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

ABSTRACT

Scalar-level predicates in SQL become increasingly inadequate to support a new class of operations that require *set-level* comparison semantics, i.e., comparing a group of tuples with multiple values, especially in data warehousing and OLAP applications. 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 the database engine to optimize, thus can result in costly evaluation.

To address this problem, this paper proposes to augment SQL with a new type of predicates, *set predicates*, to bring out the otherwise obscure semantics. Query writing is an easy-to-deploy but inefficient approach to implement set predicates, as it is outside the query engine. The query engine is unaware of set-level semantics and thus cannot easily optimize. In this paper, we proposed two approaches to handling set predicates within the query processing layer: an aggregate function-based method and a bitmap index-based method. The aggregate function-based approach is generally applicable and often the best choice when a table scan has to be conducted. On the other hand, when bitmap indices are available, which is not uncommon in today's data warehousing oriented databases, the bitmap index-based approach is often the winner. Moreover, we present a histogram-based probabilistic method of set predicate selectivity estimation, for optimizing queries with multiple predicates. The experiments also verify the accuracy of the estimation and its effectiveness in optimizing queries.

1. INTRODUCTION

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

In database design, without explicit support, set-valued attribute is often modeled by a separate table. For example, a table Resume_Skills (id, skill) can be used to connect a set of skills to job candidates in another table. A GROUP BY clause can then be used to dynamically group the tuples by id, with the values on attribute skill in the same group forming a set. The problem is that the current GROUP BY clause can only do scalar value comparison by an accompanying HAVING clause. For instance, the SUM/ COUNT/ AVG/ MAX functions will produce a single numeric value which is compared to a constant or another single aggregate value. A set-level comparison such as the set of skills subsuming {'Java', 'Web services'} cannot be accommodated without complex SQL queries. (Section 3 demonstrates such SQL queries.)

The data modeling literature has long recognized that set-valued attributes provide a concise and natural way to model complex data concepts such as sets [16, 21]. 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 the resume table can be defined as a set data type. Set operations can thus be natively supported on such attributes. Previous research has extensively studied query processing on set-valued attributes and set containment joins [7, 19, 11, 12]. Although set-valued attributes together with set containment joins can be used to support set-level comparisons, this solution has inherent limitations: 1) Set-valued attributes have to be pre-defined at the schema definition time; 2) A set can only cover one attribute and cannot be across multiple attributes.

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

Moreover, the definition of a set may be over multiple attributes. Take the example of online advertisements. The strategist may want to discover the Websites that publish ads for ING with good profit returns, because those Websites are already proven to be highly profitable with bank ads. In this case, the set is defined over both advertiser and profit attributes. In many systems, a set-valued attribute is only defined on a single attribute, thus cannot capture such 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.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

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

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

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

Given the above query, 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.

For the decision making example, suppose we have a table Ratings(department, avg_rating, month, year). The following query will find departments whose monthly average ratings in 2009 have always been poor (assuming ratings are from 1 to 5). Here CONTAINED BY is used to capture the set-level condition.

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

Set predicates can be defined across multiple attributes. Consider the online advertisement example. Suppose the table schema is Site_Statistics(website, advertiser, CTR). The following query finds Websites that publish ads for ING with more than 1% 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,+))}
AND
NOT (SET(advertiser) CONTAIN {'HSBC'})
```

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

Comparing to set-valued attributes and set containment joins [7, 19, 11, 12], set predicates have several critical advantages: (1) Unlike set-valued attributes, which bring significant hassles in redesigning database storage for the special set data type, set predicates require no change in the data representation and storage engine, and thus can be directly incorporated into any standard relational databases; (2) With set predicates, users can dynamically form set-level comparisons with no limitation caused by database schema. On the contrary, for set-valued attributes, sets are statically pre-defined in the schema design phase. Consequently, set-level comparisons can only be issued on set-valued attributes only. (3) Set predicates allow cross-attribute set-level comparison, which is not supported by set-valued attributes.

Another work relevant to set predicates is universal quantification [5]. Universal quantification and relational division are powerful for analyzing many-to-many relationships and set-valued attributes. An example universal quantification query is to find the students that have taken all computer science courses required to graduate. It is therefore a special type of set predicates with CONTAIN operator over a set of constants formed by all the values of an attribute in a table, e.g., Courses. By contrast, the proposed set predicates support CONTAINED BY and EQUAL, in addition to CONTAIN, with much more flexibilities to users.

It is easy to show that the semantics of set predicates can be implemented using existing SQL syntax. That is, we can rewrite an SQL query with set predicates into standard SQL queries. The query rewriting approach is easy to implement in that it does not require modifications inside the query engine. However, as Section 3 will discuss, the rewritten queries can be quite complex for users to formulate and difficult to optimize, thus can result in unnecessarily costly evaluation.

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 native efficient support of such queries in a query processor. In this paper, we developed two approaches to implement set predicates at the query processing layer:

Aggregate function-based approach: We handle a set predicate like an aggregate function. That is, we design specialized query algorithm to process set predicates in a way similar to processing a conventional aggregate function. Given a query with a set predicate, instead of decomposing the query into some subqueries as the query rewritten approach does, the aggregate function based approach only needs one pass of table scan.

Bitmap index-based approach: We notice that as an efficient index for decision support queries, bitmap index has gained broad interests. Nowadays, bitmap index is 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 [22], LucidDB), especially for applications with read-mostly or append-only data, such as OLAP and data warehouses. In these data warehousing oriented database environments, it is not uncommon that bitmap indices are pre-built for each attribute.

The state-of-the-art developments of bitmap compression methods [25, 2, 9] and encoding strategies [3, 26, 15] have substantially broaden the applicability of bitmap index. Today's bitmap index can be applied on all types of attributes (e.g., high-cardinality categorical attributes [24, 25], numeric attributes [24, 15] and text attributes [20]). Studies show that compressed bitmap index occupies less space than the raw data [25, 2] and provides better query performance for equality queries [25], range queries [24], and aggregation queries [15]. Bitmap index has also been used for indexing set-valued attributes [13]. A vector represents a value of a set-valued attribute, where each bit in the vector corresponds to a distinct member of the set.

We thus developed a bitmap index based approach to evaluate set predicates. This approach is efficient because it can focus on only tuples from the groups that satisfy the conditions and only the bitmaps for relevant columns. Our studies also show that when bitmap indices are available, which is not uncommon for today's data warehousing oriented databases, the bitmap index based approach often outperforms other approaches.

Note that, our approach only needs bitmap indices on individual attributes. As we will discuss in Section 5, based on such indices, the approach is able to cope with general queries, dynamic groups, joins, selection conditions, multi-attribute grouping and multiple set predicates. There is no need to pre-compute index for joins, selection results, or combination of attributes.

We further developed an optimization strategy to handle a query with multiple set predicates connected by logic operations (AND, OR, NOT). The naive approach to evaluating such a query is to exhaustively process the predicates and perform set operations over their resulting groups. It can be an overkill. 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 design a method to select a good order of predicate evaluation, i.e, an order that results in a small m, thus a 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 satisfying groups. We thus designed a probabilistic approach to estimating set predicate selectivity by exploiting histograms in databases.

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 2).
- We designed two query evaluation approaches for set predicates, including an aggregate function-based approach (Section 4) and a bitmap index-based approach (Section 5).
- We developed a histogram-based probabilistic method to estimate the selectivity of set predicates, for optimizing queries with multiple predicates. Particularly, our method determines the evaluation order of the predicates and stops early without evaluating unnecessary predicates. (Section 6)
- We conducted extensive experiments to evaluate and compare the approaches we proposed, over both benchmark and synthetic data. We also investigated the effects of various query parameters and data characteristics on these approaches. Moreover, our experiments verify the accuracy and effectiveness of the predicate ordering method. (Section 7)

2. THE SYNTAX OF 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 the 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_predicate ::= **attribute_set set_operator constant_values**
attribute_set ::= SET(v_1, ..., v_m)
set_operator ::= CONTAIN | CONTAINED BY | EQUAL
constant_values ::= $\{(v_1^1, ..., v_m^1), ..., (v_1^n, ..., v_m^n)\}$,

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

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

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

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

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

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

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

Figure 1: A classic student and course example.

Single-Predicate Set Operation: The following query Q1: $\gamma_{student}$ $course \supseteq$\{'CS101','CS102'\}(SC) identifies the students who took both CS101 and CS102. [2] 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 more general query can include regular WHERE selection clause and other aggregate functions in HAVING. In the following Q2: $\gamma_{student,COUNT(*)}$ $course \supseteq$ \{'CS101', 'CS102'\} && AVG($grade$) > 3.5 ($\sigma_{semester='Fall09'}$(SC)), we look for students with average grade higher than 3.5 in FALL09 and have taken both CS101 and CS102 in that semester. It also returns the number of courses they took in that semester.

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

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

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

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

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

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

Multi-Attribute Set Operation: The query syntax also allows comparing sets defined on multiple attributes, e.g., we use `SET(course, grade) CONTAIN` \{`('CS101',4), ('CS102',2)`\} for all the students who received grade 4 in CS101 but 2 in CS102.

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

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

3. GOING BACKWARD: THE QUERY REWRITING APPROACH

An easy-to-deploy approach is to rewrite queries with set predicates into standard SQL queries. The advantage of the rewriting approach is that it requires no change inside the query engine. However, the rewritten query loses the set-level semantics and usually results in complex queries, which are difficult to optimize. From this perspective, the query rewriting approach is essentially contradictory to the motivation of the proposed concise syntax of set predicates, because the final execution plan may go backward to disregard the set semantics. It is easy to notice the drawbacks of lacking direct support for set predicates in query processor.

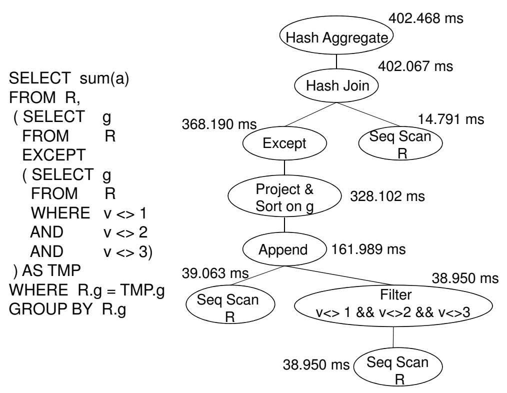
Since rewriting technique is not our focus (Our focus is to make the query processor aware of set predicates.), we only provide a summary here due to space limitations. Details of possible rewriting procedures and examples can be found in the extended version of the paper [1].

First, we prove that queries with set predicates can always be written into standard SQL queries using relational algebra [1]. Second, we realize that there are multiple ways to rewrite even a single set predicate. For instance, for a CONTAIN predicate with m constant values, we can rewrite the query using m-1 INTERSECT operations. Or, we can build a temporary table T with the constant values, join T with the table being queried, and then process a sequence of distinction, group-by, group selection using COUNT operations. As another example, a CONTAINED BY predicate can be rewritten using EXCEPT (as Figure 2 will show), or using subqueries with the CASE condition control. To rewrite more complex queries with multiple predicates, we use the rewriting of individual predicates as the building blocks and connect them together by their logical relationships. If the SELECT clause in the query contains aggregate values, the rewritten query needs to be joined with the original table on the grouping attributes.

Given a set predicate, our experience is that no matter which query rewriting approach we take, it inevitably involves the combination of a number of operations such as JOIN / UNION / INTERSECT / EXCEPT, DISTINCT, GROUP-BY / HAVING. As a result, the performance is usually unsatisfactory. The example below is to illustrate this observation.

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

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



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

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

This is also a good example to indicate that the rewritten query obscures the semantics of set predicate, making it very difficult for the query optimizer to optimize. From this perspective, it is more clear that the query rewriting approach is contradictory to the motivation of the set predicate concept. Thus it is not surprising to see the approach may not perform well. Although this work is not to provide the optimal rewriting strategy, our empirical studies do confirm this belief that the rewriting approach has inherent limitations. Hence, it is critical to investigate new approaches that work at the query processor level to obtain better performance.

4. AGGREGATE FUNCTION-BASED APPROACH

With the new syntax in Section 2, which brings forward the semantics explicitly, a set predicate-aware query plan could potentially be much more efficient by just scanning the table and processing the tuples sequentially. The key to such a direct approach seems to be processing the grouping and the set-level comparison together, through a one-pass iteration of the tuples, instead of using set operations and joins. In light of this idea, we design a method that handles set predicates as aggregate functions. The outline of the method is shown in Algorithm 1.

For the ease of presentation, we first assume the simplest query, $\gamma_{g,\oplus a}\ v$ ***op*** $\{v^1, ..., v^n\}(R)$, i.e., a query with one grouping attribute (g), one aggregate for output ($\oplus a$), and one set predicate defined by a set operator ($\supseteq$, $\subseteq$, or $=$) over a single attribute (v). The sketch in Algorithm 1 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. Follow the common implementation of processing aggregates in database systems, a set predicate is treated as an aggregate function, defined by an initial state (line 3), a state transition stage (line 4-14), and a final calculation stage (line 16-18). The hash table M maintains a mask value for each unique group. The bits in the binary representation of a mask value indicate which of the values v^1, ..., v^n in the set predicate are contained in the corresponding group. If the mask value equals $2^n - 1$, where n is the number of the constant values, the corresponding group contains all the constant values. The hash table G maintains a Boolean value for each group, indicting if it is a qualified group. The values in G were initialized to $True$ for every group if the operator is $\subseteq$ or $=$ (line 3), otherwise $False$. The hash table A maintains the aggregated values for the groups.

In detail, a tuple is skipped if the corresponding group is already identified as disqualified and the operator is $\subseteq$ or $=$. It is also

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 constant 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 $\in \{\subseteq,=\}$)&&(! $G[g]$) 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 $\supseteq$)&&($G[g]$) then continue to next tuple
 8:     if $v$ == $v^j$ for some $j$ then
 9:         /* In $M[g]$, set the mask for the $j$th constant value. */
10:         $M[g] \Leftarrow M[g] \mid 2^{j-1}$
11:         if ($M[g] == 2^n - 1$)&&(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 qualifying groups and their aggregates. */
16: for every group $g$ in hash table $M$ do
17:     if (op is $=$) && ($M[g]$ != $2^n - 1$) then $G[g] \Leftarrow False$
18:     if $G[g]$ then print ($g$, $A[g]$)
```

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 non-skipped tuple, if its value on attribute v matches some v^i in v^1, ..., v^n, we set the ith-bit of the group's mask value to 1, indicating the existence of v^i 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^i, 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 the answer is no, it means the group does not contain all the values v^1, ..., v^n. Therefore we mark the group as disqualified (line 17).

The above algorithm can be extended for general queries with set predicates. As introduced in Section 2, such queries can be denoted by $\gamma_{\mathcal{G},\mathcal{A}}\mathcal{C}(R)$. The aggregates that are in $\mathcal{A}$ (to appear in the SELECT clause) or in the conditions in $\mathcal{C}$ (to appear in HAVING) can be evaluated simultaneously with set predicates. The aggregate $\oplus a$ in Algorithm 1 already demonstrates that. We just need to accumulate all the aggregates in $\mathcal{A}$ at line 6. [3]

The algorithm described above is a one-pass algorithm where memory is available for storing the hash tables for all the groups. We have only implemented and experimented with such an 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 the memory, we can adopt the standard two-pass hash-based or sort-based aggregation method in DBMSs. In the first pass the table is sorted or partitioned by a hash function. In the second pass the tuples in the same group are loaded into memory and aggregations over different groups are handled independently.

Such two-pass method can be further improved by *early aggregation* strategies [10]. In early aggregation, intermediate aggregation results are computed in the sorting or hashing step, so that the final step computes over those intermediate results instead of original tuples. To make early aggregation algorithms applicable for a set predicate, we would need to define the intermediate aggregation results to be kept. For instance, given query $\gamma_{g,\oplus a}\ v \supseteq \{v^1, ..., v^n\}(R)$, we can define a bitmap variable $supset$, which maps v^i to its $(i-1)$th bit. For a single tuple, if it matches with v^i, then $supset$=1 $<<$ $(i-1)$, else $supset$ = 0. To combine two $supset$ values (for both aggregating the tuples in each subset and aggregating the intermediate results from the subsets of the same group), $supset$ = $supset_1 \cup supset_2$. In the final step, if $supset$ & C equals C, the corresponding group satisfies the set predicate, else it is disqualified. Here C is a bitmap constant with all n bits set to 1.

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

5. BITMAP INDEX-BASED APPRAOCH

Our second approach is based on bitmap index [14, 15]. Given a bitmap index on an attribute, there exists a bitmap (a vector of bits) for each unique attribute value. The length of the vector equals the number of tuples in the indexed relation. In the vector for value x of attribute v, its ith bit is set to 1, when and only when the value of v on the ith tuple is x, otherwise 0. With bitmap indices, complex selection queries can be efficiently answered by bitwise operations (AND (&), OR(|), XOR(ˆ), and NOT(∼)) over the bit vectors. Moreover, bitmap indices enable efficient computation of aggregates such as SUM and COUNT [15].

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

As we discussed in Section 1, the latest bitmap compression methods [25, 2, 9] and encoding strategies [3, 26, 15] have substantially broaden the applicability of bitmap index on all types of attributes. Today, bitmap index becomes more and more popular as an efficient index method for data warehousing and decision making applications. Many data warehousing oriented databases, such as column oriented databases, or OLAP query processing engines, use bitmap index as the default indexing method for each attribute. In those environments, it is not uncommon to assume that bitmap indices are pre-built by the databases.

With the aforementioned effective compression and encoding schemes, it is affordable to build bitmap indexes on many attributes. Moreover, index selection based on query workload could allow a system to selectively create indexes on attributes that are more likely to be used in queries.

Some systems (e.g., DB2, PostgreSQL) only build bitmap indices on the fly at query-time when the query optimizer determines they are beneficial. We do not consider such scenario and we focus

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

```
Input:  Table $R(g, a, v)$ with $t$ tuples;
        Query $Q = \gamma_{g,\oplus a}\ v$ op $\{v^1, ..., v^n\}(R)$;
        bit-sliced index BSI($g$), BSI($a$), and bitmap index BI($v$).
Output: Qualifying groups $g$ and their aggregate values $\oplus a$
    /* $gID$: array of size $t$, storing the group ID of each tuple */
    /* $A$: hash table for aggregate values */
    /* $M$: hash table for constant value masks */
    /* $G$: hash table for Boolean indicators of qualifying groups */
    /* Step 1. get the vector for each constant in the predicate */
 1: for each $v^j$ do
 2:     $vec_{v^j} \Leftarrow$ QUERYBI (BI($v$), $v^j$)
    /* Step 2. get the group ID for each tuple */
 3: Initialize $gID$ to all zero
 4: for each bit slice $B_i$ in BSI($g$), $i$ from 0 to $s$-1 do
 5:     for each set bit $b_k$ in bit vector $B_i$ do
 6:         $gID[k] \Leftarrow gID[k] + 2^i$
 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:         $M[gID[k]] \Leftarrow 0$; $G[gID[k]] \Leftarrow False$; also initialize
            $A[gID[k]]$ according to the aggregate function.
    /* Step 3. find qualifying 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 $\sim(vec_{v^1} \mid ... \mid vec_{v^n})$ do
18:         $G[gID[k]] \Leftarrow False$
    /* Step 4. aggregate the values of $a$ for qualifying 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 print ($g$, $A[g]$)
```

on bitmap indexes that are built before query time.

5.1 Basic Set Predicate Queries

In presenting the bitmap index-based approach, we first focus on the simplest case– a query with only one set predicate, $\gamma_{g,\oplus a}\ v$ ***op*** $\{v^1, ..., v^n\}(R)$, where ***op*** can be any of the set operators $\supseteq$, $\subseteq$, and $=$. Section 5.2 discusses the extension to more general queries.

One particular type of bitmap index that we use is *bit-sliced index* (BSI) [20]. 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 unique categorical values to integer numbers.

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

Example 1: Given the data in Figure 1 and query $\gamma_{student,AVG(grade)}$ $course$ ***op*** {'CS101','CS102'}(SC), Figure 3 shows the bitmap in-

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;Tom=>1;John=>2

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

dexes 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 unique attribute value has a corresponding bit vector. For instance, the bit vector B_{CS101} is 1000100, indicating that the 1st and the 5th tuples have 'CS101' as the value of *course*.

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

Step 2 is to get the group IDs, i.e., the values of attribute g, for the tuples in R, by querying BSI(g). The group IDs are calculated by iterating through the slices of BSI(g) and summing up the corresponding values for tuples with bits set in these vectors. (See the example of BSI(student) in Example 1.)

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

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

CONTAIN ($\supseteq$): In Step 3, a hash table M maintains a mask value $M[g]$ for each group g. The mask value has n bits, corresponding to the n constant values (v^1, ..., v^n) in the set predicate. A bit is set to 1 if there exists at least one tuple in group g that has the corresponding constant value on attribute v. Therefore for each set bit b_k in vector vec_{v^j}, the jth bit of $M[gID[k]]$ will be set by a bitwise OR operation (line 13 of Algorithm 2). Therefore if $M[g]$ equals 2^n-1 at the end, i.e., all the n bits are set, the group g contains tuples matching every constant value in the set predicate, thus satisfies the query (line 15). We use another hash table G to record the Boolean indicators of qualifying 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 $\sim(vec_{v^1} \mid ... \mid vec_{v^n})$ (i.e., the kth tuple does not match any constant v^j), the corresponding group $gID[k]$ could not satisfy the query condition and thus will be 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 2: Suppose the query is $\gamma_{student,AVG(grade)}$ $course = \{$'CS101','CS102'$\}(SC)$, using the bitmap indexes in Figure 3. After the 1st bit of vec_{CS101} (i.e., v^1='CS101') is encountered in line 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 $\sim(vec_{CS101} \mid vec_{CS102})$ 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').

5.2 General Queries with Set Predicates

A general query $\gamma_{\mathcal{G},\mathcal{A}}\mathcal{C}(R)$ may use multiple grouping attributes and multiple set predicates, and each predicate may be defined on multiple attributes. Below we discuss how to extend the method in Section 5.1 for such general queries. A query may also return multiple aggregates, e.g., $\gamma_{g,\oplus_1 a,\oplus_2 b}$ v ***op*** $\{v^1, ..., v^n\}(R)$, which can be simply handled by repeating the Step 4 of Algorithm 2 for multiple aggregates. Thus such case will not be further discussed.

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

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

Multi-Predicate Set Operation: Given a query with multiple set predicates, the straightforward approach is to evaluate the individual predicates independently to obtain the qualifying groups for each predicate. Then we can follow the logic operations between the predicates (AND, OR, NOT) to perform the intersection, union, and difference operations over the qualifying groups. Different orders of evaluating the predicates can lead to different evaluation costs, due to their different "selectivities", i.e., the number of qualifying groups. We discuss this query optimization issue in Section 6.

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

Integration and Interaction with Conventional SQL Operations: In a general query $\gamma_{\mathcal{G},\mathcal{A}}\mathcal{C}(R)$, 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 the bitmap index-based solutions for selection conditions [3, 26, 15] and join queries [14]. That is, only bitmap-index on the underlying tables (instead of the dynamic result R) is required. In Section 7.2 we present the experimental results of such general queries on TPC-H data [23]. The detailed method of dealing with general queries is in [1].

6. OPTIMIZING QUERIES WITH MULTIPLE SET PREDICATES: SELECTIVITY ESTIMATION BY HISTOGRAM

The straightforward approach to evaluate a query with multiple set predicates is to exhaustively process the predicates and perform set operations over their resulting groups. However, this approach can be an overkill. An obvious example is query $\gamma_{g,\oplus a}$ $v \supseteq \{1\}(R)$ AND $v \supseteq \{1, 2\}(R)$. The constant set of the first predicate is a subset of the second constant set. Evaluating the first predicate is unnecessary because its satisfying groups are always a superset of the second predicate's satisfying 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 constant sets are disjoint. Without evaluating either predicate, we can safely report empty result. These examples show that, given multiple predicates 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' constant sets. We do not further elaborate on such logical optimization since query minimization and equivalence [4] has been a well-established database topic.

The above 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 satisfying 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 the query evaluation, without processing the 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 the smaller the value m is, the cheaper the evaluation cost is. (The cost difference between predicates will also have an impact. We are assuming 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 the evaluation order of predicates. For instance, suppose the query has three conjunctive predicates p_1, p_2, p_3, which are satisfied by 10%, 50%, and 90% of the 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 satisfying groups after p_1 and p_2 are evaluated. Therefore 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 a good efficient one is slim, not to mention the optimal evaluation plan. This is a typical scenario in query optimization. Our goal is thus to design a method 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.[4] Our query optimization rule

[4]Therefore our technique does not extend to queries that have both

is to evaluate conjunctive (disjunctive) predicates in the ascending (descending) order of selectivities, where the selectivity of a predicate is its number of satisfying groups. Therefore the key challenge in optimizing multi-predicate queries becomes estimating predicate selectivity.

To optimize an SQL query with multiple selection predicates that have different selectivities and costs, the idea of *predicate migration* [6] is to evaluate the most selective and cheapest predicates first. The intuition of our method is similar to that. However, we focus on set predicates, instead of the tuple-wise selection predicates studied in [6]. Consequently the concept of "selectivity" in our setting stands for the number of satisfying groups, instead of the typical definition based on the number of satisfying tuples.

Our method of estimating the selectivity of set predicate is a probabilistic approach that exploits histograms in databases. Histogram is widely used in cost estimation of query plans [8]. 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 the buckets have the same number of tuples. Some other schemes partition by value frequencies. One example is the *v-optimal* histogram [17], where the buckets have minimum internal variances of value frequencies.

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* [8]. If the histogram partitions by frequency (e.g., v-optimal histogram), each bucket directly records $w_i(x)$ and all the 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* [8]. 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 the 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 and the set predicate attributes. Multi-dimensional histograms, such as *MHIST* [18], capture the correlation between attributes. Therefore the techniques developed in this section can be extended to the scenario of multi-attribute grouping and multi-attribute set predicate, by using multi-dimensional histograms. For simplicity of discussion, from now on we assume single-attribute grouping and single-attribute set predicate. Moreover, we also assume the grouping and set predicate attributes are independent of each other. Multi-dimensional histograms can extend our techniques to such scenario as well.

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 constant 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 constant 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:

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

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

$P_{\mathbf{op}}(b_i(v), V_i, R_{ij})$ 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 constant values in V_i, and the number of tuples in R_{ij}, i.e.,

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

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 the bucket $b_k(g)$ that contains value g_j:

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

Note that 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 $P_{\mathbf{op}}(N, M, T)$ (i.e., $w_i(v)$=N, $|V_i|=M$, $|R_{ij}|$=T), for each operator. Apparently $P_{\mathbf{op}}(N, M, 0)$=0. Note that $M{\le}N$, by the way V was partitioned into $\{V_1, ..., V_s\}$.

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 constant in V_i, since there are N distinct values in bucket $b_i(v)$, each tuple in group g_j has the probability of $\frac{1}{N}$ to have the value of that constant 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 constants 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. Therefore we have:

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

By solving the above recursive formula, we get:

$$P_{\supseteq}(N, M, T) = \frac{1}{N^T} \sum_{r=0}^{M} (-1)^r \binom{M}{r} (N-r)^T \tag{7}$$

AND and OR in connecting the multiple set predicates.

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 $1{<}M{\leq}T$, 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 constant 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 $M{-}1$ constant values. Solving this equation, we get

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

7. EXPERIMENTS

7.1 Experimental Settings

We conducted experiments to evaluate the effectiveness of the methods we developed. We also investigated how the methods are affected by important factors with a variety of configurations. The experiments were conducted on a Dell PowerEdge 2900 III server 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 the performance data were obtained with cold buffer.

The aggregate function based method, denoted as *Agg*, is implemented in C++. In our experiments, we always allocate enough memory so that we did not experiment with the scenario when the number of groups exceeds what the memory can accommodate. Thus it was unnecessary to use two-pass hashing-based or sorting-based methods for aggregation.

The bitmap index based method, denoted as *Bitmap*, is also implemented in C++ and leverages FastBit [5] 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 [25]. The index is pre-built before query time and stored into index files.

Our goals are: 1) To demonstrate that implementing set predicates at the query processor layer (instead of by query rewriting) is the better choice from the performance perspective. 2) To show the effectiveness of *Agg* as a general method to handle set predicates. 3) To illustrate that when bitmap indices are available, *Bitmap* often has the best performance.

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

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

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

Table 1: Configuration parameters.

Data Tables: 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. Due to space limitations, more details on how the column values were generated are in [1].

7.2 Experimental Results

(A) Overall comparison of the methods:

We first did an overall performance comparison of *Agg* and *Bitmap*. To illustrate, we also added the performance of the query rewriting approach, denoted as *Rewrt*. We used PostgreSQL 8.3.7 to store the data and execute the rewritten queries. PostgreSQL provides several table scan and join methods. We made our best effort to obtain the most efficient query plan for each query, by manually investigating alternative plans and turning on/off various physical query operators. Below we only report the numbers obtained by using these hand-picked plans.

Note that we are aware that *Rewrt* uses a full-fledged database engine PostgreSQL, while both *Agg* and *Bitmap* are implemented externally. The comparison between their results is not totally fair, as *Rewrt* would incur extra overhead from query optimizer, tuple formatting, and so on. On the other hand, our argument is that set predicates are used in OLAP queries, which often takes much longer query processing time than OLTP queries. Our observation is that for OLAP queries, PostgreSQL will spend most time on real query processing, and the extra overheads will usually not dominate. Our goal is to show that as *Rewrt* is often one order of magnitude (or even more) slower, it is unlikely that all the slowness comes from the extra overheads, considering the nature of OLAP queries. Thus, we believe the claim that implementing set predicates at query processing layer has better performance is still valid. The results could encourage database vendors to incorporate the proposed approaches into a database engine.

We measure the 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 pretty similar. Therefore we only show the result for one query in Figure 4: $\gamma_{g,SUM(a)} v \subseteq \{1, \ldots, 10\}(R)$. The top part of Figure 4 shows the results for data table 0 to data table 30, and the bottom part shows the results for data table 31 to 60. For instance, data table 54 in Figure 4 is for 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 comparison is provided in Figure 5.) It is rather to show the performance gap between several algorithms that is consistently observed in all data tables.

Figure 4 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 the translated queries and the incompetence of PostgreSQL in processing such queries, even though the semantics being processed is fairly

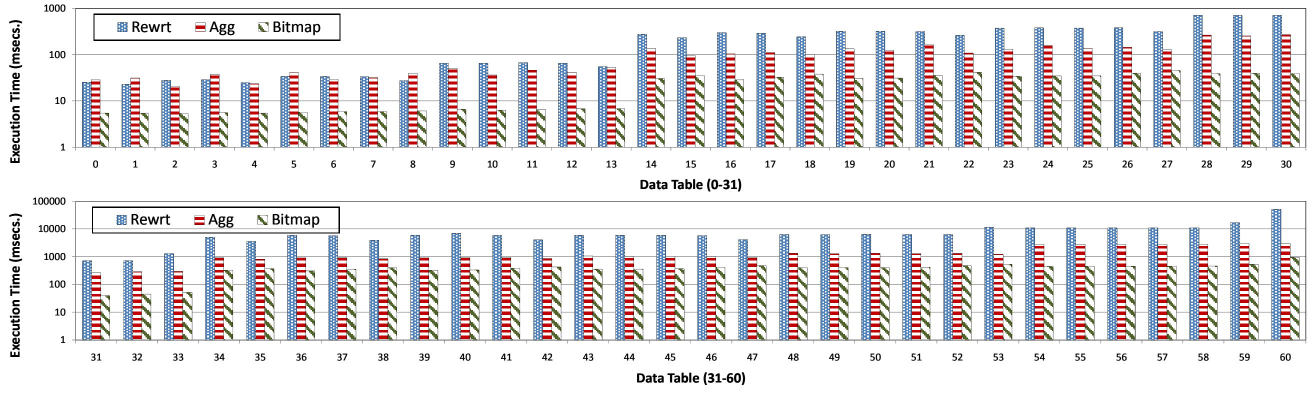


Figure 4: Overall comparison of the methods, O=$\subseteq$, C=10.

Query	*Rewrt*+Join	*Agg*+Join	*Bitmap*
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.

Table 2: Results on general queries.

simple. The performance advantage of *Agg* over *Rewrt* shows that the simple query algorithm could improve the efficiency significantly. The shown advantage of *Bitmap* over *Agg* is due to the fast bit-wise operations and the skipping of chunks of 0s enabled by bitmap index, compared to the verbatim comparisons used by *Agg*.

(B) Experiments on general queries:
Two of the advantages of *Bitmap* mentioned in Section 5.2 could not be demonstrated by the above experiment. First, it only needs to process the necessary columns, while *Agg* (and *Rewrt*) have to scan the full table before irrelevant columns can be projected out. The tables used in the experiments for Figure 4 have the schema $R(a, v, g)$ which does not include other columns. We can expect the costs of *Rewrt* and *Agg* to increase by the width of the table, while *Bitmap* will stay unaffected. Second, the *Bitmap* method also enables convenient and efficient integration with selections and joins, while the above experiment is on a single table.

We thus designed queries on TPC-H benchmark database [23] and compared the performances of the three methods. The results of six such queries are in Table 2. The first test query TPCH-1 is described below and the remaining queries can be found in [1].

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

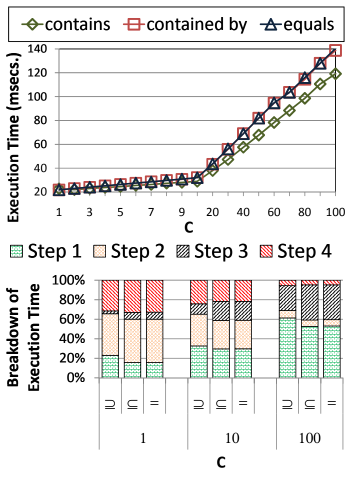
```
CREATE VIEW R1 AS
select P_BRAND, L_QUANTITY, N_NAME
FROM LINEITEM, ORDERS, CUSTOMER, PART, NATION
WHERE L_ORDERKEY=O_ORDERKEY
    AND O_CUSTKEY=C_CUSTKEY
    AND C_NATIONKEY=N_NATIONKEY
    AND L_PARTKEY=P_PARTKEY;

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

In these queries, grouping and set predicates are defined over the join result of multiple tables. Note that the joins are star-joins between keys and foreign keys. For *Rewrt* and *Agg*, we first pre-joined the tables to generate a single joined table, and then executed the algorithms over that joined table. The data tables were generated by TPC-H data generator. The joined table has about 6 million tuples for TPCH-1 and TPCH-6 and 0.8 million tuples for the other 4 queries. The sizes of individual tables and other detailed information are in [1]. Regular B+ tree indexes were created on both individual tables and the joined table, to improve query efficiency. Hence in Table 2 we report the costs of *Rewrt* and *Agg* together with the costs of pre-join. For *Bitmap* we created bitmap join index [14] according to the foreign key joins in the query, based on the description in Section 5.2. 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 similarly other queries), the *Bitmap* method works in the same way as for a single table, without pre-joining the tables.

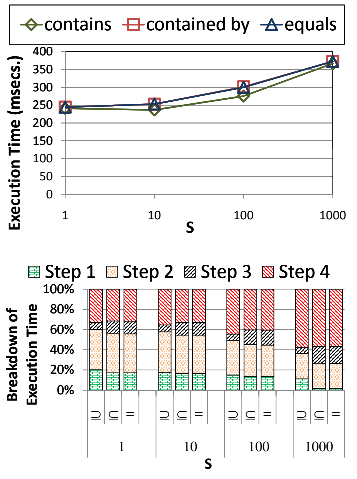
Table 2 shows that, even with the tables pre-joined for *Rewrt* and *Agg*, *Bitmap* is still often 3-4 times faster than *Agg* and more than 10 times faster than *Rewrt*. If we consider the cost of join, the performance gain is even more significant.

(C) Detailed breakdown and the effect of parameters:
To better understand the performance difference between the three methods, we looked at the detailed breakdown of their execution time. We further investigated the effect of various configuration parameters. In these experiments, we measured the execution time under various groups of configurations of four relevant parameters, C, T, G, and S. In each group of experiments, we varied the value of one parameter and fixed the remaining three. We then executed all three methods, for all 3 operators. 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. Therefore we only report the result from four representative configuration groups, varying one parameter in each group. Due to space limitations, we only report the results for *Bitmap* in Figure 5. The detailed breakdown for *Rewrt* and *Agg* are in [1].

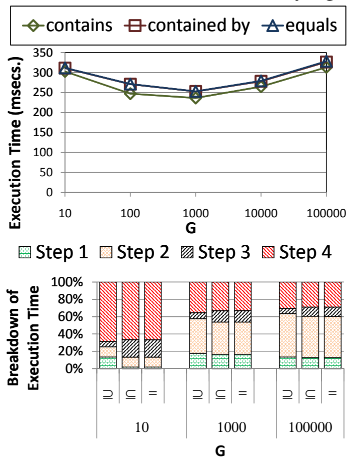
Figure 5 shows the results of four configuration groups for *Bitmap*. For each group, the upper and lower figures show the execution time and its detailed breakdown. Each vertical bar represents the execution time given one particular query (O,C) and the staked components of the bar represent the percentages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



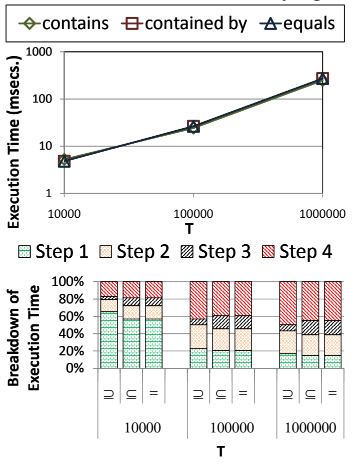
(a) T=100K, G=1K, S=10, varying C.



(b) T=1M, G=1K, C=4, varying S.



(c) T=1M, S=10, C=4, varying G.



(d) G=100, S=10, C=4, varying T.

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

the effect of the configuration parameters below.

Figure 5(a) shows that the execution time of *Bitmap* increases linearly with C (number of constant values in set predicate). (Note that the values of C increase by 1 initially and by 10 later, therefore the curves appear to have an abrupt change at C=10, although they are linear.) This observation can be explained by the detailed breakdown in Figure 5(a). The costs of Step 1 and Step 3 increase as C increases, thus get higher and higher percentages in the breakdown. This is because the method needs to obtain the corresponding vector for each constant and find the qualifying groups by considering all the constants. On the other hand, the costs of Step 2 and 4 have little to do with C, thus get decreasing percentages.

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

As Figure 5(c) shows, the cost of *Bitmap* does not change significantly with G (number of groups). However, the curves do show that the method is least efficient when there are very many or very few groups. When G increases, with the number of satisfying groups (S) unchanged, the number of tuples matching the constants becomes smaller, resulting in cheaper cost of bit vector operations in Step 1. The number of tuples per group decreases, thus the cost of Step 4 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 becomes dominating, making the overall cost higher again.

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

(D) Selectivity Estimation and Predicate Ordering:
We also conducted experiments to verify the accuracy and effectiveness of the selectivity estimation method in Section 6. 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 3. Predicate p_{i3} is most selective, with 10% to 20% satisfying groups; p_{i1} is least selective, with around 90% satisfying 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 3 shows that the method of Section 6 produces selectivity estimation that is sufficiently accurate to capture the order of different predicates by their selectivity.

		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 3: 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 4: Comparison of execution time of different plans (in seconds).

Table 4 shows that our method is effective in choosing efficient query plans. As discussed in section 6, based on the 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 satisfying groups. Given each query MPQ$_i$, there are 6 possible orders in evaluating the three predicates, shown as plan$_1$−plan$_6$ in table 4. 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. Developing such a cost model is in our future work plan. 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.

8. CONCLUSION

We propose to extend SQL to support set-level comparisons by a new type of set predicates. 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 queries with multiple set predicates, 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.

9. REFERENCES

[1] Ananymous. To comply with double-blind reviewing.

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

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

[4] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.

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

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

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

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

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

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

[11] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *SIGMOD Conference*, 2003.

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

[13] M. Morzy, T. Morzy, A. Nanopoulos, and Y. Manolopoulos. Hierarchical bitmap index: An efficient and scalable indexing technique for set-valued attributes. In *Proc. of East-European Conference on Advances in Databases and Information Systems*, pages 236–252, 2003.

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

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

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

[17] 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.

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

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

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

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

[22] 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.

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

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

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

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