Designing and Using Views To Improve Performance of Aggregate Queries (September 9, 2004)

Foto Afrati[1], Rada Chirkova[2], Shalu Gupta[2], and Charles Loftis[2]

[1] Computer Science Division, National Technical University of Athens,
157 73 Athens, Greece
afrati@cs.ece.ntua.gr
[2] Computer Science Department, North Carolina State University,
Raleigh, NC 27695, USA
{chirkova,sgupta5,celoftis}@csc.ncsu.edu

Abstract. Data-intensive systems routinely use derived data, such as indexes or materialized views, to improve query-evaluation performance. In this context, the problem of *designing derived data* is as follows: Given a set of queries and a database, return definitions of derived data that, when materialized in the database, would reduce the evaluation costs of the queries. Designing materialized views and indexes is an important part of *automated query-performance tuning* in data-management systems that experience changes over time, where a system addresses the performance requirements of *current* frequent and important queries by periodically reconsidering and rematerializing the stored derived data.

In this paper we present an extensible system architecture for Query-Performance Enhancement by Tuning (QPET). QPET combines design and use of derived data in an end-to-end approach to automated query-performance tuning, and selects appropriate data-design algorithms depending on the characteristics of the prevalent queries. Our focus in automated query-performance tuning is on a tradeoff between the amount of system resources spent on designing derived data and the degree of the resulting improvement in query performance. We present algorithms and experimental results in designing and using materialized views for practically important classes of aggregate queries, including range-aggregate queries on star-schema data warehouses.

Keywords: Data warehouses, query language and query optimization, query processing, materialized views.

1 Introduction

Derived data, such as materialized views or indexes, are routinely used in data-intensive systems to improve query-evaluation performance. In this context, the problem of *designing derived data* is as follows: Given a set of queries, a database, and a set of constraints on derived data (e.g., a storage limit), return definitions of derived data that, when materialized in the database, would satisfy the constraints and reduce the evaluation costs of the queries. Automated design of materialized views and indexes to answer queries is an important component of *automated query-performance tuning* in data-management systems that change over time, where a system addresses the performance requirements of *current* frequent and important queries by periodically reconsidering and rematerializing the stored derived data. For this reason, developing techniques for designing materialized views and indexes to improve query-answering performance is a recognized research direction within the discipline of self-administering database systems [1–3]. In this paper we describe our approach within this research direction and discuss its implementation and validation in our extensible system architecture for Query-Performance Enhancement by Tuning (QPET) [4].

Generally, spending more time on designing materialized views or indexes for a given query workload tends to pay off, as greater improvement can thereby be achieved in the performance of evaluating the workload queries using the resulting stored derived data. Because the number of potentially beneficial views or indexes tends to be prohibitive even for very simple query workloads [5–7], in many cases it is not practical to obtain sets of derived data that would *globally minimize* the evaluation costs of the input queries. Several approaches, including those described in [6–9], have been proposed in the past to design good-quality sets of derived data for evaluating SQL queries, without spending an inordinate amount of time on the design. Unfortunately, even with their runtime advantages, it is not always possible to use these and other known algorithms in automated query-performance tuning in systems that change over

time. One reason is, in many practical scenarios the amount of time or other system resources available for designing derived data is limited. The problem is even more pronounced when, under a requirement to provide continuous services to customers, a system cannot go down for redesign of the stored materialized views or indexes.

We study design and use of materialized views and indexes in automated query-performance tuning in relational data-management systems where the set of prevalent queries changes over time. Our objective is to minimize the evaluation costs of a given query workload subject to given restrictions on design time, by designing and using derived data. To address the tradeoff between the time allotted for designing derived data and the quality of the resulting views or indexes, our approach is to develop specialized algorithms for specific practically important query types. For each type of queries, when designing derived data we use information about how the resulting materialized views or indexes will be used to evaluate the workload queries, that is, at the design stage we use information about rewriting the queries using the outputs of the design stage. In this paper we present techniques for designing and using materialized views; the techniques are applicable to a practically important class of range-aggregate queries on star-schema data warehouses.

In the experimental results reported in this paper we use a greedy algorithm BPUS introduced in [7]. We say "workload of star-schema queries" to refer here to a set of aggregate queries where all queries have the same `FROM` clause and the same join conditions in the `WHERE` clause.[3] To design aggregate views for a workload of star-schema queries, BPUS explores iteratively views in a *view lattice*, which is a representation of the search space of views for the workload, with directed edges between views denoting which view can be evaluated using another view. At each step, BPUS selects from the lattice a view with the greatest *benefit*, that is, a view that reduces the most the average cost of answering a query, per unit space. For any materialized view designed by the BPUS algorithm of [7], if the view is usable in evaluating some query, then the answer to the view is *the only* relation needed in the evaluation. That is, views produced by the algorithm of [7] determine joinless rewritings of queries. In addition to joinless rewritings, in our approach we use rewritings that are computed via joins of aggregate views with other relations, namely *central rewritings* introduced in [10].

We now show two examples that provide an intuition for our results. The first example demonstrates that our approach to rewriting star-schema queries can result in better *design* time than BPUS [7], and that the types of possible rewritings of the workload queries using views determine the definitions (specifically the grouping arguments) of the aggregate views we consider to materialize.

Example 1. Consider a data warehouse with stored relations `Sales`, `Customer`, and `Time`:

```
Sales(CustID,DateID,ProductID,SalespersonID,QuantitySold,TotalAmount,Discount)
Customer(CustID,CustName,Address,City,State,RegistrDateID)
Time(DateID,Month,Year)
```

Keys of the tables are underlined. `Sales` is the fact table, and `Customer` and `Time` are dimension tables.

Let the query workload of interest have two star-schema queries, `Q1` and `Q2`. Query `Q1` asks for the total quantity of products sold per customer in the second quarter of the year 2004. `Q2` asks for the total product quantity sold per year for all years after 1997 to customers in North Carolina.

```
Q1: SELECT c.CustID, SUM(QuantitySold)
    FROM Sales s, Time t, Customer c
    WHERE s.DateID = t.DateID AND s.CustID = c.CustID
    AND Year = 2004 AND Month >= 4 AND Month <= 6
    GROUP BY c.CustID;
```

```
Q2: SELECT t.Year, SUM(QuantitySold)
    FROM Sales s, Time t, Customer c
    WHERE s.DateID = t.DateID AND s.CustID = c.CustID
    AND Year > 1997 AND State = 'NC'
    GROUP BY t.Year;
```

By running BPUS [7] on { `Q1, Q2` } we would obtain materialized views that could be used to evaluate `Q1` and `Q2` using just selection, grouping, and aggregation. In finding grouping arguments of the views, BPUS would consider all subsets of a set of four attributes { `CustID, Year, Month, State` }.

We now show two central rewritings [10], `R1` and `R2`, of the query `Q1`, which use joins of aggregate views with other relations, and discuss the tradeoffs in using each rewriting to evaluate `Q1`. Consider an aggregate view `V1`, which returns the total quantity of products sold to each customer based on just the `Sales` data:

[3] Formally, additional requirements need to be satisfied by star-schema queries; see Section 2.

```
V1: SELECT CustID,DateID,SUM(QuantitySold) AS SumQS FROM Sales GROUP BY CustID,DateID;
```

A rewriting R1 of the query Q1 uses a join of the view V1 with relations Customer and Time:

```
R1: SELECT c.CustID, sum(SumQS) FROM V1, Time t, Customer c
    WHERE V1.DateID = t.DateID AND V1.CustID = c.CustID AND Year = 2004 AND Month >= 4 AND Month <= 6
    GROUP BY c.CustID;
```

Another equivalent rewriting of Q1, R2, uses a view V2, which has *two* relations in its FROM clause:

```
V2: SELECT c.CustID AS CID,
    DateID AS DID, SUM(QuantitySold) AS SumQS
    FROM Sales s, Customer c
    WHERE s.CustID = c.CustID
    GROUP BY c.CustID, DateID;
```

```
R2: SELECT CID, sum(SumQS)
    FROM V2, Time t
    WHERE V2.DID = t.DateID AND Year = 2004
    AND Month >= 4 AND Month <= 6
    GROUP BY CID;
```

Assuming that there are typically many product IDs and product categories per sale event, and that different salespeople are responsible for selling products in different categories, the relation for each of V1 and V2 will likely be much smaller than the fact table Sales. Accordingly, the evaluation time of each of R1 and R2 will likely be lower than that of Q1 using its original definition.[4] In addition, assuming the stored data satisfy integrity constraints that are typical for star-schema databases,[5] the evaluation time of the rewriting R2 will be strictly less on typical databases than the evaluation time of R1, as the relations for V1 and V2 in this case are of the same size and as evaluating R1 involves an extra join compared to R2.

We now compare the set { V1 } with sets of views produced by BPUS [7]. (All the observations we make here about the view V1 also hold about V2.) First, V1 *by itself* provides equivalent central rewritings of both Q1 and Q2 and thus gives a *solution* { V1 } for the workload { Q1, Q2 }. Second, recall that BPUS would consider all subsets of a set of four grouping arguments. In contrast, to design the set { V1 } we had to look at just *two* grouping arguments, CustID and DateID. The reason is, we consider only aggregate views whose FROM clause has just the Sales relation[6], and CustID and DateID are the only grouping arguments of such views that are required in constructing equivalent rewritings of the workload queries.

We now show that compared to other approaches, one novelty of our approach is that it can be used to design materialized views for aggregate queries of a more general type than star-schema queries:

Example 2. Consider the database schema and query Q1 of Example 1, and let query Q3 ask for the total quantity of products sold per customer, for customers who got *registered* in the second quarter of 2004:

```
Q3: SELECT c.CustID, SUM(QuantitySold) FROM Sales s, Time t, Customer c
    WHERE s.CustID = c.CustID AND c.RegistrDateID = t.DateID AND Year = 2004 AND Month >= 4 AND Month <= 6
    GROUP BY c.CustID;
```

As the join condition c.RegistrDateID = t.DateID of Q3 differs from the condition s.DateID = t.DateId of Q1, { Q1, Q3 } is not a workload of star-schema queries. Thus, BPUS is not applicable. At the same time, view V1 from Example 1 can be used to evaluate both Q1 and Q3; we give here an equivalent rewriting of Q3 using V1.

```
R3: SELECT c.CustID, sum(SumQS) FROM V1, Time t, Customer c
    WHERE c.RegistrDateID = t.DateID AND V1.CustID = c.CustID AND Year = 2004 AND Month >= 4 AND Month <= 6
    GROUP BY c.CustID;
```

Similarly, if we add RegistrDateID to the list of grouping arguments of view V2 in Example 1, the resulting view could be used to evaluate each of Q1 and Q3.

Contributions The paper's contributions are as follows. (1) We propose a system architecture for automated query-performance tuning in data-management systems over time, by periodically designing materialized views and indexes that reduce the evaluation costs of current frequent and important queries. (2) We present a theoretical study of the problem of designing materialized views subject to input restrictions on design time. (3) We present a parameterized algorithm for designing and using materialized views subject

[4] Similarly, evaluation costs of the other workload query, Q2, can be reduced by using rewritings with V1 or V2.
[5] These are a referential-integrity constraint on CustID from Sales to Customer and key constraints on the stored relations.
[6] Or just the relations Sales and Customer in case of { V2 }.

to input restrictions on design time. Our algorithm uses algorithms such as BPUS [7] as a subroutine but is applicable to a more general class of aggregate queries than just queries on star-schema data warehouses. (Our approach can be extended in a straightforward manner to designing materialized views *and indexes*, by using, instead of BPUS, its extension described in [8]. In addition, the approach can be used to design derived data that satisfy varying maintenance-cost requirements.) (4) We validate the approach using experimental results in our QPET implementation of the proposed system architecture.

Related Work

Designing and using derived data to improve the evaluation performance of complex queries has long been a direction of research and practical efforts in data-intensive systems. Over time, a wealth of theoretical results (see [11] for a survey) and some practical solutions [12–14] have been accumulated on using materialized views and indexes in query answering. The problem of answering aggregate queries using views has been considered in relation to data warehouses and data cubes [15–18]; results on answering each query using a single view are presented in [19, 20]. Recent work [10, 21] has considered the problem of rewriting aggregate queries using multiple views; each rewriting format can be used in the results we present in this paper.

Considerable work has been done on efficiently selecting views and indexes for general SQL queries [6, 22] and in particular for aggregate queries (e.g., [7–10]). [6, 14] have introduced an end-to-end approach and a system architecture for designing and using materialized views and indexes to answer queries. That end-to-end architecture involves optimizer-based choice of best sets of views and indexes for given queries, thereby increasing the likelihood of using the resulting materialized derived data by the optimizer to answer the queries. In our framework we extend the architecture of [6]; to the best of our knowledge, we are the first to specifically address architectural issues in periodic redesign of derived data. In addition, we demonstrate that to design derived data under a design-time constraint, one has to consider the design and use (i.e., rewriting) problems together.

2 Preliminaries and Problem Specification

We consider select-project-join queries with equality-based joins and with aggregation *sum*, *count*, *max*, or *min*. Our approach is applicable to queries with inequality comparisons with constants, including the important class of range-aggregate queries. We study workloads of *parameterized* queries: For any query with constants, the parameterized version of the query has placeholders instead of the constants. (As an illustration, the parameterized version of query `Q1` in Example 1 can be obtained from `Q1` by replacing its three constants, `2004, 4,` and `6,` by placeholders `$year, $month1,` and `$month2` respectively.)

To measure query-evaluation performance, our cost model is as follows. We assume that the size of a database relation is the number of bytes in it, and that the cost of computing a join of two relations is the sum of the sizes of the input relations and of the output relation; this faithfully models the cost of, for instance, hash joins.[7] The results in this paper can be extended in a straightforward way to the model where the cost of a join is proportional to $N \log N$ for relations of size N, as in sort-merge joins.

For select-project-join queries, we measure the cost of evaluating the query on a database as the sum of the costs of all the (binary) joins in the evaluation [5]. (We assume that all selections are pushed down as far as they go, projection is the last operator in the query plan, and only left-linear-join query plans are used.) To estimate the cost of evaluating a parameterized select-project-join query with inequality comparisons, we use the cost of evaluating an instance of the query with a "typical number" of values in the range. For an aggregate query, we say that the `FROM` and `WHERE` clauses of the query define its select-project-join *core relation*. Let N be the number of tuples in the core relation of a query `Q`; then the cost of evaluating `Q` is the sum of (1) the cost of computing the core relation of `Q`, and of (2) the cost $N \log N$ of applying the grouping and aggregation operators to the core relation.[8] Finally, the *total cost* of evaluating a finite query workload $\mathcal{Q}$ on a database $\mathcal{D}$ is the sum of the costs of evaluating all individual queries in the workload on $\mathcal{D}$. The sum can be weighted to reflect the relative frequency or importance of individual workload queries.

[7] We use a variation of this join-cost model for the case of index joins under a referential-integrity constraint, where the cost of a join is proportional to the sum of the sizes of the output relation and of the *un*indexed input relation.

[8] The cost of grouping the tuples in a relation is the same as the cost of sorting the relation, and one can compute *sum*, *count*, *max* or *min* aggregation on a sorted relation in a single pass.

We consider the problem of improving the evaluation performance of workloads of SQL queries on relational databases, by using precomputed stored derived data such as materialized views or indexes. Our *problem inputs* are of the form $\mathcal{I} = (\mathcal{D}, \mathcal{Q}, \mathcal{C}_1, \mathcal{C}_2)$, where $\mathcal{D}$ is a database and $\mathcal{Q}$ is a workload of parameterized queries. $\mathcal{C}_1$ is a constraint on the time available for the design stage; $\mathcal{C}_2$ is a constraint on the derived data to be materialized, such as a storage limit or an upper bound on the maintenance costs for the stored data. Our goal is to define, within time $\mathcal{C}_1$, a set $\mathcal{V}$ of derived data, such that $\mathcal{V}$ satisfies $\mathcal{C}_2$ and reduces as much as possible the total cost of evaluating $\mathcal{Q}$ on the database $\mathcal{D}$. (We consider only *equivalent* rewritings of the queries in $\mathcal{Q}$ using the set $\mathcal{V}$, i.e., we require that exact answers to all the queries in $\mathcal{Q}$ can be computed using $\mathcal{V}$.) For data-management systems that periodically redesign their derived stored data, we explore the tradeoff between the value of $\mathcal{C}_1$ (i.e., the maximal runtime of algorithms that design derived data) and the costs of evaluating the prevalent queries using the resulting stored derived data.

In this paper, we focus on designing and using materialized views, rather than indexes. For any parameterized query in the given query workload, our goal is to design views that can be used in evaluating *any instance* of the query. Thus, similarly to [7, 8] we consider only views without comparisons with constants. We use the following definitions of admissible and optimal viewsets, or solutions:

Definition 1. *(Admissible viewset) Let $\mathcal{I} = (\mathcal{D}, \mathcal{Q}, \mathcal{C}_1, \mathcal{C}_2)$ be a problem input. A set of views $\mathcal{V}$ is said to be an* admissible viewset *for the problem input $\mathcal{I}$, if the following four conditions hold:*
(1) $\mathcal{V}$ gives equivalent rewritings of all the queries in $\mathcal{Q}$,
(2) for every view $V \in \mathcal{V}$, there exists an equivalent rewriting of a query in $\mathcal{Q}$ that uses V,
(3) the time required to generate $\mathcal{V}$ does not exceed $\mathcal{C}_1$, and
(4) $\mathcal{V}$ satisfies the constraint $\mathcal{C}_2$.

Definition 2. *(Optimal viewset) For a problem input $\mathcal{I} = (\mathcal{D}, \mathcal{Q}, \mathcal{C}_1, \mathcal{C}_2)$, an* optimal viewset *is a set of views $\mathcal{V}$ defined on $\mathcal{D}$, such that:*
(1) $\mathcal{V}$ is an admissible viewset *for $\mathcal{I}$, and*
(2) $\mathcal{V}$ minimizes the total cost of evaluating the queries in $\mathcal{Q}$ on the database $\mathcal{D}_\mathcal{V}$, among all admissible sets of views for $\mathcal{I}$; here, $\mathcal{D}_\mathcal{V}$ is the database that results from adding, to the stored data in $\mathcal{D}$, the relations for all the views in $\mathcal{V}$ computed on $\mathcal{D}$.

In our QPET framework, the focus is on using specialized algorithms for designing and using derived data to reduce the evaluation costs for each specific class of queries. In this paper we discuss algorithms that are applicable to three classes of queries with aggregation and without self-joins[9]:

1. *Workload of queries that aggregate the same table.* All queries in a query workload aggregate the same table, `T`, if the aggregated arguments of each workload query come from `T`. In this case, we refer to the table that provides the arguments of aggregation for all the workload queries as the *central table* for the workload.
2. *Workload of queries that have the same join conditions.* A workload of queries that have the same join conditions is a workload of queries that aggregate the same table, such that an additional condition is satisfied: For any pair of queries in the workload, if the queries share the same tables in the `FROM` clause then they share, in the `WHERE` clause, all the join conditions between the tables. As an illustration, the same-join condition is satisfied in workloads of queries where all joins are natural joins. (Note that, similarly to workloads of queries that aggregate the same table, workload queries in this case are not required to all have the same `FROM` clause.)
3. *Workload of star-schema queries.* A workload of star-schema queries is a workload of queries that have the same join conditions, such that two additional conditions are satisfied. First, the database schema is a star schema [23], with a designated fact table and dimension tables, such that each join of the fact table with a dimension table is on attributes that satisfy a referential-integrity constraint from the fact table to the dimension table. Second, the fact table is the central table for the workload (i.e., each query aggregates an argument of the fact table).

[9] A query has a self-join if some stored relation is mentioned in the query's `FROM` clause at least twice.

3 The QPET Framework for Designing and Using Derived Data

In this section we describe our system architecture, QPET. We use the architecture of Figure 1 to implement and validate our work in designing and using derived data to improve the evaluation performance of frequent and important queries. Our concentration and contributions in the system architecture are threefold. First, we use *specialized* algorithms for defining and using views and indexes for specific practically important classes of queries, such as the star-schema queries we discuss in this paper. Thus, our framework for designing and using derived data is *extensible*. We argue that specialized algorithms are required to ensure a guaranteed degree of improvement in query-evaluation performance, with respect to the *best possible* performance for the queries. In addition, different specialized algorithms are needed under different constraints on derived data materialized in the system, such as a storage limit on materialized views.

Second, as discussed in Section 1, we look in particular into developing a system architecture for periodic *online* (re)design of materialized views and other derived data in data-management systems. In that context, it is imperative that algorithms for designing and using derived data be lightweight, efficient, and scalable. Third, we argue the need to consider the interaction and interdependence of techniques for *generating* materialized views and other derived data with techniques for *rewriting* the given queries using the data that end up being materialized. For instance, while simple rewriting techniques can be used for star-schema query workloads, see Section 4, they would not be sufficient for workloads of more general queries with aggregation [10].

Further, as we discuss in Section 4, different requirements (e.g., on query-evaluation time or on the runtime of view-design algorithms) could influence the choice of different `FROM` clauses of — and thus of other requirements on — aggregate views that could be materialized for star-schema queries. As an illustration, Example 1 in Section 1 shows two views, `V1` and `V2`, whose different "formats" stem from the different `FROM` clauses, { `Sales` } for `V1` and { `Sales, Customer` } for `V2`. The preference for the `FROM` clause of the view `V1` could result from, for instance, the requirement of better view-maintenance costs, whereas the choice of the `FROM` clause of the view `V2` could arise due to a stronger restriction on the evaluation costs of the resulting rewriting `R2`. Different choices of `FROM` clauses determine different requirements on the view definitions — such as the presence of certain grouping arguments in some but not all view definitions — at the *view-design stage*, to ensure that it is possible to use the designed views to construct an equivalent rewriting of the input queries, such as `Q1` and `Q2` in Example 1, at the *query-rewriting* stage in QPET.

For these reasons, the third contribution of our approach is a component of the system architecture that determines the "format" of views that should be materialized, based on the rewriting types considered for the given queries. This component of the QPET framework is used at the stage of designing derived data, to determine the search space of views or other derived data considered for materialization, see Section 3.1.

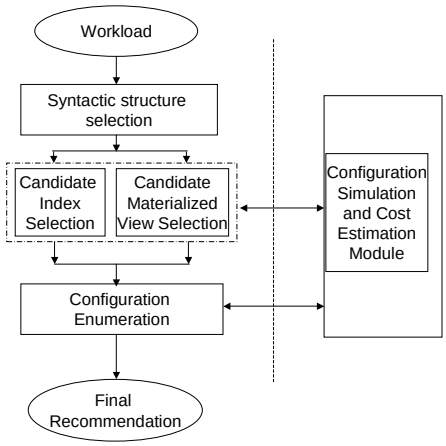

Fig. 1. Overall system architecture [6, 14].

In the remainder of the section we describe in more detail our approach to designing and using derived data to improve the performance of given workload queries. While this discussion centers on materialized views, we use the same QPET framework (within the general architecture of Figure 1) and the same overall approach in our ongoing and future work on designing and using other types of derived data, especially indexes, for improving query-evaluation performance.

3.1 Designing Derived Data in the QPET Framework

In Figure 2, shaded boxes with solid-line borders represent those modules in the QPET framework that are active at the stage of *designing* materialized views to improve the evaluation performance of frequent and important queries. We assume that we are given the definitions of workload queries. (The query workload

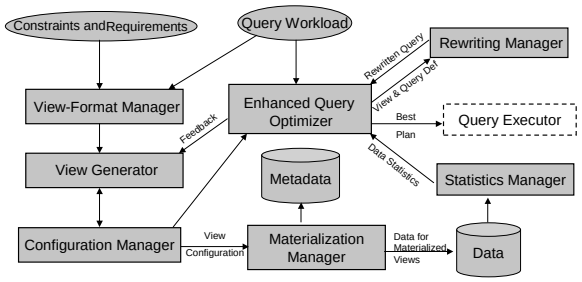

Fig. 2. Designing derived data in QPET.

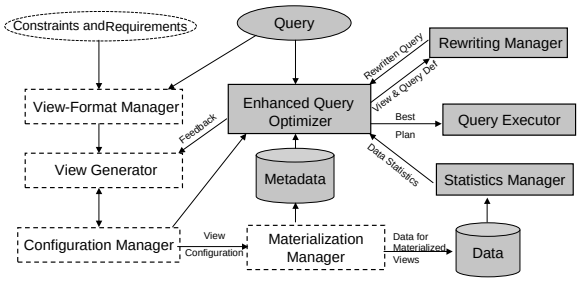

Fig. 3. Using derived data in QPET.

can be assembled using, e.g., the query log [6].) We also assume that QPET has access to the relevant constraints on the outputs of the design stage, such as the amount of disk space available for storing the materialized views. Other things that can potentially be known in advance include requirements on possible rewritings of the workload queries using the views, such as, for instance, runtime ranges for the rewritings. Finally, for *online* redesign of materialized views, constraints on the amount of resources available for the design stage can be specified. (One direction of our current work is developing algorithms for designing derived data under hard resource constraints, with an emphasis on how the quality of the outputs depends on the values of the constraints.)

Using our QPET framework, data-management systems could design and materialize derived data by using specialized plug-and-play algorithms for each class of workload queries, such as star-schema queries. For this reason, each module active at the data-design stage uses algorithms that are appropriate for the given class of workload queries. In Section 4, we describe specific algorithms we use for workloads of aggregate queries under certain restrictions, in particular for queries on the star schema.[10]

In the design stage, the first thing to do is to delineate the search space of views that will be considered for materialization. This function is performed by the *view-format manager*, which determines the suggested format of views and rewritings and the related search space of candidate views based on the type of the workload queries and on other requirements. The *view generator* considers one by one the elements of the resulting search space of candidate views and passes those of the views that satisfy the input constraints (e.g., the given storage limit) on to the next module. The *configuration manager* puts together some of the candidate views and tests the potential of the resulting view configurations to improve the evaluation performance of the workload queries. The tests are performed by an *enhanced query optimizer*, which estimates the costs of answering the workload queries using the configurations. (We use a *statistics manager* and the architecture of [6] to obtain query-cost estimates without materializing the views in the candidate configurations.) When obtaining query-cost estimates, the enhanced optimizer considers potential rewritings of the workload queries by calling a *rewriting manager*, in the manner of [12].[11] After a quality threshold is reached, typically after several iterations of designing and testing view configurations, the *materialization manager* materializes the views in the best configuration as new stored data.

3.2 Using Derived Data in the QPET Framework

In Figure 3, shaded boxes with solid-line borders represent those modules in the QPET framework that are active at the stage of *using* derived data to answer customer queries. Suppose the system chooses to minimize the evaluation costs of a given query using the currently stored derived data. In this case, QPET first finds a "good" query plan while taking the materialized views into account, by using the enhanced query optimizer and rewriting manager in essentially the same way the modules are used in the view-design phase, see Section 3.1. (Depending on how appropriate the currently materialized views are for answering the query, the resulting query plan may or may not refer to stored view relations.) The resulting plan is

[10] If the given query workload has queries that correspond to several distinct query classes such that a general solution would be too generic, one strategy to improve the quality of the solution could be to obtain separate specialized solutions for the sub-workloads according to the query classes, and then to try to merge some of the views, as proposed in, e.g., [6].

[11] The paper [12] describes an algorithm for select-project-join queries and views without aggregation. Using the results in [10], we have designed a straightforward extension applicable to aggregate queries and views.

passed to the *query executor*, which obtains an answer to the query in a standard way using the stored base or view relations.

Other than considering the stored derived data in generating query plans, using materialized views in QPET to answer a given query is not different from standard query processing. What can, however, make a difference in periodic online redesign of materialized views in our framework is which user queries are answered by taking the stored derived data into account. One approach could be to evaluate all user queries using the materialized views; the drawback here is the runtime penalty incurred by the optimizer for all input queries. As the currently materialized views were designed to improve the evaluation costs of some query workload that was fixed at the design stage, at the other extreme we could take the views into account only when evaluating those user queries that are part of that workload.[12] Between the two extremes, multiple strategies are possible; those that we currently focus on in QPET make the decision for each query based on whether it could help characterize changes in the prevalent query workload over time. (For instance, QPET could use materialized views to evaluate frequent queries that are not in the fixed query workload. This way, the system can accumulate, for the future design stage, useful statistics on how the currently materialized views are used in answering important non-workload queries.) Other strategies are possible, based on which changes over time are tracked in the given data-management system; possible objects of interest include prevalent queries and stored data.

4 Complexity of the Problem and Parameterized Algorithm

We now discuss an approach we use within our QPET framework to improve the efficiency of evaluating aggregate queries without self-joins, by computing the queries using specially designed aggregate views. Recall that the BPUS approach of [7] evaluates aggregate star-schema queries using rewritings that have a materialized aggregate view as the *only* relation in the `FROM` clause. Using the results in [10], we generalize the approach of [7] into an approach that uses one *or more* views in each rewriting and that applies to a *more general class* of aggregate queries — queries that aggregate the same table, see Section 2 for a definition. In this framework, when looking for views that are potentially usable in computing given queries, we can examine all the views considered in [7], as well as additional views (with or without aggregation) that are defined on *subsets* of the set of relations in the `FROM` clause of the query. For instance, Examples 1 and 2 in Section 1 show an aggregate view `V1` that is defined on just the fact table `Sales` and that can be used, in joins with other relations, to evaluate aggregate queries `Q1` through `Q3`.

The idea of our approach is to extract, from a workload of queries that aggregate the same table, view *templates*, which serve as input queries to the view-selection algorithm that we use as a subroutine. (In our current implementation of QPET, this subroutine is the BPUS algorithm of [7].) The views returned by the subroutine are then materialized and used to automatically construct rewritings of the workload queries; the rewritings have one or more relations in their `FROM` clause and may or may not be aggregate queries. (That is, some of the rewritings of the given aggregate queries can be conjunctive queries *without* aggregation, see [10] for details and examples.) This approach can be tuned to explore different subspaces of the search space of views, depending on the input constraints such as the amount of system resources available for designing derived data.

4.1 Dealing with Inequality Comparisons

We first show how to obtain materialized views and rewritings for aggregate queries with inequality comparisons with constants, including range-aggregate queries, without having to deal with the inequalities. (As shown in [24], in general the presence of inequalities increases the complexity of the problem of rewriting queries using views.) Recall that we consider only views without selection conditions, to maximize rewriting benefits for parameterized queries. Intuitively, we reduce the problem of designing and using materialized views for aggregate queries with inequalities to the problem for queries *without* inequalities, by allowing rewritings with inequalities. Theorem 1 says that the procedure results in correct processing of aggregate queries with inequality comparisons. More precisely, any set of views that is a solution for the modified queries without inequalities is also a solution for the original queries with inequalities.

[12] As we consider workloads of parameterized queries, the number of specific user queries that are "covered" by a given finite workload is potentially infinite.

Theorem 1. *On a database $\mathcal{D}$, let $\mathcal{Q}$ be a workload of select-project-join queries without self-joins and with aggregation sum, max, min, and count, such that at least one query in $\mathcal{Q}$ has inequality comparisons with constants. Let $\mathcal{C}$ be either a storage limit or a view-maintenance-cost constraint. By $\mathcal{Q}'$ we denote a workload of queries obtained by replacing inequality comparisons in $\mathcal{Q}$ with equalities on the same arguments. Consider a set $\mathcal{V}$ of select-project-join views with aggregation and without constants. If $\mathcal{V}$ is an admissible solution for a problem input $(\mathcal{D}, \mathcal{Q}', \mathcal{C})$, then $\mathcal{V}$ is also an admissible solution for a problem input $(\mathcal{D}, \mathcal{Q}, \mathcal{C})$, assuming select-project-join rewritings with inequalities and with or without aggregation.*[13]

4.2 View Selection: Sources of Exponentiality

We now observe that, as in the case of queries and views without aggregation [25], the view-selection problem for queries with aggregation has several sources of exponentiality, even under our restriction that materialized aggregate views do not have inequality comparisons or selection conditions. In our setting, the problem has four sources of exponentiality. First, even if we restrict ourselves to evaluating a single query using aggregate views, potentially useful views could be defined using subsets *or supersets* of the relations in the query's `FROM` clause. (Workloads of select-project-join queries with aggregation and self-joins might have useful aggregate views defined using exponentially more relations than used in any workload query [5, 10].) Second, useful views could be defined using various subsets of the join conditions in the query's `WHERE` clause. Third, the set of grouping arguments in a useful view could be a superset of the query's grouping arguments [7], or could even skip at least one grouping argument of the query [10]. The latter case holds for rewritings whose `FROM` clause has more than one relation, such as rewriting `R1` in Example 1. (There exist central rewritings [10] in which the sets of grouping arguments of the aggregate view and of the query can even be disjoint.)

Finally, different queries in a query workload could use different aggregate functions (e.g., *max* and *sum*) and could aggregate different arguments in base tables, as can be seen in, for instance, queries in the TPC-H benchmark [26]. In our approach we eliminate this source of exponentiality in view selection by having *all* views under consideration have all the aggregated arguments (i.e., all the combinations of the attribute to be aggregated with the aggregate function) that occur in the input query workload.

4.3 Complexity Results

We now show that for the practically important class of queries that aggregate the same table, two more of the four sources of exponentiality can be removed without removing *any* potentially useful aggregate views. The last result, Theorem 4, shows how removing yet another source of exponentiality that arises from varying subsets of relations in the `FROM` clauses of views, reduces the search space of views even further while preserving useful views. As a result, in our approach we need to deal with only one source of exponentiality, which arises from varying grouping arguments in the definitions of aggregate views. (All the results in this section hold for workloads of star-schema queries.)

We first establish that in evaluating workloads of aggregate queries without self-joins using materialized views, each optimal aggregate view can be defined using a low number of relations in the `FROM` clause. More precisely, the number of base relations in each view's `FROM` clause is at most the number of relations used to define some workload query. Thus, for aggregate queries without self-joins, Theorem 2 removes the source of exponentiality in view selection that comes from varying, in view definitions, up to exponentially-sized combinations of relations in the `FROM` clauses of the input queries. As a result (Corollary 1), the problem has lower complexity for this class of queries, NP-complete rather than exponential-time lower bound.

Theorem 2. *(Restricted definitions of useful central views) On a database $\mathcal{D}$, let $\mathcal{Q}$ be a workload of select-project-join queries without self-joins and with aggregation sum, max, min, and count. Let the aggregated arguments of all the queries come from a single base table in $\mathcal{D}$. Let $\mathcal{V}$ be an optimal set of central aggregate views for a problem input $(\mathcal{D}, \mathcal{Q}, \mathcal{C})$, where $\mathcal{C}$ is a storage limit or a view-maintenance-cost constraint. Then each view in $\mathcal{V}$ can be defined using a subset of the* `FROM` *clause of some query $Q \in \mathcal{Q}$.*

Corollary 1. *For problem inputs as Theorem 2, the view-selection problem is NP-complete.*

[13] This and all other results in this section hold provided some admissible set of materialized views exists for the given problem input. Note that when, for instance, the input storage limit is too low, no solution may exist.

The next result, Theorem 3, holds for query workloads that additionally have the same join conditions (see Section 2). Intuitively, under this restriction we can generate all useful central aggregate views by combining all meaningful subsets of the `FROM` clause of the input queries (each such subset determines all join conditions in the `WHERE` clause) with all possible sets of useful grouping arguments. That is, the same-join condition removes the source of exponentiality that comes from varying join conditions in the `WHERE` clauses of the views.

Theorem 3. *(Restricted search space of central views) For problem inputs as in Theorem 2, and if the queries in the workload $\mathcal{Q}$ additionally have the same join conditions, then the search space of optimal central aggregate views is at most doubly exponential in the size of the input query workload $\mathcal{Q}$.*

Thus, for workloads of aggregate queries that have the same join conditions we need to deal with just two sources of exponentiality in view definitions instead of the four that are discussed in Section 4.2.

So far, we have shown that for a practically important class of aggregate queries, some of the sources of exponentiality (Section 4.2) can be removed without removing *any* potentially useful views from consideration. Our last result in this section says that focusing on a certain important subclass of views (views that all share the same `FROM` clause) removes an additional exponent from the complexity of the view-selection problem.[14]

Theorem 4. *For problem inputs as in Theorem 3, and if in addition all views under consideration have the same* `FROM` *clause, then the search space of optimal central aggregate views is at most singly exponential in the size of the input query workload $\mathcal{Q}$.*

We will see in Section 4.4 how this restriction to views that share the same `FROM` clause allows us to use a view-selection method for just star-schema queries (such as BPUS [7]) as a subroutine in our approach to view selection for a *more general* class of queries that aggregate the same table, such as the query workload of Example 2 in Section 1.

4.4 Outline of the Algorithm

We now discuss our parameterized algorithm for selecting and using aggregate views to reduce the evaluation costs of workloads of aggregate queries. Suppose we are given a database $\mathcal{D}$ and a workload $\mathcal{Q}$ of aggregate select-project-join queries that aggregate the same table, for instance the fact table in the star-schema setting. In addition, we are given (1) a constraint $\mathcal{C}_1$ on the runtime of the view-design algorithm, and (2) a constraint $\mathcal{C}_2$ on the materialized views — either a storage limit or a view-maintenance-cost constraint. To reduce the costs of evaluating the workload $\mathcal{Q}$ on $\mathcal{D}$ under the constraint $\mathcal{C}_2$, we select, under the time constraint $\mathcal{C}_1$, views with aggregation (all the views will share the same `FROM` clause) and construct rewritings of the queries using the views as follows:

Algorithm *Composite-Aggregate-Rewritings* (database $\mathcal{D}$, query workload $\mathcal{Q}$, constraints $\mathcal{C}_1$ and $\mathcal{C}_2$).
Output: set $\mathcal{R}$ of rewritings of $\mathcal{Q}$ using aggregate views $\mathcal{V}$.

1 Begin
2 Initialize $\hat{\mathcal{Q}}, \mathcal{R}$ to empty sets;
3 $\mathcal{F}$:= *Nonempty-Overlap*($\mathcal{Q}, \mathcal{C}_1$); `/* Set` $\mathcal{F}$ `to relations that occur in all queries in` $\mathcal{Q}$ `*/` `/* (1) */`
4 `/* Determine input queries` $\hat{\mathcal{Q}}$ `to the` *Lattice-Select-Views* `subroutine: */` `/* (2) */`
5 For each query $Q \in \mathcal{Q}$ do: Begin $\hat{Q}$:= *Input-Lattice-Select*($Q, \mathcal{F}$); $\hat{\mathcal{Q}} := \hat{\mathcal{Q}} \cup \{\hat{Q}\}$; End
6 `/* Collect into a set` $\mathcal{A}$ `the aggregated arguments in all the queries in` $\mathcal{Q}$ `*/` `/* (3) */`
7 $\mathcal{V}$:= *Lattice-Select-Views*($\{\hat{\mathcal{Q}}, \mathcal{F}, \mathcal{D}, \mathcal{C}_2, \mathcal{A}$); `/* Generate central aggregate views` $\mathcal{V}$ `*/` `/* (4) */`
8 `/* Construct rewritings` $\mathcal{R}$ `of the queries in` $\mathcal{Q}$ `using the aggregate views` $\mathcal{V}$`: */` `/* (5) */`
9 For each query $Q \in \mathcal{Q}$ do: Begin R := *Build-Rewriting*(Q, $\mathcal{V}$, $\mathcal{F}$); $\mathcal{R} := \mathcal{R} \cup R$; End
10 Output $\mathcal{R}$;
11 End.

[14] Recall that all the queries we consider take their aggregate arguments from a single table. In the simplest case, all the views in the subclass can be defined on just that central table.

Conceptually, the algorithm finds central aggregate views $\mathcal{V}$ for the queries $\mathcal{Q}$ using a view-selection algorithm for star-schema queries (such as BPUS [7]) as a subroutine *Lattice-Select-Views*, see step `(4)`, and then uses the views $\mathcal{V}$ to build equivalent rewritings $\mathcal{R}$ of the queries $\mathcal{Q}$. In the rewritings $\mathcal{R}$, the views $\mathcal{V}$ "cover" all *or some* subgoals of the queries: In each rewriting, $\mathcal{F}$ are the subgoals "covered" by a view from $\mathcal{V}$. As the size of the set $\mathcal{F}$ determines the runtime of the view-design subroutine of step `(4)`, lower values of the design-time constraint $\mathcal{C}_1$ result in choosing smaller sets $\mathcal{F}$, see step `(1)`. In the overall algorithm (steps `(1)` through `(5)`), the subroutines for generating input queries $\hat{\mathcal{Q}}$ for *Lattice-Select-Views*, step `(2)`, and for constructing the rewritings $\mathcal{R}$ of $\mathcal{Q}$, step `(5)`, ensure the equivalence of the rewritings to the queries, while *Lattice-Select-Views*, step `(4)`, produces views $\mathcal{V}$ that are guaranteed to reduce the evaluation costs *w.r.t. the subgoals* $\mathcal{F}$ in each query on the database $\mathcal{D}$, subject to the constaint $\mathcal{C}_2$.

Theorem 5. *Algorithm Composite-Aggregate-Rewritings is sound for problem inputs as in Theorem 2 whenever all the queries in the workload $\mathcal{Q}$ have the same join condition w.r.t. the set $\mathcal{F}$.*[15] *That is, in this case the algorithm returns equivalent rewritings of the queries in $\mathcal{Q}$ using central aggregate views.*

Different choices of the `FROM` clause $\mathcal{F}$ of the views determine different search spaces of views with aggregation for the input query workload. (For each such choice, note the performance guarantees of the algorithm induced by the use of the *Lattice-Select-Views* subroutine, e.g., BPUS of [7], in case the algorithm is applied to queries on the star schema with the associated integrity constraints.) Varying the value of the overlap set $\mathcal{F}$ results in different degrees of "goodness" of the output of the algorithm *Composite-Aggregate-Rewritings* along several metrics: As the number of base relations in $\mathcal{F}$ increases,

- the evaluation costs of the workload queries $\mathcal{Q}$ using the output rewritings $\mathcal{R}$ tend to decrease;
- the runtime of the *Lattice-Select-Views* subroutine tends to increase;
- the total size (and thus disk-space requirement) for the output materialized views $\mathcal{V}$ tends to increase;
- the total maintenance costs for the output materialized views $\mathcal{V}$ tend to increase.

Intuitively, all these trends are caused by the "degree of coverage" of the evaluation plans for the workload queries by the evaluation plans for the views. (The sizes of evaluation plans for the views are determined by the number of base relations in the definitions of the views $\mathcal{V}$.) As an illustration, view `V1` in Example 1 has just the `Sales` table in its `FROM` clause and thus "covers less" of the query `Q1` than the view `V2`, which is defined using tables `Sales` and `Customer`.

5 Implementation and Preliminary Experimental Evaluation

Our implementation of the QPET framework is written in C and is based on an open-source relational data-management system PostgreSQL [27] version 7.3.4. The current version of QPET is available online [4]; it incorporates support for answering SQL queries using materialized views, as described in [12], and for generating aggregate views using the BPUS algorithm of [7]. Implementing other view-generation algorithms, including that of [9], and implementing automated evaluation of arbitrary select-project-join queries with aggregation using materialized views and indexes [8] is part of our ongoing work.

We have conducted preliminary experiments to evaluate the system architecture and techniques presented in Sections 3 and 4. All experiments were run on a machine with a 2.8GHz Intel P4 processor, 512MB RAM, and an internal 80GB hard drive running Linux RedHat 9 with kernel version 2.4.20-8 and our implementation of QPET [4]. The experimental results show the following. (1) Using materialized views designed by our approach results in query runtimes comparable to runtimes of the queries using views output by the BPUS algorithm, which we used for comparison purposes. (2) Disk-space requirements for storing materialized views in our approach are acceptable, compared to the requirements for storing views produced by BPUS. (In addition, by design of views in our approach, the view relations require less system resources for maintenance

TPC-H Tables

Name	Size (bytes)
Lineitem	2,147,483,647
Part	1,193,906
Supplier	14,188,544
PartSupp	5,830,541
Customer	244,883,456
Orders	482,877,440
Nation1	2,103
Nation2	2,103
Region	396

Fig. 4. Sizes of base TPC-H tables (in bytes).

[15] This condition determines a more general class of queries than queries that have the same join conditions.

than the relations for BPUS views, see Section 4.4.) (3) Finally, the time required to *design* views in our approach can be drastically lower than the time required to design BPUS views for the same queries.

We give here just a brief summary of the experiments; a detailed account of the experimental setup and results can be found in Appendix B. The goal of the experiments was to compare evaluation costs for a workload of aggregate star-schema queries in three settings:

- using just the stored relations in the original database;
- using materialized views obtained by our approach; and
- using materialized views obtained by an approach in the literature.

As an approach in the literature, we chose the BPUS algorithm of [7] because it gives strong performance guarantees when applied to a wide range of types of view lattices. (The approach of [9] achieves the same guarantees as BPUS for a more restricted set of view lattices.) For our approach in the experiments, we chose to design and materialize *fact-table views*, that is, views whose `FROM` clause has just the fact table in the star schema. Recall that query rewritings with fact-table views have at least as many joins as rewritings of the same query using other view types, and are thus the "least efficient" rewritings in our approach. Thus, by choosing fact-table views for the experiments, we were interested in seeing how much worse query-evaluation costs can be when using our approach than when using the BPUS approach. In the remainder of this section, by "BPUS views or rewritings" ("fact-table views or rewritings," respectively) we refer to the views or rewritings based on the BPUS approach (based on our approach, respectively).

Query ID	Rows Returned	Fact-Table Views Used		BPUS Views Used	
		View ID	View Size	View ID	View Size
1	1	112	121,758	112	121,758
3	1,281	113	199,699,334	5105	410,344,447
5	5	5	141,452,410	6656	4,938,267
6	1	120	1,461,958	120	1,461,958
7	1	101	219,496,362	6496	127,090,353
8	3	39	192,158,348	7936	761,573,474
9	752,916	39	192,158,348	3074	77,529,577
10	505	17	43,995,526	5008	922,962,047

Fig. 5. Answer sizes for workload queries and views; view sizes are in bytes. Fact-table views are the same for `Q8` and `Q9`.

We did the experiments on a TPC-H database benchmark [26]; the sizes of the stored tables are shown in Figure 4. (We used the scale factor of one for the stored data.) The query workload for the experiments had eight aggregate queries, with between one and eight tables in the `FROM` clause and with aggregation on the `Lineitem` table, which is the fact table in the experiments.[16] The queries can be used as an input to the BPUS algorithm because the referential-integrity constraints on the stored TPC-H data guarantee that the views produced by BPUS, with all eight tables in the `FROM` clause, can be used to equivalently rewrite even those queries whose `FROM` clause has fewer tables.

We used the BPUS algorithm to design views for both fact-table rewritings (as a subroutine in our algorithm, see Section 4.4) and BPUS rewritings of the workload queries. In both cases, size estimates for relations in the view lattices were obtained by running the queries for all possible lattice views on TPC-H stored data with scale factor of 0.1 and by extrapolating the sizes of the answers to the queries to the size of the stored data used to evaluate the workload queries and their rewritings.

	Time to Generate Views (seconds)	Number of Nodes In the Lattice
Fact-Table Lattice	40	128
BPUS Lattice	1,920	8,192

Fig. 6. View-generation runtimes and lattice sizes, for fact-table and BPUS views.

To design views for BPUS rewritings of the workload queries, we ran the BPUS algorithms using the workload queries as inputs. As explained in Section 4, in our approach the `FROM` clause of the views we design determines automatically the rewriting of each query *before* we design the actual views. (The BPUS approach also determines the rewritings automatically based on which views can be used to evaluate the input queries.) Thus, once we settled on fact-table views for the rewritings in our approach, the rewritings we obtained determined automatically the input queries to the BPUS subroutine, which then designed

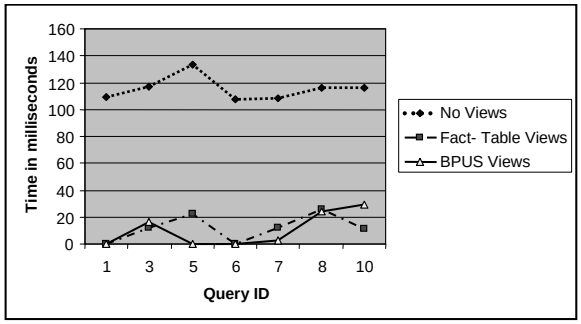

Fig. 7. Runtimes of the workload queries.

[16] Appendix B has definitions of all the queries.

fact-table views for the workload. Note that by design of BPUS, in each case we obtained a set of views for the entire query workload, rather than for its individual queries. In addition, recall that by design of BPUS, views produced by the algorithm have no comparisons with constants and can thus be used to evaluate any instantiation of the *parameterized* versions of the workload queries.

The fact-table and BPUS rewritings of the workload queries determined view lattices in the BPUS algorithm for seven and thirteen grouping attributes respectively. (The grouping arguments in a view lattice are a union of the grouping arguments of all the input queries for BPUS.) Figure 6 shows the number of nodes in view lattices and the runtimes for BPUS when called to generate views for fact-table rewritings and for BPUS rewritings. Increasing the number of tables in the `FROM` clause of views in our approach would tend to cause an increase in the number of required output arguments in the views. (These required arguments include those attributes of the tables in each view's `FROM` clause that participate in join and selection conditions in the rewritings, as well as those attributes of the tables in the `FROM` clause that are grouping arguments of the query.) Accordingly, the size of the view lattice explored by the BPUS subroutine would tend to increase exponentially with each table added to the `FROM` clause of views designed in our approach. The reason is, the size of a view lattice in the BPUS algorithm is exponential in the number of grouping attributes in the algorithm input.

Figure 5 has information on views produced by the BPUS subroutine for fact-table rewritings (seven views; note that the same view was produced for queries `Q8` and `Q9`) and for BPUS rewritings (eight views) for our query workload. Intuitively, storing materialized fact-table views should require more disk space than storing BPUS views, because a grouping argument of a fact-table view is a *key* argument of a dimension table whenever the corresponding grouping argument of a BPUS view (for the same workload query) is a nonkey argument of the same dimension table. When the views were materialized on the database for the experiments, storing the relations for the fact-table views did not require significantly more disk space than storing the relations for the BPUS views.

As discussed in Section 4, once the view-design subroutine returns aggregate views in our approach, the most appropriate materialized views (i.e., views that have a minimal-size relation among the views that have all the required grouping arguments for the rewriting) are selected automatically into query rewritings that were designed before calling the subroutine. Similarly, selecting the most appropriate views to rewrite workload queries is straightforward in the BPUS approach, which we use for comparison in the experiments. (In fact, building rewritings in the BPUS approach is a special case of building rewritings in our approach.)

Figure 5 shows which views were used to evaluate which queries using fact-table and BPUS rewritings. To measure the evaluation costs of the workload queries with and without materialized views, we ran the original workload queries and their fact-table and BPUS rewritings on the test database; each runtime measurement is the average based on ten runs on the database. Figure 7 shows the resulting averaged runtimes for seven of the eight workload queries. (The runtimes of the modified TPC-H query `Q9` are not shown in the diagram, as those runtimes were over 5000ms, both for the original query formulation and for the fact-table rewriting. This effect seems to be linked to the number of rows in the answer to the query, see Figure 5.) The dotted line at the top of Figure 7 shows runtimes for the original queries; at the bottom, the dashed line shows runtimes for fact-table rewritings, and the solid line shows runtimes for BPUS rewritings of the queries. As can be seen from Figure 7, using materialized views designed in our approach resulted in query runtimes comparable to runtimes of the queries using views produced by the BPUS algorithm; both runtimes were markedly lower than those for the same queries without views. Note that for the class of queries we study in this paper, building rewritings in the optimizer, once the views are materialized, is just a matter of replacing the fact table by a suitable view and thus takes negligible time.

The experiments described above were run for a workload of point queries, that is, queries without inequality selection conditions. By design, views and rewritings in our approach can be used to evaluate either point queries or queries with inequality comparisons on the same arguments, including range-aggregate queries. (That is, if a point query has equality selection conditions on arguments `A1,...,Ak`, all comparisons in the corresponding query with inequality must be on the same arguments.) Our experiments with range-aggregate versions of some of the point queries in the test workload show that the runtimes for the

rewritings of the range-aggregate queries are comparable with the runtimes of the rewritings (both fact-table and BPUS) of the corresponding point queries and are significantly lower than the runtimes of the queries when evaluated without views.

Acknowledgments

We are grateful to Kyoung-hwa Kim and Simran Sandhu who have been part of the implementation team.

References

1. Shasha, D., Bonnet, P. Database Tuning: Principles, Experiments, and Troubleshooting Techniques. Morgan Kaufmann (2002) `http://www.distlab.dk/dbtune/`.
2. Microsoft Research AutoAdmin Project: Self-Tuning and Self-Administering Databases. (`http://research.microsoft.com/dmx/autoadmin/default.asp`)
3. IBM Autonomic Computing. (`http://www.research.ibm.com/autonomic/`)
4. Chirkova, R., Gupta, S., Kim, K.H., Sandhu, S. Extensible framework for query-performance enhancement by tuning. Code downloads and documentation are available from `http://research.csc.ncsu.edu/selftune/` (2004)
5. Chirkova, R., Halevy, A., Suciu, D. A formal perspective on the view selection problem. VLDB Journal **11** (2002) 216–237
6. Agrawal, S., Chaudhuri, S., Narasayya, V. Automated selection of materialized views and indexes in SQL databases. In: Proceedings of VLDB (2000) 496–505
7. Harinarayan, V., Rajaraman, A., Ullman, J. Implementing data cubes efficiently. In: Proc. SIGMOD (1996) 205–216
8. Gupta, H., Harinarayan, V., Rajaraman, A., Ullman, J. Index selection for OLAP. In: Proceedings of ICDE (1997) 208–219
9. Shukla, A., Deshpande, P., Naughton, J. Materialized view selection for multidimensional datasets. In: Proceedings of VLDB (1998) 488–499
10. Afrati, F., Chirkova, R. Selecting and using views to compute aggregate queries. Submitted for publication; earlier version available at `http://dbgroup.ncsu.edu/aggregAquv.pdf` (2004)
11. Halevy, A.Y. Answering queries using views: A survey. VLDB Journal **10** (2001) 270–294
12. Chaudhuri, S., Krishnamurthy, R., Potamianos, S., Shim, K. Optimizing queries with materialized views. In: Proceedings of the Eleventh International Conference on Data Engineering (ICDE) (1995) 190–200
13. Chaudhuri, S., Narasayya, V. AutoAdmin 'What-if' index analysis utility. In: Proc. SIGMOD (1998) 367–378
14. Agrawal, S., Chaudhuri, S., Narasayya, V. Materialized view and index selection tool for Microsoft SQL Server 2000. In: Proceedings of ACM SIGMOD (2001)
15. Widom, J. Research problems in data warehousing. In: Proceedings of CIKM (1995)
16. Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M. Data cube: A relational aggregation operator generalizing Group-by, Cross-Tab, and Sub Totals. Data Mining and Knowledge Discovery **1** (1997) 29–53
17. Chaudhuri, S., Dayal, U. An overview of data warehousing and OLAP technology. SIGMOD Record **26** (1997) 65–74
18. Agarwal, S., Agrawal, R., Deshpande, P., Gupta, A., Naughton, J., Ramakrishnan, R., Sarawagi, S. On the computation of multidimensional aggregates. In: Proceedings of VLDB (1996) 506–521
19. Gupta, A., Harinarayan, V., Quass, D. Aggregate-query processing in data warehousing environments. In: Proceedings of VLDB (1995) 358–369
20. Srivastava, D., Dar, S., Jagadish, H., Levy, A. Answering queries with aggregation using views. In: Proc. VLDB (1996) 318–329
21. Cohen, S., Nutt, W., Serebrenik, A. Rewriting aggregate queries using views. In: Proceedings of PODS (1999) 155–166
22. Chaudhuri, S., Narasayya, V. An efficient cost-driven index selection tool for Microsoft SQL server. In: Proceedings of VLDB (1997) 146–155
23. Kimball, R., Ross, M. The Data Warehouse Toolkit (second edition). Wiley Computer Publishing (2002)
24. Afrati, F., Li, C., Mitra, P. Answering queries using views with arithmetic comparisons. In: Proc. PODS (2002) 209–220
25. Levy, A., Mendelzon, A., Sagiv, Y., Srivastava, D. Answering queries using views. In: Proceedings of PODS (1995) 95–104
26. TPC-H: TPC Benchmark H (Decision Support). (Available from `http://www.tpc.org/tpch/spec/tpch2.1.0.pdf`)
27. PostgreSQL (Open source database-management system) `http://www.postgresql.org/`.
28. TPC-H (TPC Benchmark H (Decision Support).) Available from `http://www.tpc.org/tpch/ spec/tpch2.1.0.pdf`.

A Example: Materialized Views Explored by the BPUS Algorithm of [7]

This example shows the format of materialized views explored by the greedy algorithm BPUS of [7] for the query workload { `Q1, Q2` } in Example 1 of Section 1.

Example 3. Queries `Q1` and `Q2` in Example 1 are defined using a fact table `Sales` and two dimension tables, `Customer` and `Time`. Depending on the data in these three tables and on the amount of disk space available for storing materialized views, the approach BPUS of [7] could let us evaluate both queries using a set of two views, { `W1, W2` }, with the following definitions:

```
W1: SELECT c.CustID AS CID1, Year, Month,
    SUM(QuantitySold) AS SQS1
    FROM Sales s, Time t, Customer c
    WHERE s.DateID = t.DateID
    AND s.CustID = c.CustID
    GROUP BY c.CustID, Year, Month;
```

```
W2: SELECT t.Year AS Year2, State,
    SUM(QuantitySold) AS SQS2
    FROM Sales s, Time t, Customer c
    WHERE s.DateID = t.DateID
    AND s.CustID = c.CustID
    GROUP BY t.Year, State;
```

Query `Q1` (`Q2`, respectively) can be answered using selection, grouping, and aggregation on the view `W1` (`W2`, respectively). Finding the grouping arguments of the views `W1` and `W2` using BPUS requires considering all subsets of the set of four attributes { `CustID, Year, Month, State` } of the stored tables.

Another BPUS solution for the workload { `Q1, Q2` } could comprise a *single* aggregate view, `W`. For instance, query `Q1` could be evaluated using this rewriting `R`:

```
R: SELECT CustID, CustName,Addr, sum(SumQS)
   FROM W WHERE Year = 2004 AND Month >= 4 AND Month <= 6
   GROUP BY CustID, CustName, Addr;
```

That is, suppose we have a materialized aggregate view `W` that has all "necessary" arguments; we will elaborate shortly on which arguments are necessary in `W` for evaluating `Q1` and `Q2`. Let `W` have an attribute `SumQS` whose value is the sum of `QuantitySold` (see the schema of `Sales` in Example 1) for each combination of the grouping arguments of `W`. Then the answer to `Q1` is computed using the rewriting `R` by doing `Q1`'s grouping and aggregation on those rows in the answer to `W` that satisfy the conditions `Year = 2004 AND Month >= 4 AND Month <= 6`. The answer to `Q2` could be computed using `W` in a similar way.

Here is one possible definition for `W`:

```
W: SELECT s.CustID, Year, Month, State, SUM(QuantitySold) AS SumQS
   FROM Sales s, Time t, Customer c
   WHERE s.DateID = t.DateID AND s.CustID = c.CustID
   GROUP BY s.CustID, Year, Month, State;
```

That is, `W` returns total sales per customer ID, year, month, and customer state.

BPUS [7] considers all sets of views — including { `W1, W2` } and { `W` } — that (1) can be used to evaluate the query `Q1` using the rewriting `R` above, and (2) can be used to evaluate the query `Q2` using the same approach. All such sets $\mathcal{W}$ of views need to satisfy several requirements. First, each view in the set of views $\mathcal{W}$ "covers" the body of each workload query, by using all the relations in the `FROM` clause and all the join conditions of each query. Second, for each workload query, at least one view in $\mathcal{W}$ needs to have in its `GROUP BY` clause at least all the grouping arguments of the query and the arguments in all the query's selection conditions. (In this sense, the above SQL definition of `W` is "minimal" for the queries `Q1` and `Q2`, because all the arguments in the `SELECT` clause of `W` are used in rewritings of the two queries.) Finally, each view in $\mathcal{W}$ should have an attribute (`SumQS` in our example) whose value is the result of aggregating the selected argument (`QuantitySold` in our example) for each combination of the grouping arguments of the view.

B Experimental Setup and Results

We have used the TPC-H [28] database with scale factor of one and a query workload based on TPC-H queries.

B.1 The Modified TPC-H Schema

The only modification of the schema concerns the `Nation` table: To create a workload of queries without self-joins (i.e., without duplicate occurrences of the same table in the `FROM` clause), we used two copies of the `Nation` table, named `Nation1` and `Nation2`. In addition, we do not show here the schema of the `Partsupp` relation, as `Partsupp` is not in definitions of the queries we used to define our query workload.

Argument names in boldface denote key attributes.

Part(**PARTKEY**,NAME,MFGR,BRAND,TYPE,SIZE,CONTAINER,RETAILPRICE,COMMENT)
Table size: ScaleFactor × 200,000

Supplier(**SUPPKEY**,NAME,ADDRESS,NATIONKEY,PHONE,ACCTBAL,COMMENT)
Table size: ScaleFactor × 10,000

Customer(**CUSTKEY**,NAME,ADDRESS,NATIONKEY,PHONE,ACCTBAL,MKTSEGMENT,COMMENT)
Table size: ScaleFactor × 150,000

Nation1(**NATIONKEY**,NAME,REGIONKEY,COMMENT)
Table size: 25

Nation2(**NATIONKEY**,NAME,REGIONKEY,COMMENT)
Table size: 25

Lineitem(**ORDERKEY**,PARTKEY,SUPPKEY,**LINENUMBER**,QUANTITY,
EXTENDEDPRICE,DISCOUNT,TAX,RETURNFLAG,LINESTATUS,SHIPDATE,
COMMITDATE,RECEIPTDATE,SHIPINSTRUCT,SHIPMODE,COMMENT)
Table size: ScaleFactor × 6,000,000

Orders(**ORDERKEY**,CUSTKEY,ORDERSTATUS,TOTALPRICE,ORDERDATE,
ORDERPRIORITY,CLERK,SHIPPRIORITY,COMMENT)
Table size: ScaleFactor × 1,500,000

Region(**REGIONKEY**,NAME,COMMENT)
Table size: 5

Referential-integrity constraints on the schema:

From Lineitem(PARTKEY) to Part(PARTKEY)
From Lineitem(SUPPKEY) to Supplier(SUPPKEY)
From Customer(NATIONKEY) to Nation2(NATIONKEY)
From Supplier(NATIONKEY) to Nation1(NATIONKEY)
From Nation2(REGIONKEY) to Region(REGIONKEY)
From Orders(CUSTKEY) to Customer(CUSTKEY)
From Lineitem(ORDERKEY) to Orders(ORDERKEY)

B.2 The Product-of-Dimensions Views

For the experiments, each product-of-dimensions (POD) view is defined as follows:

- the `FROM` and `WHERE` clauses of the query represent a join of all the relations in Section B.1, as given in queries in [28];
- for each query, the `GROUP BY` and `SELECT` clauses of the query is as specified (in Datalog) in Section B.3.

B.3 The Query Workload: Modified TPC-H Queries

In this section, the IDs of the modified queries are the IDs of the corresponding original TPC-H queries in [28]. In the experiments, all the placeholders for constants in the queries below were replaced by constant values occurring in the TPC stored data. The query workload for the experiments consists of the modified queries `Q1, Q3, Q5, Q6, Q7, Q8, Q9, Q10`.

In the Datalog formulations, the naming of stored relations is as follows: `Nat1` refers to `Nation1`, `Nat2` refers to `Nation2`, and the remaining tables are referred to by the first letter of their name.

Pricing Summary Report Query (Q1)

1. Objective: Reports the amount of business that was billed, shipped, and returned.
2. Modified query Q1 for the experiments, in SQL:

```
SELECT
      l_returnflag,
      l_linestatus,
      SUM(l_extendedprice),
      COUNT(*)
FROM
      lineitem
WHERE
      l_shipdate = date '[DATE]'
GROUP BY
      l_returnflag,
      l_linestatus;
```

3. Modified query Q1 for the experiments, in Datalog:

$q_1(RF, LS, sum(EX), count)$:-
$L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, `date', CD, RD, SI, SM, C8)$.

4. Input for product-of-dimensions lattice:

$q_1^P(RF, LS, SD)$

That is, to be answered using the outputs of the BPUS algorithm of [7] on the product-of-dimensions lattice, this query needs a view whose grouping arguments include arguments `Returnflag, Linestatus,` and `Shipdate` of the relation `Lineitem`.
In general, the rule for forming q_i^P for a query q_i is as follows: take all grouping arguments of q_i and all the arguments of q_i that are involved in (in)equality comparisons with constants.
5. Definition of product-of-dimension view for the query: on `prodOfDims`, see Section B.2.
In SQL:

```
SELECT returnflag, linestatus, shipdate,
SUM(extendedprice) AS sumex, count(*) AS totalcount
INTO _0112
FROM prodOfDims
GROUP BY returnflag, linestatus, shipdate;
```

In Datalog:

$V0112(RF, LS, SD, sum(EX), count(*))$:-
$prodOfDims$.

6. Rewriting of the query using the product-of-dimension view:
In SQL:

```
SELECT returnflag, linestatus, SUM(sumex), SUM(totalcount) AS count
FROM _0112
WHERE shipdate = date '1998-12-01'
GROUP BY returnflag, linestatus;
```

In Datalog:

$Q1(RF, LS, SUM(sumex), SUM(totalcount))$:-
$V0112(RF, LS, `1998-12-01', sumex, totalcount)$.

7. Input for fact-table lattice[17]:

   $q_1^F(RF, LS, SD)$

   That is, to be answered using the outputs of the BPUS algorithm of [7] on the fact-table lattice, this query needs a view whose grouping arguments include arguments `Returnflag, Linestatus,` and `Shipdate` of the relation `Lineitem`. (Inputs q_1^P and q_1^F are the same because the body of q_1 has just one table. See below that whenever the body of a query has at least two tables, q^P and q^F for the query will be different.)
   In general, the rule for forming q_i^F for a query q_i is as follows: take all arguments A of `Lineitem` (in the body of the query q_i), such that: (1) A is a grouping argument in the head of q_i, or (2) A is involved in (in)equality comparisons with constants in the body of q_i, or (3) A is needed for joins with other relations in the body of q_i.
8. Definition of fact-table view for the query:
   In SQL:

```
SELECT returnflag, linestatus, shipdate,
SUM(extendedprice) AS sumex, count(*) AS totalcount
INTO _112
FROM lineitem
GROUP BY returnflag, linestatus, shipdate;
```

   In Datalog:

$$V112(RF, LS, SD, sum(EX), count(*)) :- \\ L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, SD, CD, RD, SI, SM, C8).$$

9. Rewriting of the query using the fact-table view:
   In SQL:

```
SELECT returnflag, linestatus, SUM(sumex), SUM(totalcount) AS count
FROM _112
WHERE shipdate = date '1998-12-01'
GROUP BY returnflag, linestatus;
```

   In Datalog:

$$Q1(RF, LS, SUM(sumex), SUM(totalcount)) :- \\ V112(RF, LS, '1998-12-01', sumex, totalcount).$$

Shipping Priority Query (Q3)

1. Objective: Retrieves the ten unshipped orders with the highest value.
2. Modified query Q3 for the experiments, in SQL:

```
SELECT
      l_orderkey,
      sum(l_extendedprice),
      o_orderdate,
      o_shippriority
FROM
      customer,
      orders,
      lineitem
```

[17] In our experiments, `Lineitem` was the designated fact table; we treated the remaining tables as dimension tables.

```
WHERE
      l_returnflag = '[RETURNFLAG]'
      AND c_custkey = o_custkey
      AND l_orderkey = o_orderkey
      AND l_shipdate = date '[DATE]'
GROUP BY
      l_orderkey,
      o_orderdate,
      o_shippriority;
```

3. Modified query Q3 for the experiments, in Datalog:

$$q_3(OK, OD, SP, sum(EX)) \text{ :-}$$
$$C(CK, N4, A4, NK, P4, B4, MS, C4),$$
$$O(OK, CK, OS, TR, OD, OP, CL, SP, C7),$$
$$L(OK, PK, SK, LN, QT, EX, DS, TX, `r', LS, `date', CD, RD, SI, SM, C8).$$

4. Input for product-of-dimensions lattice:

$q_3^P(OK, OD, SP, SD, RF)$

5. Definition of product-of-dimension view for the query: on `prodOfDims`, see Section B.2.
In SQL:

```
SELECT orderkey, returnflag, linestatus, shipdate,
custkey, mktsegment, nationkey2, orderdate, shippriority,
regionkey, SUM(extendedprice) AS sumex, count(*) AS totalcount
INTO _5105
FROM prodOfDims
GROUP BY orderkey, returnflag, linestatus, shipdate, custkey,
mktsegment, nationkey2, orderdate, shippriority, regionkey;
```

In Datalog:

$$V5105(OK, RF, LS, SD, CK, MS, NK2, OD, SP, RK, SUM(ex), count(*)) :-$$
$$prodOfDims.$$

6. Rewriting of the query using the product-of-dimension view:
In SQL:

```
SELECT orderkey, SUM(sumex), orderdate, shippriority
FROM _5105
WHERE returnflag = 'R'
AND shipdate = '1995-04-16'
GROUP BY orderkey, orderdate, shippriority;
```

In Datalog:

$$Q3(OK, SUM(sumex), OD, SP) :-$$
$$V5105(OK, `R', LS, `1995-04-16', CK, MS, NK2, OD, SP, RK, SUMEX, totalcount).$$

7. Input for fact-table lattice:

$q_3^F(OK, SD, RF)$

8. Definition of fact-table view for the query:
In SQL:

```
SELECT orderkey, returnflag, linestatus, shipdate,
SUM(extendedprice) AS sumex, count(*) AS totalcount
INTO _113
FROM lineitem
GROUP BY orderkey, returnflag, linestatus, shipdate;
```

In Datalog:

$$V113(OK, RF, LS, SD, SUM(ex), COUNT(*)) :- \\ L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, SD, CD, RD, SI, SM, C8).$$

9. Rewriting of the query using the fact-table view:
   In SQL:

```
SELECT l.orderkey, SUM(sumex), orderdate, shippriority
FROM customer c, orders o, _113 l
WHERE returnflag = 'R'
AND o.custkey = c.custkey
AND l.orderkey = o.orderkey
AND shipdate = date '1995-04-16'
GROUP BY l.orderkey, orderdate, shippriority;
```

In Datalog:

$$Q3(OK, SUM(sumex), OD, SP) :- \\ V113(OK,' R', LS,' 1995-04-16', sumex, totalcount), \\ C(CK, N4, A4, NK, P4, B4, MS, C4), O(OK, CK, OS, TR, OD, OP, CL, SP, C7).$$

Local Supplier Volume Query (Q5)

1. Objective: Lists the revenue volume done through local suppliers.
2. Modified query Q5 for the experiments, in SQL:

```
SELECT
      n_name,
      SUM(l_extendedprice)
FROM
      customer,
      orders,
      lineitem,
      supplier,
      nation1,
      region
WHERE
      c_custkey = o_custkey
      AND l_orderkey = o_orderkey
      AND l_suppkey = s_suppkey
      AND s_nationkey = n_nationkey
      AND n_regionkey = r_regionkey
      AND r_name = '[REGION]'
      AND o_orderdate = date '[DATE]'
GROUP BY
      n_name;
```

3. Modified query Q5 for the experiments, in Datalog:

$q_5(N51, sum(EX))$:-
$C(CK, N4, A4, NK2, P4, B4, MS, C4)$,
$O(OK, CK, OS, TR, `date', OP, CL, SP, C7)$,
$L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, SD, CD, RD, SI, SM, C8)$,
$S(SK, N2, A2, NK1, P2, B2, C2)$,
$Nat1(NK1, N51, RK, C51)$,
$R(RK, `region', C6)$.

4. Input for product-of-dimensions lattice:

   $q_5^P(N51, OD, N6)$

5. Definition of product-of-dimension view for the query: on `prodOfDims`, see Section B.2.
   In SQL:

```
SELECT orderdate, nationkey1, n1_name, regionkey, name,
SUM(extendedprice) AS sumex, count(*) AS totalcount
INTO _6656
FROM prodOfDims
GROUP BY orderdate, nationkey1, n1_name, regionkey, name
```

   In Datalog:

   $V6656(OD, NK1, N1.name, RK, R.name, SUM(ex), count(*)) :-$
   $prodOfDims$.

6. Rewriting of the query using the product-of-dimension view:
   In SQL:

```
SELECT n1_name, SUM(sumex)
FROM _6656
WHERE name='AFRICA'
AND orderdate= '1993-07-01'
GROUP BY n1_name;
```

   In Datalog:

   $Q5(N1.name, SUM(sumex)) :-$
   $V6656('1993-07-01', NK1, N1.name, RK, ' AFRICA', SUMEX, totalcount)$.

7. Input for fact-table lattice:

   $q_5^F(OK, SK)$

8. Definition of fact-table view for the query:
   In SQL:

```
SELECT orderkey, suppkey, SUM(extendedprice) AS sumex, count(*) AS totalcount
INTO _005
FROM lineitem
GROUP BY orderkey, suppkey;
```

   In Datalog:

   $V5(OK, SK, SUM(EX), count(*)) :-$
   $L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, SD, CD, RD, SI, SM, C8)$.

9. Rewriting of the query using the fact-table view:
   In SQL:

```
SELECT n.name, SUM(sumex)
FROM customer c, orders o, _005 l, supplier s, nation1 n, region r
WHERE c.custkey=o.custkey
AND l.orderkey=o.orderkey
AND l.suppkey = s.suppkey
AND s.nationkey= n.nationkey
AND n.regionkey=r.regionkey
AND r.name='AFRICA'
AND orderdate= '1993-07-01'
GROUP BY n.name;
```

In Datalog:

$$Q5(N5, SUM(sumex)) :- \\ C(CK, N4, A4, NK, P4, B4, MS, C4), O(OK, CK, OS, TR, '1993-07-01', OP, CL, SP, C7), \\ V5(OK, SK, SUMEX, totalcount), S(SK, N2, A2, NK, P2, B2, C2), \\ Nat1(NK, N5, RK, C5), R(RK, 'AFRICA', C6).$$

Forecasting Revenue Change Query (Q6)

1. Objective: Quantifies the amount of revenue increase that would have resulted from eliminating certain companywide discounts in a given percentage range in a given year. Asking this type of "what if" query can be used to look for ways to increase revenues.
2. Modified query Q6 for the experiments, in SQL:

```
SELECT
      SUM(l_extendedprice)
FROM
      lineitem
WHERE
      l_shipdate = date '[DATE]'
      AND l_discount = '[DISCOUNT]'
      AND l_returnflag = '[RETURNFLAG]';
```

3. Modified query Q6 for the experiments, in Datalog:

$$q_6(sum(EX)) :- \\ L(OK, PK, SK, LN, QT, EX, 'discount', TX, 'r', LS, 'date', CD, RD, SI, SM, C8).$$

4. Input for product-of-dimensions lattice:

$$q_6^P(DS, SD, RF)$$

5. Definition of product-of-dimension view for the query: on `prodOfDims`, see Section B.2.
   In SQL:

```
SELECT discount, returnflag, linestatus, shipdate,
SUM(extendedprice) AS sumex, count(*) AS totalcount
INTO _0120
FROM prodOfDims
GROUP BY discount, returnflag, linestatus, shipdate;
```

In Datalog:

$$V120(DS, RF, LS, SD, SUM(ex), count(*)) :- \\ prodOfDims.$$

6. Rewriting of the query using the product-of-dimension view:
   In SQL:

```
SELECT SUM(sumex)
FROM _0120
WHERE shipdate = date '1994-09-01'
AND discount = 0.06
AND returnflag = 'R';
```

   In Datalog:

$Q6(SUM(sumex)) :-$
$V120(0.06,' R', LS,' 1994-09-01', SUMEX, totalcount).$

7. Input for fact-table lattice:

   $q_6^F(DS, SD, RF)$

8. Definition of fact-table view for the query:
   In SQL:

```
SELECT discount, returnflag, linestatus, shipdate,
SUM(extendedprice) AS sumex, count(*) AS totalcount
INTO _120
FROM lineitem
GROUP BY discount, returnflag, linestatus, shipdate;
```

   In Datalog:

$V120(DS, RF, LS, SD, SUM(EX), count(*)) :-$
$L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, SD, CD, RD, SI, SM, C8).$

9. Rewriting of the query using the fact-table view:
   In SQL:

```
SELECT SUM(sumex)
FROM _120
WHERE shipdate ='1994-09-01'
AND discount = 0.06
AND returnflag = 'R';
```

   In Datalog:

$Q6(SUM(sumex)) :-$
$V120(0.06,' R', LS,' 1994-09-01', SUMEX, totalcount).$

Volume Shipping Query (Q7)

1. Objective: Determines the value of goods shipped between certain nations to help in the renegotiation of shipping contracts.
2. Modified query Q7 for the experiments, in SQL:

```
SELECT
      n1.n_nationkey AS supp_nation,
      n2.n_nationkey AS cust_nation,
      SUM(l_extendedprice) AS revenue
FROM
      supplier,
```

```
      lineitem
      orders,
      customer,
      nation n1,
      nation n2
WHERE
      s_suppkey = l_suppkey
      AND o_orderkey = l_orderkey
      AND c_custkey = o_custkey
      AND s_nationkey = n1.n_nationkey
      AND c_nationkey = n2.n_nationkey
      AND n1.n_name = '[NATION1]'
      AND n2.n_name = '[NATION2]'
      AND l_shipdate = date '1995-01-01'
GROUP BY
      supp_nation,
      cust_nation;
```

3. Modified query Q7 for the experiments, in Datalog:

   $q_7(NK1, NK2, sum(EX))$:-
   $S(SK, N2, A2, NK1, P2, B2, C2)$,
   $L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, `1995-01-01', CD, RD, SI, SM, C8)$,
   $O(OK, CK, OS, TR, OD, OP, CL, SP, C7)$,
   $C(CK, N4, A4, NK2, P4, B4, MS, C4)$,
   $Nat1(NK1, `nation1', RK1, C51)$,
   $Nat2(NK2, `nation2', RK, C52)$.

4. Input for product-of-dimensions lattice:

   $q_7^P(NK1, NK2, N51, N52, SD)$

5. Definition of product-of-dimension view for the query: on `prodOfDims`, see Section B.2.
   In SQL:

```
SELECT linestatus, shipdate, nationkey2, n2_name, nationkey1,
n1_name, regionkey, SUM(extendedprice) AS sumex,
count(*) AS totalcount
INTO _6496
FROM prodOfDims
GROUP BY linestatus, shipdate, nationkey2, n2_name, nationkey1,
n1_name, regionkey;
```

   In Datalog:

   $V6496(LS, SD, NK2, N2.name, NK1, N1.name, RK, SUM(ex), count(*)) :-$
   $prodOfDims.$

6. Rewriting of the query using the product-of-dimension view:
   In SQL:

```
SELECT nationkey1 AS supp_nation, nationkey2 AS cust_nation,
SUM(sumex) AS revenue
FROM _6496
WHERE n1_name = 'CHINA'
AND n2_name = 'GERMANY'
AND shipdate =  '1995-03-31'
GROUP BY supp_nation, cust_nation;
```

In Datalog:

$Q7(NK1, NK2, SUM(sumex)) :-$
$V6496(LS,'1995-03-31', NK2,'GERMANY', NK1,'CHINA', RK, sumex, totalcount).$

7. Input for fact-table lattice:

   $q_7^F(OK, SK, SD)$

8. Definition of fact-table view for the query:
   In SQL:

```
SELECT orderkey, suppkey, linestatus, shipdate,
SUM(extendedprice) AS sumex, count(*) AS totalcount
INTO _101
FROM lineitem
GROUP BY orderkey, suppkey, linestatus, shipdate;
```

   In Datalog:

$V101(OK, SK, LS, SD, SUM(EX), count(*)) :-$
$L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, SD, CD, RD, SI, SM, C8).$

9. Rewriting of the query using the fact-table view:
   In SQL:

```
SELECT n1.nationkey AS supp_nation, n2.nationkey AS cust_nation, SUM(sumex) AS revenue
FROM supplier s, _101 l, orders o, customer c, nation1 n1, nation2 n2
WHERE l.suppkey = s.suppkey
AND l.orderkey = o.orderkey
AND o.custkey = c.custkey
AND s.nationkey = n1.nationkey
AND c.nationkey = n2.nationkey
AND n1.name = 'CHINA'
AND n2.name = 'GERMANY'
AND shipdate = date '1995-03-31'
GROUP BY supp_nation, cust_nation;
```

   In Datalog:

$Q7(N1.NK1, N2.NK2, SUM(sumex)) :-$
$S(SK, N2, A2, NK1, P2, B2, C2), V101(OK, SK, LS,'1995-03-31', SUMEX, totalcount),$
$O(OK, CK, OS, TR, OD, OP, CL, SP, C7), C(CK, N4, A4, NK2, P4, B4, MS, C4),$
$Nat1(NK1,'CHINA', RK1, C51), Nat2(NK2,'GERMANY', RK2, C52).$

National Market Share Query (Q8)

1. Objective: Determines how the market share of a given nation within a given region has changed over two years for a given part type.
2. Modified query Q8 for the experiments, in SQL:

```
SELECT
      n1.n_name,
      n2.n_name,
      SUM(l_extendedprice)
FROM
      part,
```

```
      supplier,
      lineitem
      orders,
      customer,
      nation n1,
      nation n2,
      region
WHERE
      p_partkey = l_partkey
      AND s_suppkey = l_suppkey
      AND l_orderkey = o_orderkey
      AND o_custkey = c_custkey
      AND c_nationkey = n2.n_nationkey
      AND n1.n_regionkey = r_regionkey
      AND r_name = '[REGION]'
      AND s_nationkey = n1.n_nationkey
      AND o_orderdate = date '1995-01-01'
      AND p_type = '[TYPE]'
GROUP BY
      n1.n_name,
      n2.n_name;
```

3. Modified query Q8 for the experiments, in Datalog:

$q_8(N52, N51, sum(EX))$:-
$P(PK, N1, MF, BR, `type', SZ, CN, RT, C1)$,
$S(SK, N2, A2, NK1, P2, B2, C2)$,
$L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, SD, CD, RD, SI, SM, C8)$,
$O(OK, CK, OS, TR, `1995-01-01', OP, CL, SP, C7)$,
$C(CK, N4, A4, NK2, P4, B4, MS, C4)$,
$Nat2(NK2, N52, RK1, C52)$,
$Nat1(NK1, N51, RK, C51)$,
$R(RK, `region', C6)$.

4. Input for product-of-dimensions lattice:

$q_8^P(N51, N52, TP, OD, N6)$

5. Definition of product-of-dimension view for the query: on `prodOfDims`, see Section B.2.
In SQL:

```
SELECT nationkey2, n2_name, orderdate, type, nationkey1,
n1_name, regionkey, name, SUM(extendedprice) AS sumex,
count(*) AS totalcount
INTO _7936
FROM prodOfDims
GROUP BY nationkey2, n2_name, orderdate, type, nationkey1,
n1_name, regionkey, name;
```

In Datalog:

$V7936(NK2, N2.name, OD, TP, NK1, RK, R.name, SUM(ex), count(*)) :-$
$prodOfDims.$

6. Rewriting of the query using the product-of-dimension view:
In SQL:

```
SELECT n1_name, n2_name, SUM(sumex)
FROM _7936
WHERE orderdate= '1996-11-30'
AND name='ASIA'
AND type= 'ECONOMY BURNISHED TIN'
GROUP BY n1_name, n2_name;
```

In Datalog:

$$Q8(N1.name, N2.name, SUM(sumex)) :- \\ V7936(NK2, N2.name, '1996-11-30', TP, NK1, RK, 'ASIA', sumex, totalcount).$$

7. Input for fact-table lattice:

   $q_8^F(PK, SK, OK)$

8. Definition of fact-table view for the query:
   In SQL:

```
SELECT orderkey, partkey, suppkey, linestatus,
SUM(extendedprice) AS sumex, count(*) AS totalcount
INTO _039
FROM lineitem
GROUP BY orderkey, partkey, suppkey, linestatus;
```

   In Datalog:

$$V039(OK, PK, SK, LS, SUM(EX), count(*)) :- \\ L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, SD, CD, RD, SI, SM, C8).$$

9. Rewriting of the query using the fact-table view:
   In SQL:

```
SELECT n1.name, n2.name, SUM(sumex)
FROM part p, supplier s, _039 l, orders o, customer c,
nation1 n1, nation2 n2, region r
WHERE p.partkey = l.partkey
AND s.suppkey = l.suppkey
AND l.orderkey = o.orderkey
AND o.custkey = c.custkey
AND c.nationkey = n2.nationkey
AND n1.regionkey = r.regionkey
AND r.name='ASIA'
AND s.nationkey = n1.nationkey
AND o.orderdate= '1996-11-30'
AND p.type= 'ECONOMY BURNISHED TIN'
GROUP BY n1.name, n2.name;
```

   In Datalog:

$$Q8(N51, N52, SUM(sumex)) :- \\ P(PK, N1, MF, BR, 'ECONOMYBURNISHEDTIN', SZ, CN, RT, C1), \\ S(SK, N2, A2, NK1, P2, B2, C2), V039(OK, PK, SK, SD, SUMEX, totalcount), \\ O(OK, CK, OS, TR, '1996-11-30', OP, CL, SP, C7), C(CK, N4, A4, NK2, P4, B4, MS, C4), \\ Nat1(NK1, N51, RK, C51), Nat2(NK2, N51, RK1, C52), R(RK, 'ASIA', C6).$$

Product Type Profit Measure Query (Q9)

1. Objective: Determies how much profit is made on a given line of parts, broken out by supplier nation and year.
2. Modified query Q9 for the experiments, in SQL:

```
SELECT
      n_name,
      p_name,
      SUM(l_extendedprice)
FROM
      part,
      supplier,
      lineitem,
      orders,
      nation
WHERE
      s_suppkey = l_suppkey
      AND p_partkey = l_partkey
      AND o_orderkey = l_orderkey
      AND s_nationkey = n_nationkey
GROUP BY
      n_name,
      p_name;
```

3. Modified query Q9 for the experiments, in Datalog:

$q_9(N51, N1, sum(EX))$:-
$P(PK, N1, MF, BR, TP, SZ, CN, RT, C1)$,
$S(SK, N2, A2, NK1, P2, B2, C2)$,
$L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, SD, CD, RD, SI, SM, C8)$,
$O(OK, CK, OS, TR, OD, OP, CL, SP, C7)$,
$Nat1(NK1, N51, RK1, C51)$.

4. Input for product-of-dimensions lattice:

$q_9^P(N51, N1)$

5. Definition of product-of-dimension view for the query: on `prodOfDims`, see Section B.2.
In SQL:

```
SELECT partkey, type, p_name, nationkey1, n1_name,
SUM(extendedprice) AS sumex, count(*) AS totalcount
INTO _3074
FROM prodOfDims
GROUP BY partkey, type, p_name, nationkey1, n1_name;
```

In Datalog:

$V3074(PK, TP, P.name, NK1, N1.name, SUM(ex), count(*)) :-$
$prodOfDims.$

6. Rewriting of the query using the product-of-dimension view:
In SQL:

```
SELECT n1_name, p_name, SUM(sumex)
FROM _3074
GROUP BY n1_name, p_name;
```

In Datalog:

$Q9(N1name, Pname, SUM(sumex)) :-$
$V3074(PK, TP, Pname, NK1, N1name, SUMEX, totalcount).$

7. Input for fact-table lattice:

   $q_9^F(OK, PK, SK)$

8. Definition of fact-table view for the query: same as for Q8.
9. Rewriting of the query using the fact-table view:
   In SQL:

```
SELECT n.name, p.name, SUM(sumex)
FROM part p, supplier s, _039 l, orders o, nation1 n
WHERE s.suppkey = l.suppkey
AND p.partkey = l.partkey
AND o.orderkey = l.orderkey
AND s.nationkey = n.nationkey
GROUP BY n.name, p.name;
```

In Datalog:

$Q9(N51, N1, SUM(sumex)) :-$
$P(PK, N1, MF, BR, TP, SZ, CN, RT, C1), S(SK, N2, A2, NK, P2, B2, C2),$
$V039(OK, PK, SK, SUMEX, totalcount), O(OK, CK, OS, TR, OD, OP, CL, SP, C7),$
$Nat1(NK, N51, RK, C5).$

Returned Item Reporting Query (Q10)

1. Objective: Identifies customers who might be having problems with the parts that are shipped to them.
2. Modified query Q10 for the experiments, in SQL:

```
SELECT
      c_custkey,
      c_name,
      SUM(l_extendedprice),
      c_acctbal,
      n_name,
      c_address,
      c_phone,
      c_comment
FROM
      customer,
      orders,
      lineitem,
      nation
WHERE
      c_custkey = o_custkey
      AND l_orderkey = o_orderkey
      AND o_orderdate = date '[DATE]'
      AND l_returnflag = 'R'
      AND c_nationkey = n_nationkey
GROUP BY
      c_custkey,
```

```
c_name,
c_acctbal,
c_phone,
n_name,
c_address,
c_comment;
```

3. Modified query Q10 for the experiments, in Datalog:

$q_{10}(CK, N4, B4, N52, A4, P4, C4, sum(EX))$:-
$C(CK, N4, A4, NK2, P4, B4, MS, C4)$,
$O(OK, CK, OS, TR, `date', OP, CL, SP, C7)$,
$L(OK, PK, SK, LN, QT, EX, DS, TX, `r', LS, SD, CD, RD, SI, SM, C8)$,
$Nat2(NK2, N52, RK, C52)$.

4. Input for product-of-dimensions lattice:

$q_{10}^{P}(CK, N4, B4, N52, A4, P4, C4, OD, RF)$

5. Definition of product-of-dimension view for the query: on `prodOfDims`, see Section B.2.
In SQL:

```
SELECT returnflag, custkey, c_name, acctbal, address,
phone, comment, nationkey2, n2_name, orderdate, regionkey,
SUM(extendedprice) AS sumex, count(*) AS totalcount
INTO _5008
FROM prodOfDims
GROUP BY returnflag, custkey, c_name, acctbal, address,
phone, comment, nationkey2, n2_name, orderdate, regionkey;
```

In Datalog:

$V5008(RF, CK, C.name, C.B4, C.A4, C.P4, C.C4, NK2, N2.name, OD, RK, SUM(ex), count(*)) :-$
$prodOfDims$.

6. Rewriting of the query using the product-of-dimension view:
In SQL:

```
SELECT custkey, c_name, SUM(sumex), acctbal, n2_name,
address, phone, comment
FROM _5008
WHERE orderdate= '1993-07-01'
AND returnflag= 'R'
GROUP BY custkey, c_name, acctbal, phone, n2_name,
address, comment;
```

In Datalog:

$Q10(CK, C.name, SUM(sumex), C.B4, N2.name, C.A4, C.P4, C.C4) :-$
$V5008('R', CK, C.name, C.B4, C.A4, C.P4, C.C4, NK2, N2.name, `1993-07-01', RK, sumex, totalcount)$.

7. Input for fact-table lattice:

$q_{10}^{F}(RF, OK)$

8. Definition of fact-table view for the query:
In SQL:

```
SELECT orderkey, returnflag, SUM(extendedprice) AS sumex,
count(*) AS totalcount
INTO _017
FROM lineitem
GROUP BY orderkey, returnflag;
```

In Datalog:

$$V017(OK, RF, SUM(EX), count(*)) :-$$
$$L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, SD, CD, RD, SI, SM, C8).$$

9. Rewriting of the query using the fact-table view:
In SQL:

```
SELECT c.custkey, c.name, SUM(sumex), c.acctbal,
n.name, c.address, c.phone, c.comment
FROM customer c, orders o, _017 l, nation2 n
WHERE c.custkey = o.custkey
AND l.orderkey = o.orderkey
AND o.orderdate= '1993-07-01'
AND l.returnflag= 'R'
AND c.nationkey = n.nationkey
GROUP BY c.custkey, c.name, c.acctbal,
c.phone, n.name, c.address, c.comment;
```

In Datalog:

$$Q10(CCK, CN4, SUM(sumex), CB4, NN5, CA4, CP4, CC4) :-$$
$$C(CCK, CN4, CA4, NK, CP4, CB4, MS, CC4), V017(OK, 'R', LS, SUMEX, totalcount),$$
$$O(OK, CK, OS, TR, '1993-07-01', OP, CL, SP, C7), Nat2(NK, NN5, RK, C5).$$