Groupwise Processing of Relational Queries

Damianos Chatziantoniou* Kenneth A. Ross*
Department of Computer Science, Columbia University
damianos,kar@cs.columbia.edu

Abstract

In this paper, we define and examine a particular class of queries called group queries. Group queries are natural queries in many decision-support applications. The main characteristic of a group query is that it can be executed in a group-by-group fashion. In other words, the underlying relation(s) can be partitioned (based on some set of attributes) into disjoint groups, and each group can be processed separately. We give a syntactic criterion to identify these queries and prove its sufficiency. We also prove the strong result that every group query has an equivalent formulation that satisfies our syntactic criterion. We describe a general evaluation technique for group queries, and demonstrate how an optimizer can determine this plan. We then consider more complex queries whose components are group queries with potentially different partitioning attributes. We give two methods to identify group query components within such a query. We also give some performance results for group queries expressed in standard SQL, comparing a commercial database system with our optimized plan on top of the same commercial system. These results indicate that there are significant potential performance improvements.

*This research was supported by a grant from the AT&T Foundation, by a David and Lucile Packard Foundation Fellowship in Science and Engineering, by a Sloan Foundation Fellowship, by NSF CISE grant CDA-9625374, and by an NSF Young Investigator award.

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

1 Introduction

With the recent interest in decision support systems and data warehousing has come a demand for techniques to evaluate and optimize very complex relational queries involving both aggregation and joins. Current commercial systems do not find good plans for many very complex queries.

In this paper we examine a particular class of complex queries, called *group queries*. Many complex decision support queries describe the following idea: for each value x in a dimension D (e.g. for each customer), evaluate a query Q'. This query Q' can be something simple (e.g. compute avg(sales)) or complex (e.g. include joins, selections, further aggregations, etc). To specify this kind of complex query in SQL, one has to embed D within many places of a complex piece of SQL code. This may have many drawbacks in terms of performance if the optimizer is not aware of the query's structure. This particular structure is often amenable to the following evaluation strategy: partition the data along dimension D, and evaluate Q' independently on each partition. Therefore it is important to be able to identify whether an SQL specification has the form mentioned above. In that case, that SQL specification constitutes a group query. Our main contributions are:

Group Queries (Section 2) We define the concept of a group query. A query Q is a group query with respect to certain partitioning attributes S if, for all databases, it is possible to answer Q by (a) partitioning the data according to the values for attribute(s) S, (b) evaluating another query Q' on each partition of the database, and (c) taking the union of the results. Common decision support queries require complex operations *within* groups and not just simple aggregation. We provide a syntactic criterion for identifying group queries and prove its sufficiency. We also prove the surprising result that *every* group query can be expressed in a way that satisfies our syntactic criterion. We consider arbitrary relational queries expressed in SQL, and do not restrict ourselves to a special syntax.

Execution Plan (Section 3) We demonstrate how a group query can be evaluated by partitioning the data and applying a significantly simpler query to each partition. Further, this evaluation plan can be automatically generated given a query that satisfies the criterion. The main benefits of this approach are that the simpler query is often much easier to optimize and evaluate, and often partitions are small enough to fit in main memory, reducing the I/O cost significantly. In the presence of multiple CPUs, the partitions could be processed in parallel. We do not assume that the data is partitioned in advance.

Performance (Section 4) We present some preliminary performance results indicating that there are often large potential gains in performance for group queries.

Partial Group Queries (Section 5) We consider queries that are made up of group query components. These components can potentially be evaluated in a groupwise fashion. We show how to recognize such components, and outline some evaluation techniques.

Motivating Examples

In this paper we will use the following relations typical of a decision-support database:

```
FYI_LOG(id,day,month,year,time,section,duration)
      FYI_BILL(cust-id,month,year,payment)
```

A news organization called FYI maintains a web server with up-to-date news information. The FYI_LOG relation stores the log of customer requests as they access different sections of the news. The id attribute is unique to each customer, the day, month, year and time attributes represent the time of the connection, the section attribute denotes some section of the news (e.g., sports, world, business, travel, politics, etc.) and the duration attribute gives the time spent on this section. The unit of abstraction is the connection, not the day, since we assume that the web server is updated continuously. The FYI_BILL relation stores information about customers' monthly payments. The FYI organization would like to ask a variety of queries on their database in order to understand user access patterns. Such queries are likely to heavily use aggregation; examples include:

Q1. For each customer, show how many times the world section was accessed and how many times the time spent in the world section was greater than the customer's average section time.

Q2. For each customer, find the maximum among the average durations for each section, along with the section's name.

Q3. For each customer and for the world section accessed on connection t, show the average time spent on that section during the connections prior to t, and the average time spent on that section during the connections following t.

Q4. For those customers and months for which the customer's payment was greater than the customer's average monthly payment, find the section with the maximum average duration during that month.

The answers to queries like these could be utilized for marketing purposes, to allocate resources to sections based on usage, or for behavioral research on customers over extended periods. It is not straightforward to express these queries in standard SQL. Multiple views joined together with complex conditions are necessary (correlated subqueries can also be used.) Let's see how query Q1 could be expressed in standard SQL.

```
create view V1 as
    select id, avg_d=avg(duration)
    from FYI_LOG
    group by id

create view V2 as
    select id, cnt=count(*)
    from FYI_LOG
    where section="WORLD"
    group by id

create view V3 as
    select C.id, cnt=count(*)
    from FYI_LOG C, V1
    where  C.id = V1.id and
           C.section = "WORLD" and
           C.duration > V1.avg_d
    group by C.id

 Q1:   select V2.id, V2.cnt, V3.cnt
       from V2, V3
       where V2.id=V3.id
```

We need several joins on the id attribute between the FYI_LOG relation and views on FYI_LOG. Although this query is *expressed* using joins, this query does not have to be *evaluated* using joins. While we won't go into details yet, it is apparent that for this query we could partition the FYI_LOG relation on the id attribute. Each of the views and the query can then be evaluated separately on each partition, with no interaction between the partitions being necessary. The answer to the query is simply the union of the answers in each partition.

One major advantage of this approach is that partitions may be sufficiently small that they fit in memory. If so, then the I/O complexity no longer depends on the complexity of the query. Complex in-memory operations can be done on the whole partition, each partition being processed in turn. In a parallel database system, multiple partitions could be simultaneously processed.

Query Q2 can be expressed in SQL as

```
create view V1 as
    select id, section, avg_d=avg(duration)
    from FYI_LOG
    group by id,section

create view V2 as
    select id, max_s=max(avg_d)
    from V1
    group by id

 Q2:   select V1.id, V1.section, V2.max_s
       from V1, V2
       where V1.id=V2.id and V2.max_s=V1.avg_d
```

A similar observation about partitioning can be made: the query (without the join condition on id) could be applied separately to partitions of FYI_LOG based in id, and the partial results could be unioned together.

Query Q3 is expressed in standard SQL as follows. View V1 contains the average duration of the sections read in the days before each access to the world section, and view V2 the average duration of the sections read in the days after. We join V1 and V2 in order to get the result in single tuples. The actual SQL formulation is omitted due to lack of space and appears in [Cha97]. Once again, FYI_LOG can be partitioned on id attribute and a simpler query can be posed separately on each partition. We shall discuss Query Q4 in Section 2.4.

2 Theoretical Framework

In this section, we introduce our terminology and define what we mean by a group query. We give a syntactic criterion for identifying group queries and prove that this condition is sufficient. We also show that *every* group query can be expressed in a form that satisfies our criterion.

2.1 Assumptions and Terminology

We assume that queries are written in terms of views, with no subqueries. This is a valid assumption since there are many ways to rewrite a subquery as a join of two (or more) views [Kim82, Day87, SPL96]. We initially assume that the database contains a single relation R. (Multiple relations will be considered in Section 2.4.) R may itself be a view or the result of another query, but from our point of view it is treated as an encapsulated table. (I.e., if R was a view, then we don't consider unfolding the definition of R into queries over R.)

We shall define below the notion of a query graph. A query graph has nodes that are relational operations. We consider three kinds of relational operations:

Basic Blocks A basic block is some combination of projections and selections applied to a join of relations. In SQL such operations are expressed as SELECT-FROM-WHERE queries without aggregates or attribute renaming. A base relation is also treated as a basic block.

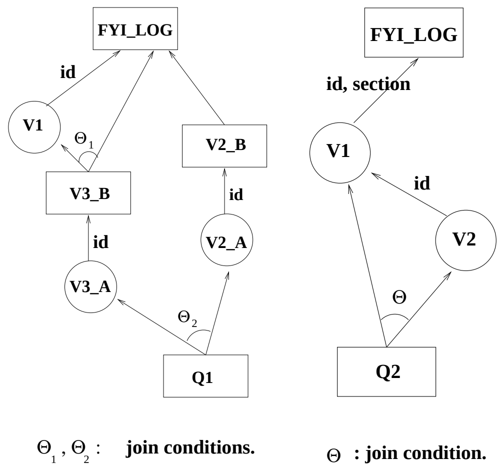
Aggregation Blocks An aggregation block is a single aggregation operation specifying a set of grouping attributes and a list of aggregate functions to be computed over the groups. The aggregation can optionally be followed by a selection. In SQL, such operations are expressed as SELECT-FROM-GROUPBY-HAVING queries where the SELECT clause includes all grouping attributes (plus some aggregates), and the FROM clause contains a single relation.

Set Blocks Set blocks express the set-oriented operations, namely union, intersection and difference of relations with the same schema. The corresponding SQL constructs are UNION, INTERSECT and EXCEPT.

Any relational query can be specified using these three operations. A query can be split into two blocks if necessary, for example if the query has both a WHERE clause and a GROUPBY clause then we can rewrite it as an aggregation block applied to a basic block. Note that we do *not* allow constant relations in queries, nor do we allow attribute renaming. (In [Cha97] we describe the extensions needed to handle the renaming operator.)

Definition 2.1: (Query Graph) Suppose that we are given a query Q. The nodes of the query graph for Q are the relational operations (as defined above) used in Q and its subviews. Basic blocks are written as rectangles, aggregation blocks as circles, and set blocks as diamonds. Every block is given a label corresponding to the view (or relation) name of that block. There is an edge from node labeled N_1 to node labeled N_2 if N_2 is mentioned in the FROM clause of N_1, or if N_2 is an argument of a set operation in N_1. An edge whose source is an aggregation block N is labeled with the grouping attributes of N. Edges coming out of a single basic block N are linked together with an arc, and jointly labeled by the join condition of N. Edges coming out of set blocks are not labeled. $\Box$

A query graph is a directed acyclic graph with a single source (root) node representing the query result. We shall conventionally draw the graph with edges "pointing up." For single-relation databases, there will always be a single sink node at the top of the picture. Figure 1 shows the query graphs for queries Q1, and Q2. Notice the separation of V2 and V3 from query Q1 into basic blocks (V2_B, V3_B) and aggregation blocks (V2_A, V3_A).



(a) query Q1 (b) query Q2

Figure 1: Query graphs

Definition 2.2: (Partitioning) Let S be a set of attributes from the underlying relation R, and let D represent the domain over which the tuples of S values range. For every node N in the query graph and for $x \in D$ we write $N(x)$ to represent the answer at node N when the extension of R is replaced by $\sigma_{S=x}(R)$. $\Box$

Observe that the attributes in S do not have to be attributes of the answer at node N for $N(x)$ to be well-defined. Also, observe that $R = \bigcup_{x \in D} R(x)$.

Definition 2.3: (Equality) If Q and Q' are queries we say $Q = Q'$ if the answer to query Q is the same as the answer to query Q' for all databases. □

2.2 Group Queries

Given a relation R and a set of attributes S of R, imagine the following computation: First, R is partitioned based on S. Then, a *new* query is executed on every partition. Finally, the partial results are unioned to form one relation. Any query on R that can be translated to such a computation is called a *group query* on attributes S.

Definition 2.4: We say that a query Q on a relation R is a *group query* on attributes S of R, iff there exists a query Q' such that $Q = \bigcup_{x \in D} Q'(x)$, where D is the domain of attributes S. □

It is the job of the optimizer to identify group queries and "decorrelate" them. We now give a criterion to identify group queries, and we prove its sufficiency.

Criterion 2.1: Let G be a query graph on relation R, and let S be a set of attributes of R.

- Every outgoing edge of any aggregate block in G must have as label a *superset* of S, and
- If a basic block N represents a join, then the join condition must conjunctively contain an equijoin on attributes S of each child of N. In other words, if there are edges from N to each of $f_1, f_2, \ldots, f_n$, then the joining condition θ should have the form $(f_1.S = f_2.S = \cdots = f_n.S) \wedge \theta'$, for some θ'. □

Example 2.1: Queries Q1, Q2, and Q3 satisfy Criterion 2.1 when S is {id}. Consider queries Q1 and Q2 in Figure 1. The join conditions are conjunctions of an equijoin on the id attribute with some other condition θ'. Also, id is among the grouping attributes in each of the aggregation operations. □

Criterion 2.1 is a *syntactic criterion*. In other words, given a query expressed in standard SQL and a set of attributes S, it is immediate to check whether the query meets the Criterion. We just examine the attributes of the group-by clauses and the join conditions. Furthermore, checking Criterion 2.1 need not be performed for all possible sets of partitioning attributes S. For a given query graph we need only consider sets S that (a) are subsets of the grouping attributes at all aggregation nodes, and (b) have all members equated in the appropriate way in each join node.

The following theorem demonstrates the *sufficiency* of Criterion 2.1.

Theorem 2.1: Every query graph that satisfies Criterion 2.1 defines a group query.
Proof: See [Cha97] where a more general result is proved. ∎

The proof of Theorem 2.1 is constructive: it shows how to construct a Q' that operates on partitions of R given Q. Essentially, Q' is Q with join nodes modified by removing the equality conditions between partitioning attributes. We call the resulting query Q' the *partial query* of Q, which we write as $pq(Q)$.

2.3 Necessity

Criterion 2.1 is not a necessary condition for a query to be a group query. For example, any query that returns the empty answer for all databases is a (trivial) group query whether or not its query graph satisfies Criterion 2.1. A nontrivial group query violating Criterion 2.1 is the following:

```
create view V1 as
select * from FYI_LOG where year<=1995
create view V2 as
select * from FYI_LOG where year>=1995
select year from V1, V2
where V1.year>=V2.year
```

The selection conditions together with the join condition imply the equality of the year attribute, and thus ensure that this is a group query on year. It is relatively straightforward to construct a query Q^* from a query Q such that Q^* is a group query if and only if Q is empty for all databases. However, deciding whether a query returns the empty set as an answer is an undecidable problem[DiP69].

Nevertheless, Theorem 2.2 shows that *every group query can be expressed using a query graph satisfying Criterion 2.1.* In other words, by restricting attention to queries satisfying Criterion 2.1 we can still express all group queries.

Theorem 2.2: For every group query Q there exists a query Q^* such that (i) $Q = Q^*$, and (ii) The query graph of Q^* satisfies Criterion 2.1.
Proof: This is a special case of a more general result shown in [Cha97]. ∎

2.4 Multiple Base Relations

In this section we extend our formalism to handle queries over multiple relations.

Example 2.2: Consider Query Q4 from Section 1. Its SQL version appears in [Cha97]. The query graph for this query is given in Figure 2. □

Multiple views on both FYI_LOG and FYI_BILL are defined and joined together. However, the key idea is still present. We can partition FYI_LOG on the id attribute, and partition FYI_BILL on the cust-id attribute and apply a query on partitions of FYI_LOG and FYI_BILL with the same value of id and cust-id respectively. A partition still needs to be evaluated if some (but not all) of the relations are empty on that partition; for example $R_1 - R_2$ could be nonempty on a partition even if R_2 were empty on that partition. At the end, we union the partial results to get the answer to the initial query.

The two attributes, `id` and `cust-id` must be from the same domain, although they may have different names. Thus we need to extend our concept of a set of partitioning attributes. The basic idea is that we pick an attribute from a common domain from each relation, and require that these attributes are *equivalent* for the purposes of partitioning. More generally, we can pick a *tuple* of attributes from each relation, and require an equivalence.

We would choose $\{(\texttt{FYI\_LOG}.\text{id}), (\texttt{FYI\_BILL}.\text{cust-id})\}$ as our partitioning set in Example 2.2. The idea is that we simultaneously partition the base relations on attributes with common domains. Queries can operate on multiple relations, so that query graphs can have multiple sinks.

The definition, criterion, and theorems can be appropriately extended. The details appear in [Cha97]. Query Q4 of Example 2.2 satisfies the more general criterion, and it is a group query (it satisfies $Q = \bigcup_{x \in D} Q(x)$ under a more general definition of $Q(x)$).

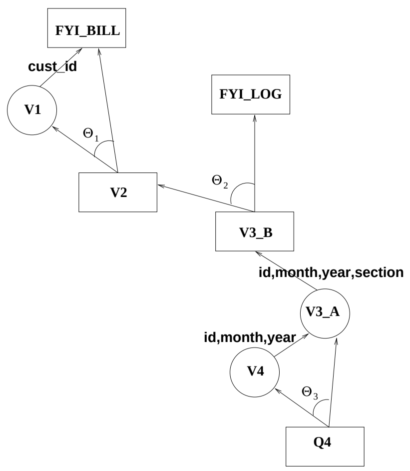


Figure 2: Query graph of Example 2.2

3 Execution Plan

Given a group query Q satisfying Criterion 2.1, we would like to be able to find efficient techniques for the evaluation of Q. It should come as no surprise that the first step of our suggested execution plan is to partition the base relation(s) according to the partitioning set. Thus we expect to have a number of separate partitions of the base relations, and we will subsequently process each partition independently, perhaps even in parallel. Note that we do not assume that the underlying data is partitioned in advance.

On each partition we can execute the query $pq(Q)$, with the system locally optimizing this query on the partitioned data set. Since the partitioned data set is likely to be much smaller than the original relation, it is likely that the partitioned data will fit in memory, putting an upper bound on the amount of I/O needed.

Nevertheless, for complex queries we cannot simply ignore CPU time. CPU time may contribute a large fraction of the cost of answering a query. (In Section 4 it will become apparent that CPU time does dominate for queries like Query Q3.) Thus it is essential to make the query on each partition as simple as possible, and to convey as much information to the optimizer as possible in the form of the query, so that the optimizer can do a good job of optimization.

Looking at $pq(Q)$ we can see that it is not ideal according to the discussion above. In particular, the partitioning attributes are still present throughout the query. This has several drawbacks.

(a) The partitioning attribute(s) will be redundantly represented in every tuple of the base relation(s). Since they take a single value on the partition, this representation is not necessary. The optimizer may also unnecessarily represent the partitioning attribute(s) in intermediate results.

(b) While it is possible to expect an optimizer to have (and use) information on the cardinalities of relations being queried, it is unreasonable to expect the optimizer to notice that there is a single value of one or more attribute(s) within a relation. As a result, operations like aggregation will group by more attributes than necessary. By making an inaccurate estimate of the number of groups, the optimizer may chose a suboptimal plan.

Fortunately, we can restructure the query $pq(Q)$ to avoid the difficulties above. Rather than simply partitioning R into partitions $R(x)$, we partition R into two relations. The first is $\hat{R}(x) = \hat{\pi}_S(R(x))$, where $\hat{\pi}_S$ *projects out* the attributes in S, leaving all other attributes. The second is a relation X with schema S and a single tuple x. If there are multiple sink relations then all can share the same X relation.

We can transform $pq(Q)$ to the *partition partial query* $ppq(Q)$ as follows: The sink node(s) are all replaced by $\hat{R}(x)$, and attributes S are removed from all nodes in $pq(Q)$. In particular, attributes in S are removed from the list of grouping attributes in aggregation nodes. Any node that needs the x value for selections or for inclusion in the final result (at the root node), can obtain it by first taking a cross-product (i.e., a join with an empty condition) of its argument(s) with X.

Because there is only one x value in the partition, it should be clear that $pq(Q) = ppq(Q)$. The advantages of $ppq(Q)$ are that: (a) The x value is not represented redundantly, and is used without duplication exactly where it is needed. (b) The optimizer has access to the cardinality information for X. Thus the optimized query plan for $ppq(Q)$ can reflect the fact that X has one tuple. (c) $ppq(Q)$ is considerably simpler than $pq(Q)$ or Q, and hence is likely to be more easily optimized.

If the partitioning is likely to be relatively uniform, then it is also possible to optimize $ppq(Q)$ *once* for a partition of the expected size, and to use the resulting plan on each partition.

The final step in the plan is to form the union of the partial results. This step can be avoided if all partition results are written to a common output relation.

Example 3.1: Let `FYI_LOG'` denote $\hat{\pi}_{\texttt{id}}(\texttt{FYI\_LOG})$ and let

X denote the single relation with schema `id` containing the single `id` value of the current partition. The partial query ppq(Q1) is the following:

```
create view V1_PPQ as
    select avg_d=avg(duration)
    from FYI_LOG'

create view V2_PPQ as
    select cnt=count(*)
    from FYI_LOG'
    where section="WORLD"

create view V3_PPQ as
    select cnt=count(*)
    from FYI_LOG' C, V1_PPQ
    where  C.duration > V1.avg_d and
           C.section = "WORLD"

 ppq(Q1):   select id, V2_PPQ.cnt, V3_PPQ.cnt
            from X, V2_PPQ, V3_PPQ
```

The form of V1_PPQ, V2_PPQ, and V3_PPQ ensure that each view represents a single value, so the optimizer should consider them as such in the joins that they appear in. With the `id` attribute in the views this would not have been possible without requiring the optimizer to know about the number of distinct attribute values in (temporary) relations. This query is substantially simpler than the original specification of Query Q1. □

Another reason why we expect our partitioning algorithm to perform well is that it reduces the complexity of the evaluation problem. In general, joins and aggregations have nonlinear complexity. Thus, if we can divide a query evaluation problem of size n into s (less difficult) subproblems of size n/s with overhead[1] linear in n, then we win in terms of complexity for any operations that take $\Omega(n \log n)$ time. This observation holds for both I/O time and CPU time. In the next section we demonstrate the disparity between the performance of a commercial system on a full group query and the performance of the same commercial system when partitioning and processing each partition in turn.

4 Performance Results

We used a popular commercial system, running on a Sparcserver 630MP with Solaris 2.5. We used the standard SQL of that system to run both the standard version of our examples and our method. Each group of `FYI_LOG` for a given `id` contains 50 tuples. 30 of these tuples belong to the "WORLD" section. In all cases, the cost is the average cost (elapsed time) of multiple runs on a lightly loaded machine.

Given a group query Q on attributes S, the partial query $ppq(Q)$ is found very easily, by dropping the equijoins on S from the join conditions and removing S from the grouping attributes in the group-by's clauses. Furthermore, S is replaced by parameters in the select clause. This is $ppq(Q)$. Then we sort the underlying relation(s) on attributes S and apply $ppq(Q)$ on each group. The total time is calculated by adding the sorting cost to the sum of costs of bringing each group in memory and processing it. For these measurements we use a single CPU and a single disk device; we do not take advantage of the potential parallelism available for processing the partitions. Furthermore, we sort the base relation(s) (instead of hash partition, for example), because this is the mechanism provided by the commercial system and we wanted an implementation on top of that system. Having incorporated hash-partitioning and parallelism, the performance would be substantially better.

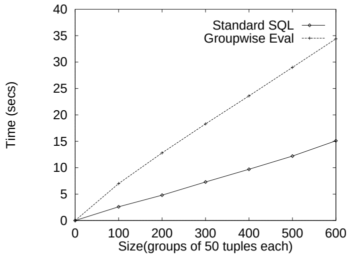
Figures 3(a),(b),(c) and (d) show the performance of queries $Q1$ to $Q4$ for both the commercial system's standard method and our method. We used relation sizes of 100 groups to 600 groups in increments of 100 groups. Note that the performance of our method is better than the standard method in this range only for queries Q3 and Q4. For queries Q1 and Q2, our method performs poorly for two reasons. First, the number of groups is small, and the joins are not expensive to perform (the standard plan). Second, the sorting step in our method is quite expensive. In cases (a) to (d) the number of groups is small. In (a) and (b), where the computation performed *within* each group is quite simple, the standard method beats our method. However, in cases (c) and (d), where the computation performed within each group is more complex (e.g. incorporates joins), our method is better, even for a small number of groups.

We also wish to consider relations of comparable sizes with a substantially larger number of groups. We have conducted performance experiments for queries Q1 and Q2 for relation sizes 4000 groups to 20000 groups in increments of 4000 groups. Each group contains 5 tuples this time. The performance results are shown in Figure 3(e) and (f). Cases (e) and (f) represent a much larger number of smaller groups. Our method performs better than the native commercial system's engine for (e) but not for (f).

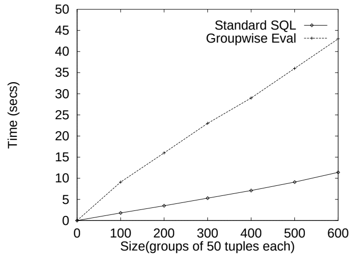
Note that we are biased in our experiments against our method, since we are not exploiting possible benefits of hash partitioning and parallelism. Further, our method is implemented *on top* of the commercial system rather than as part of it.

The cost of our method can be estimated as the sum of the cost of the partitioning step, plus the cost to apply the partial query $ppq(Q)$ on each group multipled by the estimated number of groups. In general, before our method is applied, its estimated cost should be compared with the estimated cost of other methods, since it is not always better than conventional techniques. See [Cha97] for a discussion of cost formulas for our techniques. Furthermore, the presence of indices should be considered since with sorting or hash-partitioning, indices may become useless for later computations. As a very general rule of thumb, our method is better when the complexity of the partial computation is high (like in queries Q3 and Q4) and/or the number of groups is large (like Q1, in Figure 3(e)).

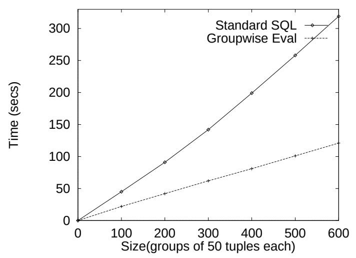
[1]The overhead is $O(n \log s)$ here.



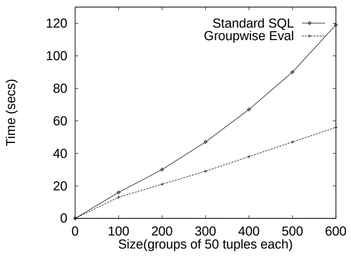
(a) Performance for Q1



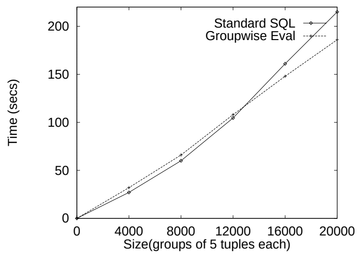
(b) Performance for Q2



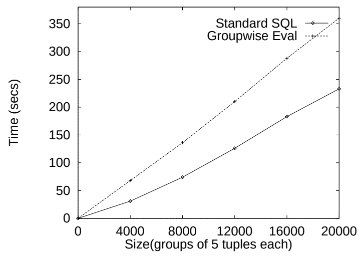
(c) Performance for Q3



(d) Performance for Q4



(e) Performance for Q1



(f) Performance for Q2

Figure 3: Performance of standard method versus groupwise processing for different relation sizes

5 Group Query Components

So far we have considered queries such that *the whole query* is a group query. A number of natural queries are not group queries with respect to a set of partitioning attributes S, but have components that can be classified as group queries on S. These components could then be evaluated as suggested in Section 3, within a larger query execution plan. In this section we define what it means for part of a query to be a group query and we describe techniques for identifying such parts of a query.

To describe better the idea of *group query components* within a query, consider the following example.

Example 5.1: (Group queries within a query)

Q5. Find the tuples that have duration greater than the average duration (over all customers) of the whole section. Then, from the selected tuples, find each user's average duration.

In this example, we have to find first the average duration for each section, select the tuples that have duration greater than the corresponding section's average duration, and then group over the `id` attribute. The query graph for this example is given in Figure 4(a). Note that this query is not a group query with respect to any set of attributes of `FYI_LOG`. However, we can find *parts* of the query that *are* group queries, i.e. we can "decompose" the query graph in two components, as shown in Figure 4(b). The first component represents a group query on the `section` attribute and the second component represents a group query on the `id` attribute. Each component represents *the part of the query that is relevant to grouping according to a set of attributes.* □

A common characteristic of decision support queries is that they aggregate according to several different sets of grouping attributes and then correlate the results through joins (cross-dimensional queries). This idea is implicit in papers that model multi-dimensional databases [LW96, AGS97]. In our framework this characteristic results in a query graph with several, possibly overlapping, component group queries.

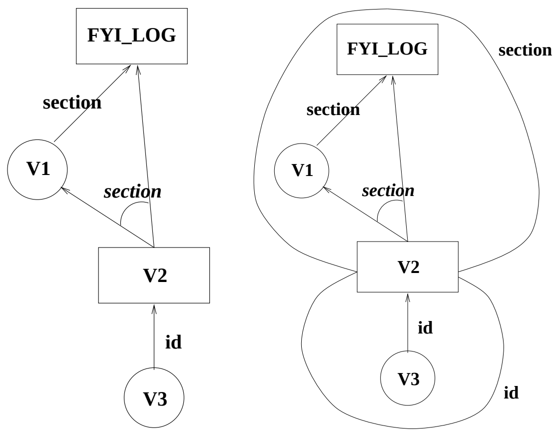
Having a complex decision support query, it is important to identify its group query components for a number of reasons.

Firstly, each component can be optimized locally. As a result, simpler and more efficient plans can be identified. For example, the group query component on the `section` attribute of query Q5 constitutes a multi-feature query (discussed in [CR96]).

Secondly, depending on the structure and the interrelation of these components, special cases with efficient plans can be identified. Results of one component can be pipelined to another, or components can be evaluated in parallel.

Finally, one of the key ideas in multiple-query optimization is to allow queries to be decomposed into smaller subqueries that now become the unit of execution [Sel88]. The idea of group query components is very similar: Group queries become now the unit of execution. In fact, our framework can be used for multiple-query optimization in the case of aggregate queries. Given a set of decision support queries, the first step is to identify the group query components of each query. To derive an execution plan, all these components should be considered. Research on multiple-query optimization has been focused on queries not involving aggregation. However, with the increased demand for decision support systems and data mining, we believe that multiple-query optimization in the context of complex aggregate queries will be very important.

In summary, we believe that our approach provides a



(a) query Q5 (b) group query components

Figure 4: Query graphs

framework that is suitable for complex aggregate query analysis and optimization, since it identifies the parts of a query that can be evaluated in a partitioned fashion.

5.1 Identifying Multiple Group Queries

We first define the notion of a component of a query graph, and use this notion to represent group query components of queries.

Definition 5.1: (Component) Given a query graph G, we say G' is a *component* of G if G' is a connected subgraph of G, and for every node N in G' of arity greater than one, either *all* outgoing edges of N are in G', or *none* is in G'. □

A component of a query graph may have one or more sink nodes, and one or more source (root) nodes. Having multiple source nodes presents no additional technical difficulties, and all of the concepts defined for query graphs also apply to components. A *maximal* group query component with respect to a set of attributes S is a component that satisfies Criterion 2.1 and it is maximal in the sense that it can not be extended with neighboring nodes to form a larger group query component on S. In general, we are interested only in maximal group query components. For optimization purposes it is better to partition as large a subquery as possible in order to simplify the processing of that subquery. Below, we briefly discuss two algorithms to identify group query components (for a detailed discussion, see [Cha97]). The idea of maximal group query components is fundamental in both.

The first algorithm has running time $O(mn)$, where m is the number of aggregate and join nodes in the query graph and n the total number of nodes.

Algorithm 5.1: Initially, each aggregate node is associated with its grouping attributes and each join node with the set of attributes equated in a conjunct within the join condition. These nodes are the initial components. We keep expanding these components with neighboring nodes such that the Criterion 2.1 is satisfied with respect to the set of attributes associated with these components. We repeat this step until we can not expand any component further (note that we do not expand from any base relation(s)).

Example 5.2: Consider a query with query graph as shown in Figure 5(a). Algorithm 5.1 would give the decomposition of Figure 5(b). □

The ouput of Algorithm 5.1 is *one* decomposition with possibly overlapping maximal group query components. However, overlap means repeated work. We would like to have a decomposition with no overlapping group query components. For that reason, we modify slightly Algorithm 5.1.

Algorithm 5.2: *Choose* an aggregate node and a set S to be the node's grouping attributes, or *choose* a join node and let S be the set of attributes equated in a conjunct within the join condition. Keep expanding by adding neighboring nodes so that Criterion 2.1 is satisfied with respect to S. Do not expand from any of the base relation(s). When you can not expand further, choose an aggregate or join node that either has not been selected so far, or is a sink node for some already defined group query component. Expand from this node like before, but do not expand from nodes that belong to other group query components or base relation(s). Repeat this, until there are no more join or aggregate nodes to choose from.

Different node selections at choice points give different decompositions. Applying Algorithm 5.2 for all possible choices, we get all possible decompositions into group query components with at least one maximal group query component.

Example 5.3: Figures 5(c), (d), and (e) show the decompositions given by Algorithm 5.2 for all possible node choices for the query graph of Figure 5(a).

6 Related Work

Query processing. A large body of work exists on query optimization in databases. Graefe surveys various principles and techniques [Gra93]. The issues of aggregation and join have been studied separately until quite recently, when a number of papers on optimization of *both* aggregation and join have appeared [YL94, YL95, CS94, CS96, GHQ95]. Yan and Larson in [YL94, YL95] describe a class of transformations that allow the query optimizer to push a group-by past a join (eager aggregation) or pull a group-by above a join (lazy aggregation). In a similar direction, Chaudhuri and Shim in [CS94, CS96] present a similar class of pull-up and push-down transformations. Furthermore, they incorporate these transformations in optimizers and propose a cost-based optimization algorithm to pick a plan. In [GHQ95], Gupta, Harinarayan and Quass try to unify these transformations, viewing aggregation as an extension of duplicate-eliminating projection. Our approach is quite

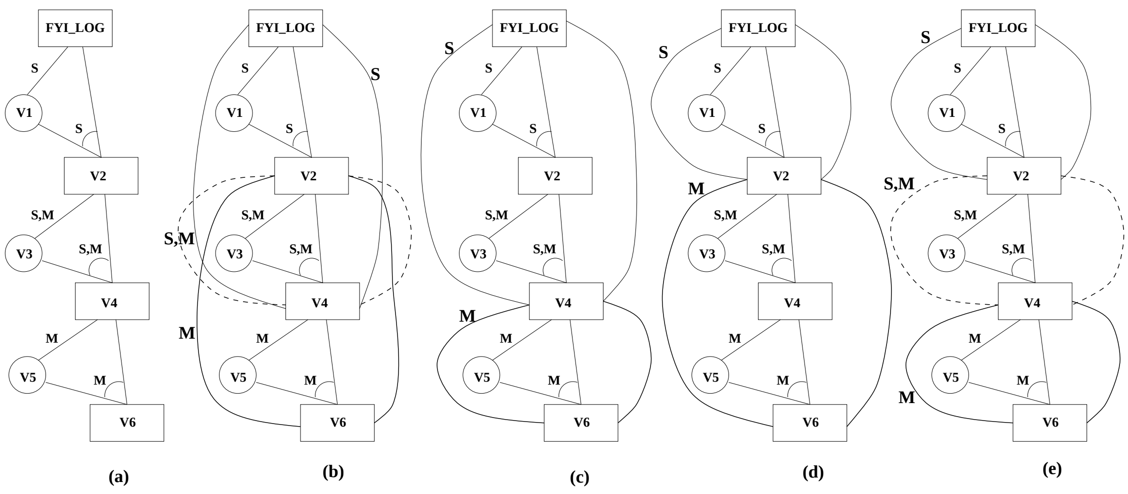


Figure 5: Output of Algorithms 5.1 and 5.2

different from these in that we try to optimize a query by partitioning the query (or components of a query) rather than by changing the order of operations.

Similarities exist in the context of distributed query processing and horizontally partitioned relations [SA80, ESW78, Day83]. In these papers, one of the key ideas for query optimization is that selection, projection and join distribute over union, a key idea also in our work.

Kim in [Kim82], Dayal in [Day87], Seshadri in [SPL96] propose decorrelation techniques, where an SQL correlated query is transformed to a query that is a join of base tables and one or more aggregate views. When rewritten using views, the attributes used for equality correlations tend to become partitioning attributes in our sense. Thus our techniques can have impact on optimizing correlated subqueries.

Although our method is presented in the context of a single-processor database system, parallelism can be exploited by partitioning the base relations on the partitioning attributes and then applying the partition partial query on each partition simultaneously. This is similar to *intraoperator* parallelism [Gra93, DG92]. However it is different in the sense that an entire complex query may be executed in parallel, rather than a single operation. While it has always been clear that multiple operators could be parallelized together [DG92], our work is novel in two important respects: Firstly, we have given a criterion (Criterion 2.1) that allows the expression of *every* group query on a given set of partitioning attributes. Thus, we have answered the problem of *how much* partitioning parallelism is possible for an arbitrary relational query in a relational system. Secondly, our partitioning algorithm is designed to achieve benefits beyond the linear speedup associated with partitioning a process among multiple CPUs. We have argued (and illustrated on a commercial system) that partitioning very complex queries makes the partitioned queries simpler and easier to optimize. Partitions of the input relation(s) can often fit in memory leading to reduced I/O costs. This strategy can improve query performance by more than a linear factor, whether one uses a parallel database system or a single-processor system.

Shatdal and Naughton have described a performance comparison of various algorithms for performing aggregation in the presence of data partitioning [SN95]. While we have not considered data partitioning here, our partitioned queries would obviously benefit in a parallel system if the data were partitioned to CPUs according to the partitioning attributes. A similar kind of performance evaluation for group queries rather than single aggregates would be an interesting direction for future work.

If the data is partitioned by the partitioning attributes then view maintenance algorithms can also be simplified. On addition or deletion of a tuple, only one group is affected. Therefore, a differential form of the partial query $ppq(Q)$ can be applied *only* to that group, where Q is the query defining the materialized view.

Extended syntaxes. A number of papers and systems propose extensions of standard SQL in order to more easily express and optimize complex queries [CR96, KS95, RBVG96, Syb94]. Each of these proposals makes the argument that with an extended syntax, complex queries are easier to write and easier to optimize. We did not consider an extended syntax here, because (a) we wanted our work to apply to present-day systems, (b) we wanted to see how much optimization potential there was for queries without utilizing the hints inherent in a special syntax, and (c) we did not want to restrict the class of queries we considered in any way. Our techniques apply to all relational queries, including those expressed in a special syntax.

Decision support. Very recently, a number of papers appeared on data models for decision support queries [AGS97, LW96]. In these papers databases are treated as multidimensional constructs and operators such as push (similar to grouping with aggregation) and join are proposed. We believe that Section 5 is in the same spirit as these papers and gives a framework for optimization of complex decision support queries, using the notion of group queries.

7 Conclusions

We defined the class of *group queries*, a class of relational queries that can, in principle, be evaluated by evaluating a simpler query on each partition of the data. This class includes a number of complex queries that are typical in decision support applications. We have provided a syntactic criterion that we show is a sufficient condition for a query to be a group query. We also show that *every* group query can be written in a form that satisfies our criterion. We have provided an execution plan for evaluating group queries that (a) partitions the relations, (b) applies a significantly simpler query to each partition, and (c) unions the results. Because the query applied to each partition has simpler join conditions and grouping attributes, it is often easier to optimize. We have presented some performance results using a commercial database system that indicate that large potential performance improvements are possible. We have described techniques for identifying components of queries that are group queries, and have outlined evaluation algorithms for nested group queries.

There are several issues that remain open and we would like to investigate them further in future work. Probably the most interesting and challenging questions lie in the area of decision support query optimization. How can one optimize the evaluation of a complex query graph, with more than one relation and multiple group query components? A cost-based approach seems imperative. What will constitute the costs and how will they be calculated? As we have seen, CPU cost plays a significant role.

References

[AGS97] Rakesh Agrawal, Ashish Gupta, and Sunita Sarawagi. Modeling multidimensional databases. In *IEEE International Conf. on Data Engineering*, 1997.

[Cha97] Damianos Chatziantoniou. *Optimization of Complex Aggregate Queries in Relational Databases.* PhD thesis, Department of Computer Science, Columbia University, 1997.

[Syb94] Sybase Corporation. *Sybase SQL Server, Reference manual, Vol. 1.* Sybase, Inc, 1994.

[CR96] Damianos Chatziantoniou and Kenneth Ross. Querying multiple features of groups in relational databases. In *22nd VLDB Conference*, pages 295–306, 1996.

[CS94] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *VLDB Conference*, pages 354–366, 1994.

[CS96] Surajit Chaudhuri and Kyuseok Shim. Optimizing queries with aggregate views. In *Extending Database Technology*, pages 167–182, 1996.

[Day83] Umeshwar Dayal. Processing queries over generalization hierarchies in a multidatabase system. In *Proceedings of the 9th VLDB Conference*, pages 342–353, 1983.

[Day87] Umeshwar Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the 13th VLDB Conference*, pages 197–208, 1987.

[DG92] David DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[DiP69] R. A. DiPaola. The recursive unsolvability of the decision problem for a class of definitite formulas. *Journal of ACM*, 16(2):324–327, 1969.

[ESW78] Robert Epstein, Michael Stonebraker, and Eugene Wong. Distributed query processing in a relational data base system. In *ACM SIGMOD, Conference on Management of Data*, pages 169–178, 1978.

[GHQ95] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-query processing in data warehousing environments. In *VLDB Conf.*, pages 358–369, 1995.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[Kim82] Won Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, 7(3):443–469, 1982.

[KS95] Ralph Kimball and Kevin Strehlo. Why decision support fails and how to fix it. *SIGMOD RECORD*, 24(3):92–97, 1995.

[LW96] Chang Li and Sean W. Wang. A data model for supporting on-line analytical processing. In *to appear in International Conference on Information and Knowledge Management*, pages 81–88, 1996.

[RBVG96] Sudhir Rao, Antonio Badia, and Dirk Van Gucht. Providing better support for a class of decision support queries. In *ACM SIGMOD, Conference on Management of Data*, pages 217–227, 1996.

[SA80] Patricia Selinger and Michel Adiba. Access path selection in distributed database management systems. In *International Conference on Databases*, pages 204–215, 1980.

[Sel88] Timos Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[SN95] Ambuj Shatdal and Jeffrey F. Naughton. Adaptive parallel aggregation algorithms. In *ACM SIGMOD, Conference on Management of Data*, pages 104–114, 1995.

[SPL96] Praveen Seshadri, Hamid Pirahesh, and T.Y. Cliff Leung. Complex query decorrelation. In *International Conference of Data Engineering*, pages 450–458, 1996.

[YL94] Weipeng P. Yan and Per-Ake Larson. Performing Group-By before Join. In *IEEE International Conference on Data Engineering*, pages 89–100, 1994.

[YL95] Weipeng P. Yan and Per-Ake Larson. Eager aggregation and lazy aggregation. In *VLDB Conference*, pages 345–357, 1995.