Efficient Detection of Empty-Result Queries

Gang Luo
IBM T.J. Watson Research Center
luog@us.ibm.com

Abstract

Frequently encountered in query processing, empty query results usually do not provide users with much useful information. Yet, users might still have to wait for a long time before they disappointingly realize that their results are empty. To significantly reduce such unfavorable delays, in this paper, we propose a novel method to quickly detect, without actual execution, those queries that will return empty results. Our key idea is to remember and reuse the results from previously-executed, empty-result queries. These results are stored in the form of so-called atomic query parts so that the (partial) results from multiple queries can be combined together to handle a new query without incurring much overhead. To increase our chance of detecting empty-result queries with only a limited storage, our method (1) stores the most "valuable" information about empty-result queries, (2) removes redundant information among different empty-result queries, (3) continuously updates the stored information to adapt to the current query pattern, and (4) utilizes a set of special properties of empty results. We evaluate the efficiency of our method through a theoretical analysis and an initial implementation in PostgreSQL. The results show that our method has low overhead and can often successfully avoid executing empty-result queries.

1. Introduction

DBMSs are being used more and more for interactive exploration [7, 14, 37], where users keep refining queries based on previous query results. Due to the large data set size, users are forced to form rather precise queries in order to avoid getting too many query results. Unfortunately, this often causes queries to return empty result sets, which usually do not provide users with much useful information [19, 26]. This is especially true at the beginning, when users do not have sufficient knowledge of the data set. Even worse, users cannot discover empty result sets until query execution finishes, which can take a long time. We call this the *empty-result* problem [19].

Empty results are frequently encountered in query processing. For example, in a query trace that contains 18,793 SQL queries and is collected in a Customer Relationship Management (CRM) database application developed by IBM, 18.07% (3,396) queries are empty-result ones. In another real estate database application developed by IBM, 5.75% SQL queries are discovered to return empty result sets. As a third example, [17] and [30] reported that the percentages of empty-result queries are 10.53% and 38%, respectively.

One might think that empty-result queries can finish in a short amount of time. However, this is often not the case. For example, consider a query that joins two relations (possibly after some selection and projection). Regardless of whether the query result set is empty, the query execution time will be longer than the time required to do the join. Even if this query can finish execution in a few seconds in a lightly loaded RDBMS, it can last longer than a minute in a heavily loaded RDBMS.

In general, it is desirable to quickly detect empty-result queries. Not only does it facilitate the exploration of massive data sets but also it provides important benefits to users. First, users can quickly realize that they encounter the empty-result problem rather than waiting for the empty result. With a much shortened latency time, users can quickly go ahead with other trials. Second, by avoiding the unnecessary execution of empty-result queries, the load on the RDBMS can be reduced, thus further improving the system performance.

In this paper, we propose a novel method for fast detection of empty-result queries. To the best of our knowledge, this direction of handling the empty-result problem has never been explored before. We observe that as users often submit similar queries, the probability that they can reuse each other's query results is usually high. For example, among the 3,396 empty-result queries collected from the CRM database application at IBM, only 1,287 queries are distinct. All the other 2,109 queries are repeated ones. Hence, we reuse the evaluation results from previously-executed, empty-result queries by storing the information about query parts in the RDBMS, where each query part is a sub-tree of a query plan tree. If such reuse is always successful, the execution of at least 11% (=2109/18793) of all the 18,793 queries in the above mentioned CRM application can be saved.

Under a fixed storage budget, to improve our chance of using the remembered information to detect whether a new query will return an empty result set, four techniques are used. First, from previous queries' execution, only the lowest-level query parts that lead to empty result sets, rather than all empty-result query parts, are stored. Second, redundancy in the information provided by different empty-result queries is removed and only the most "valuable" information is stored. Third, as query pattern changes, the stored empty-result query parts get continuously updated to adapt to the current situation. Fourth, our method utilizes a set of special properties of empty result sets so that its coverage detection capability is often more powerful than that of the traditional materialized view method (e.g., if $\pi(R)=\emptyset$, we know immediately that $R=\emptyset$, $\sigma(R)=\emptyset$, and $R \bowtie S=\emptyset$.)

Our method works as follows. From previous queries' execution, the lowest-level query parts that lead to empty result sets are found. These query parts are decomposed into so-called atomic query parts and stored in the RDBMS. In this way, the (partial) execution results from multiple, previously-executed, empty-result queries can be combined together to handle a new query without incurring much overhead. When a new query comes, it is decomposed into atomic query parts, which are then matched with the remembered ones. If such a match exists for each decomposed atomic query part, we know immediately that this new query will return an empty result set and thus its execution is saved.

The empty-result problem has been studied in the research literature. Existing solutions either explain what leads to the empty result set [10, 16, 18, 21, 26, 27] or automatically generalize the query so that the generalized query will return some answers [7, 18, 19, 27]. For most such solutions, the first step is to execute the query and see whether the result set is empty. Our method can be used in conjunction with these solutions and significantly speed up this first step.

Throughout this paper, we assume a read-mostly environment where periodic batch updates are allowed. This is the case with traditional data warehouses. The data sets of all the database applications mentioned earlier satisfy this property. When relations get updated, our current method is to delete all the information in the RDBMS about previously-executed, empty-result queries.

We investigate the performance of our fast detection method for empty-result queries with a theoretical analysis and an initial implementation in PostgreSQL. The results show that our method has minor overhead and can often bring significant benefit by avoiding executing empty-result queries unnecessarily.

The rest of the paper is organized as follows. Section 2 describes our fast detection method for empty-result queries. Section 3 investigates the performance of the proposed method. We discuss related work in Section 4 and conclude in Section 5.

2. A Fast Detection Method for Empty-Result Queries

In this section, we describe our fast detection method for empty-result queries. Unless otherwise specified, we focus on select-project-join queries. By select-project-join queries, we mean those queries whose logical query plans contain only scan, selection, projection, join, sort, and duplicate elimination operators. Nested queries that can be rewritten into such a form are included while outer join operators are excluded. All the operators are physical operators. All the query plans are physical query plans. In Section 2.5 below, we show how to extend our techniques beyond select-project-join queries.

Our main observation is that query plans of different queries often share some common parts [32, 33, 39]. For example, during interactive exploration, users often issue a series of queries. Each query is a refinement of the previous one [34]. Also, users often submit canned queries by filling parameter values into a pre-defined template [37]. If for some parameters, different users fill in the same values (e.g., certain parts of the data set can be frequently accessed), then the query plans of these queries are likely to share some common parts. During previous queries' execution, if the RDBMS remembers the query parts that lead to empty result sets, then it is likely that the RDBMS can use them to tell whether future queries will return empty result sets. The details of our method are described in the following subsections.

2.1 Definitions

We first introduce some definitions.

Empty-result-propagating operator. Consider an operator *Op* that has one or more inputs. *Op* is an empty-result-propagating operator if it satisfies the following condition: the output of *Op* is empty if any input of *Op* is empty.

Empty-result-propagating query. An empty-result-propagating query is a query whose query plan contains only empty-result-propagating operators.

A large number of operators (e.g., scan, selection, projection, join, sort, duplicate elimination) are empty-result-propagating. As a result, empty-result-propagating queries include a large class of queries. For example, all the select-project-join queries are empty-result-propagating queries.

Query part. A query plan is a tree of operators. Every sub-tree of the query plan is called a query part.

In our discussion, we assume that each relation is used at most once in a query part. In the case that the same relation R is used multiple times in a query part (e.g., R is joined with itself), the first occurrence of R is untouched while each other occurrence of R is given a different name. Then our techniques still work. (This may reduce the detection capability of our method. However, as mentioned in Section 2.6 below, our method wants to achieve a balance between efficiency and detection capability.)

Atomic query part. Each atomic query part is an ordered pair (relation names R_N, selection condition S_C). It represents a relational algebra formula that first product joins all the relations in R_N, and then applies S_C. S_C is a conjunction of primitive terms, where each primitive term is a comparison (e.g., $A.a=B.b$, $A.a<B.b+C.c$, $A.a=100$). In the rest of the paper, we do not differentiate between an atomic query part and the relational algebra formula represented by the atomic query part.

Cover (selection condition). A selection condition S_{C1} covers another selection condition S_{C2} iff whenever S_{C2} is true, S_{C1} is also true.

Cover (atomic query part). We say that an atomic query part $P_1=(R_{N1}, S_{C1})$ covers another atomic query part $P_2=(R_{N2}, S_{C2})$ if $R_{N1} \subseteq R_{N2}$ and S_{C1} covers S_{C2}. For example, $P_1=\sigma_{A.a<40}(A)$ covers $P_2=\sigma_{A.a=20 \wedge A.c=B.d}(A \times B)=(\sigma_{A.a=20}(A)) \bowtie_{A.c=B.d} B$.

The following two theorems will be used repeatedly in our discussion:

Theorem 1: Any query part of an empty-result-propagating query is an empty-result-propagating query. For an empty-result-propagating query, if the output of some query part P is empty, the output of any higher-level query part that contains P (and thus the output of the entire query) is empty.

Proof. Omitted.

Theorem 2: Suppose that atomic query part P_1 covers atomic query part P_2. For a given database, if the output of P_1 is empty, the output of P_2 is also empty.

Proof. Suppose that $P_1=(R_{N1}, S_{C1})$. $P_2=(R_{N2}, S_{C2})$. The output U_1 of P_1 is

$$U_1 = \sigma_{S_{C1}}(\prod_{R \in R_{N1}} R) \cdot$$

The output U_2 of P_2 is

$$U_2 = \sigma_{S_{C2}}(\prod_{R \in R_{N2}} R) \cdot$$

From $R_{N1} \subseteq R_{N2}$ and $U_1=\varnothing$, we know that

$$\begin{aligned} U_3 &= \sigma_{S_{C1}}(\prod_{R \in R_{N2}} R) \\ &= \sigma_{S_{C1}}((\sigma_{S_{C1}}(\prod_{R \in R_{N1}} R)) \times (\prod_{R \in R_{N2}-R_{N1}} R)) \\ &= \sigma_{S_{C1}}(U_1 \times (\prod_{R \in R_{N2}-R_{N1}} R)) \\ &= \varnothing . \end{aligned}$$

From S_{C1} covers S_{C2}, we know that $U_2 \subseteq U_3$. Hence, we have $U_2 = \varnothing$. ■

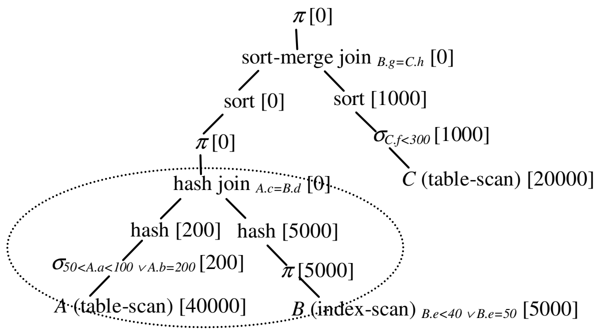
2.2 Overview of the Method

In this section, we give an overview of our fast detection method for empty-result queries. The RDBMS keeps a collection C_{aqp} of atomic query parts. The output of each atomic query part in C_{aqp} is empty. The information about previously-executed, empty-result queries is stored in C_{aqp}. For efficiency purposes, C_{aqp} is always kept in memory. Initially, C_{aqp} is empty.

When the execution of a query returns an empty result set, the RDBMS does the following two operations:

Operation O_1: Display the query plan to the user. For each operator in the query plan, the output cardinality is displayed (the output cardinalities of the operators are kept as collected statistics during query execution [23]). These output cardinalities can facilitate the user to find first the query sub-expression (and the reason) that causes the empty result set, and then the right follow-up queries [10]. The follow-up queries submitted by the user can still return empty result sets. This is because after optimizer's rewriting, the query plan often looks very different from the original query. The user may not always be able to find the genuine cause of the empty result set. Also, there can be multiple causes that lead to the empty result set.

Operation O_2: Find the lowest-level query part(s) whose output is empty. Each such query part P is broken into one or more atomic query parts. Then the atomic query parts are stored in the collection C_{aqp}. According to Theorem 1, the output of any higher-level query part P_h that contains P is empty, while the output of any lower-level query part P_l that is contained in P is not empty. However, the information about P_h is not stored in C_{aqp}, since this information is redundant. This is similar to the technique in [10, 21] of only presenting the minimal zero results to the user.



π [0]
sort-merge join $_{B.g=C.h}$ [0]
sort [0]
sort [1000]
π [0]
$\sigma_{C.f<300}$ [1000]
hash join $_{A.c=B.d}$ [0]
C (table-scan) [20000]
hash [200]
hash [5000]
$\sigma_{50<A.a<100 \vee A.b=200}$ [200]
π [5000]
A (table-scan) [40000]
B (index-scan) $_{B.e<40 \vee B.e=50}$ [5000]

Figure 1. A first query plan example.

For example, consider the query plan in Figure 1. The numbers in the square brackets represent the output cardinalities of the operators. The subscripts represent the selection conditions. (Join conditions are also selection conditions.) Only the query part that is represented by the oval is broken into atomic query parts.

From query execution, the RDBMS can only obtain output cardinalities of physical operators in physical query plans [23]. Hence, in Operations O_1 and O_2, our method works with physical query plans rather than logical query plans. In contrary, as described in Section 2.4 below, when our method uses the information stored in C_{aqp} to check whether a new query Q will return an empty result set, the logical query plan of Q is used.

In Operation O_2, the query part P is stored in the collection C_{aqp} in the form of one or more atomic query parts. This is to facilitate the future checking of whether a new query Q will return an empty result set, as our method may need to use the (partial) execution results from multiple previous queries (see Section 2.4 below for details). For example, suppose that $Q=\sigma_{A.a=50 \wedge A.a=60}(A)$. From previous queries' execution, the RDBMS knows that both queries $Q_1=\sigma_{A.a=50 \wedge A.b=30}(A)$ and $Q_2=\sigma_{A.a=60 \wedge A.b=40}(A)$ return empty result sets. Q_1 is broken into two atomic query parts: $P_1=\sigma_{A.a=50}(A)$ and $P_2=\sigma_{A.b=30}(A)$. Q_2 is broken into two atomic query parts: $P_3=\sigma_{A.a=60}(A)$ and $P_4=\sigma_{A.b=40}(A)$. Then by using P_1 and P_3, our method can tell that Q will return an empty result set.

A constant C_{cost} is used to decide whether a query is a *low-cost query* or a *high-cost query*. When the query plan of a new query Q comes, our method first checks whether the optimizer's estimated cost of Q, $cost(Q)$, is bigger than C_{cost}. There are two possible cases:

(1) If $cost(Q) \le C_{cost}$ (i.e., Q is a low-cost query), Q is executed.
(2) If $cost(Q) > C_{cost}$ (i.e., Q is a high-cost query), our method first uses the information stored in the collection C_{aqp} to check whether Q will return an empty result set. If not, Q is executed. Otherwise the empty result set is returned directly.

Our heuristics is that low-cost queries can be executed quickly, and it takes time to use the information stored in C_{aqp} to check whether a query will return an empty result set. Hence, there is no need to use the information stored in C_{aqp} to check whether a low-cost query will return an empty result set. Similarly, we do not store the information about low-cost empty-result queries in C_{aqp}.

C_{cost} is an empirical number. Its value can be decided based on past statistics. For example, how expensive it is to use the information stored in C_{aqp} to check whether a query will return an empty result set, how likely a query will return an empty result set, etc.

2.3 Storing Atomic Query Parts

In this section, we show how to break a lowest-level query part P whose output is empty into one or more atomic query parts and then store the atomic query parts in the collection C_{aqp}. Our method proceeds in three steps:

Step 1: P is transformed into a simplified query part P_s.
Step 2: P_s is broken into one or more atomic query parts.
Step 3: The atomic query parts are stored in C_{aqp}.

Organization of C_{aqp}

To facilitate search and save storage space, the collection C_{aqp} of atomic query parts is organized as a list of entries. Each entry in C_{aqp} represents a set of relation names R_N. All the atomic query parts with the same relation names R_N are stored in the same entry as a linked list of selection conditions S_C.

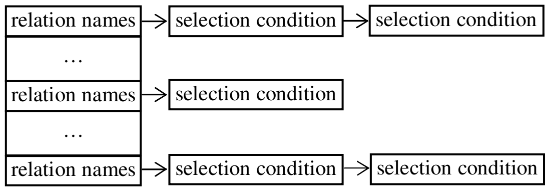


Figure 2. The collection C_{aqp} of atomic query parts.

The number of atomic query parts stored in the collection C_{aqp} cannot exceed a constant N_{max} (as will be shown in Section 3.1, N_{max} can be quite large). This is because it is not desirable to let C_{aqp} consume too much storage space. Also, as discussed in Section 2.4 below, to check whether a new query Q will return an empty result set, our method needs to check whether the atomic query parts generated from Q are covered by those stored in C_{aqp}. The more atomic query parts stored in C_{aqp}, the slower the checking.

Our method only stores in the collection C_{aqp} those atomic query parts that are frequently used [38]. To achieve this goal, the standard clock algorithm (an approximation of the LRU algorithm) is used to manage C_{aqp}. Each time the RDBMS wants to store a new atomic query part P_{new} in C_{aqp},

(1) If the number of atomic query parts stored in C_{aqp} is no more than N_{max}, P_{new} is stored in C_{aqp} directly.
(2) If the number of atomic query parts stored in C_{aqp} is larger than N_{max}, the RDBMS uses the clock algorithm to find an atomic query part P_{old} in C_{aqp} that is not frequently used. P_{old} is deleted from C_{aqp}. Then P_{new} is stored in C_{aqp}.

Our method is more dynamic than the method in [2, 40] of only keeping the most frequently used materialized views, since creating or deleting an atomic query part is cheaper than creating or deleting a materialized view and thus can be done more frequently.

Step 1: $P \Rightarrow P_s$

When getting a lowest-level query part P whose output is empty, the RDBMS converts P into a simplified query part P_s by performing the following three transformations:

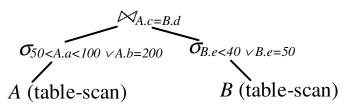
Transformation T_1: Drop all the operators in P that have no influence on the emptiness of the output. These operators include projection, hash, sort, and duplicate elimination.

Transformation T_2: Each physical join operator (e.g., hash join, sort-merge join, nested-loops join) in P is replaced with a logical join operator.

Transformation T_3: Each index-scan operator in P is replaced with a table-scan operator followed by a selection operator, where the selection condition is the same as the index-scan condition.

The result of a query is independent of its evaluation method. Hence, none of these three transformations has influence on the emptiness of the output. Essentially, these three transformations convert the "physical" query part P into a "logical" query part – the simplified query part P_s. P_s represents a relational algebra formula without projection and duplicate elimination. It is more "general" than P. This can increase our chance of using the query part to detect whether a new query will return an empty result set.

For example, after applying these three transformations, the query part that is represented by the oval in Figure 1 becomes the simplified query part in Figure 3. Our method can use this simplified query part to tell that the query in Figure 4 will return an empty result set (see Section 2.4 below for details).


$\bowtie_{A.c=B.d}$
$\sigma_{50<A.a<100 \vee A.b=200}$
$\sigma_{B.e<40 \vee B.e=50}$
A (table-scan)
B (table-scan)

corresponding relational algebra formula:

$$(\sigma_{50<A.a<100 \vee A.b=200}(A)) \bowtie_{A.c=B.d} (\sigma_{B.e<40 \vee B.e=50}(B))$$

Figure 3. A simplified query part example.

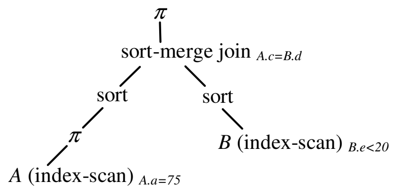

π
sort-merge join $_{A.c=B.d}$
sort
sort
π
B (index-scan) $_{B.e<20}$
A (index-scan) $_{A.a=75}$

Figure 4. A second query plan example.

Step 2: $P_s \Rightarrow$ *Atomic Query Parts*

Now the simplified query part P_s is broken into one or more atomic query parts. Suppose that P_s contains n selection conditions. Each selection condition comes from either a selection operator or a join operator. Each selection condition is rewritten into a disjunctive normal form (DNF). DNF is a disjunction of terms, where each term is a conjunction of primitive terms and each primitive term is a comparison. During the rewriting,

(1) Negations on numeric or string attributes are removed by using complementary operators. For example, $not(A.a<20)$ is rewritten into $A.a\geq 20$. $not(A.a=20)$ is rewritten into $(A.a<20)\vee(A.a>20)$.
(2) Interval-based comparison is treated as a single primitive term. For example, $10<A.a<20$ is treated as a single primitive term rather than a conjunction of two primitive terms $10<A.a$ and $A.a<20$. This is to facilitate the later checking of whether one atomic query part covers another atomic query part.

After the above rewriting, the combination of all n selection conditions in the simplified query part P_s becomes a conjunction of n DNFs $D_1\wedge D_2\wedge\ldots\wedge D_n$, where each D_i ($1\leq i\leq n$) is a DNF of a selection condition. Then $D_1\wedge D_2\wedge\ldots\wedge D_n$ is rewritten into another DNF P_{DNF}. Each term in P_{DNF} is a selection condition. It is a conjunction of n terms, where the i-th ($1\leq i\leq n$) term comes from D_i. Note that the rewriting into the DNF P_{DNF} is an exponential step. However, usually this step is not expensive, as selection conditions are not very complex. Also, for queries with extremely complex selection conditions, our method may not be used.

Let R_N denote the input relations of all the table-scan operators in the simplified query part P_s. Each term S_C in P_{DNF}, combined with R_N, is an atomic query part (R_N, S_C). For example, after rewriting, the simplified query part in Figure 3 becomes the four atomic query parts in Figure 5.

$$(\sigma_{50<A.a<100}(A)) \bowtie_{A.c=B.d} (\sigma_{B.e<40}(B))$$
$$(\sigma_{A.b=200}(A)) \bowtie_{A.c=B.d} (\sigma_{B.e<40}(B))$$
$$(\sigma_{50<A.a<100}(A)) \bowtie_{A.c=B.d} (\sigma_{B.e=50}(B))$$
$$(\sigma_{A.b=200}(A)) \bowtie_{A.c=B.d} (\sigma_{B.e=50}(B))$$

Figure 5. Atomic query part examples.

It is easy to see that the simplified query part P_s and the generated atomic query parts have the following property:

Theorem 3: The following three assertions are equivalent to each other:

(a) The output of the query part P is empty.

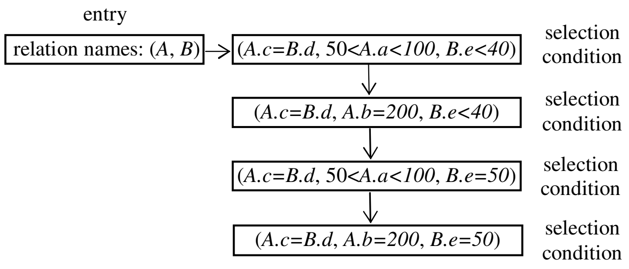
(b) The output of P_s is empty.

(c) The output of each generated atomic query part is empty.

Proof. Omitted.

Step 3: Storing Atomic Query Parts in C_{aqp}

For each atomic query part $P_a=(R_N, S_C)$ generated in Step 2, all the atomic query parts that are covered by P_a are removed from the collection C_{aqp}. Then P_a is inserted into C_{aqp}. That is, only the most "general" atomic query parts are kept in C_{aqp}, since they are most useful in deciding whether a new query will return an empty result set. This is similar to the materialized view merging technique in [2].



entry
relation names: (A, B)
(A.c=B.d, 50<A.a<100, B.e<40)
selection condition
(A.c=B.d, A.b=200, B.e<40)
selection condition
(A.c=B.d, 50<A.a<100, B.e=50)
selection condition
(A.c=B.d, A.b=200, B.e=50)
selection condition

Figure 6. An example entry in the collection C_{aqp} of atomic query parts.

For example, suppose that the collection C_{aqp} is originally empty. Then after inserting the four atomic query parts in Figure 5, the corresponding entry E in C_{aqp} becomes what is shown in Figure 6.

For an atomic query part $P_a=(R_N, S_C)$, atomic query parts that are covered by P_a can only be contained in those entries of C_{aqp} whose relation names form a superset of R_N. To find out which atomic query parts stored in C_{aqp} are covered by P_a, our method only searches in those entries. (The signature method in [31] is used to speed up the process of checking set containment.)

Deciding Coverage

In general, it is computationally expensive to decide precisely whether one atomic query part P_1 covers another atomic query part P_2 (many problems are NP-complete) [3, 9, 11, 13, 20, 22, 28, 36]. In our case, false positives are not allowed while false negatives are tolerable. That is, if our method decides that P_1 covers P_2, it must be true that P_1 covers P_2. However, if P_1 covers P_2 but our method fails to detect it, it does not matter except that efficiency is sacrificed. For example, storage in the collection C_{aqp} is wasted, or a new query Q is executed unnecessarily because our method fails to use the information stored in C_{aqp} to detect that Q will return an empty result set (see Section 2.4 below for details).

In deciding whether $P_1=(R_{N1}, S_{C1})$ covers $P_2=(R_{N2}, S_{C2})$, the RDBMS uses a method that attempts to achieve a reasonable balance between efficiency and detection capability [3, 36]. Suppose that $S_{C1}=p_1 \wedge p_2 \wedge \ldots \wedge p_n$ and $S_{C2}=q_1 \wedge q_2 \wedge \ldots \wedge q_m$. Each p_i ($1 \leq i \leq n$) is a primitive term. Each q_j ($1 \leq j \leq m$) is a primitive term.

The method works as follows. The RDBMS decides that S_{C1} covers S_{C2} if they satisfy the following two conditions:

(1) $n \leq m$.

(2) For each i ($1 \leq i \leq n$), the RDBMS decides that p_i covers some q_j ($1 \leq j \leq m$).

If one of the following three conditions is satisfied, the RDBMS decides that p_i covers q_j:

(1) $p_i=q_j$.

(2) Both p_i and q_j are interval-based comparison on the same attribute of the same relation. The interval in p_i contains the interval in q_j. Note that point-based comparison is a special form of interval-based comparison. Also, the endpoints of some intervals can be plus or minus infinity. For example, p_i is of the form $A.a<50$ and q_j is of the form $20<A.a<40$. As a second example, p_i is of the form $A.a<50$ and q_j is of the form $A.a=30$.

(3) The comparisons in p_i and q_j are on the same attribute of the same relation. p_i is of the form $A.a \neq c_1$. q_j is of the form $A.a=c_2$. $c_1 \neq c_2$.

As mentioned in Section 2.1, P_1 covers P_2 if $R_{N1} \subseteq R_{N2}$ and S_{C1} covers S_{C2}.

[9] proposed an algorithm that uses safe substitutions to decide whether a materialized view MV can be used to answer a query Q. MV "covers" Q if a safe substitution of MV can be used to generate an equivalent query of Q. Our method can be regarded as a simplified version of the algorithm in [9], where safe substitutions are replaced by naive substitutions. This is because in generating atomic query parts, all the projection operators are dropped (Transformation T_1). Hence, there is no "dangling" selection condition. The computational overhead of our method is lower than that of the algorithm in [9], while in certain cases, the coverage detection capability of our method is weaker than that of the algorithm in [9]. This results from our design philosophy that is explained in Section 2.6.

2.4 Checking Empty Result Set

In this section, we describe how to use the information stored in the collection C_{aqp} to check whether a query Q will return an empty result set. Suppose that the logical query plan of Q is Q_p. Techniques similar to that in Section 2.3 are used to break Q_p into one or more atomic query parts P_i ($i=1, 2, \ldots, m$). For each i ($1 \leq i \leq m$), the RDBMS checks whether there exists any atomic query part A_i in C_{aqp} that covers P_i. (As explained at the end of Step 3 in Section 2.3, our method only needs to search in those entries of C_{aqp} whose relation names form a subset of the relation names of P_i.) If such an A_i exists, the RDBMS knows that the output of P_i is empty (Theorem 2). If the RDBMS can find such an A_i for each i ($1 \leq i \leq m$), it knows that the output of Q_p is empty (Theorem 3).

2.5 Extension beyond Select-Project-Join Queries

The above discussion focuses on select-project-join queries. In certain cases, our techniques can be extended beyond select-project-join queries. For example,

(1) Suppose that the root operator of the query plan is an aggregate operator (either with or without a group by clause). Except for this root aggregate operator, the rest of the query plan forms a select-project-join query. Since the root aggregate operator has no influence on the emptiness of the query output, the RDBMS can ignore it when deciding whether the query will return an empty result set. *count()* needs some special handling, as *count(∅)=0*.

(2) Set union is not an empty-result-propagating operator. Suppose that the RDBMS wants to check whether the output of $Q_1 \cup Q_2$ is empty, where Q_1 and Q_2 are select-project-join queries. The techniques in Sections 2.2~2.4 are used to check whether the output of Q_1 is empty, and whether the output of Q_2 is empty. There are three possible cases:
   (a) The checking says that both outputs are empty. Then the output of $Q_1 \cup Q_2$ is empty.
   (b) The checking only says that the output of Q_1 (or the output of Q_2) is empty. Then $Q_2 = Q_1 \cup Q_2$ (or $Q_1 = Q_1 \cup Q_2$) needs to be evaluated.
   (c) The checking says nothing. $Q_1 \cup Q_2$ needs to be evaluated.

(3) Outer join can be handled in a way similar to set union.

(4) Set difference is not an empty-result-propagating operator. Suppose that the RDBMS wants to check whether the output of Q_1-Q_2 is empty, where Q_1 is a select-project-join query. The techniques in Sections 2.2~2.4 are used to check whether the output of Q_1 is empty. If so, the output of Q_1-Q_2 is empty. Otherwise Q_1-Q_2 needs to be evaluated.

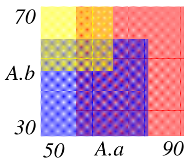
We leave it as an interesting area for future work to investigate how to handle other classes of queries. For example, set intersection is an empty-result-propagating operator. In the presence of set intersection in the query plan, projection operators influence the emptiness of the query output – from $A \cap B = \varnothing$, it cannot be deduced that $\pi(A) \cap \pi(B) = \varnothing$ (the attributes that are projected out in A and B can have different values and cause $\pi(A) \cap \pi(B) \neq \varnothing$). As a result, our current method of storing the information of empty-result queries in the collection C_{aqp} does not work for set intersection. One possible solution is that in the presence of set intersection, Transformation T_1 does not drop projection operators.

2.6 Comparison between Our Method and the Traditional Materialized View Method

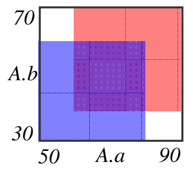
Our fast detection method for empty-result queries uses some data structure similar to materialized views – each atomic query part stored in the collection C_{aqp} can be regarded as a "mini" materialized view. However, our method utilizes a set of special properties of empty result sets and is different from the traditional method of using materialized views to answer queries.

In deciding whether a query will return an empty result set, our method ignores those operators (e.g., projection, duplicate elimination) that have no influence on the emptiness of the query output. Hence, in certain cases, the coverage detection capability of our method is more powerful than that of the traditional materialized view method. For example, consider the following two queries: $Q_1 = A \bowtie_{A.c=B.d} B$ and $Q_2 = \pi((\sigma_{A.a=100}(A)) \bowtie_{A.c=B.d} B)$. $A.a$ is projected out by the projection operator in Q_2. In general, materialized view $MV = \pi(A \bowtie_{A.c=B.d} B)$ cannot be used to answer either Q_1 or Q_2. However, if the RDBMS knows that query $Q_3 = MV$ returns an empty result set, our method can tell that both Q_1 and Q_2 will return empty result sets.

Usually, a large number of atomic query parts can be stored in the collection C_{aqp} and our method wants to speed up the checking process. Therefore, when checking whether a query will return an empty result set, our method uses a limited number of approaches to check the coverage of atomic query parts. In contrast, the traditional materialized view method uses the extensive query rewriting approach, which is more capable in certain cases but also more expensive. For example, consider query $Q = \sigma_{(50<A.a<90) \wedge (30<A.b<70)}(A)$. There are three materialized views that are all empty: $MV_1 = \sigma_{(50<A.a<80) \wedge (30<A.b<60)}(A)$, $MV_2 = \sigma_{60<A.a<90}(A)$, and $MV_3 = \sigma_{(50<A.a<70) \wedge (50<A.b<70)}(A)$. The traditional materialized view method can rewrite Q into $Q = \sigma_{30<A.b<70}(MV_1 \cup MV_2 \cup MV_3)$ and thus tell that Q will return an empty result set. In contrast, our method cannot tell that if MV_1, MV_2, and MV_3 are all empty, the output of Q is also empty.



To save storage space, the traditional materialized view method can merge multiple materialized views together [2]. For example, the two materialized views $MV_1 = \sigma_{(50<A.a<80) \wedge (30<A.b<60)}(A)$ and $MV_2 = \sigma_{(60<A.a<90) \wedge (40<A.b<70)}(A)$ can be merged into a single materialized view $MV_3 = \sigma_{(50<A.a<90) \wedge (30<A.b<70)}(A)$. Then all the queries that can be answered by using either MV_1 or MV_2 are answerable by using MV_3. However, if both MV_1 and MV_2 are empty, the RDBMS cannot deduce that MV_3 is also empty. As a result, our method cannot arbitrarily merge multiple atomic query parts in the collection C_{aqp} into a single atomic query part.



In general, the design philosophy of our method is to achieve a reasonable balance between efficiency and detection capability. Our method strives for the detection capability that is both mostly needed and easily achievable [3, 36]. It makes up the "loss" in the other detection capability by storing a large number of atomic query parts in the collection C_{aqp}.

2.7 Summary of Advantages

Our fast detection method for empty-result queries has the following advantages:

(1) It has small storage and computation overhead.

(2) A large number of atomic query parts can be stored in the collection C_{aqp}.

(3) As query pattern changes, the atomic query parts stored in C_{aqp} get continuously updated to reflect the current situation.

(4) The hotter an empty-result query Q is (i.e., the more frequently users submit this Q), the more likely Q can be successfully detected by our method. This is desirable for those applications where users care more about hot spots in the data set.

(5) Due to the utilization of a set of special properties of empty result sets, its coverage detection capability is often more powerful than that of a traditional materialized view method.

After enough information about previously-executed, empty-result queries has been accumulated in C_{aqp}, our method can often successfully detect empty-result queries and avoid the expensive query execution. This both reduces the load on the RDBMS and speeds up the interactive exploration process.

3. Performance Evaluation

In this section, we evaluate the performance of our method, first with experiments in PostgreSQL [29], and then with a theoretical analysis.

3.1 Evaluation in PostgreSQL

We first describe experiments we performed with a prototype implementation of our techniques in PostgreSQL Version 7.3.4 [29]. Our measurements were performed with the PostgreSQL client application and server running on a Dell Inspiron 8500 PC with one 2.2GHz processor, 512MB main memory, one 40GB disk, and running the Microsoft Windows XP operating system. The default setting of PostgreSQL was used, where the buffer pool size is 1,000 pages. (We also tested larger buffer pool sizes. The results were similar and thus omitted.)

The relations used for the experiments followed the schema of the standard TPC-R Benchmark relations [35]:

customer (custkey, nationkey, …),
orders (orderkey, custkey, orderdate, …),
lineitem (orderkey, partkey, …).

Table 1. Test data set.

	number of tuples	total size
customer	$0.15\times s$ M	$23\times s$ MB
orders	$1.5\times s$ M	$114\times s$ MB
lineitem	$6\times s$ M	$755\times s$ MB

s is the scale factor of the database. In our experiments, on average, each *customer* tuple matches ten *orders* tuples on the attribute *custkey*. Each *orders* tuple matches 4 *lineitem* tuples on the attribute *orderkey*.

We used the following two queries:

Query Q_1: Find the information about certain parts that were sold on certain days.

```
select *
from orders o, lineitem l
where o.orderkey=l.orderkey
        and (o.orderdate=$d_1$ or … or o.orderdate=$d_e$)
        and (l.partkey=$p_1$ or … or l.partkey=$p_f$);
```

Query Q_2: Find the information about certain parts that were sold to certain customers on certain days.

```
select *
from orders o, lineitem l, customer c
where o.orderkey=l.orderkey and o.custkey=c.custkey
        and (o.orderdate=$d_1$ or … or o.orderdate=$d_e$)
        and (l.partkey=$p_1$ or … or l.partkey=$p_f$)
        and (c.nationkey=$n_1$ or … or c.nationkey=$n_g$);
```

We built an index on each selection or join attribute. Before we ran queries, we ran the PostgreSQL statistics collection program on all the relations. We also tested other database schemas. The results are similar and thus not presented here.

For each query Q_i (i=1, 2) we tested, if Q_i returns an empty result set, the minimal zero result [10, 21] is Q_i itself. That is, except for the entire Q_i, any part of Q_i (e.g., selection on a single relation) has some non-empty output. For Q_1, its combination factor is defined as $F=e\times f$. For Q_2, its combination factor is defined as $F=e\times f\times g$. F represents the number of atomic query parts that will be generated from the query.

We performed three experiments. In these experiments, we purposely use large N's to demonstrate good scalability, where N is the number of atomic query parts that have already been stored in the collection C_{aqp} when the new query comes. As will be shown in Section 3.2, our method can often use a much smaller N to achieve good detection probability. In each experiment, we measured the overhead of our techniques in the following two cases:

Case 1: Only Q_1's are executed. That is, the new query is Q_1. Also, the information stored in C_{aqp} comes from previous Q_1's.
Case 2: Only Q_2's are executed. That is, the new query is Q_2. Also, the information stored in C_{aqp} comes from previous Q_2's.

For each reported number, the experiment was repeated twenty times (twenty runs). Each run used different (and random) d_i's ($1\le i\le e$), p_j's ($1\le j\le f$), and n_k's ($1\le k\le g$). Unless otherwise specified, the reported overhead of our techniques is the largest one observed during these twenty runs. The reported query execution time is the shortest one observed during these twenty runs. This is to approximate the maximum overhead of our techniques and the shortest query execution time. In this way, when comparing the overhead of our techniques with the query execution time, we always do favor to the query execution time.

C_{aqp} Size Experiment

In this experiment, we fixed $F=2$ and $s=2$. We varied N from 1,000 to 3,000. Recall that N is the number of atomic query parts that have already been stored in the collection C_{aqp} when the new query comes.

Figure 7 shows the overhead of our techniques. By "check succeeds/fails," we mean that our method can/cannot use the information stored in the collection C_{aqp} to tell that a query will return an empty result set. The lines for query Q_1 (Q_2) represent the overhead of our techniques in Case 1 (Case 2).

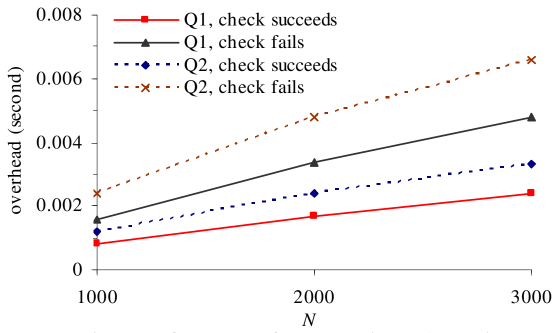


Figure 7. Overhead of our techniques (C_{aqp} size experiment).

The overhead of our techniques in Case 2 is larger than that in Case 1. This is because query Q_2 is more complex than query Q_1: Q_2 joins three relations, while Q_1 joins two relations. As a result, the atomic query parts generated from Q_2 are more complex than those generated from Q_1. Also, the atomic query part coverage checking in Case 2 is more time consuming than that in Case 1.

The overhead of our techniques increases with N, the number of atomic query parts stored in the collection C_{aqp}. This is easy to understand. Given a query, C_{aqp} needs to be searched for matching atomic query parts. The more atomic query parts stored in C_{aqp}, the longer time the search takes.

When the check succeeds, the collection C_{aqp} only needs to be searched once. When the check fails, if the query returns an empty result set, C_{aqp} needs to be searched twice. The first time is to check whether the query will return an empty result set. The second time is to store the information about the empty-result query into C_{aqp}. Hence, when the check fails, the maximum overhead of our techniques is about two times that when the check succeeds.

Query Combination Factor Experiment

In this experiment, we fixed $s=2$ and $N=2,000$. We varied the query combination factor F from 1 to 8. Figure 8 shows the overhead of our techniques. The meaning of the four lines in Figure 8 is the same as that in Figure 7. The larger the F is, the more atomic query parts the query generated. Then our method needs to spend more time on both searching for the matching atomic query parts in the collection C_{aqp}, and storing the atomic query parts generated from an empty-result query into C_{aqp}. Therefore, the overhead of our techniques increases with F.

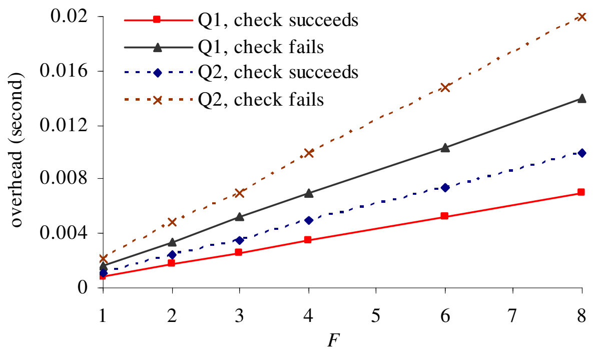


Figure 8. Overhead of our techniques (query combination factor experiment).

Database Scale Factor Experiment

In this experiment, we fixed $F=2$ and $N=2,000$. We varied the database scale factor s from 1 to 3. The purpose of this experiment is to show that the overhead of our techniques is negligible compared to the query execution cost.

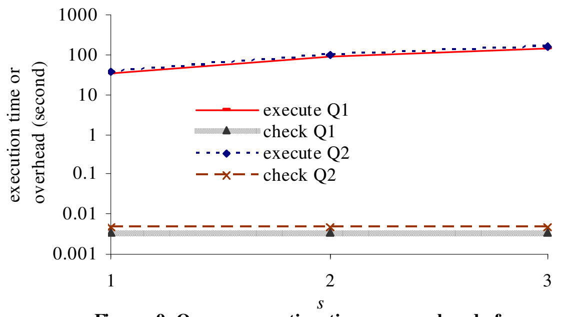


Figure 9. Query execution time vs. overhead of our techniques (database scale factor experiment).

Figure 9 shows both the overhead of our techniques and the query execution time. The lines for "check Q_1/Q_2" represent the overhead of our techniques. The lines for "execute Q_1/Q_2" represent the query execution time. The y-axis uses logarithmic scale.

Our techniques do not examine the data set. Also, our techniques only perform fast in-memory operations (recall that the entire collection C_{aqp} is kept in memory). Hence, compared to the query execution time, the overhead of our techniques is about four orders of magnitude smaller.

The query execution time increases with the data set size. In contrary, the overhead of our techniques is independent of the data set size, since our techniques do not examine the data set.

3.2 Theoretical Analysis of Detection Probability

We now turn to describe a first-principle mathematical analysis on the probability that our method can successfully detect empty-result queries. The goal of this analysis is to gain insight into the main performance trends of our method. For this purpose, our analysis picks up several typical cases that are frequently encountered in practice (e.g., canned queries).

We assume that all the queries are of the form $\pi(\sigma_{S_C}(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_k))$, where S_C is the selection condition and the join condition is omitted. From previous queries' execution, N atomic query parts have been stored in the collection C_{aqp}. The detection probability D_p is defined as the probability that our method can successfully detect empty-result queries. That is, in the case that the output of a new query Q is empty, with probability D_p our method can detect that Q will return an empty result set without executing Q. We analyze the following three cases.

Case 1: Point-based Comparisons

Suppose that the selection condition S_C is a disjunction of m terms, where each term is a conjunction of n primitive terms and the i-th ($1 \le i \le n$) primitive term is of the form $R_{e_i}.a_i = c_i$ ($1 \le e_i \le k$). Each term can be represented as an n-tuple (c_1, c_2, …, c_n). Assume that when $m=1$, there are totally K such n-tuples that can cause the query to return an empty result set. All these K n-tuples form a set S_K. Each atomic query part stored in the collection C_{aqp} corresponds to an n-tuple $t \in S_K$. We say that an n-tuple $t \in S_K$ is stored in C_{aqp} if the corresponding atomic query part is stored in C_{aqp}. Therefore, N n-tuples $t \in S_K$ are stored in C_{aqp}.

Consider a new query Q whose result set is empty. That is, for each one of the m terms in the selection condition S_C of Q, the corresponding n-tuple $t \in S_K$. For a given n-tuple $t \in S_K$, the probability that t is stored in C_{aqp} is $p=N/K$. The detection probability D_p is equal to the probability that for each one of the m terms in the S_C of Q, the corresponding n-tuple $t \in S_K$ is stored in C_{aqp}. Therefore, $D_p=p^m$.

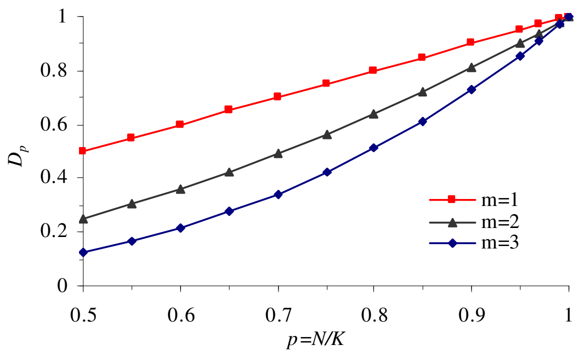


Figure 10. The detection probability D_p (point-based comparisons).

Figure 10 shows the detection probability D_p of Case 1. For a fixed m, D_p increases with p. When p is close to 1, D_p is also close to 1. This is easy to understand, since p represents the amount of information stored in the collection C_{aqp}. In the extreme case that $p=1$, our method can always successfully detect that query Q will return an empty result set.

For a fixed p, the detection probability D_p decreases with m. That is, the more terms in the selection condition S_C of query Q, the less likely our method can detect that Q will return an empty result set. This is because only in the case that the information of all m terms is stored in the collection C_{aqp}, our method can detect that Q will return an empty result set.

Case 2: Unbounded-Interval-based Comparisons

Suppose that the selection condition S_C is a conjunction of n primitive terms, where the i-th ($1 \le i \le n$) primitive term is of the form $c_i < R_{e_i}.a_i$ ($1 \le e_i \le k$). Assume that $\prod_{i=1}^{n}(U_i, V_i)$ is the entire empty result space. That is, whenever $c_i \in (U_i, V_i)$ for each i ($1 \le i \le n$), the query output is empty. Without loss of generality, we assume that the entire empty result space has been normalized. That is, for each i ($1 \le i \le n$), $U_i=0$ and $V_i=1$.

The information of N empty-result queries is stored in the collection C_{aqp}. Essentially, N selection conditions are stored in C_{aqp}. For each such selection condition and each i ($1 \le i \le n$), we assume that the corresponding c_i is uniformly distributed between 0 and 1. (For simplicity, "merging" atomic query parts in C_{aqp} is not considered. In the case that only the most "general" atomic query parts are kept in C_{aqp}, our derivation is still valid while the number of atomic query parts stored in C_{aqp} can be less than N.)

Consider a new query Q whose result set is empty. For each i ($1 \le i \le n$), the c_i of Q is uniformly distributed between 0 and 1. We derive the detection probability D_p in the following way:

(1) Consider any selection condition S_C' stored in the collection C_{aqp}. The selection condition S_C of Q is covered by S_C' iff for each i ($1 \le i \le n$), the c_i of S_C is bigger than the c_i of S_C' (whose probability is ½). Hence, the probability that S_C is covered by S_C' is $1/2^n$.

(2) The detection probability D_p is equal to the probability that the S_C of Q is covered by one of the N selection conditions stored in C_{aqp}. Therefore, $D_p = 1-(1-1/2^n)^N$. This is because:

   (a) Consider any selection condition S_C' stored in C_{aqp}. The probability that the S_C of Q is not covered by S_C' is $1-1/2^n$.

   (b) The probability that the S_C of Q is not covered by any of the N selection conditions stored in C_{aqp} is $(1-1/2^n)^N$.

Figure 11 shows the detection probability D_p of Case 2. For a fixed n, D_p increases with N. When N goes to infinity, D_p becomes close to 1. This is easy to understand. N represents the amount of information stored in the collection C_{aqp}. The more information our method has, the more likely it can detect that query Q will return an empty result set.

For a fixed N, the detection probability D_p decreases with n. That is, the more primitive terms in the selection condition S_C of query Q, the less likely our method can detect that Q will return an empty result set. This is because only in the case that all n primitive terms are "matched" by the information stored in the collection C_{aqp}, our method can detect that Q will return an empty result set.

Now suppose that each unbounded-interval-based comparison $c_i < R_{e_i}.a_i$ ($1 \le i \le n$) is replaced by a bounded-interval-based comparison $c_i < R_{e_i}.a_i < d_i$. From a similar analysis, we can show that in the bounded-interval-based comparison case, the detection probability D_p is $D_p = 1-(1-1/6^n)^N$.

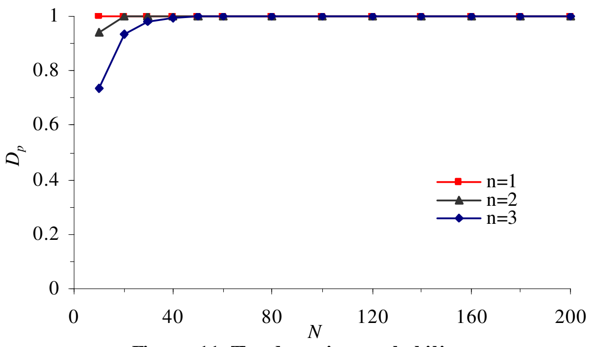


Figure 11. The detection probability D_p (unbounded-interval-based comparisons).

Case 3: Mixed Case

Suppose that the selection condition S_C is a disjunction of m terms, where each term is a conjunction of n primitive terms. Among these $n=n_1+n_2+n_3$ primitive terms, n_1 are point-based comparisons, n_2 are bounded-interval-based comparisons, and n_3 are unbounded-interval-based comparisons.

Consider a new query Q whose result set is empty. Suppose that for any term T of Q and any atomic query part P stored in the collection C_{aqp}, the probability that T is "covered" by P is q. Then for any term T of Q, the probability that T is covered by some atomic query part stored in C_{aqp} is $1-(1-q)^N$. The detection probability D_p is $D_p = (1-(1-q)^N)^m$.

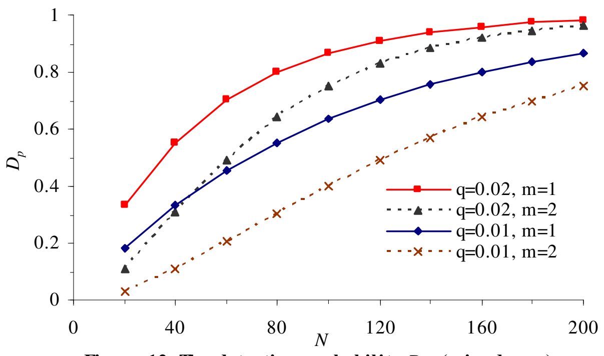


Figure 12. The detection probability D_p (mixed case).

Figure 12 shows the detection probability D_p of Case 3. For a fixed q and m, D_p increases with N. When N goes to infinity, D_p becomes close to 1. This is easy to understand, since N represents the amount of information stored in the collection C_{aqp}.

Only in the case that the information of all m terms is stored in C_{aqp}, our method can detect that query Q will return an empty result set. Hence, for a fixed q and N, D_p decreases with m.

For a fixed m and N, D_p increases with q, since q represents the probability that an atomic query part stored in C_{aqp} can be used to match an empty-result query.

Summary of Experimental Results

In all the above three cases, when N, the number of atomic query parts stored in the collection C_{aqp}, goes to infinity, D_p becomes close to 1. As shown in Section 3.1, our method can afford to store many atomic query parts in C_{aqp}. This increases our chance of successfully detecting empty-result queries. In practice, we would expect the data set to contain hot spots. Therefore, in a large number of cases, after enough information about previously-executed, empty-result queries has been accumulated in C_{aqp}, we would expect D_p to converge to 1. Section 3.1 shows that the overhead of our techniques is negligible compared to the query execution overhead. Thus, when the check succeeds, our method can bring significant performance benefit by avoiding the expensive query execution. This benefit increases with the data set size.

4. Related Work

The empty-result problem has been studied in the research literature. Existing solutions fall into two categories:

(1) Explain what leads to the empty result set [10, 16, 18, 21, 26, 27].
(2) Automatically generalize the query so that the generalized query will return some answers [7, 18, 19, 27].

Both categories can be regarded as an AI approach: letting the computer make trials and guess what users want. However, none of the existing solutions is completely satisfactory. Some solutions require domain specific knowledge that is often not available. Some solutions only apply to a restricted form of queries. Some solutions require an excessive amount of time. In certain cases, some solutions give a large number of reasons why the result set is empty. Users do not know which one is the genuine reason. Even worse, these reasons may not always include the genuine reason.

For most AI-style solutions, the first step is to execute the query and see whether the result set is empty. This is also the default step when no specific AI-style solution is used. However, none of the existing solutions allows users to reuse each other's query results, which is the focus of our method. For certain applications, our method works while the AI approach does not. For example, in anomaly detection, users submit queries to detect anomaly. In a large number of cases, users would expect that anomaly does not exist and hence submitted queries should return empty result sets. In this case, empty result sets are not a surprise. There is no need to either explain what leads to empty result sets or generalize queries so that users can get some answers. Rather, users care how soon they can get query results.

To facilitate interactive exploration of massive data sets, [14] proposed providing early approximate answers for aggregate queries. In contrast, our method needs to know exactly whether a query will return an empty result set.

There has been a lot of work on using materialized views to speed up query processing [12]. In general, it is expensive to decide which queries can be answered with which materialized views (many problems are NP-complete) [3, 9, 11, 13, 20, 22, 28, 36]. Our method uses the idea in [3, 36] and focuses on a restricted set of ways of using stored query parts to check whether a query will return an empty result set. Hence, our solution is scalable in the presence of a large number of queries and stored query parts.

[15] proposed using materialized views to answer preference queries that maximize a linear function over the relation's attributes. [15] only considers numeric attributes and selection on a single relation, while our method needs to handle arbitrary attributes and multiple relations.

[4, 5] proposed using statistics on query expressions to facilitate query optimization. Each set of collected statistics S is on an attribute r. If r does not appear in a query Q, then S cannot be used for the cardinality estimation of Q. In contrast, our method does not consider attributes. Rather, it only cares whether or not the cardinality is zero. Moreover, [4, 5] only consider select-project-join queries whose filter expression is a conjunction of simple predicates, while our method needs to handle arbitrary filter expressions.

[1, 6] proposed building histograms by analyzing query results rather than checking the relation. [1, 6] cannot detect correlation among multiple relations, which is a common reason that leads to empty result sets.

Multiple-query optimization [32, 33, 39] allows multiple concurrent queries to reuse each other's intermediate results. However, it does not work for non-concurrent queries, which are common in our case.

5. Conclusion

In this paper, we propose a fast detection method for empty-result queries. Our key idea is to reuse the evaluation results from previous empty-result queries. Under a fixed storage budget, our method only stores the most valuable and non-redundant information, continuously updates the stored information, and utilizes a set of special properties of empty result sets to increase its coverage detection capability. Our experiments with both a theoretical analysis and a prototype implementation in PostgreSQL show that our method has low overhead. In a large number of cases, our method can bring significant benefit by avoiding executing empty-result queries unnecessarily. This can both reduce the load on the RDBMS and expedite the interactive exploration process.

Like traditional materialized views, we expect our techniques to be most helpful in read-mostly environments. Extending our techniques to more update-intensive environment is an interesting area for future work. We suspect that many of the techniques used in materialized view maintenance for detecting irrelevant updates [8, 25] can be modified to be useful in our scenario. That is, C_{aqp} keeps those atomic query parts that are not influenced by the updates to relations. The atomic query parts used in our techniques have the special property that most (or all) updates to relations are irrelevant. As a result, we can create in-memory filtering relations that summarize the most relevant information in the base relations and use them to quickly filter out as many irrelevant updates as possible. The details of our current progress are available in [24].

6. Acknowledgements

We would like to thank Vikram Aggarwal, Jeffrey F. Naughton, Chunqiang Tang, Naval K. Verma, Michael W. Watzke, Zhen Wen, Kun-lung Wu, and Fan Ye for helpful discussions.

7. References

[1] Aboulnaga, A., and Chaudhuri, S. Self-tuning Histograms: Building Histograms without Looking at Data. SIGMOD Conf. 1999: 181-192.
[2] Agrawal, S., Chaudhuri, S., and Narasayya, V.R. Automated Selection of Materialized Views and Indexes in SQL Databases. VLDB 2000: 496-505.
[3] Afrati, F.N., Li, C., and Mitra, P. On Containment of Conjunctive Queries with Arithmetic Comparisons. EDBT 2004: 459-476.

[4] Bruno, N., and Chaudhuri, S. Exploiting Statistics on Query Expressions for Optimization. SIGMOD Conf. 2002: 263-274.

[5] Bruno, N., and Chaudhuri, S. Conditional Selectivity for Statistics on Query Expressions. SIGMOD Conf. 2004: 311-322.

[6] Bruno, N., Chaudhuri, S., and Gravano, L. STHoles: A Multidimensional Workload-Aware Histogram. SIGMOD Conf. 2001: 211-222.

[7] Bruno, N., Chaudhuri, S., and Gravano, L. Top-k Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation. TODS 27(2): 153-187, 2002.

[8] Blakeley, J.A., Coburn, N., and Larson, P. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. TODS 14(3): 369-400, 1989.

[9] Chaudhuri, S., Krishnamurthy, R., and Potamianos, S. et al. Optimizing Queries with Materialized Views. ICDE 1995: 190-200.

[10] Corella, F., Kaplan, S.J., and Wiederhold, G. et al. Cooperative Responses to Boolean Queries. ICDE 1984: 77-85.

[11] Goldstein, J., and Larson, P. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. SIGMOD Conf. 2001: 331-342.

[12] Gupta, A., and Mumick, I.S. Materialized Views: Techniques, Implementations, and Applications. MIT Press, 1999.

[13] Halevy, A.Y. Answering Queries Using Views: A Survey. VLDB J. 10(4): 270-294, 2001.

[14] Hellerstein, J.M., Haas, P.J., and Wang, H. Online Aggregation. SIGMOD Conf. 1997: 171-182.

[15] Hristidis, V., Koudas, N., and Papakonstantinou, Y. PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. SIGMOD Conf. 2001: 259-270.

[16] Janas, J.M. Towards More Informative User Interfaces. VLDB 1979: 17-23.

[17] Jones, S., Cunningham, S.J., and McNab, R.J. et al. A Transaction Log Analysis of a Digital Library. Int. J. on Digital Libraries 3(2): 152-169, 2000.

[18] Kao, M., Cercone, N., and Luk, W. Providing Quality Responses with Natural Language Interfaces: The Null Value Problem. IEEE Trans. Software Eng. 14(7): 959-984, 1988.

[19] Kießling, W., and Köstler, G. Preference SQL - Design, Implementation, Experiences. VLDB 2002: 990-1001.

[20] Kolaitis, P.G., Martin, D.L., and Thakur, M.N. On the Complexity of the Containment Problem for Conjunctive Queries with Built-in Predicates. PODS 1998: 197-204.

[21] Lee, R.M. Conversational Aspects of Database Interactions. VLDB 1978: 392-399.

[22] Levy, A.Y., Mendelzon, A.O., and Sagiv, Y. et al. Answering Queries Using Views. PODS 1995: 95-104.

[23] Luo, G., Naughton, J.F., and Ellmann, C.J. et al. Toward a Progress Indicator for Database Queries. SIGMOD Conf. 2004: 791-802.

[24] Luo, G., and Yu, P.S. Efficient Real-time Materialized View Maintenance. Submitted for publication, available at http://www.cs.wisc.edu/~gangluo/detect_full.pdf, 2006.

[25] Levy, A.Y., and Sagiv, Y. Queries Independent of Updates. VLDB 1993: 171-181.

[26] Motro, A. SEAVE: A Mechanism for Verifying User Presuppositions in Query Systems. ACM Trans. Inf. Syst. 4(4): 312-330, 1986.

[27] Motro, A. FLEX: A Tolerant and Cooperative User Interface to Databases. TKDE 2(2): 231-246, 1990.

[28] Pottinger, R., and Levy, A.Y. A Scalable Algorithm for Answering Queries Using Views. VLDB 2000: 484-495.

[29] PostgreSQL homepage, 2005. http://www.postgresql.org.

[30] Rebhan, M., Chalifa-Caspi, V., and Prilusky, J. et al. GeneCards: a Novel Functional Genomics Compendium with Automated Data Mining and Query Reformulation Support. Bioinformatics 14(8): 656-664, 1998.

[31] Ramasamy, K., Patel, J.M., and Naughton, J.F. et al. Set Containment Joins: the Good, the Bad and the Ugly. VLDB 2000: 351-362.

[32] Roy, P., Seshadri, S., and Sudarshan, S. et al. Efficient and Extensible Algorithms for Multi Query Optimization. SIGMOD Conf. 2000: 249-260.

[33] Sellis, T.K. Multiple-Query Optimization. TODS 13(1): 23-52, 1988.

[34] Shneiderman, B. Improving the Human Factors Aspect of Database Interactions. TODS 3(4): 417-439, 1978.

[35] TPC Homepage. TPC-R benchmark, www.tpc.org.

[36] Tsatalos, O.G., Solomon, M.H., and Ioannidis, Y.E. The GMAP: A Versatile Tool for Physical Data Independence. VLDB J. 5(2): 101-118, 1996.

[37] Williamson, C., and Shneiderman, B. The Dynamic HomeFinder: Evaluating Dynamic Queries in a Real-Estate Information Exploration System. SIGIR 1992: 338-346.

[38] Yu, C.T., Philip, G., and Meng, W. Distributed Top-N Query Processing with Possibly Uncooperative Local Systems. VLDB 2003: 117-128.

[39] Zhao, Y., Deshpande, P., and Naughton, J.F. et al. Simultaneous Optimization and Evaluation of Multiple Dimensional Queries. SIGMOD Conf. 1998: 271-282.

[40] Zilio, D.C., Rao, J., and Lightstone, S. et al. DB2 Design Advisor: Integrated Automatic Physical Database Design. VLDB 2004: 1087-1097.