A Case for Parallelism in Data Warehousing and OLAP

Anindya Datta*
Dept. of MIS
University of Arizona
Tucson, AZ 85721
adatta@loochi.bpa.arizona.edu

Bongki Moon
Dept. of Computer Science
University of Arizona
Tucson, AZ 85721
bkmoon@cs.arizona.edu

Helen Thomas
Dept. of MIS
University of Arizona
Tucson, AZ 85721
helen@loochi.bpa.arizona.edu

Abstract

In recent years the database community has experienced a tremendous increase in the availability of new technologies to support efficient storage and retrieval of large volumes of data, namely data warehousing *and* On-Line Analytical Processing *(OLAP) products. Efficient query processing is critical in such an environment, yet achieving quick response times with OLAP queries is still largely an open issue. In this paper we propose a solution approach to this problem by applying* parallel processing *techniques to a warehouse environment. We suggest an efficient partitioning strategy based on the relational representation of a data warehouse (i.e.,* star schema*). Furthermore, we incorporate a particular indexing strategy,* DataIndexes*, to further improve query processing times and parallel resource utilization, and propose a preliminary parallel star-join strategy.*

1 Introduction

In recent years, there has been an explosive growth in the use of databases for decision support. This phenomenon is a result of the increased availability of new technologies to support efficient storage and retrieval of large volumes of data, namely *data warehousing* and *On-Line Analytical Processing* (OLAP) products. A data warehouse can be defined as an on-line repository of historical enterprise data that is used to support decision making [8]. OLAP refers to the technologies that allows users to efficiently retrieve data from the data warehouse [2]. Throughout this paper, we refer to the combination of a data warehouse and its corresponding OLAP techniques as an *OLAP system*.

The characteristics of an OLAP system are quite different from those of transactional database systems, referred to as *On-line Transaction Processing* (OLTP) systems. OLTP systems are designed to perform repetitive, structured tasks where detailed records are updated (e.g., order entry), and therefore, emphasis is placed on maximizing transaction throughput. In contrast to OLTP systems, data warehouses are designed for decision support purposes and contain long periods of historical data. For this reason, data warehouses tend to be extremely large - it is quite possible for a data warehouse to be hundreds of gigabytes to terabytes in size [1]. The information in a warehouse is usually multidimensional in nature, requiring the capability to view the data from a variety of perspectives. In this environment, aggregated and summarized data are much more important than detailed records. Queries tend to be complex and ad-hoc, often requiring computationally expensive operations such as joins and aggregation. Further complicating this situation is the fact that such queries must be performed on tables having potentially millions of records. Moreover, the results have to be delivered interactively to the business analyst using the system.

In this paper, we propose a solution to the problem of OLAP query processing by applying parallel processing techniques. To the best of our knowledge, this is one of the first works to propose a parallel physical design for data warehousing. Some of the major DBMS vendors offer products that support various levels of parallel processing. We describe this work in more detail in Section 4.

The remainder of the paper is organized as follows. In Section 2 we discuss why parallelism is especially well suited for OLAP systems, in Section 3 we describe how parallelism can be implemented in OLAP systems with a focus on physical design issues, in Section 4 we discuss related work, and in Section 5 we conclude the paper.

*The author list is in alphabetical order.

2 Why Parallelism is Appropriate for Data Warehousing

The appeal of parallel processing is especially strong for the data warehouse environment due to the inherent nature of such an environment. As mentioned previously, in OLAP systems the emphasis is on interactive processing of complex queries. Given these characteristics as well as the often extreme size of a warehouse, methods are clearly needed for more rapid query execution. By partitioning data among a set of processors, OLAP queries can be executed in parallel, potentially achieving *linear speedup* and thus significantly improving query response times.

In addition to the already large size of most warehouses, growth is another important factor in OLAP systems. Data warehouses tend to grow quite rapidly. For example, AT&T has a data warehouse containing call detail information that grows at a rate of approximately 18 GB per day [10]. Thus, a *scalable* architecture is crucial in a warehouse environment. Shared-nothing architectures have been shown to achieve near linear speedups and scale-ups in OLTP environments as well as on complex relational queries [5], and so it is worth investigating their performance in OLAP systems.

Still another reason that the appeal for parallelism is so strong in an OLAP system relates to the logical design of a warehouse. This point is best illustrated using an example. In a ROLAP environment, the data is stored in a relational database using a star schema. A star schema usually consists of a single *fact* table and set of *dimension* tables. Consider the star schema presented in Figure 1, which was derived from the TPC-D benchmark database [15] (with a scale factor of 1). The schema models the activities of a world-wide wholesale supplier over a period of 7 years. The fact table is the `SALES` table, and the dimension tables are the `PART`, `SUPPLIER`, `CUSTOMER`, and `TIME` tables. The fact table contains foreign keys to each of the dimension tables. This schema suggests an efficient data partitioning as we will soon show.

A common type of query in OLAP systems is the *star-join* query. In a star-join, one or more dimension tables are joined with the fact table. For example, the following query is a three-dimensional star-join that identifies the volumes sold locally by suppliers in the United States for the period between 1996 and 1998 [15]:

Query 1

```
SELECT     U.Name, SUM(S.ExtPrice)
FROM       SALES S, TIME T, CUSTOMER C, SUPPLIER U
WHERE      T.Year BETWEEN 1996 AND 1998
AND        U.Nation='United States' AND C.Nation='United
States'
AND        S.ShipDate = T.TimeKey AND S.CustKey = C.CustKey
AND        S.SuppKey = U.SuppKey
GROUP BY   U.Name
```

A set of attributes that are frequently used in join predicates can be readily identified in the structure of a star schema. In the example star schema, `ShipDate`, `CustKey`, `SuppKey`, and `PartKey` of the `SALES` table can be identified as attributes that will often participate in joins with the corresponding dimension tables. We can thus use this information to apply a vertical partitioning method on these attributes to achieve the benefits of parallelism. The specific vertical partitioning method that we propose is referred to as *DataIndexes*. A DataIndex is a storage structure that serves both as an index as well as data [16]. We discuss DataIndexes in more detail in a later section. For now, we turn our attention to how parallelism can be achieved in OLAP systems.

3 Achieving Parallelism in Data Warehousing

In this section we propose several preliminary ideas to achieve parallelism in data warehousing. An important issue in parallel processing is the *declustering* of data. Declustering involves decomposing the database into chunks or partitions. We now propose a declustering scheme for a ROLAP star schema based on the principle of DataIndexes, a physical design strategy recently proposed in the literature [16]. We present the basic principles behind DataIndexes as a starting point.

3.1 Designing Data Warehouses to Exploit Parallelism

Database systems, thus far, have considered index and data separately, i.e., in conventional database design, one envisions a set of relations or table structures, and a separate set of indices or access structures. A DataIndex is a storage structure that serves both as an index as well as data. This is best illustrated through an example. Consider again the TPC-D star schema presented in Figure 1. In a typical *Relational Data Warehouse Management System* (RDWMS), these tables would be stored "as-is", i.e., all 5 tables would be stored. In addition to the base tables, for retrieval efficiency, index structures would typically be defined (especially in data warehouses where the fact table is very large). More specifically, the `SALES` table will likely be indexed on each of its six dimensional attributes (namely, `PartKey`, `SuppKey`, `CustKey`, `ShipDate`, `CommitDate` and `ReceiptDate`). A number of indexing schemes have been proposed in the literature. Among these, four index types are shown in [13] to be particularly appropriate for OLAP systems: *B^+-trees* , *bitmapped indexes* [12], *projection indexes* and *bit-sliced indexes* [13].

A DataIndex, like the projection index, exploits a *positional* indexing strategy. A projection index is simply a mirror image of the column being indexed. In particular, when indexing columns of the fact table, both the index and the corresponding column in the

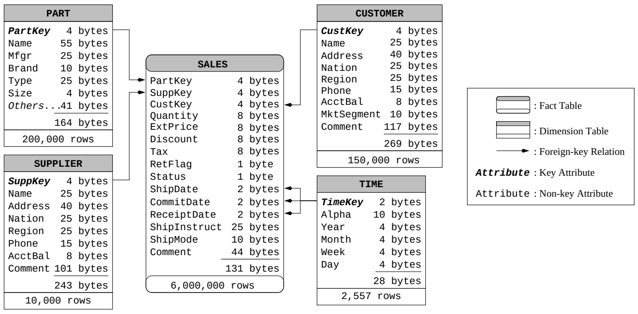

Figure 1: A Sample Warehouse Star Schema

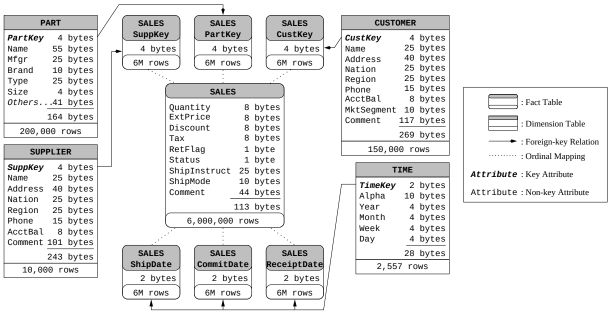

Figure 2: Example Warehouse Schema with DataIndex

fact table are stored, resulting in a duplication of data. In a DataIndex, however, only the index is stored. By applying this idea, we can divide the `SALES` table in Figure 1, into seven smaller tables, as shown in Figure 2. The new scheme is then composed of 7 vertical partitions: one for each of the dimensional attributes and one for the remaining columns from the original `SALES` table. A record in the original `SALES` table is now partitioned into 7 records, one in each of the resulting tables. Any such record can easily be re-built from these since its component rows in the resulting tables all share the same ordinal position. This ordinal mapping is key to the idea of positional indexing. See [13, 16] for details. Each of the 7 new tables is a DataIndex. We now describe two specific types of DataIndexes, *Basic DataIndexes* and *Join DataIndexes*.

Basic DataIndex (BDI) A DataIndex can be simply created as a vertical partition of a relational table. This sort of partition can contain any number of columns from the original table, unlike projection indexes which are restricted to single columns. In this sort of partitioning, the columns being indexed are removed from the original table and stored separately, with each entry being in the same position as its corresponding base record. The isolated columns can then be used like a projection index for fast access to data in the table. This partition is referred to as a *Basic DataIndex* (BDI). Retrievals using BDIs rely on ordinal position to RID mapping. For exact details of the mapping from RID to ordinal position, refer to [16].

Join DataIndex (JDI) In decision support databases, a large portion of the workload consists of queries that operate on multiple tables. Many queries on the star schema of Figure 1 would access one or more dimension tables and the central `SALES` table. Access methods that efficiently support join operations thus become crucial in decision support environments [12, 14]. The idea of a BDI presented in the previous section can very easily be extended to support such operations. Consider for instance, an analyst who is interested in possible trends or seasonalities in discounts offered to customers. This analysis would be based on the following query:

Query 2

```
SELECT     TIME.Year, TIME.Month, average(SALES.Discount)
FROM       TIME, SALES
WHERE      TIME.TimeKey = SALES.ShipDate
GROUP BY   TIME.Year, TIME.Month
```

Using the conventional relational approach, the association between the two tables `TIME` and `SALES` in Figure 1 is implemented through the primary key/foreign key relationship linking the columns `ShipDate` and `TimeKey`. To perform a join operation on these two tables, the two columns must be accessed to determine the records that are join candidates. There exist relatively fast algorithms (e.g., merge and hash joins) for evaluating joins. However, approaches that use pointers to the underlying data, instead of the actual records, tend to give a better performance than other join strategies [6]. Indeed, we can significantly reduce the number of data blocks to be accessed while processing a join by storing the RIDs of the matching records in the corresponding dimension table – instead of the corresponding key values – in a BDI for a foreign key column. This structure is a *Join DataIndex* (JDI). The JDI on `SALES.ShipDate` would then consist of a list of RIDs on the `TIME` table. In this structure, instead of storing the data corresponding to the `ShipDate` column, the JDI provides a *direct* mapping between individual tuples of the `SALES` and `TIME` tables. It has been shown in [16] that the join required to answer Query 2 can thus be performed in a single scan of the JDI. This property of JDIs is indeed attractive, since the size of this index is, of course, proportional to the number of tuples in the table from which it was derived.

The application of the DataIndex principle to a data warehouse results in a physical design that exploits parallelism. For instance, consider again Query 1. This query entails a 3-dimensional star join, which must join the 3 dimensional tables `TIME`, `CUSTOMER`, and `SUPPLIER` with the `SALES` table. A typical DataIndexing scheme on this star schema may be obtained using the procedure described in Algorithm 1. The notation used in this procedure and for the remainder of the paper is presented in Table 1.

Symbol	Description
F	Fact table
A_F	Set of fact table attributes
A_F^d	Set of fact table dimensional attributes
A_F^m	Set of fact table metric attributes
$\mathcal{D}$	Set of all dimension tables
D_i	Dimension table i where $D_i \in \mathcal{D}$
A_D	Set of dimensional attributes for all dimension tables
$\mathcal{J}_{\mathtt{a_j}}$	JDI for attribute a_j
$\mathcal{B}_{\mathtt{a_j}}$	BDI for attribute a_j
B_{A_D}	Set of BDIs for dimensional attributes
$B_{A_F^m}$	Set of BDIs for metric attributes
A_P^d	Set of dimensional projection attributes
A_P^m	Set of metric projection attributes
P_σ	Set of restriction predicates
A_R	Set of restriction attributes
$P_{\bowtie}$	Set of join predicates
G	Set of processor groups
$\mathcal{R}_{\mathtt{a_j}}$	Rowset for attribute a_j
$\mathcal{P}_{\mathtt{a_j}}$	Projection column for attribute a_j
$\mathcal{N}$	Number of processors
$\mathcal{N}_i$	Number of processors in group i
S_i	Size of D_i in bytes
r_f	Ratio of metric data to the entire data volume of the warehouse

Table 1: Table of Notation

For the star schema of Figure 1, the application of this procedure would result in the following JDIs and BDIs:

- JDIs: $\mathcal{J}_{\mathtt{SALES.SuppKey}}$ for the `SUPPLIER` dimension, $\mathcal{J}_{\mathtt{SALES.CustKey}}$ for the `CUSTOMER` dimension, $\mathcal{J}_{\mathtt{SALES.PartKey}}$ for the `PART` dimension, and $\mathcal{J}_{\mathtt{SALES.ShipDate}}$, $\mathcal{J}_{\mathtt{SALES.CommitDate}}$, and $\mathcal{J}_{\mathtt{SALES.ReceiptDate}}$ for the `TIME` dimension
- BDIs for `SALES` fact table: $\mathcal{B}_{\mathtt{SALES.Quantity}}$, $\mathcal{B}_{\mathtt{SALES.ExtPrice}}$, $\mathcal{B}_{\mathtt{SALES.Discount}}$, $\cdots$
- BDIs for dimension tables: $\mathcal{B}_{\mathtt{TIME.TimeKey}}$, $\mathcal{B}_{\mathtt{TIME.Alpha}}$, $\mathcal{B}_{\mathtt{TIME.Year}}$, $\mathcal{B}_{\mathtt{TIME.Month}}$, $\mathcal{B}_{\mathtt{TIME.Week}}$, $\mathcal{B}_{\mathtt{TIME.Day}}$ for the `TIME` dimension. BDIs for all attributes in each of the remaining dimension tables would be established in the same way.

Algorithm 1 DataIndex Procedure for Physical Design

Input:

1. Fact table F with schema A_F. The schema is decomposable into disjoint sets A_F^d, a set of dimensional keys, and A_F^m, a set of metric attributes.
2. A set of d dimension tables, $D_1, \ldots, D_d$, with corresponding schemas $R_1, \ldots, R_d$, respectively.

for each dimensional key attribute $a_j \in A_F^d$ **do**
  Store a_j as JDI $\mathcal{J}_{\mathtt{a_j}}$
for each metric attribute $a_k \in A_F^m$ **do**
  Store a_k as BDI $\mathcal{B}_{\mathtt{a_k}}$
for each dimension table D_i **do**
  for each $a_l \in R_i$ **do**
    Store a_l as BDI $\mathcal{B}_{\mathtt{a_l}}$

Having presented the use of DataIndexes as a physical design strategy in a data warehouse, we now turn our attention to describing our data placement strategy.

3.2 Data Placement

To describe our data placement strategy, we assume a shared-nothing architecture with $\mathcal{N}$ processors. We further assume a d-dimensional data warehouse physically designed according to the DataIndexing mechanism outlined in the previous section. Our basic approach is as follows: we will partition the $\mathcal{N}$ processors into $d+1$ (potentially mutually non-exclusive) processor groups. Subsequently, dimension table i, i.e., D_i, (designed according to the DataIndexing strategy outlined in Algorithm 1) and the fact table JDI corresponding to the key value of D_i will be assigned to processor group i. Inside processor group i, a hybrid strategy will be used to allocate records to individual processors. The metric BDIs will be allocated to group $d+1$. Below we motivate the above approach and provide more details.

There are three fundamental motivations behind this approach. First, the task of data placement can be *hinted* by the structure of the star schema. For example, the primary key of a dimension table and its associated foreign key of a fact table can be the most favorable candidates for the partitioning attributes, because they are expected to be used as join attributes frequently.

Second, the use of JDIs makes it possible to co-locate the fact table with multiple dimension tables *at the same time* by grouping each dimension table with its associated JDI and partitioning them by the same strategy. (In general, a relation can be co-located with only one other relation with a traditional horizontal partitioning method.) Therefore, the join of a dimension table and a fact table can be computed efficiently in parallel without data redistribution, and completely independent of other join computations that involve different dimension tables and the same fact table.

Third, it is generally the case that the size of a dimension table is much smaller than that of a fact table, and often small enough to be fit in main memory. Thus, given the number of available processors and aggregate main memory capacity, the relative sizes of dimension tables can be used to determine an ideal degree of parallelism for each *dimension*, that is, a dimension table and its associated JDI.

Algorithm 2 Data Placement

Note: This algorithm determines the size ($\mathcal{N}_i$) of a processor group for each i-th dimension ($1 \leq i \leq d$), the size ($\mathcal{N}_{d+1}$) of a processor group for the metric attributes, and suggests an intergroup partitioning strategy across individual processors.

Input:

Given $\mathcal{N}$ processors each with main memory of size m in bytes, a data warehouse represented by a d-dimensional star schema. The size of the i-th dimension table in bytes is $\mathcal{S}_i$, and the ratio of the metric data to the entire data volume of the star schema is r_F.

PROCESSOR GROUP PARTITIONING PHASE

for $i(1 \leq i \leq d+1)$ **do**

1. Choose $\mathcal{N}_i$ $(1 \leq i \leq d)$ to be the smallest number such that the dimension table can be fit in the aggregate memory of the processor group. That is, $\mathcal{N}_i = \min(\mathcal{N}, \lceil \mathcal{S}_i/m \rceil)$.
2. Choose $\mathcal{N}_{d+1}$ to be proportional to the data volume stored. That is, $\mathcal{N}_{d+1} = \lceil \mathcal{N} \times r_F \rceil$.
3. If there are more available processors than required in the previous step (that is, $\sum_{i=1}^{d+1} \mathcal{N}_i < \mathcal{N}$), then choose $\mathcal{N}_i$ such that $\mathcal{S}_i/(m \times \mathcal{N}_i)$ is maximal, and increment it by one. Repeat this process until $\sum_{i=1}^{d+1} \mathcal{N}_i = \mathcal{N}$.
4. Now $\sum_{i=1}^{d+1} \mathcal{N}_i \geq \mathcal{N}$. Assign d+1 (virtual) processor groups to $\mathcal{N}$ (physical) processors such that the overlap of processor groups is minimized.

PHYSICAL DATA PLACEMENT PHASE

1. Allocate the i^{th} dimension table and its associated JDI to processor group i.
2. If $\mathcal{N}_i > 1(1 \leq i \leq d)$, then partition the i-th dimension table and its JDI horizontally such that they are *co-located* with respect to the join attribute.
3. Allocate metric BDIs to processor group $d+1$
4. If $\mathcal{N}_{d+1} > 1$, then partition horizontally the metric data.

Now we describe our algorithm. Essentially there are two phases: (a) a processor group partitioning phase where we partition the set of given processors into $d+1$ groups and (b) a physical data placement phase where we allocate data fragments to individual processors. This is shown in Algorithm 2.

3.3 Parallel Star Join

In this section we present an algorithm to perform a star join in parallel. We assume a DataIndexing scheme as described in Section 3.1 and a partitioning strategy as described in Section 3.2. Let $\mathcal{D}$ be the set of dimension tables where $|\mathcal{D}| = d$, so there are d dimension tables and $k \leq d$. Let G be the set of processor groups such that $|G| = d+1$. We represent a general star-join query as follows (refer to Table 1 for notation):

```
SELECT  A_P^d, A_P^m
  FROM  F, D_1, ..., D_k
 WHERE  P_⋈ AND P_σ
```

Here $D_1, \ldots, D_k$ are the k dimensional tables participating in the join. We assume that each individual restriction predicate in P_σ only concerns one table and is of the form a_j $\langle op \rangle$ `constant`, where a_j is any attribute in the warehouse schema and $\langle op \rangle$ denotes a comparison operator (e.g., $=, \leq, \geq$). We assume each join predicate in $P_{\bowtie}$ is of the form $a_l = a_t$ where a_t is any dimensional key attribute and a_l is the foreign key referenced by a_t in the fact table.

A high-level description of the algorithm is presented in Algorithm 3. We emphasize the potential for efficiency gains in this algorithm, which result primarily from two factors: parallel processing of the query (due to the partitioning) and the use of DataIndexes. Using DataIndexes provides the capability to perform many of the operations in a bitwise fashion, thus lending a high degree of efficiency. Although measuring the actual performance of the proposed algorithm is beyond the scope of this paper, we expect significant gains in terms of both speedup and scale-up.

4 Related Work

Two main approaches have been proposed to improve OLAP query performance (i.e., response times) in OLAP systems: *precomputation strategies* and indexing strategies. Precomputation strategies involve deriving tables that store precomputed answers to queries. Such tables are often referred to as *summary tables* or *materialized views* [1]. There is a tradeoff between response times and the storage requirements of precomputed data. Determining how much data to precompute is an issue that has been addressed in [7]. The work in indexing strategies includes traditional approaches, such as tree-based indexing (e.g., [9]), as well as non-traditional approaches, such as positional indexing, which has been proposed in [13, 16]. DataIndexes is a form of positional indexing.

Algorithm 3 Parallel Star Join

Note: Requires the DataIndexing scheme described in Section 3.1 and the partitioning strategy described in Section 3.2.

Input:
G, $D_1, \ldots, D_k$, A_D, B_{A_D}, $B_{A_F^m}$, A_P^d, A_P^m, A_F^d, A_F^m, P_σ, and A_R where each $a_{R_i} \in A_R$ exists in some predicate in P_σ. Refer to Table 1 for explanation.

```
for each dimensional attribute a_j ∈ A_D do
    Locate B_{a_j} in group g_i
for each metric attribute a_k ∈ A_F^m do
    Locate B_{a_k} in group g_{d+1}
R_all_dim ← NULL
for each restriction attribute a_l ∈ A_R do
    Generate restriction rowset R_{a_l} based on predicate p_{σ.t}
    Send R_{a_l} to group g_{d+1}
    R_all_dim ← R_all_dim ∧ R_{a_l}
Result ← B_{A_F^m}
for each dimension projection attribute a_l ∈ A_P^d do
    Fetch R_all_dim
    Generate projection column P_{a_l} based on R_all_dim
    Send P_{a_l} to group g_{d+1}
    Result ← Result ∘ P_{a_l}
Output Result.
```

A large body of work exists in applying parallel processing techniques to relational database systems (e.g., [4]). From this work has emerged the notion that highly-parallel, shared-nothing architectures can yield much better performance than equivalent closely-coupled systems [4]. Indeed, may commercial database vendors have capitalized on this fact [5]. An issue that is closely related to parallelism is that of declustering large data sets over a number of nodes in a parallel database machine. Various methods have been developed over the years to distribute data across sites. Some early work in the area has concentrated on *hash-* or *range-partitioning* based on a single key [3]. This approach is also supported by a number of database vendors (e.g., Oracle, Informix and NCR). More recently, multi-attribute declustering techniques have been proposed and analyzed [11]. As mentioned previously, we are not aware of any published academic work related to parallelism in data warehouses. There are, however, several DBMS vendors that claim to support parallel warehousing products to various degrees.

5 Conclusion

In this paper we have presented a framework for applying parallel processing to OLAP systems. Our proposed approach entails a physical design based on DataIndexes, which takes advantage of the efficient partitioning suggested by the star schema representation of a data warehouse. We have proposed a declustering strategy which incorporates both task and data partitioning. We have also presented the Parallel Star Join Algorithm, which provides a means to perform a star join in parallel using efficient operations involving only rowsets and projection columns.

The approach we have presented is preliminary, leaving open many issues. Future work in this area includes further refinement of the Parallel Star Join Algorithm, implementation and testing of the algorithm, and the development of algorithms for other OLAP operations such as slice, dice, roll-up, and drill-down. It is also important to devise meaningful metrics to judge the performance of algorithms in parallel warehouses.

References

[1] S. Chauduri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, March 1997.

[2] E. Codd, S. Codd, and C. Salley. Providing OLAP (on-line analitycal processing) to user-analysts: An IT mandate. Technical report, E.F. Codd & Associates, 1993.

[3] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in Bubba. In *Proc. ACM SIGMOD*, Chicago, IL, May 1988.

[4] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Comm. of the ACM*, 35(6):85–98, June 1992.

[5] S. Engelbert, J. Gray, T. Kocher, and P. Stah. A benchmark of non-stop SQL Release 2 demonstrating near-linear speedup and scaleup on large databases. Technical Report 89.4, Tandem Computers, May 1989. Tandem Part No.27469.

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

[7] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proc. ACM SIGMOD*, pages 205–216, Montreal, Canada, June 4-6 1996.

[8] W. Inmon. *Building the Data Warehouse*. J. Wiley & Sons, Inc., second edition, 1996.

[9] T. Johnson and D. Shasha. Some approaches to index design for cube forests. *Bulletin of the Technical Committee on Data Engineering*, 20(1), March 1997.

[10] K. Lyons, AT&T Research. Private communication, July-August 1997.

[11] B. Moon and J. H. Saltz. Scalability analysis of declustering methods for multidimensional range queries. *IEEE Transactions on Knowledge and Data Engineering*, 10(2), Mar/Apr 1998.

[12] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, September 1995.

[13] P. O'Neil and D. Quass. Improved query performance with variant indexes. In J. M. Peckman, editor, *Proc. ACM SIGMOD*, volume 26(2) of *SIGMOD Record*, pages 38–49, Tucson, Arizona, May 13-15 1997.

[14] Red Brick Systems. Star schema processing for complex queries. White Paper, July 1997.

[15] Transaction Processing Performance Council, San Jose, CA. *TPC Benchmark D (Decision Support) Standard Specification*, revision 1.2.3 edition, June 1997.

[16] I. Viguier, A. Datta, and K. Ramamritham. "have your data and index it, too", efficient storage and indexing for data warehouses. Technical Report GOOD-TR-9702, Dept. of MIS, University of Arizona, June 1997. Submitted for Publication URL: `http://loochi.bpa.arizona.edu`.