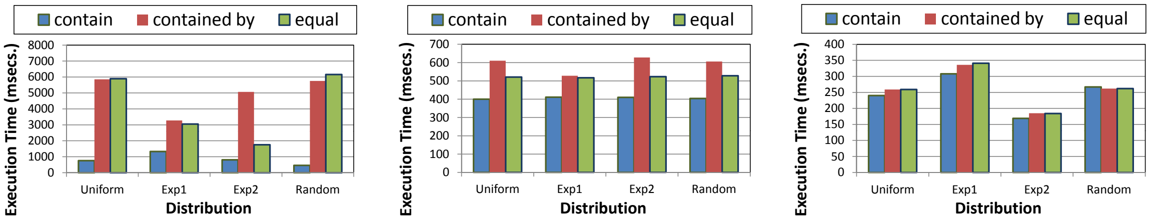
Queries: We thus designed six queries (TPCH-1 below, TPCH-2 to TPCH-6 in the supplemental materials to this paper) on the TPC-H benchmark database [31] and compared the performance of the three methods. In these queries, grouping and set predicates are defined over the join result of multiple tables. Note that the joins are key-foreign key joins. For *Rewrt* and *Agg*, we first joined the tables to generate a single joined table, and then executed the algorithms over that joined table.

(TPCH-1) *Get the total sales of each brand that has business in both USA and Canada.*

```
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'}
```

Data: The data tables were generated by the TPC-H data generator, with scale factors 0.1, 1, and 10, respectively. Table 2 shows the sizes of the original TPC-H tables under



Rewrt *Agg* *Bitmap*

Fig. 6. Execution time of three methods over different skewness of group size.

TPC-H table	Size	Joined table	Size
PART	200000	R1	6001215
PARTSUPP	800000	R2	800000
SUPPLIER	10000	R3	800000
CUSTOMER	150000	R4	800000
NATION	25	R5	800000
LINEITEM	6001215	R6	6001215
ORDERS	1500000		

TABLE 2
Sizes of tables in TPC-H data with scale factor=1.

scale factor 1 and the sizes of the corresponding joined tables used in our experiments, i.e., R1 to R6 in queries TPCH-1 to TPCH-6, respectively. The sizes of data tables under scale factor 0.1 (10) are 10 times smaller (larger) than the sizes of the tables under scale factor 1.

Results: The results are shown in Table 3. Regular B+-tree indices were created on both individual tables and the joined table, to improve query efficiency. Hence in Table 3 we report the costs of *Rewrt* and *Agg* together with the cost of join. For *Bitmap* we created bitmap join index [22] according to key-foreign key joins, based on the description in Section 7 (F). For example, we index 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 other queries, the *Bitmap* method works in the same way as for a single table, without pre-computing joined tables.

Table 3 shows that, even if the tables are already joined for *Rewrt* and *Agg*, *Bitmap* is still often 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. For all three methods (except the join result materialization for *Rewrt* and *Agg*), the execution time grows almost linearly with scale factor from 0.1 to 10.

(C) Experiments over WordCup98 data:

To measure the performance of the three methods under large and skewed data in the real world, we conducted experiments over the WorldCup98 dataset. [5]

Data: The WorldCup98 dataset contains 1,352,804,107 tuples, which correspond to all the access requests made to the 1998 World Cup Website between April 30, 1998 and July 26, 1998. Each tuple records information such as the timestamp of the request, the type of the requested file, the file size, the server that handled the request, the client identifier (which maps to an IP address), and so on. This dataset by nature is skewed. For example, 88.16% of the requests were images, 44.5% of the requests were handled by servers in Plano, TX, and more than 5% of all requests were made on a single day, June 30th [3].

Query	*Rewrt*+Join	*Agg*+Join	*Bitmap*
scale factor = 0.1			
TPCH-1	0.71+14.50 secs.	0.30+14.50 secs.	0.11 secs.
TPCH-2	0.30+0.13 secs.	0.09+0.13 secs.	0.07 secs.
TPCH-3	0.10+0.09 secs.	0.04+0.09 secs.	0.02 secs.
TPCH-4	0.08+0.09 secs.	0.06+0.09 secs.	0.03 secs.
TPCH-5	0.20+0.09 secs.	0.07+0.09 secs.	0.03 secs.
TPCH-6	0.19+1.85 secs.	0.36+1.85 secs.	0.15 secs.
scale factor = 1			
TPCH-1	10.64+31.64 secs.	2.65+31.64 secs.	0.83 secs.
TPCH-2	4.37+1.51 secs.	0.77+1.51 secs.	0.19 secs.
TPCH-3	1.20+1.76 secs.	0.37+1.76 secs.	0.23 secs.
TPCH-4	0.92+0.95 secs.	0.36+0.95 secs.	0.23 secs.
TPCH-5	3.28+0.94 secs.	0.41+0.94 secs.	0.23 secs.
TPCH-6	26.98+7.09 secs.	3.16+7.09 secs.	1.53 secs.
scale factor = 10			
TPCH-1	110.89+710.76 secs.	28.07+710.76 secs.	8.43 secs.
TPCH-2	60.71+22.52 secs.	8.17+22.52 secs.	2.64 secs.
TPCH-3	64.08+24.40 secs.	4.46+24.40 secs.	1.40 secs.
TPCH-4	28.75+25.15 secs.	4.29+25.15 secs.	2.55 secs.
TPCH-5	92.85+30.92 secs.	5.01+30.92 secs.	2.58 secs.
TPCH-6	287.82+566.24 secs.	33.71+566.24 secs.	16.06 secs.

TABLE 3
Results on TPC-H data with different scale factors.

Query	C	S	G	T	Tuples in S
WC98-1	3	1.4K	2.8M	1.4B	77.0M
WC98-2	3	16.8K	89.9K	1.4B	21.2K
WC98-3	3	76.1K	2.8M	1.4B	3.9M

TABLE 4
Characteristics of the WorldCup98 dataset with regard to queries WC98-1, WC98-2, and WC98-3.

Queries: We designed three queries on this dataset, as follows.

(WC98-1) *Find the total traffics for clients who had visited in three consecutive days– July 24th, July 25th, and July 26th.*

```
SELECT clientID, SUM(bytes) GROUP BY clientID
HAVING SET(date) CONTAIN {0724,0725,0726}
```

(WC98-2) *Find the total traffics for files that had only been retrieved from US servers #1, #2, and #3.*

```
SELECT objectID, SUM(bytes) GROUP BY objectID
HAVING SET(server) CONTAINED by {1,2,3}
```

(WC98-3) *Find the total traffics of clients who had accessed file types HTML(1), JPG(2), and GIF(3), but nothing else.*

```
SELECT clientID, SUM(bytes) GROUP BY clientID
HAVING SET(type) EQUAL {1,2,3}
```

Results: The results on the WorldCup98 dataset are shown in Table 5. We observed that while *Rewrt* required hours to finish a query, *Bitmap* and *Agg* only used several minutes. This clearly shows the enlarged performance gains of our methods on billion-tuple dataset.

[5] The WorldCup98 dataset is collected from http://ita.ee.lbl.gov/html/contrib/WorldCup.html.

Query	*Rewrt*	*Agg*	*Bitmap*
WC98-1	16061 secs.	569 secs.	427 secs.
WC98-2	20692 secs.	698 secs.	380 secs.
WC98-3	15571 secs.	689 secs.	468 secs.

TABLE 5
Results on the WorldCup98 dataset.

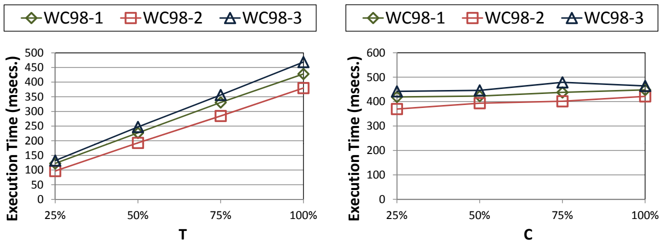


Fig. 7. Execution time of *Bitmap* on the WorldCup98 dataset under different data sizes and query complexities.

We further investigated how *Bitmap* performs under different table sizes and query complexities. We varied number of tuples (T) by using 20%, 50%, 75%, and 100% of the original dataset. We varied number of values in set predicate (C) by using 20%, 50%, 75%, and 100% of the distinct attribute values in the original dataset as the values in set predicate. The results are shown in Figure 7. The execution time of *Bitmap* grew linearly by the data size and grew very slowly by the query complexity.

Below we explain the results in Table 5. For better understanding of the results, we list in Table 4 the characteristics of the dataset with regard to the three queries. In addition to the four variables–number of values in set predicate (C), number of qualified groups (S), number of groups (G), and number of tuples (T)–Table 4 also shows the number of tuples in qualified groups (Tuples in S). Note that the focus of the analysis is to understand the individual results. It is not to say which query is more efficient than others. The three queries are different in set operations (i.e. CONTAIN, CONTAINED BY, and EQUAL), set predicate attributes, and grouping attributes. Hence it is less meaningful to compare the three queries against each other, as the performance of a query can be highly dependent on the parameters *C*, *S*, *G*, *T* and skewness of data.

For *Rewrt*, the performance difference between WC98-1 and WC98-3 was small. WC98-2 was considerably less efficient. From Figure 2, we can see that the query plan needs to perform a set difference operation between the original table and the set of tuples that do not match any set predicate values. In query WC98-2, many tuples did not match the set predicate values. That made the set difference operation expensive. For *Agg*, query WC98-1 took less time than the other two queries. This can be explained by our observation that *Agg* is sensitive to data size T and less sensitive to other parameters and under the same data size CONTAIN operation is more efficient than CONTAINED BY and EQUAL with regard to *Agg* (cf. Figure 6 and supplemental materials). For *Bitmap*, query WC98-2 had the best performance since it had much fewer groups and tuples in qualified groups, in comparison with WC98-1 and WC98-3 (cf. Table 4). This is also consistent with the observation in Section 9.2 (A).

		estimated selectivity	real selectivity
MPQ$_1$	p_{11}	99.88%	95.71%
	p_{12}	62.88%	80.32%
	p_{13}	25.75%	15.79%
MPQ$_2$	p_{21}	99.99%	95.56%
	p_{22}	69.54%	83.81%
	p_{23}	13.16%	9.61%
MPQ$_3$	p_{31}	99.95%	90.09%
	p_{32}	73.51%	35.61%
	p_{33}	13.87%	20.13%

TABLE 6
Comparison of estimated and real selectivity.

	MPQ$_1$ (i=1)	MPQ$_2$ (i=2)	MPQ$_3$ (i=3)
plan$_1$: $p_{i1}p_{i2}p_{i3}$	0.69	0.79	0.68
plan$_2$: $p_{i1}p_{i3}p_{i2}$	0.69	0.79	0.68
plan$_3$: $p_{i2}p_{i1}p_{i3}$	0.69	0.79	0.68
plan$_4$: $p_{i2}p_{i3}p_{i1}$	0.31	0.33	0.16
plan$_5$: $p_{i3}p_{i1}p_{i2}$	0.69	0.79	0.68
plan$_6$: $p_{i3}p_{i2}p_{i1}$	0.32	0.33	0.16

TABLE 7
Execution time of different plans (in seconds).

(D) Selectivity estimation and predicate ordering: We also conducted experiments to verify the accuracy and effectiveness of the selectivity estimation method in Section 8. Here we use the results of three queries (MPQ$_1$, MPQ$_2$, MPQ$_3$), each on a different synthetic data table, to demonstrate. The values of grouping attribute g and set predicate attribute v are independently generated, each following a normal distribution. Each MPQ$_i$ has three conjunctive set predicates, p_{i1}, p_{i2}, and p_{i3}. The predicates are manually chosen so that they have different selectivities, shown in the real selectivity column of Table 6. Predicate p_{i3} is most selective, with 10% to 20% qualified groups; p_{i1} is least selective, with around 90% qualified groups; p_{i2} has a selectivity in between.

To estimate set predicate selectivity, we employed two histograms over g and v, respectively. The data tables have about $40-60$ distinct values in v and 10000 distinct values in g. We built 10 and 100 equi-width buckets, on v and g, respectively. Table 6 shows that the estimated selectivities are sufficiently accurate to capture the order of different predicates by selectivity.

Table 7 shows that our method is effective in choosing efficient query plans. As discussed in Section 8, based on estimated selectivity, our optimization method chooses a plan that evaluates conjunctive predicates in the ascending order of selectivity. The execution terminates early when the evaluated predicates result in empty qualified groups. Given each query MPQ$_i$, there are 6 possible orders in evaluating three predicates, shown as plan$_1-$plan$_6$ in Table 7. Since the order of estimated selectivity is $p_{i3} < p_{i2} < p_{i1}$, our method chooses plan$_6$ over other plans, based on the speculation that it has a better chance to stop the evaluation earlier. Plan$_6$ evaluates p_{i3} first, followed by p_{i2}, and finally p_{i1} if necessary.

In all three queries, the chosen plan$_6$ terminated after p_{i3} and p_{i2}, because no group satisfies both predicates. By contrast, other plans (except plan$_4$) evaluated all predicates. Therefore their execution time is 3 to 4 times of that of plan$_6$. Note that plan$_6$ saves the cost by about 60%, by just avoiding p_{i1} out of 3 predicates. This is due to different evaluation costs of predicates. The least selective predicate, p_{i1}, naturally is also the most expensive one. This indicates that, selectivity and cardinality will be the basis of cost-model in a cost-based query optimizer for set predicates, consistent with the common practice in DBMSs. We also note that plan$_4$ is equally efficient as plan$_6$ for these queries, because they both terminate after p_{i2} and p_{i3} and no plan can stop after only one predicate.

10 Conclusion

We propose to extend SQL by set predicates to support set-level comparisons. Such predicates, combined with grouping, allow selection of dynamically formed groups by comparison between a group and a set of values. We presented two evaluation methods to process set predicates. Comprehensive experiments on synthetic and TPC-H data show the effectiveness of both the aggregate function-based approach and the bitmap index-based approach. For optimizing multi-predicate queries, we designed a histogram-based probabilistic method to estimate the selectivity of set predicates. The estimation governs the evaluation order of multiple predicates, producing efficient query plans.

Acknowledgment

This material is based upon work partially supported by NSF Grant IIS-1018865, CCF-1117369, and 2011, 2012 HP Labs Innovation Research Award. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the funding agencies. The authors would also like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper.

References

[1] Jaql: Query language for javascript object notation (json). http://code.google.com/p/jaql/.

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

[3] M. Arlitt and T. Jin. A workload characterization study of the 1998 world cup web site. *IEEE Network*, 14(3):30 –37, 2000.

[4] D. Chamberlin, M. Astrahan, K. Eswaran, P. Griffiths, R. Lorie, J. Mehl, P. Reisner, and B. Wade. Sequel 2: A unified approach to data definition, manipulation, and control. *IBM Journal of R & D*, 20(6):560 –575, 1976.

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

[6] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, 1977.

[7] D. Chatziantoniou. Using grouping variables to express complex decision support queries. *Data Knowl. Eng.*, 61(1):114–136, 2007.

[8] D. Chatziantoniou and K. A. Ross. Querying multiple features of groups in relational databases. In *VLDB*, pages 295–306, 1996.

[9] D. Chatziantoniou and K. A. Ross. Groupwise processing of relational queries. In *VLDB*, pages 476–485, 1997.

[10] D. Chatziantoniou and E. Tzortzakakis. Asset queries: a declarative alternative to mapreduce. *SIGMOD Rec.*, 38(2):35–41, Oct. 2009.

[11] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 2011.

[12] G. Graefe and R. L. Cole. Fast algorithms for universal quantification in large databases. *ACM TODS*, 20(2), 1995.

[13] J. M. Hellerstein and M. Stonebraker. Predicate migration: optimizing queries with expensive predicates. In *SIGMOD*, pages 267–276, 1993.

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

[15] Y. Ioannidis. The history of histograms (abridged). In *VLDB*, 2003.

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

[17] P.-A. Larso. Grouping and duplicate elimination: Benefits of early aggregation. Technical report, 1997.

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

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

[20] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Commun. ACM*, 54:114–123, June 2011.

[21] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

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

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

[24] G. Özsoyoğlu, Z. M. Özsoyoğlu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM TODS*, 12(4), 1987.

[25] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. *SIGMOD Rec.*, 25(2):294–305, 1996.

[26] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB*, 1997.

[27] K. Ramasamy, J. Patel, R. Kaushik, and J. Naughton. Set containment joins: The good, the bad and the ugly. In *VLDB*, 2000.

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

[29] M. Roth, H. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *TODS*, 1988.

[30] 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.

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

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

[33] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM TODS*, 31(1), 2006.

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

Chengkai Li is an Assistant Professor in the Department of Computer Science and Engineering at the University of Texas at Arlington. His research interests are in the areas of databases, Web data management, data mining, and information retrieval. In particular, he works on computational journalism, database exploration, database testing, entity search and query, and ranking and skyline queries. He received his Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign in 2007, and an M.E. and a B.S. degree in Computer Science from Nanjing University, China, in 2000 and 1997, respectively.

Bin He is a Research Staff Member and Master Inventor at IBM Almaden Research. He got his Ph.D. in the Department of Computer Science at the University of Illinois at Urbana-Champaign in 2006. He also received the M.S. degree in Computer Science from the University of Illinois at Urbana-Champaign in 2002, and M.S. and B.S. degrees in Mathematics from Peking University, China in 2000 and 1998 respectively. He has strong expertise in business intelligence, cloud computing, databases, data warehousing, data mining, and data integration.

Ning Yan received his B.E. degree in software engineering and M.S. degree in computer science from Southeast University in China. He is currently a Ph.D. candidate in the Department of Computer Science and Engineering at the University of Texas at Arlington. His research interests include faceted search, entity retrieval, and database system.

Muhammad Assad Safiullah received his M.S. degree in Computer Science from the University of Texas at Arlington in 2008 and his B.S. degree in Computer Science from the National University of Computer and Emerging Sciences, Islamabad, Pakistan in 2005. His M.S. thesis focused on efficient processing of set-predicate queries using bitmap indexes. He is currently a software engineer at Microsoft Corporation.

Supplemental Materials to “Set Predicates in SQL: Enabling Set-Level Comparisons for Dynamically Formed Groups”

A Details of the Query Rewriting Approach

We can formally prove by relational algebra that a query with set predicates can be translated into standard SQL queries. There could be multiple ways in rewriting a query. We will not enumerate all of them.

First, it is easy to show

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

$\gamma_{\mathcal{G}}\mathcal{C}(R)$ selects the qualified groups. Joining the qualified groups with R on grouping attributes and then computing aggregates will generate the correct query results.

Now we consider the rewriting of $\gamma_{\mathcal{G}}\mathcal{C}(R)$. Given two predicates, i.e., $\mathcal{C}$ = $\mathcal{C}_1$ AND$|$OR $\mathcal{C}_2$, we rewrite for each set predicate separately and then put them together, by using INTERSECT for AND and UNION for OR.

$\gamma_{\mathcal{G}}\mathcal{C}_1$ AND $\mathcal{C}_2$ (R) = $\gamma_{\mathcal{G}}\mathcal{C}_1$ $(R) \cap \gamma_{\mathcal{G}}\mathcal{C}_2$ (R)

$\gamma_{\mathcal{G}}\mathcal{C}_1$ OR $\mathcal{C}_2$ (R) = $\gamma_{\mathcal{G}}\mathcal{C}_1$ $(R) \cup \gamma_{\mathcal{G}}\mathcal{C}_2$ (R)

Given a predicate with a preceding NOT, i.e., $\mathcal{C}$ = NOT $\mathcal{C}'$, we rewrite by using set difference operation.

$\gamma_{\mathcal{G}}$ NOT $\mathcal{C}'$ (R) = $\pi_{\mathcal{G}}(R) - \gamma_{\mathcal{G}}$ $\mathcal{C}'$ (R)

If $\mathcal{C}$ contains multiple predicates $\mathcal{C}_1$ ... $\mathcal{C}_m$ connected by Boolean operators (AND, OR, NOT), we rewrite by keep applying the above algebraic rules.

Now we focus on how to rewrite a single set predicate. A CONTAIN predicate $\mathcal{C} \supseteq \{c_1, ..., c_n\}$ can be rewritten by:

$\gamma_{\mathcal{G}}\mathcal{C} \supseteq \{c_1, ..., c_n\}(R)$ = $\pi_{\mathcal{G}}\sigma_{\mathcal{C}=c_1}(R) \cap ... \cap \pi_{\mathcal{G}}\sigma_{\mathcal{C}=c_n}(R)$

A CONTAINED BY predicate $\mathcal{C} \subseteq \{c_1, ..., c_n\}$ can be rewritten by:

$\gamma_{\mathcal{G}}\mathcal{C} \subseteq \{c_1, ..., c_n\}(R)$ = $\pi_{\mathcal{G}}(R)$ - $\pi_{\mathcal{G}}\sigma_{\mathcal{C}\neq c_1 \wedge ... \wedge \mathcal{C}\neq c_n}(R)$

An EQUAL predicate $\mathcal{C} = \{c_1, ..., c_n\}$, which is a combination of CONTAIN and CONTAINED BY, is rewritten by:

$\gamma_{\mathcal{G}}\mathcal{C} = \{c_1, ..., c_n\}(R)$ =

$(\pi_{\mathcal{G}}\sigma_{\mathcal{C}=c_1}(R) \cap ... \cap \pi_{\mathcal{G}}\sigma_{\mathcal{C}=c_n}(R))$ - $\pi_{\mathcal{G}}\sigma_{\mathcal{C}\neq c_1 \wedge ... \wedge \mathcal{C}\neq c_n}(R)$

Below are some examples to show how to apply the above algebraic rules in query rewriting.

Rewriting CONTAIN: Consider the query Q1 in Section 3, 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 3 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 3 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 3, 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 General Set Predicate Queries (Section 7 Cont'd)

(G) Range-Based Set Predicate: Set predicates on data types such as numeric attributes and dates can use range-based values (e.g., Example 3 in Section 1), in two different ways.

(1) The operand is a set and a value in this set can be a range. For example, predicate SET(size) CONTAIN {[1,10],[25,30)} requires a group to have at least two size values such that the first one is within range [1,10] and the second one is within range [25,30). A group such as {3,4,15,27} would qualify because 3 (4 too) satisfies the first range and 27 satisfies the second range. Similarly {3,15} does not satisfy SET(size) CONTAINED BY {[1,10], [25,30)} because 15 is not in either [1,10] or [25,30).

With regard to the bitmap index-based method, we modify Line 1 and 2 of Algorithm 2. The $QueryBI$ function returns a bit vector for each range, which is naturally supported by both bit-sliced index and regular bitmap indices [28], [32]. For extending the aggregate function-based method, we replace v==v^j by $v \in range^j$ in Line 8 of Algorithm 1.

(2) The whole operand itself is a range. For example, predicate SET(size) CONTAINED BY [1,10] requires the values of size in a qualified group to be subsumed by {1, 2, …, 10}. Another example is SET(size) CONTAIN [1,10] which requires a group to subsume {1, 2, …, 10}. Note that such a CONTAIN predicate is not meaningful on attribute of floating point numbers.

With regard to the bitmap index-based method, we replace Line 1 and 2 of Algorithm 2 by a $QueryBI$ function to obtain a bit vector for the range. For extending the aggregate function-based method, we replace $v == v^j$ by $v \in range$ in Line 8 of Algorithm 1.

(H) Set-Level Comparisons with Subquery Results, between Multiple Groups, and with Partial Satisfaction: The syntax in Section 3 can be extended to allow comparison with not only literal values, but also the result of a subquery. An example query is SELECT student FROM SC GROUP BY student HAVING SET (course) CONTAINED BY (SELECT course_id FROM Courses WHERE dept='CS'). (Finding those students that have only taken CS courses.) The impact of such extension on the proposed evaluation approaches is minimal– The subquery is evaluated first by conventional methods and the original set predicate operand is replaced by the subquery's result values.

Furthermore, the subquery can be correlated. For example, the following query returns those courses that have more diverse students in 2011 than in 2010. Note that, although the syntax can allow correlated subqueries, our proposed methods are not suitable for such extension. In order to find qualifying groups, our algorithms produce bitmap vectors for set predicate operand (returned values from correlated subquery in this case), which in turn depend on correlated qualifying groups.

```
SELECT   SC.course    FROM  SC, Students AS S
WHERE    SC.student=S.name AND SC.year=2011
GROUP BY SC.course
HAVING   SET(S.nationality) CONTAIN
         ( SELECT S.nationality
           FROM   SC AS SC1, Students AS S
           WHERE  SC1.course=SC.course AND
                  SC1.student=S.name AND
                  SC1.year=2010 )
```

In [8], [9], [7], [10] the concept of *grouping variable* was introduced as an SQL extension to allow comparisons of multiple aggregates over the same grouping condition. That line of work only considered regular aggregates such as SUM and COUNT. Combining the concepts of set predicate and grouping variable– set predicates for set-level comparisons and grouping variable for group-wise comparisons– can allow simpler syntax for complex queries. For example, the above student diversity query can be simplified as follows. The simpler query syntax can potentially ease the job of query optimization in producing more efficient query evaluation plans. For instance, the aggregate function-based approach in Section 5 can be adapted to maintain the multiple aggregates during one-pass iteration over tuples.

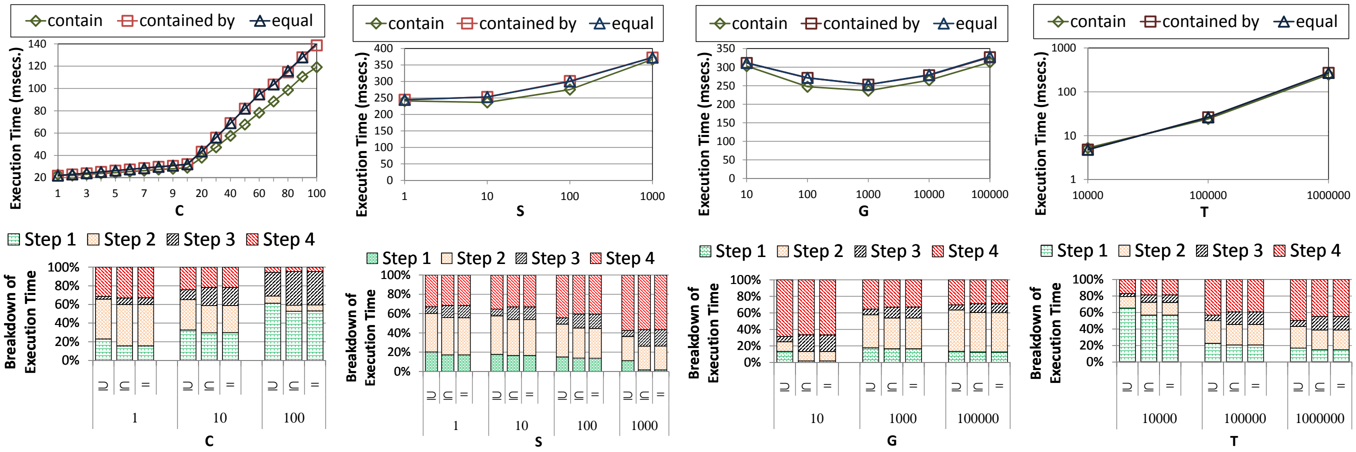
```
SELECT   SC.course   FROM  SC, Students AS S
WHERE    SC.student=S.name
GROUP BY SC.course : X, Y
SUCHTHAT X.year=2010 AND Y.year=2011 AND
SET(X.nationality) CONTAIN SET(Y.nationality)
```

The syntax can also be extended to allow partial satisfaction of set predicates. For example, set predicate SET(skill) CONTAIN 3 OF {'Java', 'Python', 'C++', 'MySQL', 'Web services'} finds the job candidates that have at least 3 of the 5 skills, and predicate SET(course) CONTAINED BY 2 of {'CS101', 'CS102', 'CS103', 'CS104'} identifies the students that have taken no more than 2 of the 4 courses and nothing else. Incorporating partial satisfaction into Algorithm 2 (and similarly Algorithm 1) is quite straightforward. For CONTAIN, it would need to check if k of the n bits in $M[g]$ are set (Line 15). For CONTAINED BY, it needs to maintain both M and G and disqualifies a group g when either some unwanted value is encountered (i.e., current Line 17) or more than k of the n bits in $M[g]$ are set.

C Detailed analysis of synthetic data results

To better understand the performance difference between the three methods, we looked at detailed breakdown of their execution time. Based on the results of 3×19 queries and 61 tables for each query mentioned in Section 9.2(A), we investigated how execution time scales under various groups of configuration parameters C, T, G, and S. In each group, we varied one parameter value and fixed the remaining three. We compared all three methods for all three different operators (O={⊇,⊆,=}). In general, the trend of curve remains fairly similar when we use different values for three fixed parameters and vary the values of the fourth parameter. Hence we only plotted the result for four representative configuration groups.

Detailed Analysis of *Bitmap*: Figure 8 is for *Bitmap* under four configuration groups. For each group, the upper and lower figures show the execution time and its detailed breakdown. Each vertical bar represents the execution time for one particular query (O,C) and the stacked components



(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.

Fig. 8. Execution time of *Bitmap* and its breakdown.

in the bar represent percentages of the costs of all individual steps. *Bitmap* has four major steps, as shown in Algorithm 2. The figures show that no single component dominates. The breakdown varies as configuration parameters change. Hence we shall analyze it in detail while we investigate the effect of parameters below.

Figure 8(a) shows that the execution time of *Bitmap* increases linearly with C (number of values in set predicate). This can be explained by the detailed breakdown in Figure 8(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 value and find qualified groups by considering all values. On the other hand, the costs of Step 2 and 4 have little to do with C, thus get decreasing percentages.

Figure 8(b) shows the execution time of *Bitmap* increases slowly with S (number of qualified groups). With more and more qualified groups, the costs of Step 1-3 increase only moderately since C and G do not change, while Step 4 becomes dominating because it has to calculate aggregates for more and more qualified groups.

As Figure 8(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 S (number of qualified groups) unchanged, less tuples match the values in predicate, resulting in cheaper cost of bit vector operations in Step 1. The number of tuples per group decreases, thus the cost of Step 4 decreases. These two factors lower the overall cost, 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 dominates, making overall cost higher again.

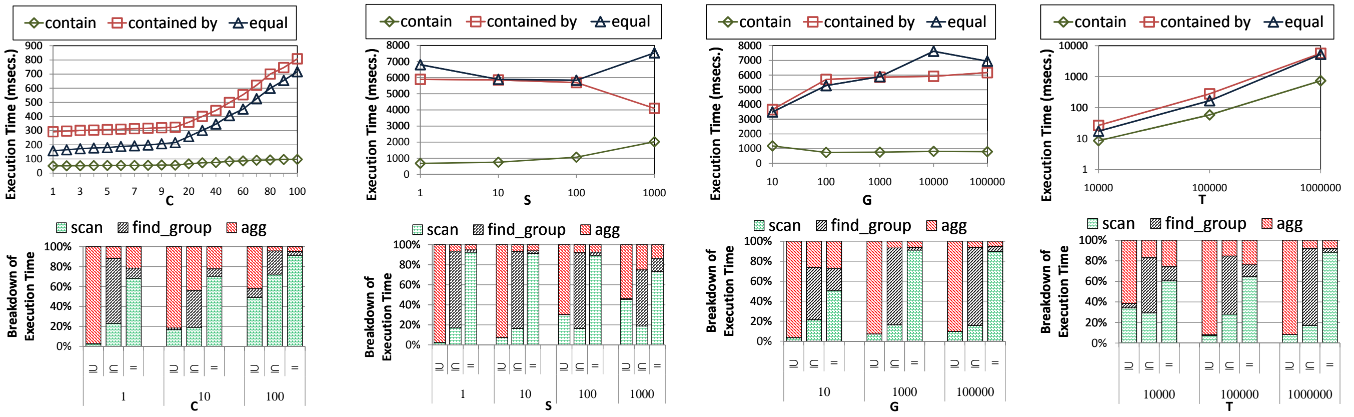
Figure 8(d) shows that *Bitmap* scales linearly with T (number of tuples). When T increases, Step 1 gets less dominating and Step 4 becomes more significant.

Detailed Analysis of *Rewrt* and *Agg*: We divided the *Bitmap* algorithm into four steps in Algorithm 2. Similarly, both *Rewrt* and *Agg* algorithms could also be divided into several steps for detailed study. In *Rewrt*, 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.

Figure 10(a) shows that the execution time of *Rewrt*, similar to that of *Bitmap*, increases linearly with C (number of values in a 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 9. 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 10(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 9 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.

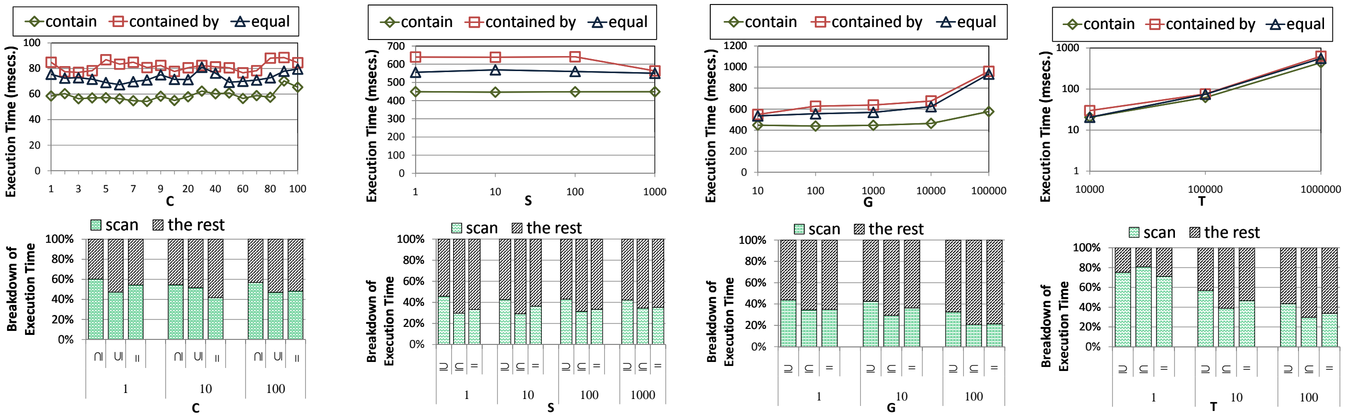
Figure 11(a) shows that the execution time of *Agg*, different from that of *Rewrt* and *Bitmap*, is not affected by C. As Algorithm 1 shows, the only cost component related to C is the checking of a tuple's v value against the given C values in a predicate (line 8 and 11 of Algorithm 1), which is much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

Figure 11(b) shows that the execution time of *Agg* increases with S (number of qualified groups) under some operators and decreases under others, and the changes are only slight in both cases. Figure 10(b) shows that the behavior of *Rewrt* is similar to that of *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



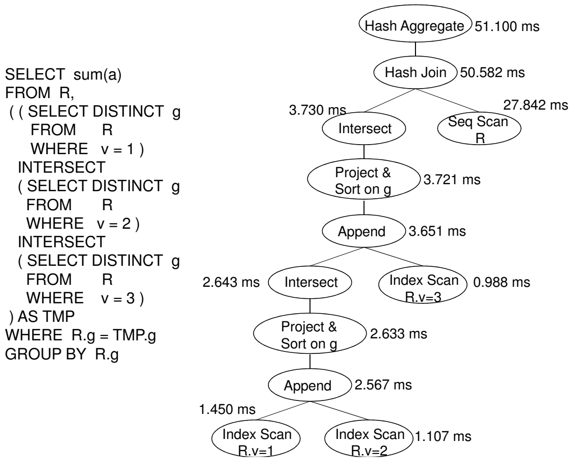
(a) T=100K, G=1K, S=10, varying C. (b) T=1M, G=1K, C=4, varying S. (c) T=1M, S=10, C=4, varying G. (d) G=100, S=10, C=4, varying T.

Fig. 10. 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.

Fig. 11. Execution time of *Agg* and its breakdown.



```
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
```

Fig. 9. *Rewrt* Query Plan for CONTAIN, 100K tuples, 1K groups, 10 qualified groups.

by S. Use Figure 2 to explain again. When more groups satisfy the query condition, the output cardinality of operator Filter becomes smaller, resulting in smaller cost of set difference (Except) operation.

Figure 10(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 qualified 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 11(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 10(d) and Figure 11(d) show that, unsurprisingly, *Rewrt* and *Agg* scale linearly with T (number of tuples), similar to *Rewrt*.

D Other Queries used in TPC-H Experiments

Due to the space limit, we only showed one of the six TPC-H queries in the paper. The other five queries together with their query semantics are given below.

(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’}
```

(TPCH-3) *Get the available quantity of parts for each supplier which provides parts of brand #13 and brand #42 :*

```
CREATE VIEW R3 AS
SELECT PS_SUPPKEY, PS_AVAILQTY, P_BRAND
FROM PARTSUPP, PART
WHERE PS_PARTKEY=P_PARTKEY;

SELECT PS_SUPPKEY, SUM(PS_AVAILQTY)
FROM R3
GROUP BY PS_SUPPKEY
HAVING SET (P_BRAND) CONTAIN {’Brand#13’, ’Brand#42’}
```

(TPCH-4) *Get the available quantity of parts for each supplier which provides parts of size 30, 31, and 32:*

```
CREATE VIEW R4 AS
SELECT PS_SUPPKEY, PS_AVAILQTY, P_SIZE
FROM PARTSUPP, PART
WHERE PS_PARTKEY=P_PARTKEY;

SELECT PS_SUPPKEY, SUM(PS_AVAILQTY)
FROM R4
GROUP BY PS_SUPPKEY
HAVING SET (P_SIZE) CONTAIN {’30’,’31’,’32’}
```

(TPCH-5) *Get the available quantity of parts for each supplier which only provides parts manufactured by MFGR#1-#5:*

```
CREATE VIEW R5 AS
SELECT PS_SUPPKEY, PS_AVAILQTY, P_MFGR
FROM PARTSUPP, PART
WHERE PS_PARTKEY=P_PARTKEY;

SELECT PS_SUPPKEY, SUM(PS_AVAILQTY)
FROM R5
GROUP BY PS_SUPPKEY
HAVING SET (P_MFGR) CONTAINED BY {’MFGR#1’,’MFGR#2’,
       ’MFGR#3’,’MFGR#4’,’MFGR#5’}
```

(TPCH-6) *Get the total price of orders for each lineitem that has orders with exactly 5 different priorities, from low to urgent:*

```
CREATE VIEW R6
SELECT L_LINENUMBER, O_TOTALPRICE, O_ORDERPRIORITY
FROM LINEITEM, ORDERS
WHERE L_ORDERKEY=O_ORDERKEY;

SELECT L_LINENUMBER, SUM(O_TOTALPRICE)
FROM R6
GROUP BY L_LINENUMBER
HAVING SET (O_ORDERPRIORITY) EQUAL {’1-URGENT’,
       ’2-HIGH’,’3-MEDIUM’,’4-NOT SPECIFIED’,’5-LOW’}
```