Automated Materialized View Selection in PostgreSQL

Brendan Fruin   Shangfu Peng
Department of Computer Science, University of Maryland
College Park, MD 20742 USA
{brendan, shangfu}@cs.umd.edu

ABSTRACT

Materialized views have been shown to greatly improve the performance of queries in certain databases. A problem with materialized views is that they are often difficult to define or need to be updated frequently with changing query workloads. This paper outlines an approach for generating and selecting materialized views in PostgreSQL by modifying the database schema to have pseudo-materialized views. The generated materialized views are analyzed by rewriting existing queries from a sample workload and analyzing the cost using built-in PostgreSQL cost analysis. We then run the algorithm on a test database and show how materialized views can increase the performance of queries.

1. INTRODUCTION

Databases are often associated with high maintenance cost in creation and upkeep, but these costs are often considered warranted if it improves the query time to complete a process. General metadata design and management has been generally left to database administrators who control many aspects of the database. This can be monetarily expensive for a company that always needs to employ a database administrator and these updates may force the database to go offline which can be unacceptable in global companies. Self-tuning in databases is the idea that both the metadata creation and maintenance can be controlled by the database automatically. This allows the database to optimize based on the current users queries and with certain implementations this can be done while keeping the system accessible.

This paper will focus on research into self-tuning a database to choose optimal materialized views that will increase query efficiency and be constrained to user defined costs such as disk-space or maintenance time. Due to the limited research on this specific problem, general self-tuning practices are presented in Section 2.1 along with its use in commercial systems in Section 2.1.2. Related works in the principles behind materialized view selection are outlined in Section 2.2. A general overview of our approach is given in Section 3. Our candidate materialized view generation algorithm is explained in Section 4 and Section 5 describes how materialized views are ultimately selected. The results of our experiments are in Section 6 followed by future work in Section 7.

2. RELATED WORK

2.1 Self-Tuning

Self-tuning or automatic tuning has been an elusive goal for database technology for some time. It is a well researched problem, [23, 21, 41], over the past decade. The total cost of ownership for DBMS-based IT solution is dominated by staff for system administrator, management, and tuning. However, since the complexity of multi-tier application services increase quickly, and DBMS offers increasingly many tuning knobs, it is hard and expensive for several experts to adjust these knobs. Thus DBMS should be autonomic: self-managing, self-monitoring, self-healing, and self-tuning.

Although this paper focuses on the problem of automated physical database design, it is closely interrelated to the other self-tuning database technologies, such as statistics management [26, 20, 4], monitoring and diagnostics [22, 33, 16, 24], automated memory [14, 25], and automated storage/data layout management. In order to automatically select a physical design, we need to recommend a set of database statistics to ensure that the optimizer has the necessary statistics to generate plans that leverage the recommended indexes.

2.1.1 Physical Design & Configuration

The performance of a query depends on the execution plan. Execution plans chosen by the optimizer depend on statistics created by the optimizer and physical design objects that exist. Physical design configuration includes two aspects, indexes and materialized views. In order to get the optimal settings, it is required to formulize the problem. In any auto-tuning problem, it is not possible to find a parameter setting that yields universally close-to-optimal performance. So the goal is usually to get the configuration from solving the below function given workload and performance goal [23]:

$$workload * config \rightarrow performance\ metrics$$

In physical design, the workload consists of queries and updates; the configuration is a set of indexes, materialized views and partitions from a search space. While the upper bound is based on storage space, the goal is to pick a configuration with the lowest cost for the given database and workload. However, early approaches before 1997 [38, 31, 27, 28, 36] did not always agree on what constitutes a workload, or what should be measured as the cost for a given query and configuration. These papers stated that it is too disruptive to estimate goodness of a physical design for a workload by execution cost. The VLDB 1997 paper [18] and the SIGMOD 1998 paper [19] from Microsoft build the "What-If" index architecture to recommend indexes for the given workload. This architecture uses optimizer estimated costs, and is sufficient to fake existence of physical design. It is a reasonable way to measure goodness of a design. "What-If" architecture also extends to handle other physical design structures like materialized views and partitioning.

While we can formalize the "goodness" of a design, the search algorithm is crucial in automated physical design. The introduction of materialized views and partitioning results in an explosion in the search space. There are three techniques: pruning table and column sets, merging, and enumeration, which enable physical design tools to explore this large space in a scalable manner. [5, 7, 9] describes the importance and methods in pruning. After we get all candidate selection, the union of these "best" configurations may violate storage constraints and increase the maintenance costs for update queries. Thus [7, 11] propose the merging algorithm for these candidates. So the search space is the union of the "locally best" and "merged" indexes and materialized views. Since the selection problem has been shown to be NP-Hard [15], the focus of related work has been on developing heuristic solutions that give high quality recommendations that are scalable. Broadly the search strategies explored thus far can be categorized -s bottom-up [18, 35, 43] or top-down [8] search. The bottom-up strategy begins with an empty set and adds structures in a greedy manner. In contrast, the top-down approach begins with a globally optimal configuration, but it would be infeasible if it exceeds the storage bound.

2.1.2 *Commercial Tools*

Microsoft, IBM and Oracle database systems ship with automated physical design tools. These include Database Engine Tuning Advisor (DTA) [18, 19, 6] from Microsoft SQL Server, DB2 Design Advisor [40, 43, 44] from IBM DB2, and the SQL Access Advisor [24] from Oracle 10g.

In 1998, Microsoft SQL Server 7.0 was the first commercial DBMS to ship a physical database design tool, called the Index Tuning Wizard. Then in the release of Microsoft SQL Server 2005, it was replaced by a full-fledged application DTA that can provide integrated recommendations for indexes, indexed views, indexes on indexed views and horizontal range partitioning. The most important attribution for Microsoft is to propose use optimizer estimated costs and build a "what-if" component in optimizer. The "what-if" component is very important. It supports Hypothetical Configuration Mode to allow create indexes and MVs but only statistics information, no physical strutures. The MV candidates are only single block, and generated by syntactic structure.

IBM's DB2 shipped the DB2 Advisor in 1999 that could recommend indexes. Subsequently, the DB2Design Advisor Tool in DB2 version 8.2 provides integrated recommendations for indexes, materialized views, shared-nothing partitioning and multi-dimensional clustering. Some basic ideas are very similar to Microsoft. It also builds a new 'EXPLAIN' mode to build hypothetical configuration. However The difference is that Candidate generation and selection. The candidate materialized views is just in the format Select-From-Where-GroupBy. This is like Microsoft. But DB2 generates more materialized views from logical views and the common sub-expressions finding from multiple query optimization (MQO). MQO is a special technique in DB2. It is to build a graph and find the common small SQL statement. So DB2 considers multi-block queries. Comparing to Microsoft, DB2 supports back-joins. It is designed to have independent advisors for each physical design structure. The search step that produces the final integrated recommendation iteratively invokes each of the advisors in a staged manner. Besides, the selection algorithm is also different. It gives some metrics to model like a knapsack problem. First generate a initial solution, then iterative improvement and randomly swaps.

Oracle 10g purposes a advisor named SQL Access Advisor. We can know it is also based on the idea of Microsoft. Oracle also builds a new mode, it changes the regular optimizer to automatic tuning optimizer. This new optimizer includes the component SQL Profile. Each SQL statement has its own SQL Profile that store its auxiliary information. SQL Profile also generates some recommendation for each SQL statement. Then SQL Access Advisor collects these advice, and consolidates them into a global advice for the entire SQL workload. The big difference of oracle advisor is that it builds a tuning life cycle. First Oracle only cares about high-load SQL statements detected by Automatic Database Diagnostic Engine (ADDM) and SQL statement defined by users from USER component. Then Automatic SQL Tuning generates and stores the SQL Profile in SQL Tuning Base. We say this is a loop. It is because after we improve the execution plan of the set of SQL statements. Hence reduce their impacts on the system. So in the next cycle, it will begin with a different set of high-load SQL statements.

These three commercial databases dominate this area. But we cannot get their code or detail implementation. This is the reason that we want to build a open source project for self-tuning. Learning from these three commercial databases, there are several common ideas. First they all estimates cost by optimizer. Second, they all need to modify optimizer to support "what-if" mode. Then define metrics and find candidate indexes and materialized views. They all need to implement a component to rewrite SQL. Finally, they all select indexes and materialized views together to generate the final configuration.

2.2 Materialized View Selection

Materialized views in database systems have been shown to greatly improve the query performance [17] in data warehouses and decision support systems. There has been much research in the selection of optimal materialized views and when to calculate them given constraints on disk-size or maintenance cost. These methods differ in whether the views are created in preprocessing in static view selection or as the data is processed in dynamic view selection.

2.2.1 *Static View Selection*

Static view selection (i.e. data warehouse configuration [39]) selects materialized views offline based on a given workload given constraints on disk-space or the maintenance cost. The disk-space is the amount of space that the views are allowed to take on the disk while the maintenance cost is the cost associated with updating the views. Maintenance is expensive and in some cases requires the database to be temporarily put offline in order for the views to be updated. Once a static materialized view has been determined, it will not be changed, but may be updated when the data in the underlying tables change.

Gupta et al. [30] and Theodoratos et al. [39] stressed the importance of materialized view selection in a data warehouse in order to minimize total query time while limiting the total cost of disk-space or maintenance. Gupta et al. developed polynomial-time heuristics for determining OR view graphs and AND view graphs within a constant factor of the optimal solution for the disk-space constraint. They further studied view selection under the maintenance cost constraint using an exponential A* heuristics which they found almost always returned the optimal solution. Theodoratos et al. focused strictly on constraining to the maintenance cost of materialized views. They proposed having auxiliary materialized views in a data warehouse in order to reduce the cost of maintaining the other materialized views by reducing the number of accesses to base relations when accessing the data warehouse for an update.

Baralis et al. [10] researched the selection of views in a multidimensional database with the intention of finding candidate views that limit the total update cost. They define the update cost associated with materializing a view to be the cost of updating the set of materialized views multiplied by the frequency of insertions to the fact table. They select candidate views for materialization with the criteria that they must be associated with some query in the workload or that the addition of a view and the removal of two other views would decrease the total cost of the materialized views. Baralis et al. prove that the addition of any non-materialized view will not decrease the total cost. A heuristic reduction is then applied on the candidate views to remove any view that is not expected to contribute to the optimal query solution.

Maintenance of static materialized views is important in ensuring that the data materialized in your views is current up to some degree. Mistry et al. [34] suggested updating materialized views by exploiting common subexpressions and whether to recompute or to incrementally update when the underlying data has changed. Zhou et al. [42] presented the idea of lazily updating views only when the system has free cycles or a query directly references a view. Chaudhari et al. [13] explored incremental updates of materialized views for streaming heterogenous data sources.

2.2.2 *Dynamic View Selection*

Static view selection is limited in that it computes the materialized views based upon a pre-compiled workload of queries with pre-compiled constraints. The pre-compiled queries may not reflect future queries and the pre-compiled constraints may become inaccurate over time such as when the maintenance window shrinks or the disk-space increases. View caching or dynamic view selection is a proposed solution to these limitations.

Caching in database systems stores pages, tuples, queries or even entire views that have been previously executed or that are executed often. This allows previously executed queries to be executed more efficiently as they have been cached. Caching also is used to reduce the amount of calls to the base relations by decomposing queries into cached parts and non-cached parts where only the non-cached parts need to be computed from the base relations. Scheuermann et al. [37] extended caching to design WATCHMAN, a database cache manager for data warehouses that aims to minimize query response time. WATCHMAN assumes that a data warehouse has infrequent updates and that past query patterns will reflect future query patterns. As a result, entire retrieved sets of queries are stored in the cache manager instead of individual pages or tuples. A query is only admitted into the cache if its addition lowers the overall cost of query execution.

For databases which are not static or where the requests may change over time, it is important to adapt your materialized views in order to reflect changes in the underlying data and changes in the users' queries. Kotidis et al. [32] created the dynamic view management system, DynaMat, which dynamically materializes views from incoming queries taking into account both the cost of the time necessary for updates and of the disk-space available. DynaMat uses dedicated disk storage in order to manage its materialized data that supports any underlying storage structure as long as there is an accompanying cost model for querying and updating the materialized views. The authors found that DynaMat outperformed optimal pre-computed static view solution, but more importantly it is a self-tunable system that changes its views based upon incoming queries.

3. OVERVIEW OF APPROACH

In this section, we describe the data that was used and the processing on the data that resulted in the selection of materialized views. We also give a brief overview of how we incorporated materialized views into PostgreSQL.

3.1 Data & Preparation

For this research, we chose to use the TPC Benchmark H(TPC-H) support benchmark [3]. TPC-H allows for the creation of variable sized databases and auto-generates a sample workload given a query template with pseudo-random values. We auto-generated a one gigabyte database with the schema shown in Figure 1. We also auto-generated our sample query workload from twenty-two unique query templates. We consider both the data and the query workload to be static as is often the case in data warehouses where materialized views are often used.

3.2 Process

After creating our database and sample query workload as described in Section 3.1, we ran through each query in the workload and generated candidate materialized views as detailed in Section 4. Once we have generated a list of candidate materialized view definitions, we create pseudo-materialized views along with their corresponding non-materialized views for each of these candidates which is detailed in Section 3.3. After the candidate materialized views have been created in the database, we want to evaluate the reduction in cost to the query execution time by first rewriting the queries as described in Section 5.1 and then evaluating their cost and selecting a subset of the candidate materialized views to keep as described in Section 5.2.

3.3 Materialized Views in PostgreSQL

As of PostgreSQL 9.2, materialized views are not built-in constructs. This means that there is a lack of the following features available in some commercial databases [1]:

- "CREATE MATERIALIZED VIEW" type syntax
- Built-in auto-updates of materialized views
- Query planner does not automatically accelerate queries by substituting applicable materialized views

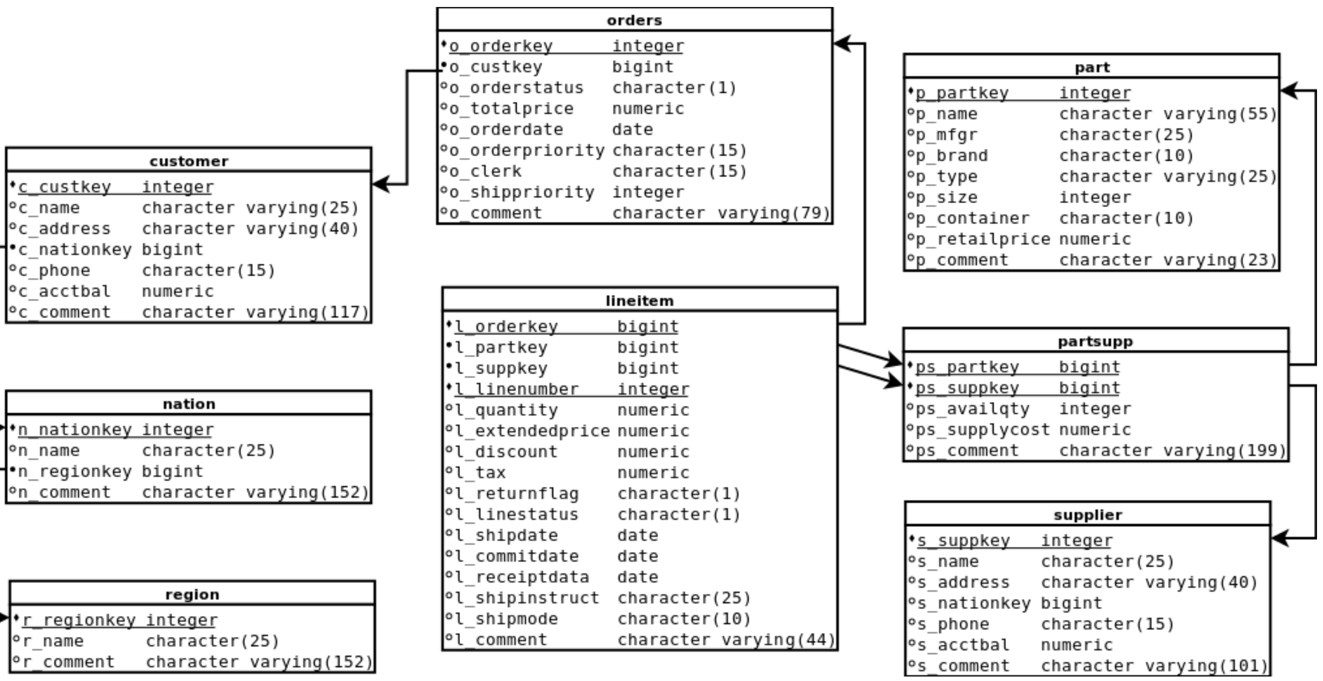

Figure 1: TPC-H Schema

However, there have been multiple workarounds for materialized views in PostgreSQL by taking advantage of user-defined functions and table triggers [12, 29].

We chose to use the pseudo-materialized view implementation proposed by Jonathan Gardner [29]. To begin we created a table, *matviews*, in our database that contains information that includes the name of the materialized view, the name of the view that populated it and the last refresh of the materialized view. We then defined a function in PL/pgSQL that creates a "materialized view" given a name for the "materialized view" and a view that will be its definition. This subsequently adds a row to the *matviews* table and defines a new table representing the materialized view with the chosen name. In addition, two additional functions are defined to drop a materialized view in the database and to refresh a materialized view. As mentioned in Section 3.1, we assume that the data is static so we do not deal with updating or refreshing a materialized view once it has been created, but include the refresh function in case of non static databases.

With the above additions to the databases, the problems mentioned before of not having built-in materialized views are resolved by creating a table to represent a materialized view, periodically taking snapshots or using triggers to update materialized views and to manually specify the materialized view to use in queries.

4. CANDIDATE MATERIALIZED VIEW GENERATION

After parsing the workload, we begin our selection of candidate materialized view. First let us think about what materialized views are relevant for a workload. A materialized view that is useful for a workload must be syntactically relevant for at least one SQL query. So there are two basic ideas for finding materialized views that reduce execution cost:

1) specific to each SQL query

2) common and high-load queries

The second idea is used in DB2 and Oracle [24, 40]. In our project, we just focus on the first idea. However, even though for a single SQL statement, considering all syntactically relevant materialized views is not scalable.

Consider a query Q:

```
SELECT c.c_name
FROM customers c, orders o
WHERE o.o_custkey = c.c_custkey
      and o.o_totalprice > 1000.
```

The following materialized views are syntactically relevant:

MV1:
```
SELECT c.c_name
FROM customers c, orders o
WHERE o.o_custkey = c.c_custkey
      and o.o_totalprice > 1000;
```

MV2:
```
SELECT c.c_name, o.o_totalprice
FROM customers c, orders o
WHERE o.o_custkey = c.c_custkey
GROUP BY o.o_totalprice;
```

MV3:
```
SELECT c.c_name, o.o_orderdate, o.o_totalprice
FROM customers c, orders o
WHERE o.o_custkey = c.c_custkey
GROUP BY o.o_totalprice;
```

MV4:
```
SELECT c.c_name, o.o_totalprice, l.l_tax
FROM customers c, lineitem l, orders o
WHERE o.o_custkey = c.c_custkey
      and o.o_orderkey = l.l_orderkey
      and o.o_totalprice > 1000;
```

The space of syntactically relevant materialized views for a query (and hence a workload) is very large, since in principle, a materialized view can be proposed on any subset of tables in the query, even containing the table that does not occur in the original query like $MV4$. Based on these, we add some constraints for candidate materialized views generating from each SQL statement.

- Focus on the class of single-block materialized views consisting of selection, join, grouping and aggregation
- For a particular query, generate materialized views just based on this query (i.e. $MV4$ won't be considered)

Adding these two constraints, there is still an explosion in the space of materialized views arising from selection conditions and group by columns in a particular query. If there are m selection conditions in the query on a table-subset T (a table-subset is a subset of tables referenced in a query in the workload.), then materialized views containing any subset of these selection conditions are syntactically relevant. Therefore, the goal of candidate materialized view generation is to quickly eliminate materialized views that are syntactically relevant for a particular query in the workload, but are not important in the whole workload. In order to achieve this goal, we should know how to measure "importance" for a materialized view. The naive approach of selecting one candidate materialized view per query that exactly matches each query in the workload does not usually work since this materialized view usually cannot be used by other queries. So the fist observation is that the materialized view that could be used by more SQL statement is more important. This observation [7] is even more severe in large workloads. The following simplified example of $Q1$ from the TPC-H benchmark illustrates this point: Example 1. Consider a workload consisting of 1000 queries of the form:

```
SELECT l_returnflag, l_linestatus, SUM(l_quantity)
FROM lineitem
WHERE l_shipdate BETWEEN <Date1> and <Date2>
GROUPBY l_returnflag, l_linestatus.
```

Assume that each of the 1000 queries has different constants for $< Date1 >$ and $< Date2 >$. Then, rather than recommending 1000 materialized views, the following materialized view that can service all 1000 queries may be more attractive for the entire workload:

```
SELECT l_shipdate, l_returnflag, l_linestatus,
       SUM(l_quantity)
FROM lineitem
GROUPBY l_shipdate, l_returnflag, l_linestatus.
```

Another observation is that a materialized view that requires more time to calculate is more important [7]. Example 2. Consider the TPC-H 1GB database and the workload specified in Section 3.1. There are several queries involving the tables: lineitem, orders, nation, and region. However, it is likely that materialized views proposed on the table-subset lineitem, orders are more useful than materialized views proposed on nation, region. This is because the tables lineitem and orders have 6 million and 1.5 million rows respectively, but the tables nation and region are very small (25 and 5 rows respectively). Hence, the benefit of pre-computing the portion of the queries involving lineitem and orders is significant.

Based on these facts, we separate the task of candidate materialized view generation into three parts: (1) from the large space of all possible table-subsets, we prune to get a small set of interesting table-subsets. (2) Based on interesting table-subsets, generate the best materialized view for each query. (3) According to these materialized views, generate some candidate materialized views that are good for the whole workload.

4.1 Interesting Table-Subsets

Since it is required to prune the space of possible materialized views. An efficient way is to find all of the interesting table-subsets in the workload as proposed by Microsoft [7]. The interesting table-subset is a set of several tables which often appear together in the workload. So it also follows the two general observations mentioned before. Then we can define two metrics to capture the importance of a table-subset. For a particular table-subset T, we define Q_T as the queries in the workload where table-subset T occurs.

$$TSCost(T) = \textstyle\sum_{q \in Q_T} COST(q)$$

$$TSWeight(T) = \sum_{q \in Q_T} \frac{COST(q) \times (sum\ of\ tables\ sizes\ in\ T)}{sum\ of\ tables\ sizes\ in\ q}$$

Here $COST(q)$ is the estimated cost from optimizer, i.e., 'EXPLAIN' q in PostgreSQL (described in more detail in Section 5.2). $TSCost$ comes from the first observation. $TSWeight$ derives from the two observations. Then our task is to find all the table-subsets T that $TSWeight(T)$ is higher than a given threshold C. Since $TSWeight(T) \leq TSCost(T)$, we can find all the table-subsets T that $TSCost(T) \geq C$, then filter the result by $TSWeight(T) \geq C$. $TSCost$ metric has the property of "monotonicity" such that for table subsets T_1, T_2, $T_1 \subseteq T_2 \Rightarrow TSCost(T_1) \geq TSCost(T_2)$. The Algorithm 1 shows the detail. Threshold C can be set as 10% of the total workload cost.

```
Data: $C$ is the threshold
Result: $ITS$ list of all interesting table-subsets
$S_1 \leftarrow \{T|T$ is a table-subset of size 1 satisfying
$TSCost(T) \geq C\}$;
$i \leftarrow 1$;
while $i <$ MAX-TABLES and $|S_i| > 0$ do
    $S_{i+1} \leftarrow \{\}$;
    $G \leftarrow \{T|T$ is a table-subset of size $i+1$ and $\exists s \in S_i$
    such that $s \subset T\}$;
    foreach $T \in G$ do
        if $TSCost(T) \geq C$ then
            $S_{i+1} \leftarrow S_{i+1} \cup \{T\}$
        end
    end
    $i++$;
end
$S \leftarrow S_1 \cup S_2 \cup .. \cup S_{MAX-TABLES}$;
return $\{T|T \in S$ and $TSWeight(T) \geq C\}$;
```

Algorithm 1: Finding interesting table-subsets

4.2 Generating Materialized Views for Each Query

After finding the interesting table-subset step, we can get some basic materialized views that simply join all the tables in each table-subset, but their sizes are usually very large. This step we want to generate the best materialized views for each query. Given a select query q, we only the generating of materialized views from interesting table-subsets. We first generate all the available materialized views for this query, and then select the best one based on the estimated cost. The estimated cost of materialized view M is the EXPLAIN cost to get the result of q using M, if M has been created.

The available materialized views for a query q must be general. Their format is only SELECT-FROM-WHERE-(GROUP BY) where GROUP BY is optional. For simplicity in this initial implementation, we only consider equality in the WHERE clause. All of the other conditions including $<$,$>$,$\leq$,$\geq$, are moved to the GROUP BY clause. From these, we can get the available materialized views from the following steps. For each interesting table-subset T that occurs in q, we can find: (1) A "pure-join" materialized view on T containing join and conditions for join tables. (2) all the combination of GROUP BY columns and aggregate expression from q on tables in T. (3) keep a subset of = conditions in query q and on tables in T, and move a subset of the other conditions related to T to GROUP BY. For example, $V2$ and $V3$ mentioned at the beginning of this section are available materialized views, while $V1$ and $V4$ are not available. If $\{customers, orders, suppliers\}$ is a interesting table-subset, the following materialized view is available.

```
SELECT c.c_name, o.o_totalprice, l.l_shipdate
FROM customers c, lineitem l, orders o
WHERE o.o_custkey = c.c_custkey
      and o.o_orderkey = l.l_orderkey
GROUP BY o.o_totalprice;
```

4.3 Generating More Common Materialized Views

Although we haven't implemented this part in our project, it is good to discuss the existing idea and design. In this step, the input is all the materialized views generated before. We require some candidate materialized views that are useful for the workload. It is an improvement on our existing work with the goal to make the materialized views more general. Microsoft uses the "Merge" algorithm; IBM DB2 implements MQO technique to generate small common queries; Oracle finds the high-load queries from ADDM and USER defined. In PostgreSQL, we believe the "Merge" algorithm is the best choice. Microsoft describes the detail of "merge" algorithm in [7] and another "merge" algorithm proposed in [11].

5. CANDIDATE SELECTION

After a list of candidate materialized views has been selected, *cand_matviews*, each materialized view needs to be evaluated with respect to performance improvement in relation to the sample query workload Q. This involves rewriting each of the sample queries in Q to use eligible candidate materialized views. After the rewrite is complete, then candidate materialized views are selected that are both eligible for the workload and improve on the performance time of one of the original queries.

5.1 Query Rewrite

Query rewriting involves rewriting each query of the query workload Q using each of the candidate materialized views in *cand_mvs* to determine if they are eligible. An eligible

```
Data: Q list of sample queries
Data: cand_matviews list of candidate materialized
      views
for curr_query in Q do
    original_cost = calculate curr_query cost;
    curr_min_cost = original_cost;
    mv_to_use = nil;
    for curr_mv in cand_matviews do
        if curr_mv is eligible for curr_query then
            curr_rewrite = rewrite(curr_query);
            curr_cost = calculate curr_rewrite cost;
            if $curr\_cost < curr\_min\_cost$ then
                curr_min_cost = curr_cost;
                mv_to_use = curr_mv;
            end
        end
    end
    if mv_to_use is not nil then
        rewrite curr_query to use mv_to_use;
    end
end
```

candidate materialized view for a given query is a materialized view that satisfies both of the following constraints:

Requirement 1: MV view definition contains at least two tables from the current query

Requirement 2: Any constraints of MV are supersets to any constraints of the current query

These two requirements for eligible candidate materialized views guarantee that the query will have the same results regardless of whether it is the original query or the query rewritten with a materialized view. For example, given the TPC-H database described in Section 3.1 the following query is used to get the name, date, amount of order and shipdate for all orders over 1000:

query:
```
select  c.c_name, o.o_orderdate,
        o.o_totalprice, l.l_shipdate
from    customer c, lineitem l, orders o
where   o.o_custkey = c.c_custkey
        and o.o_orderkey = l.l_orderkey
        and o.o_totalprice > 1000;
```

The first step is to determine whether a candidate materialized view definition is eligible for use in a query. In this case, we are only concerned about the *query* defined above which extracts data from customers, lineitem and the orders tables after joining the tables on *custkey* and *orderkey*. This statement, *query*, has the constraint that the value of *o_totalprice* must exceed 1000. In this case eligible candidate materialized views are materialized views which have at least two of the tables and their constraints are a superset of the constraints of *query*.

Let the candidate materialized views of *cand_mvs* be defined as follows:

CMV1
```
select * from customers c, orders o
where o.o_custkey = c.c_custkey;
```

Eligible: *CMV1*'s materialized view definition is a join on *customers* and *orders* on their *custkey* value. This satisfies Requirement 1 for eligible candidate materialized views as it contains at least two of the tables from the original query. It also satisfies Requirement 2 of eligible candidate materialized views by joining on the same attributes as *query* and not adding any more restrictive constraints.

Query Rewrite:

```
select mv.c_name, mv.o_orderdate,
       mv.o_totalprice, l.l_shipdate
from lineitem l, MV1 mv
where mv.o_orderkey = l.l_orderkey
      and mv.o_totalprice > 1000;
```

CMV2
```
select * from customers c, orders o
where o.o_custkey = c.c_custkey
      and o.o_totalprice > 500;
```

Eligible: *CMV2*'s materialized view definition is a join on *customers* and *orders* on their *custkey* value along with a constraint that the *o_totalprice* of the *orders* table must be greater than 500. This satisfies Requirement 1 for eligible candidate materialized views as it contains at least two of the tables from the original query. It also satisfies Requirement 2 of eligible candidate materialized views by joining on the same attributes as *query* and adding a constraint that *o.o_totalprice* be greater than 500 which is a superset of *query*'s constraint that *o.o_totalprice* be greater than 1000 (i.e. *o.o_totalprice* $>$ 500 is a superset of tuples returned by *o.o_totalprice* $>$ 1000).

Query Rewrite:

```
select mv.c_name, mv.o_orderdate,
       mv.o_totalprice, l.l_shipdate
from lineitem l, MV2 mv
where mv.o_orderkey = l.l_orderkey
      and mv.o_totalprice > 1000;
```

CMV3
```
select * from customers c, orders o
where o.o_custkey = c.c_custkey
      and o.o_totalprice > 1000;
```

Eligible: *CMV3*'s materialized view definition is a join on *customers* and *orders* on their *custkey* value along with a constraint that the *o_totalprice* of the *orders* table must be greater than 1000. This satisfies Requirement 1 for eligible candidate materialized views as it contains at least two of the tables from the original query. It also satisfies Requirement 2 of eligible candidate materialized views by joining on the same attributes as *query* and adding the same constraint that *o.o_totalprice* be greater than 1000.

Query Rewrite:

```
select mv.c_name, mv.o_orderdate,
       mv.o_totalprice, l.l_shipdate
from lineitem l, MV3 mv
where mv.o_orderkey = l.l_orderkey
```

CMV4
```
select * from customers c, orders o
where o.o_custkey = c.c_custkey
      and o.o_totalprice > 2000;
```

Not Eligible: *CMV4*'s materialized view definition is a join on *customers* and *orders* on their *custkey* value along with a constraint that the *o_totalprice* of the *orders* table must be greater than 2000. This satisfies Requirement 1 for eligible candidate materialized views as it contains at least two of the tables from the original query. However, it does not satisfy Requirement 2 as it places a more restrictive constraint on the value of *o.o_totalprice* to be greater than 2000 which would mean that we would potentially lose values between 1000 and 2000 if the query were rewritten in

CMV5
```
select * from customers c
where c.c_name = 'Bob Smith';
```

Not Eligible: *CMV5*'s materialized view definition only looks at the *customers* table with the constraint that the name equals "Bob Smith". This fails Requirement 1 in that it only has less than two tables from *query* and it also fails Requirement 2 in that it adds more constraints than *query* has. Substitution with this materialized view could give different results than execution of the original query.

In this manner, keep track of each candidate materialized view in *cand_matviews* that is eligible for the queries in *Q*. Only the eligible candidate materialized views and their rewrites will be evaluated during the rewrite phase of the candidate view selection.

5.2 Selection

For a given query in *Q*, the goal is to determine whether the original query can be sped up by substituting a materialized view into the original query. This process involves calculating the cost of the original query and comparing this cost to the cost of each of the eligible candidate materialized view rewrites described in Section 5.1. Our approach operates under no disk space constraint, but we propose a knapsack approach for dealing with disk space limits.

In calculating the cost or run time of a query, PostgreSQL has two methods which are prepending 'EXPLAIN' or 'EXPLAIN ANALYZE' to a query [2]. Prepending 'EXPLAIN' to a query outputs the expected query plan and the expected cost of execution. 'EXPLAIN' does not actually run the query so these cost estimates may be inaccurate as it relies on statistical information of the database. Prepending 'EXPLAIN ANALYZE' to a query actually runs the statement and outputs the expected cost of execution, the actual cost of execution and the actual query plan that was used. Since 'EXPLAIN ANALYZE' runs the query it can take much longer to execute and the actual costs may be influenced by query caching.Due to the speed of EXPLAIN and the requirement that we only need estimates, we used 'EXPLAIN' for candidate materialized view generation. For candidate materialize view selection, we needed actual costs not based on statistical analysis to rewrite our queries so we used 'EXPLAIN ANALYZE'. Query caching may result in longer initial query costs, but reduced costs in subsequent executions. To avoid the potential varying times and under the assumption that the sample workload is executed often we chose to keep the cache warm by executing the query two times before evaluating its cost then we average three subsequent runs to get a cost.

With the assumption that we are optimizing for query run time with no restriction on disk space, we estimate the cost for each of the original queries and their eligible candidate materialized views using the method described above. The query with the lowest cost is used whether it be the original query or a query rewrite with a materialized view. If the query with the lowest cost is a query rewrite with a materialized view, then this materialized view is marked as being used and the difference in cost between the original query's execution and its own is recorded. This process continues until all queries in the workload have completed. After all queries have been processed, the candidate materialized views that were used to improve the performance of at least one query are stored and the remaining candidate materialized views are removed. This process eliminates any candidate materialized views that do not benefit the query workload as a whole while keeping all materialized views that lower the cost of the query workload.

More often than not, a database will have a restriction on the amount of disk space that can be used by the addition of materialized views. Under a disk space constraint, the ideal solution will maximize the total benefit (i.e. minimize total cost of query workload) by adding materialized views in relation to the amount of space that they require on disk. In this case, let *reduction* be a two dimensional array of size $|Q| * |eligible\ MV|$ where the first index references a query in Q and the second index references an eligible materialized view. For example, $reduction[i][j] =$ is the reduced cost of query i by rewriting it using eligible materialized view j. If there is no reduction in cost by rewriting query i with eligible materialized view j, then assign a value of 0 to $reduction[i][j]$. Let $disk(mv\_j)$ be the amount of space that materialized view j takes up on disk. With these two definitions, an estimate on the value of a materialized view can be computed as:

$$value\ of\ mv\_j = \frac{\sum_{n=1}^{|Q|} reduction[n][j]}{disk(mv\_j)}$$

This definition accounts for the reduction in cost in terms the amount of disk space that is required to store the materialized view. A dynamic programming knapsack algorithm can then be used to calculate which materialized views should be added given the values calculated in the equation above and the weight defined to be the amount of disk space reserved for the materialized views. Note that this algorithm can be improved by recalculating the cost reduction of each materialized view each time a materialized view is added so as not to include cost benefits of materialized views that will not be used due to other views giving higher cost reductions (i.e. if materialized view A improves query q by x and materialized view B improves query q by y and A is added by knapsack algorithm and $x > y$ then do not include y in the benefit for B since it will not be used). However, when the disk space allocated for the materialized view is very large dynamic programming cannot be used. In this case, an initial solution can be found using a greedy algorithm on metric $value\ of\ mv$. First, select materialized views with high values as disk space allows. After finding an initial solution, the solution can be justed by randomly doing swaps to possibly get better knapsack results. Choose the result with the highest value as defined above.

6. EXPERIMENTS

T	ER	IR	< 10%	10-20%	20-50%	> 50%
10%	8	5	2	0	2	1
2%	10	8	2	3	2	1

Table 1: Query improvements for 10% and 2% thresholds where T stands for threshold, ER for eligible query rewrite and IR for improved query rewrites

We evaluated the algorithm proposed in this paper using the data described in Section 3.1, generated interesting tables subset:

Threshold 10%:

orders,lineitem,customer

partsupp,part

orders,lineitem

Threshold 2%:

orders,lineitem,customer,nation

partsupp,supplier

orders,lineitem

partsupp,part

orders,customer

orders,lineitem,customer

region,orders,lineitem,customer,nation

partsupp,nation,supplier

and then generated candidate materialized views for threshold values of 10% and 2% of the overall query workload. As stated in Section 5.2, candidate materialized view generation used query plan estimates while materialized view selection used actual query costs. The actual query costs were computed by first running the query twice and ignoring the results and immediately running the query three more times and taking the average of these last three runs to be the cost. An overview of the results is shown in Table 1

The 10% threshold resulted in the generation of three candidate materialized views. Of the twenty-two queries in our workload, eight queries could be written using at least one of the three generated materialized views. Five of the query rewrites lowered the cost of the query using two of the three generated candidate materialized views. Four of the five improved query rewrites were from one of the materialized views while only one was from the other materialized view and resulted in the lowest cost improvement (6%). The two materialized views require 1.85 GB on disk.

The 2% threshold resulted in the generation of eight candidate materialized views. Of the twenty-two queries in our workload, ten queries could be written using at least one of the eight generated candidate materialized views. Of the ten queries that were rewritten, eight of the queries had reduced costs by using a materialized view with performance improvements ranging from 6% to 57%. Four of the eight generated materialized views were selected and required 2.7 GB on disk.

7. FUTURE WORK

Our goal is to build an open source project for PostgreSQL to support automatic physical design advisor. Up until now, we have implemented several components such as a parser, a materialized view generator, SQL rewriting, and materialized view selection. These parts cooperate well under PostgreSQL. The next step is to improve the performance for each component. In materialized view generation, we need to follow the idea to generate all the available materialized views, and then select the best materialized view for each query. After these, we can use a "merge" algorithm to improve the performance for the whole workload. Furthermore, how to generate materialized views that are useful for the whole workload is a really open problem. We can try new algorithms. It is easy to plug in new algorithms in our design.

Second, the PostgreSQL optimizer must be modified to support "What-If" mode. It needs to change the source files of PostgreSQL. First we should modify PostgreSQL to allow user to create hypothetical indexes and materialized views. For these hypothetical physical structure, PostgreSQL only keeps statistics information, but no physical structures. The commend should like

```
CREATE INDEX A on Table WITH STATISTICS_ONLY;
CREATE MATERIALIZED VIEW B as Query
WITH STATISTICS_ONLY;
```

This step is just to remove something from CREATE source code. The second step is to modify optimizer to use these hypothetical physical structures. It should allow user to define the running mode, e.g., the hypothetical mode or the regular mode. Under hypothetical mode, all the datasets and are fixed. PostgreSQL only accepts: (1) the CREATE and DELETE statements to create and delete hypothetical physical structure; (2) the EXPLAIN query to get the estimated cost. The benefit of "What-if" mode is reducing the time to measure the importance of materialized views. We do not need to create the actual physical structure, but just statistics information used by optimizer. The difficult of this point is that we need to be more familiar with PostgreSQL source code and modify carefully.

Third, it requires to implement the components to recommend indexes on both original tables and materialized views. All the previous papers show that materialized views with indexes can improve more performance. The three commercial databases all implement index advisors, we can implement a similar one under PostgreSQL. Indexes advisor under PostgreSQL should be much easier than materialized views advisor since PostgreSQL supports index feature pretty well. After create indexes, optimizer can automatically choose the best index for each query. So it does not require query rewriting component in advisor at least. After finish index advisor, we can joint enumeration of candidate indexes and materialized views.

8. CONCLUSION

This open source project borrows some ideas from the three commercial databases: Microsoft Server, IBM DB2, and Oracle. We have implemented almost all the components except "What-If" mode. Our components run perfectly under PostgreSQL. We can generate some simple candidate materialized views, rewrite the queries to utilize materialized views, and select the best one. The experiments in TPC-H show that only using materialized views can still improve the performance significantly. TPC-H has many queries that are multi-block and cannot use the materialized views well. But in real world, single-block query is more common, and can get more benefits from materialized views. In summary, this open source project is really worth exploring. We first can break the domination of these commercial databases in this topic; second can push PostgreSQL to develop the materialized view feature; third can let researchers know that there are still many remaining problems in this topic. According the future work description, we will continue to pursue our long-term goal of a complete automatic physical design advisor for PostgreSQL.

For more information on this project including the test database and the client code feel free the authors by the emails listed on the first page.

9. REFERENCES

[1] Materialized Views. `http://wiki.postgresql.org/wiki/Materialized_Views`.

[2] PostgreSQL 9.2.4 Documentation - Using EXPLAIN. `http://www.postgresql.org/docs/9.2/static/using-explain.html`.

[3] TPC-H is an ad-hoc, decision support benchmark. `http://www.tpc.org/tpch/`.

[4] A. Aboulnaga, P. J. Haas, S. Lightstone, G. M. Lohman, V. Markl, I. Popivanov, and V. Raman. Automated statistics collection in db2 udb. In *VLDB*, pages 1146–1157, 2004.

[5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.

[6] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database tuning advisor for microsoft sql server 2005. In *VLDB*, pages 1110–1121, 2004.

[7] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, pages 496–505, 2000.

[8] S. Agrawal, E. Chu, and V. R. Narasayya. Automatic physical design tuning: workload as a sequence. In *SIGMOD Conference*, pages 683–694, 2006.

[9] S. Agrawal, V. R. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD Conference*, pages 359–370, 2004.

[10] E. Baralis, S. Paraboschi, and E. Teniente. Materialized views selection in a multidimensional database. In *VLDB*, pages 156–165, 1997.

[11] N. Bruno and S. Chaudhuri. Physical design refinement: The "merge-reduce" approach. In *EDBT*, pages 386–404, 2006.

[12] D. Chak. Materialized Views that Really Work. `http://www.pgcon.org/2008/schedule/events/69.en.html`, 2008.

[13] M. B. Chaudhari and S. W. Dietrich. Metadata services for distributed event stream processing agents. In *SEDE*, pages 307–312, 2010.

[14] S. Chaudhuri, E. Christensen, G. Graefe, V. R. Narasayya, and M. J. Zwilling. Self-tuning technology in microsoft sql server. *IEEE Data Eng. Bull.*, 22(2):20–26, 1999.

[15] S. Chaudhuri, M. Datar, and V. R. Narasayya. Index selection for databases: A hardness study and a principled heuristic solution. *IEEE Trans. Knowl. Data Eng.*, 16(11):1313–1323, 2004.

[16] S. Chaudhuri, A. C. König, and V. R. Narasayya. Sqlcm: A continuous monitoring framework for relational database engines. In *ICDE*, pages 473–484, 2004.

[17] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, pages 190–200, 1995.

[18] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *VLDB*, pages 146–155, 1997.

[19] S. Chaudhuri and V. R. Narasayya. Autoadmin 'what-if' index analysis utility. In *SIGMOD Conference*, pages 367–378, 1998.

[20] S. Chaudhuri and V. R. Narasayya. Automating statistics management for query optimizers. *IEEE Trans. Knowl. Data Eng.*, 13(1):7–20, 2001.

[21] S. Chaudhuri and V. R. Narasayya. Self-tuning database systems: A decade of progress. In *VLDB*, pages 3–14, 2007.

[22] S. Chaudhuri, V. R. Narasayya, and R. Ramamurthy. Estimating progress of long running sql queries. In *SIGMOD Conference*, pages 803–814, 2004.

[23] S. Chaudhuri and G. Weikum. Foundations of automated database tuning. In *ICDE*, page 104, 2006.

[24] B. Dageville, D. Das, K. Dias, K. Yagoub, M. ZaÃŕt, and M. Ziauddin. Automatic sql tuning in oracle 10g. In *VLDB*, pages 1098–1109, 2004.

[25] B. Dageville and M. Zaït. Sql memory management in oracle9i. In *VLDB*, pages 962–973, 2002.

[26] A. El-Helw, I. F. Ilyas, W. Lau, V. Markl, and C. Zuzarte. Collecting and maintaining just-in-time statistics. In *ICDE*, pages 516–525, 2007.

[27] S. J. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Trans. Database Syst.*, 13(1):91–128, 1988.

[28] M. R. Frank, E. Omiecinski, and S. B. Navathe. Adaptive and automated index selection in rdbms. In *EDBT*, pages 277–292, 1992.

[29] J. Gardner. PostgreSQL/Materialized Views. `http://tech.jonathangardner.net/wiki/PostgreSQL/Materialized_Views`.

[30] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. In *ICDT*, pages 453–470, 1999.

[31] M. Hammer and A. Chan. Index selection in a self-adaptive data base management system. In *SIGMOD Conference*, pages 1–8, 1976.

[32] Y. Kotidis and N. Roussopoulos. Dynamat: A dynamic view management system for data warehouses. In *SIGMOD*, pages 371–382, 1999.

[33] C. Mishra and N. Koudas. A lightweight online framework for query progress indicators. In *ICDE*, pages 1292–1296, 2007.

[34] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *SIGMOD*, pages 307–318, 2001.

[35] S. Papadomanolakis and A. Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In *SSDBM*, pages 383–392, 2004.

[36] S. Rozen and D. Shasha. A framework for automating physical database design. In *VLDB*, pages 401–411, 1991.

[37] P. Scheuermann, W.-S. Li, and C. Clifton. Watchman : A data warehouse intelligent cache. In *VLDB*, pages 51–62, 1996.

[38] M. Stonebraker. The choice of partial inversions and combined indices. *International Journal of Parallel Programming*, 3(2):167–188, 1974.

[39] D. Theodoratos and T. Sellis. Data warehouse configuration. In *VLDB*, pages 126–135, 1997.

[40] G. Valentin, M. Zuliani, D. C. Zilio, G. M. Lohman, and A. Skelley. Db2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, pages 101–110, 2000.

[41] G. Weikum, A. Mönkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *VLDB*, pages 20–31, 2002.

[42] J. Zhou, P.-A. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *VLDB*, pages 231–242, 2007.

[43] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. J. Storm, C. Garcia-Arellano, and S. Fadden. Db2 design advisor: Integrated automatic physical database design. In *VLDB*, pages 1087–1097, 2004.

[44] D. C. Zilio, C. Zuzarte, S. Lightstone, W. Ma, G. M. Lohman, R. Cochrane, H. Pirahesh, L. S. Colby, J. Gryz, E. Alton, D. Liang, and G. Valentin. Recommending materialized views and indexes with ibm db2 design advisor. In *ICAC*, pages 180–188, 2004.