Efficient OLAP Query Processing in Distributed Data Warehouses

Michael O. Akinde[1], Michael H. Böhlen[1], Theodore Johnson[2], Laks V.S. Lakshmanan[3], and Divesh Srivastava[2]

[1] Aalborg University
{strategy, boehlen}@cs.auc.dk
[2] AT&T Labs–Research
{johnsont, divesh}@research.att.com
[3] University of British Columbia
laks@cs.ubc.ca

Abstract. The success of Internet applications has led to an explosive growth in the demand for bandwidth from ISPs. Managing an IP network requires collecting and analyzing network data, such as flow-level traffic statistics. Such analyses can typically be expressed as OLAP queries, e.g., correlated aggregate queries and data cubes. Current day OLAP tools for this task assume the availability of the data in a centralized data warehouse. However, the inherently distributed nature of data collection and the huge amount of data extracted at each collection point make it impractical to gather all data at a centralized site. One solution is to maintain a distributed data warehouse, consisting of local data warehouses at each collection point and a coordinator site, with most of the processing being performed at the local sites. In this paper, we consider the problem of efficient evaluation of OLAP queries over a distributed data warehouse. We have developed the Skalla system for this task. Skalla translates OLAP queries, specified as certain algebraic expressions, into distributed evaluation plans which are shipped to individual sites. Salient properties of our approach are that only partial results are shipped – never parts of the detail data. We propose a variety of optimizations to minimize both the synchronization traffic and the local processing done at each site. We finally present an experimental study based on TPC(R) data. Our results demonstrate the scalability of our techniques and quantify the performance benefits of the optimization techniques that have gone into the Skalla system.

1 Introduction

The success of Internet applications has led to an explosive growth in the demand for bandwidth from Internet Service Providers. Managing an IP (Internet Protocol) network involves debugging performance problems, optimizing the configuration of routing protocols, and planning the rollout of new capacity, to name a few tasks, especially in the face of varying traffic on the network. Effective management of a network requires collecting, correlating, and analyzing a variety of network trace data.

Typically, trace data such as packet headers, flow-level traffic statistics, and router statistics are collected using tools like packet sniffers, NetFlow-enabled routers, and SNMP polling of network elements. A wide variety of analyses are then performed to characterize the usage and behavior of the network (see, e.g., [5, 10]). For example, using flow-level traffic statistics data one can answer questions like: "*On an hourly basis, what fraction of the total number of flows is due to Web traffic?*", or "*On an hourly basis, what fraction of the total traffic flowing into the network is from IP subnets whose total hourly traffic is within 10% of the maximum?*" Currently, such analyses are usually implemented by the networking community in an ad hoc manner using complex algorithms coded in procedural programming languages like Perl. They can actually be expressed as OLAP queries, including SQL grouping/aggregation, data cubes [12], using marginal distributions extracted by the `unpivot` operator [11], and multi-feature queries [18]. Indeed, leveraging such a well-developed technology can greatly facilitate and speed up network data analysis.

A serious impediment to the use of current-day OLAP tools for analyzing network trace data is that the tools require all the data to be available in a single, centralized data warehouse. The inherently distributed nature of data collection (e.g., flow-level statistics are gathered at network routers, spread throughout the network) and the huge amount of data extracted at each collection point (of the order of several gigabytes per day for large IP networks), make it impractical to gather all this data at a single centralized data warehouse: for example, Feldmann et al. [10] report that use of a single centralized collection server for NetFlow data resulted in a loss of up to 90% of NetFlow tuples during heavy load periods!

The natural solution to this problem is to maintain a *distributed data warehouse*, where data gathered at each collection point (e.g., router) is maintained at a local data warehouse, adjacent to the collection point, to avoid loss of collected trace data. For such a solution to work, we need a technology for *distributed processing of complex OLAP queries* — something that does not yet exist. The goal of this paper is to take the first steps in this important direction.

1.1 Outline and Contributions

The rest of this paper is organized as follows. We first present related work in Sect. 1.2. In Sect. 2, we describe a motivating application and define the GMDJ operator for expressing OLAP queries. Our technical contributions are as follows:

- We present a general strategy for the distributed evaluation of OLAP queries, specified as GMDJ expressions, and present the Skalla system, developed by us for this task (Sect. 3).
- We develop and define optimization strategies for distributed OLAP that can exploit distribution knowledge, if known, as well as strategies that do not assume any such knowledge, to minimize both the synchronization traffic, and the local processing done at each site (Sect. 4).
- We conducted a series of experiments, based on TPC(R) data, to study the performance of the Skalla approach. Our results show the effectiveness of

our strategies for distributed OLAP query processing, and also quantify the performance benefits of our optimizations. This demonstrates the validity of the Skalla approach (Sect. 5).

While we illustrate our techniques using examples drawn from the network management application, our approach and results are more generally applicable to *distributed* data warehouses and OLAP query processing in other application domains as well (e.g., with heterogenous data marts distributed across an enterprise). To the best of our knowledge, ours is the first paper on this important topic.

1.2 Related Work

The most closely related prior work is that of Shatdal and Naughton [19], who use a similar coordinator/sites model for the parallel evaluation of aggregates, and present various strategies where the aggregate computation is split between the sites and the coordinator, to optimize performance. Aggregates are also considered in a number of parallel database systems, such as in [3, 4]. There are two main differences with our work. First, their results are tuned for a parallel computer, where communication is assumed to be very cheap, which is certainly not the case in our distributed data warehouse setting. Second, they deal only with the case of simple SQL aggregates, while we consider significantly more complex OLAP queries.

A variety of OLAP queries have been proposed in the literature, allowing a fine degree of control over both the group definition and the aggregates that are computed using operators such as `cube by` [12], `unpivot` [11], and other SQL extensions (e.g., [6]). Recently, Chatziantoniou et al. [7] proposed the MDJ operator for complex OLAP queries, which provides a clean separation between group definition and aggregate computation, allowing considerable flexibility in the expression of OLAP queries. The processing and optimization of these complex OLAP queries has received a great deal of attention in recent years (see, e.g., [1, 2, 7, 8, 12, 17, 18]), but it has all been in the context of a single centralized data warehouse. Our results form the basis for extending these techniques to the distributed case.

A considerable body of research has been performed for processing and optimizing queries over distributed data (see, e.g., [15, 16]). However, this research has focused on distributed join processing rather than distributed aggregate computation. The approach we explore in this paper uses an extended aggregation operator to express complex aggregation queries. Some of the distributed evaluation optimizations that we have developed resemble previously proposed optimizations, e.g. exploiting data distributions [15] or local reduction [20]. However, our architecture for evaluating distributed aggregation queries allows for novel optimizations not exploited by conventional algorithms for distributed processing.

2 Preliminaries

In this section, we give an example of an application area that motivates the use of distributed data warehouse techniques. We then define the GMDJ operator and demonstrate how the GMDJ operator allows us to uniformly express a variety of OLAP queries.

2.1 Motivating Example

Analysis of IP flow data is a compelling application that can tremendously benefit from distributed data warehouse technology. An IP flow is a sequence of packets transferred from a given source to a given destination (identified by an IP address, Port, and Autonomous system), using a given `Protocol`. All packets in a flow pass through a given router, which maintains summary statistics about the flow and which dumps out a tuple for each flow passing through it.

Data warehouses are typically modelled using, e.g., *star schemas* or *snowflake schemas* [14, 8]. Our techniques are oblivious to which of these data warehouse models are used for conceptually modeling the data, and our results would hold in either model. For simplicity, in our examples the table `Flow` is a denormalized fact relation, with `NumPackets` and `NumBytes` as the measure attributes and with the following schema:

```
Flow ( RouterId, SourceIP, SourcePort, SourceMask, SourceAS, DestIP,
       DestPort, DestMask, DestAS, StartTime, EndTime, NumPackets,
       NumBytes )
```

We assume that flow tuples generated by a router are stored in a local data warehouse "adjacent" to the router, i.e., `RouterId` is a partition attribute. Each local warehouse is assumed to be fully capable of evaluating any complex OLAP query. We refer to this collection of local data warehouses (or sites), along with a coordinator site that correlates subquery results, as a *distributed data warehouse.*

2.2 GMDJ Expressions

The *GMDJ* operator is an OLAP operator that allows for the algebraic expression of many complex OLAP queries [2].

Let θ be a condition, b be a tuple, and R be a relation. We write $attr(\theta)$ to denote the set of attributes used in θ. $RNG(b, R, \theta) =_{\text{def}} \{r \mid r \in R \wedge \theta(b, r)\}$ denotes the set of tuples in R that satisfies θ, with respect to the tuple b. E.g., $RNG(b, R, b.A = R.B)$ denotes those tuples in R whose B-value matches the A-value of b. We use $\{\{\ldots\}\}$ to denote a multiset.

Definition 1. *Let $B(\mathbf{B})$ and $R(\mathbf{R})$ be relations, θ_i a condition with $attr(\theta_i) \subseteq \mathbf{B} \cup \mathbf{R}$, and l_i be a list of aggregate functions $(f_{i1}, f_{i2}, \ldots, f_{in_i})$ over attributes*

$c_{i1}, c_{i2}, \ldots, c_{in_i}$ *in* $\mathbf{R}$*. The GMDJ,* $MD(B, R, (l_1, \ldots, l_m), (\theta_1, \ldots, \theta_m))$*, is a relation with schema*[1]

$$\mathbf{X} = (\mathbf{B}, f_{11}\_R\_c_{11}, \ldots, f_{1n}\_R\_c_{1n_1}, \ldots, f_{m1}\_R\_c_{m1}, \ldots, f_{mn}\_R\_c_{mn_m}),$$

whose instance is determined as follows. Each tuple $b \in B$ *contributes to an output tuple* $\mathbf{x}$*, such that:*

- $\mathbf{x}[A] = b[A]$*, for every attribute* $A \in \mathbf{B}$
- $\mathbf{x}[f_{ij}\_R\_c_{ij}] = f_{ij}\{\{t[c_{ij}] \mid t \in RNG(b, R, \theta_i)\}\}$*, for every attribute* $f_{ij}\_R\_c_{ij}$ *of* $\mathbf{x}$*.*

We call B *the* base-values relation *and* R *the* detail relation*.*

Usually, one can determine a subset K *of* key attributes *of the base-values relation* B *for each* θ_i*, which uniquely determine a tuple in* B *(*K *can be* $\mathbf{B}$*). We make use of key attributes in several of our strategies.*

It should be noted that conventional SQL groupwise and hash-based aggregation techniques cannot be directly applied to GMDJ expressions, since the set of tuples in the detail relation R that satisfy condition θ with respect to tuples b_1 and b_2 of the base-values relation, i.e., $RNG(b_1, R, \theta)$ and $RNG(b_2, R, \theta)$, might not be disjoint. However, see [2, 7] for a discussion of how GMDJ expressions can be evaluated efficiently in a centralized system.

A GMDJ operator can be composed with other relational algebra operators (and other GMDJs) to create complex GMDJ expressions. While arbitrary expressions are possible, it is often the case that the result of a GMDJ expression serves as the base-values relation for another GMDJ operator. This is because the result of the GMDJ expression has exactly as many tuples as there are in the base-values relation B. In the rest of this paper, when we refer to (complex) GMDJ expressions, we mean only expressions where the result of an (inner) GMDJ is used as a base-values relation for an (outer) GMDJ.

Example 1. Given our IP Flows application, an interesting OLAP query might be to ask for the total number of flows, and the number of flows whose `NumBytes (NB)` value exceeds the average value of `NumBytes`, for each combination of source and destination autonomous system (e.g., to identify special traffic). This query is computed by the complex GMDJ expression given below.

$$\begin{array}{l} MD(\quad MD(\pi_{\texttt{SAS,DAS}}(\texttt{Flow}) \rightarrow B_0, \\ \qquad\qquad \texttt{Flow} \rightarrow F_0, \\ \qquad\qquad ((cnt(*) \rightarrow cnt1, sum(NB) \rightarrow sum1)), \\ \qquad\qquad (\texttt{F}_0\texttt{.SAS} = \texttt{B}_0\texttt{.SAS} \ \&\ \texttt{F}_0\texttt{.DAS} = \texttt{B}_0\texttt{.DAS}) \\ \qquad\qquad) \rightarrow B_1, \\ \qquad\quad \texttt{Flow} \rightarrow F_1, \\ \qquad\quad ((cnt(*) \rightarrow cnt2)), \\ \qquad\quad (\texttt{F}_1\texttt{.SAS} = \texttt{B}_1\texttt{.SAS} \ \&\ \texttt{F}_1\texttt{.DAS} = \texttt{B}_1\texttt{.DAS} \ \&\ \texttt{F}_1\texttt{.NB} \geq \texttt{sum1/cnt1}) \\ ) \end{array}$$

[1] Attributes are appropriately renamed if there are any duplicate names generated this way. We note that the renaming scheme employed in the examples will use a shorthand form.

The flow data may or may not be clustered on `SourceAS` or `DestAS`. If it is not, the distributed evaluation of this query requires correlating aggregate data at multiple sites, and alternating evaluation (in multiple passes) at the sites and at the coordinator. We develop efficient evaluation strategies for both cases.

We call queries such as in Example 1 *correlated aggregate queries* since they involve computing aggregates w.r.t. a specified grouping and then computing further values (which may be aggregates) based on the previously computed aggregates. In general, there may be a chain of dependencies among the various (aggregate or otherwise) attributes computed by such a query. In Example 1, the length of this chain is two. Several examples involving correlated aggregates can be found in previous OLAP literature [7, 6, 11, 18].

It is possible to construct many different kinds of OLAP queries and identifying distributed evaluation strategies for each would be quite tedious. The GMDJ operator provides a clean separation between the definition of the groups and the definition of aggregates in an OLAP query. This allows us to express a significant variety of OLAP queries (enabled by disparate SQL extensions) in a uniform algebraic manner [2, 7]. Thus, it suffices to consider the distributed evaluation of GMDJ expressions, to capture most of the OLAP queries proposed in the literature.

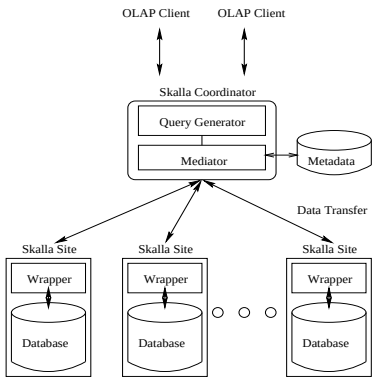
3 Distributed GMDJ Evaluation

In this section, we will describe the distributed GMDJ evaluation algorithm implemented in the Skalla prototype. We defer the presentation of query optimizations of the core Skalla evaluation algorithm until the next section.

3.1 Skalla: An Overview

The Skalla system for distributed data warehousing is based on a coordinator architecture (i.e., strict client-server) as depicted in Fig. 1. It consists of multiple local data warehouses (Skalla sites) adjacent to data collection points, together with the Skalla coordinator (we note that the coordinator can be a single instance as in Fig. 1 or may consist of multiple instances, e.g., each client may have its own coordinator instance). Conceptually, the fact relation of our data warehouse is the *union* of the tuples captured at each data collection point. However, users can pose OLAP queries against the conceptual data model of our distributed data warehouse, without regard to the location of individual tuples at the various sites.

We define a *distributed evaluation plan* (*plan* for short) for our coordinator architecture as a sequence of rounds, where a round consists of: (i) each skalla site performing some computation and communicating the results to the coordinator, and (ii) the coordinator synchronizing the local results into a global result, and (possibly) communicating the global result back to the sites. Thus, the overall cost of a plan (in terms of response time) has many components:




```
Algorithm GMDJDistribEval {
   construct empty X
   transfer X and B_0 to each site S_i
   at each site S_i ∈ S_B
      compute X_0 at the local sites
      transfer local X_0 to coordinator
   synchronize X_0
   for each GMDJ MD_k (k = 1 to m) {
      transfer X_{k-1} to each S_i ∈ S_{MD_k}
      at each S_i ∈ S_{MD_k}
         compute MD_k(X_{k-1}, R_k, l_k, θ_k)
         transfer X_k to coordinator
      synchronize X_k
   }
}
```

Fig. 1. Skalla architecture (left) and evaluation algorithm (right)

(i) communication (or, synchronization traffic), and (ii) computation (or, the query processing effort at the local sites as well as the coordinator).

The query generator of the Skalla system constructs query plans from the OLAP queries, which are then passed on to and executed by the mediator, using Alg. `GMDJDistribEval`.

3.2 Algorithm Description

We will now describe how Skalla works when receiving an OLAP query, using Example 1 to illustrate the Skalla evaluation.

First, the Skalla query engine uses *Egil*, a GMDJ query optimizer, to translate the OLAP query into GMDJ expressions. These GMDJ expressions are then optimized for distributed computation. We note that even simple GMDJ expressions can involve aggregation and multiple self-joins, which would be hard for a conventional centralized — let alone a distributed — query optimizer to handle. We defer a study of these optimizations to Sect. 4.

Alg. `GMDJDistribEval` gives an overview of the basic query evaluation strategy of Skalla for complex GMDJ expressions. Given the GMDJ query of Example 1, the coordinator will construct the empty *base-result structure* $X(\mathbf{X})$ with the schema:

$$\mathbf{X} = (\texttt{SourceAS}, \texttt{DestAS}, \texttt{count\_md1}, \texttt{sum\_md1\_numbytes}, \texttt{count\_md2})$$

The query $B_0 = \pi_{\texttt{SourceAS,DestAS}}(\texttt{Flow})$ is then shipped to each of the sites, executed locally, and the result shipped back to the coordinator. During evaluation, the coordinator maintains the base-result structure by synchronization of the sub-results that it receives.

The term *synchronization* as used in this paper refers to the process of consolidating the results processed at the local sites into the base-results structure

X. We refer to each local-processing-then-synchronization step as a *round* of processing. An evaluation of a GMDJ expression involving m GMDJ operators uses $m+1$ rounds. The notation X_k in Alg. `GMDJDistribEval` refers to the instance of X after the computation of the kth GMDJ, where we assume there are m rounds in all, m depending on the dependency chain in the original OLAP query. Aggregates of X are computed from the local aggregate values computed at the sites as appropriate. For example, to compute `count_md1`, we would need to compute the sum of the `COUNT(*)`s computed at the local sites. Following Gray et al. [12], we call the aggregates computed at the local sites the *sub-aggregates* and the aggregate computed at the coordinator the *super-aggregate*. $\mathcal{S}_B$ is the set of all local sites, while $\mathcal{S}_{MD_k}$ is set of local sites which participate in the kth round.[2] We use R_k to denote the detail relation at round k.[3] Again, depending on the query, the detail relation may or may not be the same across all rounds. This shows the considerable class of OLAP queries the basic Skalla evaluation framework is able to handle. In this paper, we give only examples where the detail relation does not change over rounds. Finally, l_k is the list of aggregate functions to be evaluated at round k and θ_k is the corresponding condition (see Definition 1).

The following theorem establishes the basis of the synchronization in Skalla:

Theorem 1. *Let $X = MD(B, R, (l_1, \ldots, l_m), (\theta_1, \ldots, \theta_m))$, where B has key attributes K. Let $R_1, \ldots, R_n$ be a partition of R. Let l'_j and l''_j be the lists of sub-aggregates and super-aggregates, respectively, corresponding to the aggregates in l_j. Let $H_i = MD(B, R_i, (l'_1, \ldots, l'_m), (\theta_1, \ldots, \theta_m))$ for $i = 1, \ldots, n$. Let $H = H_1 \sqcup \cdots \sqcup H_n$, where $\sqcup$ indicates multiset union. Then $X = MD(B, H, (l''_1, \ldots, l''_m), \theta_K)$ where θ_K is a test for equality on the attributes in K.*

In practice, the scheme given by Theorem 1 is executed very efficiently. The base-results structure maintained at the coordinator is indexed on K, which allows us to efficiently determine $RNG(X, t, \theta_K)$ for any tuple t in H and then update the structure accordingly; i.e., the synchronization can be computed in $\mathcal{O}(|H|)$. Since the GMDJ can be horizontally partitioned, the coordinator can synchronize H with those sub-results it has already received while receiving blocks of H from slower sites, rather than having to wait for all of H to be assembled before performing the synchronization. Between each computation at the local sites and synchronization, we ship the base-results structure (or fragments thereof) between the sites and the coordinator.

The following result bounds the maximum amount of data transferred during the evaluation of Alg. `GMDJDistribEval` for distributive aggregate queries.

Theorem 2. *Let the distributed data warehouse contain n sites, and the size of the result of query Q, expressed as a GMDJ expression with m GMDJ operators be $|Q|$. Let s_0 denote the number of sites participating in the computation of the*

[2] Typically, $\mathcal{S}_{MD_k} = \mathcal{S}_B$, but it is possible that $\mathcal{S}_{MD_k} \subset \mathcal{S}_B$.

[3] Note that every site refers to its local detail relation as R_k at round k. To avoid clutter, we preferred this notation to something like R^i_k.

base values relation and s_i the number of sites participating in the computation of the i'th GMDJ operator. Then the maximum amount of data transferred during the evaluation of Alg. `GMDJDistribEval` *on Q is bounded by*

$$\left(\sum_{i=1}^{n}(2 * s_i * |Q|\right) + (s_0 * |Q|).$$

Recall that Alg. `GMDJDistribEval` only ships the base-result structure X_k. Since $X_k \subseteq Q$, it follows that the maximum size of any X_k is $|Q|$. The significance of Theorem 2 is that it provides a bound on the maximum amount of data transferred as a function of the size of the query result, the size of the query (i.e. number of rounds), and the number of local sites in the distributed data warehouse, which is *independent of the size of the fact relation in the database.* $|Q|$ depends only on the size of the base values relation and the aggregates to be computed, not the detail relation. This is significant in that such a bound does not hold for the distributed processing of traditional SQL join queries (see, for example, [15, 16]), where intermediate results can be arbitrarily larger than the final query result, even when using semijoin-style optimizations.

3.3 Summary

Let B_0 be the base-values relation, $R_1, R_2, \ldots, R_m$ be detail relations, $l_1, ..., l_m$ be lists of aggregate functions, and $\theta_1, \ldots, \theta_m$ be lists of conditions.[4] Let B_k denote the result of the GMDJ MD_k. Let $\mathcal{S}_B$ be the set of sites required for the computation of B_0, and let $\mathcal{S}_{MD_i}$ be the set of sites required for the computation of the GMDJ MD_i. Alg. `GMDJDistribEval` is a simple and efficient algorithm for distributed OLAP processing that does not transfer any detailed data between the sites.

Theorem 3. *Given a GMDJ expression $Q = MD_n(\cdots(MD_1(B_0, R_1, (l_{11}, \ldots, l_{1k}), (\theta_{11}, \ldots, \theta_{1k}))\cdots), R_n, (l_{n1}, \ldots, l_{nm}), (\theta_{n1}, \ldots, \theta_{nm}))$, the set of sites $\mathcal{S}_B$ required for computing B_0, and the sets of sites $\mathcal{S}_{MD_i}, i \leq n$, required for computing GMDJ MD_i, then Alg.* `GMDJDistribEval` *correctly computes the result of Q.*

While Alg. `GMDJDistribEval` is efficient in relation to the size of the detail relations, the amount of data transfer and the computation can still be objectionably large for very large distributed data warehouses and thus these resources need to be optimized substantially. This is the subject of Sect. 4.

4 Distributed Evaluation Optimizations

Clearly, query optimization techniques used for centralized evaluation of GMDJ expressions (e.g., indexing, coalescing), which have been previously studied [2, 7] apply in an orthogonal way to their distributed processing. Classical distributed

[4] l_i is a list of aggregates and θ_i is a list of grouping conditions for each of them.

query optimization techniques developed for SQL queries such as row blocking, optimization of multi-casts, horizontal partitioning of data, or semijoin programs [15] apply to the distributed processing of GMDJs. We do not dwell on these any further. In Sect. 4.1 we generalize local reductions to GMDJ expressions. Sections 4.2 and 4.3 describe optimizations specific to distributed GMDJ query processing.

It should be stressed that in each case, we present what are best characterized as optimization schemes. Depending on the specific situation (i.e. the amount of distribution knowledge available), one may be able to come up with specific optimization strategies, which are instances of these optimization schemes. In this sense, our optimization schemes can be used for determining whether a strategy is correct in the sense that it computes the right result. Throughout this section, we give examples which illustrate this point.

4.1 Distribution-Aware Group Reduction

Recall the definition of the function, $RNG(b, R, \theta) = \{r \in R \mid \theta(b, r) \text{ is true}\}$. Given information about the partitioning of tuples in the distributed data warehouse, we may be able reduce the size of the base-results structure transferred between the sites and the coordinator. The following theorem provides the basis for this optimization.

Theorem 4. *Consider the GMDJ expression* $Q = MD(B, R, (l_1, \ldots, l_m), (\theta_1, \ldots, \theta_m))$. *Let* $R_1 \cup \cdots \cup R_n$ *be a partitioning of the detail relation* R. *For each* R_i, *let* ϕ_i *be a predicate such that for each* $r \in R_i$, $\phi_i(r)$ *is true. Let* $\psi_i(b)$ *be the formula* $\forall_r \phi_i(r) \Rightarrow \neg(\theta_1 \vee \ldots \vee \theta_m)(b, r)$. *Let* RNG_i *be* $RNG(b, R_i, \theta_1 \vee \ldots \vee \theta_m)$. *Then, we have:*

$$\sigma_{|RNG_i|>0}(MD(B, R_i, (l_1, \ldots, l_m), (\theta_1, \ldots, \theta_m))) = \sigma_{|RNG_i|>0}(MD(\sigma_{\neg\psi_i}(B), R_i, (l_1, \ldots, l_m), (\theta_1, \ldots, \theta_m)))$$

Using Alg. `GMDJDistribEval`, a local site will compute $H_i = MD(B, R_i, (l'_1, \ldots, l'_m), (\theta_1, \ldots, \theta_m))$. Let $\overline{B_i} = \{b \in B \mid \psi_i(b)\}$. Theorem 4 states that if we have a-priori knowledge about whether $RNG(b, R_i, \theta)$ is empty for any given b, we need to send to site S_i only $B - \overline{B_i}$. Given knowledge about the data distribution at the individual sites, group reductions can be performed by restricting B using the $\neg\psi_i$ condition.

Example 2. Assume that each of the packets for a specific `SourceAS` passes through a router with a specific `RouterId`. For example, site S_1 handles all and only autonomous systems with `SourceAS` between 1 and 25. The condition θ in the query of Example 1 contains the condition `Flow.SourceAS = B.SourceAS`. We can deduce that at S_1, $\psi_i(b)$ is true when b.`SourceAS` $\notin [1, 25]$. Therefore, $\neg\psi_i(b)$ is the condition b.`SourceAS` $\in [1, 25]$.

Example 2 gives a simple example of the kind of optimization possible using distribution-aware group reductions. The analysis is easy to perform if ψ_i and θ are conjunctive and the atoms of the predicates involve tests for equality.

In fact, far more complex constraints can be handled. For example, assume the condition θ in example 2 is revised to be $B.DestAS + B.SourceAS < Flow.SourceAS*2$. Then condition $\neg\psi_i(b)$ becomes $B.DestAS+B.SourceAS < 50$. The significance of Theorem 4 is that we can use it to determine the correctness of the optimizer.

Other uses of Theorem 4 are also possible. For example, `SourceAS` might not be partitioned among the sites, but any given value of `SourceAS` might occur in the `Flow` relation at only a few sites. Even in such cases, we would be able to further reduce the number of groups sent to the sites.

4.2 Distribution-Independent Group Reduction

A significant feature of the GMDJ processing, compared to traditional distributed algorithms, is the possibility of performing distribution-independent group reduction.

We extend Theorem 1 for distribution-independent group reduction:

Proposition 1. *Consider the GMDJ* $Q = MD(B, R, (l_1, \ldots, l_m), (\theta_1, \ldots, \theta_m))$ *where* B *has key attributes* K*. Let* $R_1, \ldots, R_n$ *be a partition of* R*. Let* l'_i *and* l''_i *be the lists of sub-aggregates and super-aggregates, respectively, corresponding to the aggregates in* l_i*. Then:* $MD(B, R, (l_1, \ldots, l_m), (\theta_1, \ldots, \theta_m)) = MD(B, \sigma_{|RNG|>0}(MD(B, R_1, (l'_1, \ldots, l'_m), (\theta_1, \ldots, \theta_m))) \sqcup \cdots \sqcup \sigma_{|RNG|>0}(MD(B, R_n, (l'_1, \ldots, l'_m), (\theta_1, \ldots, \theta_m))), (l''_1, \ldots, l''_m), \theta_K)$ *where* $\sqcup$ *indicates multiset union, and* θ_K *is a test for equality on the attributes in* K*.*

Let $H_1, H_2, \ldots, H_n$ be the results of processing the GMDJ expressions at the local sites. Then Proposition 1 states that the only tuples of H_i required for synchronization of the results are those tuples t such that $|RNG(t, R_i, (\theta_1 \vee \ldots \vee \theta_m))| > 0$, as otherwise the tuple does not contribute any information to the global aggregate. A simple way of detecting $|RNG| > 0$ with respect to tuples in H_i is to compute an additional aggregate $l_{m+1} =$ `COUNT`$(*)$ on H_i such that $\theta_{m+1} = (\theta_1 \vee \ldots \vee \theta_m)$. The only overhead to this optimization then becomes the additional computing time for the extra `COUNT(*)`, and to perform the selection `COUNT(*)`> 0 at the sites.

Example 3. We return to Example 1. The result of a GMDJ computation is transmitted to the coordinator by Alg. `GMDJDistribEval`. Assuming n sites, and that the size of the GMDJ is $|B|$, we transmit $n * |B|$ data. Assuming that each site, on average, computes aggregates for $1/k$ tuples in B, then distribution-independent group reduction will reduce the amount of data transmitted by each site to $|B|/k$ and the total data transmission to $n/k * |B|$.

An advantage of distribution-independent group reduction is that it improves performance even without semantic information about the distribution of R (which might not be available).

4.3 Synchronization Reduction

Synchronization reduction is concerned with reducing data transfer between the local sites and the coordinator by reducing the rounds of computation. One of the algebraic transformations possible on GMDJ operators is to coalesce two GMDJs into a single GMDJ. More precisely:

$$MD_2(MD_1(B, R,(l_{11}, \ldots, l_{1l}), (\theta_{11}, \ldots, \theta_{1l})), R, (l_{21}, \ldots, l_{2m}), (\theta_{21}, \ldots, \theta_{2m})) = \\ MD(B, R, (l_{11}, \ldots, l_{1l}, l_{21}, \ldots, l_{2m}), (\theta_{11}, \ldots, \theta_{1l}, \theta_{21}, \ldots, \theta_{2m}))$$

if the conditions $\theta_{21}, \ldots, \theta_{2m}$ do not refer to attributes generated by MD_1 [7].

However, in many instances the OLAP query may consist of only one or two simple GMDJ expressions. In this case, the advantage of the coalescing is limited, because we may still have to synchronize the base-results structure at the coordinator after its construction. We present two results specific to the distributed query processing of GMDJs, that permit synchronization reduction.

Proposition 2. *Consider the GMDJ* $Q = MD(B, R, (l_1, \ldots, l_m), (\theta_1, \ldots, \theta_m))$. *Let* B *be the result of evaluating query* $\mathcal{B}$ *on* R, *let* $R_1, \ldots, R_n$ *be a partition of* R, *and let* B_i *be the result of evaluating query* $\mathcal{B}$ *on* R_i. *Suppose that* $B = \bigsqcup_i B_i$. *Let* B *have key attributes* K. *If* θ_j *entails* θ_K,*the test for equality on the attributes in* K, $\forall j | 1 \leq j \leq m$, *then:*

$$MD(B, R, (l_1, \ldots, l_m), (\theta_1, \ldots, \theta_m)) = MD(\pi_{\mathbf{B}} H, H, (l''_1, \ldots, l''_m), (\theta_K, \ldots, \theta_K))$$

where $H = \bigsqcup_i H_i$ *and* $H_i = MD(B_i, R_i, (l'_1, \ldots, l'_m), (\theta_1, \ldots, \theta_m))$.

Proposition 2 states that, if $\mathcal{B}$ is evaluated over the relation R and each condition tests for equality on the key attributes K, then we can omit the synchronization of the base-values relation.

Example 4. Consider again Example 1. Following Proposition 2, we can compute B_0 and the first GMDJ B_1 directly, instead of synchronizing in between the two computations as would otherwise be the case. Thus the number of synchronizations can be cut down from three to two, with a potential 40% reduction in the amount of data transferred.

Theorem 5. *Consider the GMDJ* $Q = MD_2(MD_1(B, R, (l_{11}, \ldots, l_{1l}), (\theta_{11}, \ldots, \theta_{1l})), R, (l_{21},$
$\ldots, l_{2m}), (\theta_{21}, \ldots, \theta_{2m}))$. *Let* $R_1 \cup \cdots \cup R_n$ *be a partitioning of the detail relation* R. *For each* R_i, *let* ϕ_i *be a predicate such that for each* $r \in R_i$, $\phi_i(r)$ *is true. Let* $\psi_i^1(b)$ *be the formula* $\forall_r \phi_i(r) \Rightarrow \neg(\theta_{11} \vee \ldots \vee \theta_{1l})(b, r)$, *and let* $\psi_i^2(b)$ *be the formula* $\forall_r \phi_i(r) \Rightarrow \neg(\theta_{21} \vee \ldots \vee \theta_{2m})(b, r)$. *Suppose that* $\forall_j (j \neq i) \Rightarrow (\psi_j^1(b)\ \&\ \psi_j^2(b))$. *Then site* i *does not need to synchronize tuple* b *between the evaluation of* MD_1 *and* MD_2.

Theorem 5 states that it is not necessary to synchronize a tuple $b \in B$ if we know that the only site which updates b's aggregates during MD_1 and MD_2 is site i. If we have strong information about the distribution of R among the sites, we can avoid synchronizing between the evaluation of MD_1 and MD_2 altogether.

Definition 2. *An attribute A is a* partition attribute *iff* $\forall_{i \neq j} \pi_A(\sigma_{\phi_i}(R)) \cap \pi_A(\sigma_{\phi_j}(R)) = \emptyset$

Corollary 1. *Consider the GMDJ* $Q = MD_2(MD_1(B, R, (l_{11}, \ldots, l_{1l}), (\theta_{11}, \ldots, \theta_{1l})), R, (l_{21}, \ldots, l_{2m}), (\theta_{21}, \ldots, \theta_{2m}))$*. If* $\theta_{11}, \ldots, \theta_{1l}, \theta_{21}, \ldots, \theta_{2m}$ *all entail condition* $R.A = f(A)$*, where* $f(A)$ *is a bijective function on* A*, and* A *is a partition attribute, then* MD_2 *can be computed after* MD_1 *without synchronizing between the GMDJs.*

Thus, by performing a simple analysis of ϕ_i and θ, we are able to identify a significant subset of queries where synchronization reduction is possible. We note that more than one attribute can be a partition attribute, e.g., if a partition attribute is functionally determined by another attribute.

Example 5. Consider the query of Example 1. Without any synchronization reduction, the evaluation of this GMDJ expression would require multiple passes over the `Flow` relation, and three synchronizations, one for the base-values relation and one each for the results of the two GMDJs is required.

Let us assume that all packets from any given `SourceAS` only pass through a router with a particular `RouterId`. If this is the case, `SourceAS` is a partition attribute. Using Corollary 1, the second synchronization is avoided. Further, since `(SourceAS, DestAS)` form a key, Proposition 2 is applicable as well, and no synchronization of the base-values relation is needed. As a result, the query can be evaluated against the distributed data warehouse with *the entire query being evaluated locally, and with a single synchronization at the coordinator.*

Synchronization reduction is a distinctive feature of distributed GMDJ processing, which is difficult or impossible to duplicate using traditional distributed query optimizations methods. In addition, it is a key factor in keeping the distributed processing of OLAP queries scalable, as we shall show in Sect. 5.

5 Experimental Evaluation

In this section, we describe a set of experiments to study the performance of Skalla. We show the scalability of our strategies and also quantify the performance benefits of our optimizations.

5.1 Setup and Data

We used Daytona [13] as the target DBMS for GMDJ expression evaluation in Skalla, both for the local data warehouse sites and the coordinator site. We derived a test database from the TPC(R) `dbgen` program, creating a denormalized 900 Mbyte data set with 6 million tuples (named TPCR). We partitioned the data set on the `NationKey` attribute (and therefore also on the `CustKey` attribute). The partitions were then distributed among eight sites.

In each of our test queries, we compute a `COUNT` and an `AVG` aggregate on each GMDJ operator. We ran two different experiments with different attributes of

the TPCR relation as the grouping attribute. The first set of experiments (high cardinality) use the `Customer.Name` attribute, which has 100,000 unique values partitioned among eight sites. The second set of experiments (low cardinality) uses attributes with between 2000 to 4000 unique values. For the following experiments, we examine only group reduction, synchronization reduction, and combined reductions.

5.2 Speed-up Experiments

In this section, we divide the TPCR relation equally among eight sites, and vary the number of sites participating in the evaluation of a query. We use this experimental setup to evaluate the impact of the various optimizations.

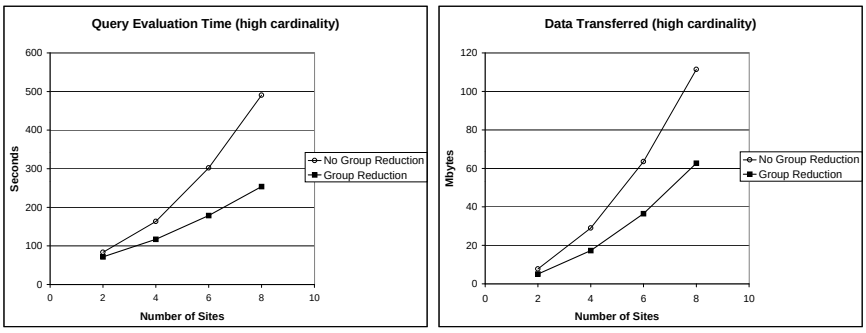


Fig. 2. Group reduction query

Figure 2 depicts graphs showing the query evaluation time (left) and the amount of data transferred for distribution-independent group-reduced and the non group-reduced versions of the group reduction query (right). The set of non group-reduced curves shows a quadratic increase in query evaluation time and in the number of bytes transferred. This behavior is due to a linearly increasing number of groups being sent to a linearly increasing number of sites; thus the communication overhead and synchronization overhead increases quadratically. When group reduction is applied, the curves are still quadratic, but to a lesser degree. The distribution-independent (i.e., site side) group reduction solves half of the inefficiency, as the sites send a linear amount of data to the coordinator, but the coordinator sends a quadratic amount of data to the sites. Distribution-aware (i.e., coordinator side) group reduction would make the curves linear.

To see this, we perform an analysis of the number of bytes transferred. Let the number of groups residing on a single site be g, the number of sites be n, and the fraction of sites' group aggregates updated during the evaluation of a grouping variable be c. In the first round, ng groups are sent from the sites to the coordinator. Without group reduction, n^2g groups are sent from the coordinator to the sites, and n^2g groups are sent back. With group reduction, only cng

groups are returned. Therefore, the proportion of groups transferred with group reduction versus without group reduction is $(ng(2c+1+2n))/(ng(4n+1)) = (2c+2n+1)/(4n+1)$. The number of bytes transferred is roughly proportional to the number of groups transferred, and in fact this formula matches the experimental results to within 5%.

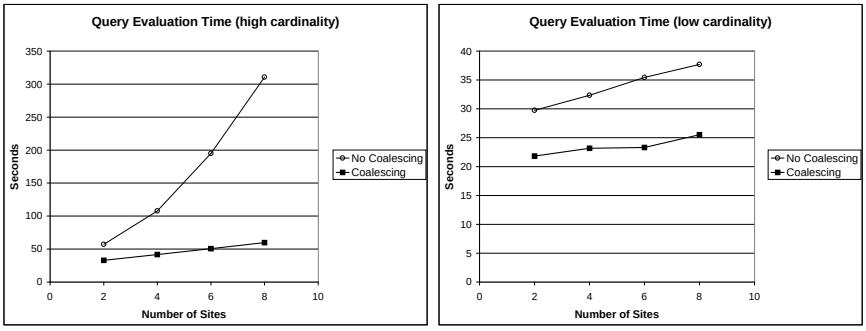


Fig. 3. Coalescing query

Figure 3 shows the evaluation time of the coalesced and non coalesced query for high cardinality (left) and low cardinality (right) queries. The non coalesced curve in the high cardinality query shows a quadratic increase in execution time. The coalesced GMDJ curve is linear. There is only one evaluation round, at the end of which the sites send their results to the coordinator, so the volume of data transferred increases linearly with the number of sites. For the low cardinality query the difference is less dramatic. Even though the amount of data transferred is small, coalescing reduces query evaluation time by 30%, primarily due to a reduction in the site computation time.

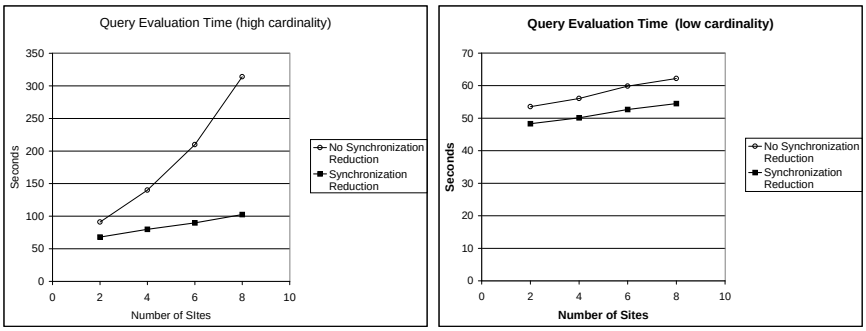


Fig. 4. Synchronization reduction query

Finally, we test the effect of synchronization reduction without coalescing. Figure 4 shows the query evaluation time of the an OLAP query evaluated with and without synchronization reduction for the high cardinality (left) and low cardinality (right) versions of the query. Without synchronization reduction in the high cardinality query, the query evaluation time is quadratic with an increasing number of sites. With synchronization reduction, the query is evaluated in a single round, and shows a linear growth in evaluation time (due to the linearly increasing size of the output). Thus, synchronization reduction removes the inefficiencies (due to attribute partitioning) seen in the previous experiments. For the low cardinality query, synchronization reduction without coalescing reduces the query evaluation time, but not to the same degree achieved with coalescing of GMDJs on the high cardinality query. This is because coalescing improves computation time as well as reducing communication; thus the work performed by the sites is nearly the same, and the difference in query evaluation time only represents the reduction in synchronization overhead.

5.3 Scale-up Experiments

In this section, we fix the number of sites at four, and vary the data set size at each of these sites. We start with the data set used in the speed-up experiments and increase its size by up to a factor of four. We used the combined reductions query, and applied either all of the reductions or none of them.

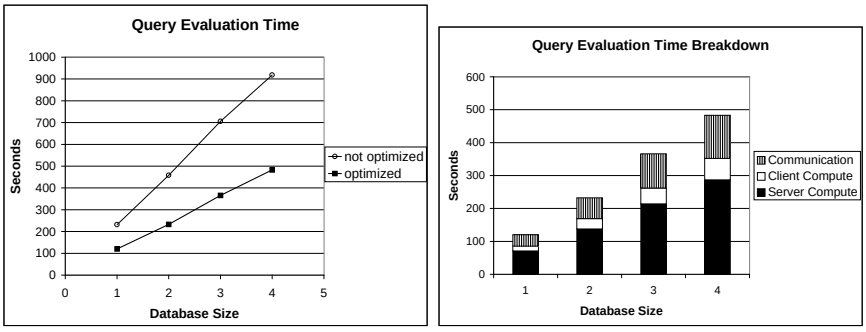


Fig. 5. Combined reductions query

In our first set of experiments, the number of groups increased linearly with the data set size. The graph in Fig. 5 (left) shows the query evaluation time when the optimizations are turned on or off. In both cases there is a linear increase in query evaluation time with increasing database size. Using the optimizations improved the query evaluation time by nearly half. The bar graph of Fig. 5 (right) breaks down the evaluation time of the optimized query into the site computation time, coordinator computation time, and communication overhead, showing linear growth in each component. We ran the same set of experiments using a

data set in which the number of groups remains constant with an increasing database size, and obtained comparable results.

5.4 Summary

For many queries (e.g., with a moderate number of groups), distributed OLAP evaluation is effective and scalable (see, e.g., Fig. 3). The optimizations discussed in this paper are effective in reducing query evaluation time by a large fraction (see, e.g., Fig. 5).

Distributed OLAP faces a scalability problem when a partition attribute is used as one of the grouping attributes, leading to a quadratic increase in query evaluation time with a linearly increasing number of sites. Most of the work is wasted as the sites do not have tuples for most of the groups sent to them. Two of the optimizations we have considered in this paper are effective in eliminating this inefficiency. Group reduction (both at the coordinator and at the sites) reduces the data traffic (and thus the query evaluation time) from quadratic to linear. Synchronization reduction further takes advantage of the partition attribute by eliminating much of the data traffic altogether.

6 Conclusions

In this paper, we have developed a framework for evaluating complex OLAP queries on distributed data warehouses. We build efficient query plans using GMDJ expressions, which allow the succinct expression of a large class of complex multi-round OLAP queries. In our distributed OLAP architecture, a coordinator manages, collects and correlates aggregate results from the distributed warehouse sites.

The use of GMDJ expressions allows us to avoid complex distributed join optimization problems. However, query plans involving GMDJ operators also require optimization for best performance. We show a collection of novel GMDJ transformations which allow us to minimize the cost of computation and of communication between the sites and the coordinator.

We built Skalla, a distributed OLAP system which implements the distributed OLAP architecture. Skalla also implements most of the optimizations discussed in this paper. We ran a collection of experiments, and found that the optimizations lead to a scalable distributed OLAP system.

The results we present in this paper are the first steps in the exploration of research issues in the important area of distributed OLAP. Future research topics could include the exploration of alternative architectures (e.g., a multi-tiered coordinator architecture or spanning-tree networks) and additional query optimization strategies.

References

1. S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. of the Int. Conf. on Very Large Databases*, pages 506–521, 1996.
2. M. O. Akinde, and M. H. Böhlen. Generalized MD-joins: Evaluation and reduction to SQL. In *Databases in Telecommunications II*, pages 52–67, Sept. 2001.
3. D. Bitton, H. Boral, D. J. DeWitt, and W. K. Wilkinson. Parallel algorithms for the executions of relational database operations. *ACM TODS* 8(3):324-353, 1983.
4. H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, a highly parallel database system. *IEEE TKDE* 2(1), March 1990
5. R. Cáceres, N. Duffield, A. Feldmann, J. Friedmann, A. Greenberg, R. Greer, T. Johnson, C. Kalmanek, B. Krishnamurthy, D. Lavelle, P. Mishra, K. K. Ramakrishnan, J. Rexford, F. True, and J. van der Merwe. Measurement and analysis of IP network usage and behavior. *IEEE Communications Magazine*, May 2000.
6. D. Chatziantoniou. Ad hoc OLAP: Expression and evaluation. In *Proc. of the IEEE Int. Conf. on Data Engineering*, 1999.
7. D. Chatziantoniou, M. O. Akinde, T. Johnson, and S. Kim. The MD-join: An operator for complex OLAP. In *Proc. of the IEEE Int. Conf. on Data Engineering*, 2001.
8. S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, Mar. 1997.
9. R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings Publishers, second edition, 1994.
10. A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving traffic demands for operational IP networks: Methodology and experience. In *Proc. of ACM SIGCOMM*, 2000.
11. G. Graefe, U. Fayyad, and S. Chaudhuri. On the efficient gathering of sufficient statistics for classification from large SQL databases. In *Proc. of Int. Conf. on Knowledge Discovery and Data Mining*, pages 204–208, 1998.
12. J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Datacube : A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.
13. R. Greer. Daytona and the fourth-generation language Cymbal. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 525–526, 1999.
14. R. Kimball. *The data warehouse toolkit*. John Wiley, 1996.
15. D. Kossman The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.
16. M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.
17. K. A. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proc. of the Int. Conf. on Very Large Databases*, pages 116–125, 1997.
18. K. A. Ross, D. Srivastava, and D. Chatziantoniou. Complex aggregation at multiple granularities. In *Proc. of the Int. Conf. on Extending Database Technology*, pages 263–277, 1998.
19. A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 104–114, 1995.
20. C. T. Yu, K. C. Guh, and A. L. P. Chen. An integrated algorithm for distributed query processing. In *Proc. of the IFIP Conf. on Distributed Processing*, 1987.