Parallel Star Join + DataIndexes : Efficient Query Processing in Data Warehouses and OLAP

Anindya Datta*, Debra VanderMeer*, Krithi Ramamritham†

Abstract

On-Line Analytical Processing (OLAP) refers to the technologies that allow users to efficiently retrieve data from the data warehouse for decision-support purposes. Data warehouses tend to be extremely large - it is quite possible for a data warehouse to be hundreds of gigabytes to terabytes in size [3]. Queries tend to be complex and ad-hoc, often requiring computationally expensive operations such as joins and aggregation. Given this, we are interested in developing strategies for improving query processing in data warehouses by exploring the applicability of parallel processing techniques. In particular, we exploit the natural partitionability of a star schema and render it even more efficient by applying DataIndexes – a storage structure that serves both as an index as well as data and lends itself naturally to vertical partitioning of the data. Dataindexes are derived from the various special purpose access mechanisms currently supported in commercial OLAP products. Specifically, we propose a declustering strategy which incorporates both task and data partitioning and present the Parallel Star Join (PSJ) Algorithm, which provides a means to perform a star join in parallel using efficient operations involving only rowsets and projection columns.

We compare the performance of the PSJ Algorithm with two parallel query processing strategies. The first is a parallel join strategy utilizing the Bitmap Join Index (BJI), arguably the state of the art OLAP join structure in use today. For the second strategy we choose a well known parallel join algorithm, namely the pipelined hash algorithm. To assist in the performance comparison, we first develop a cost model of the disk access and transmission costs for all three approaches.

Performance comparisons show that the DataIndex based approach leads to dramatically lower disk access costs than the BJI as well as the hybrid hash approaches, in both speedup and scaleup experiments, while the hash-based approach outperforms the BJI in disk access costs. With regard to transmission overhead, our performance results show that PSJ and BJI outperform the hash-based approach. Overall, our parallel star join algorithm and dataindexes form a winning combination.

Keywords: parallel star join, OLAP, query processing, dataindexes

Contact: Anindya Datta, adatta@cc.gatech.edu, Phone: 404-442-9911, Fax: 404-442-9088

1 Introduction

On-Line Analytical Processing (OLAP) refers to the technologies that allow users to efficiently retrieve data from the data warehouse for decision-support purposes. A data warehouse can be defined as an on-line repository of historical enterprise data that is used to support decision making [15]. Data warehouses tend to be extremely large - it is quite possible for a data warehouse to be hundreds of gigabytes to terabytes in size [3]. 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.

Given these characteristics, it is clear that the emphasis in OLAP systems is on query processing and response times. OLAP scenarios in data warehousing differ from standard OLTP environments in two important ways: (1) the size of the data store, and (2) the underlying data model of the warehouse. In terms of size, a data warehouse is typically orders of magnitude larger than in standard operational databases (i.e., hundreds of GBs, even TBs). These databases store historical data, not operational data, and are used primarily for decision support. Decision

*Georgia Institute of Technology, Atlanta, GA 30332
†IIT Bombay

support requires complex queries, e.g., multi-way joins. In terms of the data model, most warehouses are modeled with a star schema, i.e., a fact table and a set of data dimensions. Star schemas have an important property in terms of join processing – all dimensions join only with the fact table (i.e., the fact table contains foreign keys for each dimension). As a result, all join paths lead through the fact table, which is typically the largest table by far – usually several times the sum of the sizes of the dimensions.

Given the above, we note that joins in data warehouses are particularly expensive – the fact table (the largest table in the warehouse by far) participates in every join, and multiple dimensions are likely to participate in each join. Clearly, applying parallel processing to the join operation in this case would be beneficial.

Parallel query processing techniques, such as those described in in [19] and [1] (as well as others noted in Section 9), will clearly function in an OLAP environment. The question of interest is this: "*Can more efficient techniques be developed, given the particular characteristics of the read-mostly OLAP environment?*". For instance, the current state of the art in parallel join techniques, as exemplified by the Pipelined Hash Join using right-deep trees [4, 5], requires that either (a) all hash tables for participating build tables be co-resident in memory or (b) that temporary results be spooled to disk, allowing reclamation of memory for the hash tables used to build that intermediate result. Given the large data sizes in a data warehousing environment, it is unlikely that sufficient memory will be available, particularly in the case of multi-way joins. In this paper, we propose a novel parallel processing technique to specifically address the large data sizes inherent in OLAP query processing, and provide efficient query processing.

Performance in a parallel system is typically measured using these two key properties:

Property 1 : *In system with linear* **scale-up**, *an increase in hardware can perform a proportionately larger task in the same amount of time. 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 [20]. Thus, a scalable architecture is crucial in a warehouse environment.*

Property 2 : *In a system with linear* **speedup**, *an increase in hardware results in a proportional decrease in processing time. As we shall show, by partitioning data among a set of processors, and by developing query processing strategies that exploit this partitioning, OLAP queries can potentially achieve good speedup, significantly improving query response times.*

The first property is obvious, while the latter point is best illustrated using an example. Recall that 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 1A, which was derived from the TPC-D benchmark database [27] (with a scale factor of 1). The schema models the activities of a world-wide wholesale supplier over a period of seven 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.

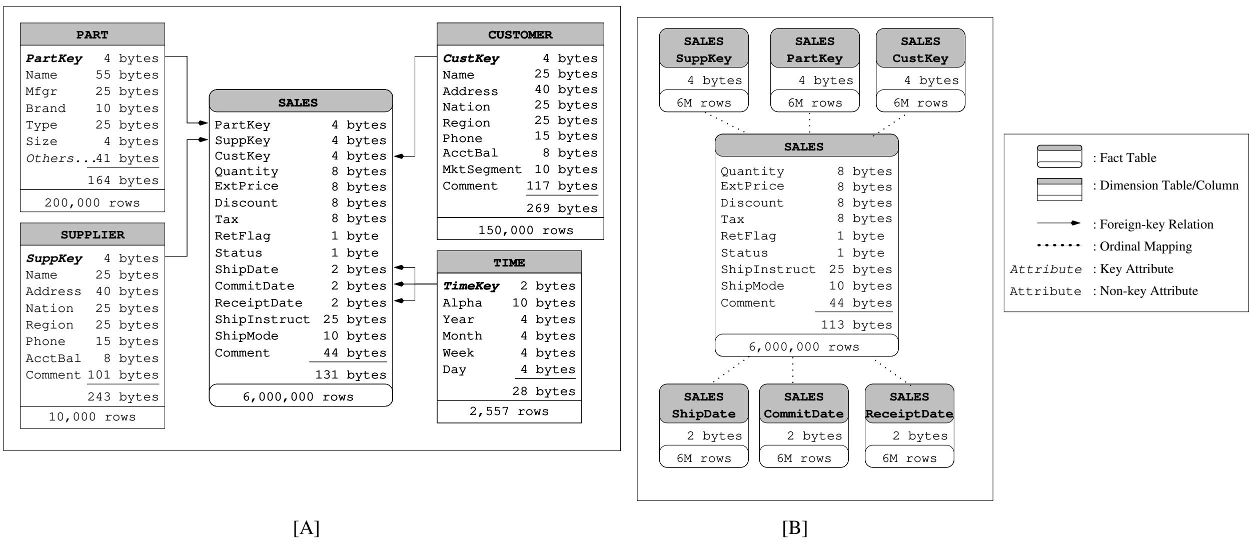

Figure 1: A Sample Warehouse Star Schema and Projection Index

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 [27]:

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 is 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. This paper shows, in fact, that one can use a combination of *vertical* and *horizontal* partitioning techniques to extract the parallelism inherent in star schemas.

Specifically, we propose a declustering strategy which incorporates both task and data partitioning and present the Parallel Star Join (PSJ) algorithm, which provides a means to perform a star join in parallel using efficient operations involving only rowsets and projection columns. To compare against PSJ, we consider two other parallel query processing strategies. The first is a parallel join strategy utilizing the Bitmap Join Index (BJI), arguably the state of the art OLAP join structure in use today. For the second strategy we choose the *Pipelined Hash* strategy [4] (HASH), one of the best performing parallel query processing strategies from the traditional OLTP literature.

Our performance results indicate that the PSJ approach leads to dramatically better performance than the pipelined hash approach, with regard to disk access costs and transmission costs, in both speedup and scaleup experiments. The pipelined hash approach, in turn, outperforms the BJI approach in terms of disk access costs (although not in terms of transmission costs). A full discussion of our results can be found in Section 8.

A large body of work exists in applying parallel processing techniques to relational database systems (e.g., [8, 26, 28, 25]). From this work has emerged the notion that highly-parallel, shared-nothing architectures can yield much better performance than equivalent closely-coupled systems [24, 17, 9]. Shared-nothing architectures have been shown to achieve near linear speedups and scale-ups in OLTP environments as well as on complex relational queries [10].

The primary contribution of this paper is in its basic theme, i.e., the exploration of parallel processing with regard to OLAP. To the best of our knowledge, this is one of the initial endeavors in this direction (we have not come across many such reports in the published literature). Specifically, the contribution of this paper is manifold: (1) It proposes a parallel physical design for data warehousing. (2) It proposes a parallel star join strategy based on this physical design and evaluates its performance. (3) It demonstrates the applicability of parallel OLTP strategies in the OLAP context. Note that some of the major DBMS vendors offer products that support various levels of parallel processing. We describe this work in more detail in Section 9 and contrast these to our work. Note also that integration with existing systems is a separate issue, and outside the scope of this paper.

The remainder of the paper is organized as follows. In Section 2 we introduce an approach to structure data warehouses by exploiting our proposed indexing strategies. The associated physical design and star join processing strategies are discussed in Section 3. This is followed by a description of two competing strategies: the parallel BJI join approach in Section 4 and the pipelined hash approach in Section 5. We then present a cost model of the disk access and transmission costs of the three approaches in Section 6, and a system model for performance comparison in Section 7. We compare the performance of these approaches in Section 8. Finally, in Section 9 we discuss related work, and in Section 10 we conclude the paper.

2 A Physical Design Principle to Exploit Parallelism

In this section we show how, by judiciously using many of the indexing schemes proposed in the literature, we can structure a data warehouse to make it amenable to parallel query processing. Four index types are shown in [22] to be particularly appropriate for OLAP systems: *B+ trees*, indexes based on *bitmaps* [22, 21], *projection indexes* and *bit-sliced indexes* [22]. Consider the division of the `SALES` table in Figure 1A, into seven smaller tables, as shown in Figure 1B. This scheme is composed of 7 vertical partitions: one for each of the dimensional attributes and one for the remaining columns from the original `SALES` table. With this division, a record in the original `SALES` table is now partitioned into 7 records, one in each of the resulting tables. Each of the 7 new tables is akin to a projection index. A projection index contains the copy of a particular column, namely, the column being indexed. 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 for fast access to data in the table. When indexing columns of the fact table, storing both the index and the corresponding column in the fact table results in a duplication of data. In such situations, it is advisable to only store the index if original table records can be reconstructed easily from the index itself. This is how Sybase IQ stores data [12, 22].

In what follows, we extend the original notion of the projection index to allow a single projection index to contain multiple columns. A graphical representation of this structure is shown in Figure 2. In this figure, we

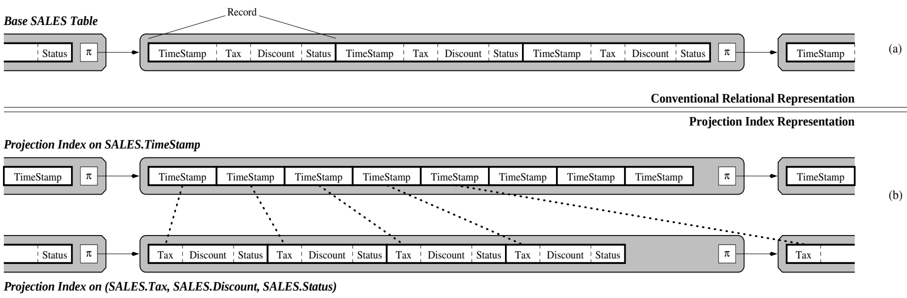

Figure 2: Projection Index

show the actual storage configurations of the two cases: a base table (Figure 2a) and the corresponding partitioned structure (Figure 2b). The base table consists of the attributes `TimeStamp, Tax, Discount, Status` and two projection indices are constructed, one on the `TimeStamp` column, and another on the `Tax, Discount` and `Status` columns. As indicated by the dotted lines joining records from the two indices, the order of the records in the base table is conserved in both indices. This allows for an efficient mapping between the entries in the two projection indexes. This mapping is accomplished through the use of *positional indexing*, which refers to accessing tuples based on their ordinal position. This ordinal mapping is key to the idea of positional indexing. For example, in the schema in Figure 1B, if we need to determine the `ShipDate` for the third `SALES` record, we would do this by accessing the third entry of the projection index for `SALES.ShipDate`. Positional indexing is made possible by row identifiers (RIDs), a feature provided by most commercial DBMS products [21, 6].

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 1A 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 [21]. The idea of a projection index 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
```

O'Neil and Graefe [21] introduced the idea of a bitmapped join index (BJI) for efficiently supporting multi-table joins. A BJI associates related rows from two tables [21], as follows. Consider two tables, T_1 (a dimension table) and T_2 (a fact table), related by a one-to-many relationship (i.e., one record of T_1 is referenced by many records of T_2). A bitmapped join index from T_1 to T_2 can be seen as a bitmapped index that uses RIDs of T_1 to index the records of T_2. (Further details of BJIs are presented in Section 4). In fact, we can further reduce the number of data blocks to be accessed while processing a join by storing the RIDs of the matching dimension table records – instead of the corresponding key values – in a projection index for a foreign key column. Such an index from T_2 to T_1 is called a *Join Index* (JI) in the sequel. For instance, the JI on `SALES.ShipDate` would consist of a list of RIDs on the `TIME` table. (One can also achieve an equivalent, and sometimes more efficient, representation by storing actual ordinal positions corresponding to the `TIME` table, rather than the RIDs). Such a JI is shown in Figure 3. As before, we show both the conventional relational and the JI representations. In the conventional approach, we show referential integrity links between the `SALES` and `TIME` tables as dashed arrows. For the JI approach, we use solid arrows to show the rows to which different RIDs point and dotted lines to show that the order of the records in the JI and the `SALES` projection index is preserved from the base table.[1]

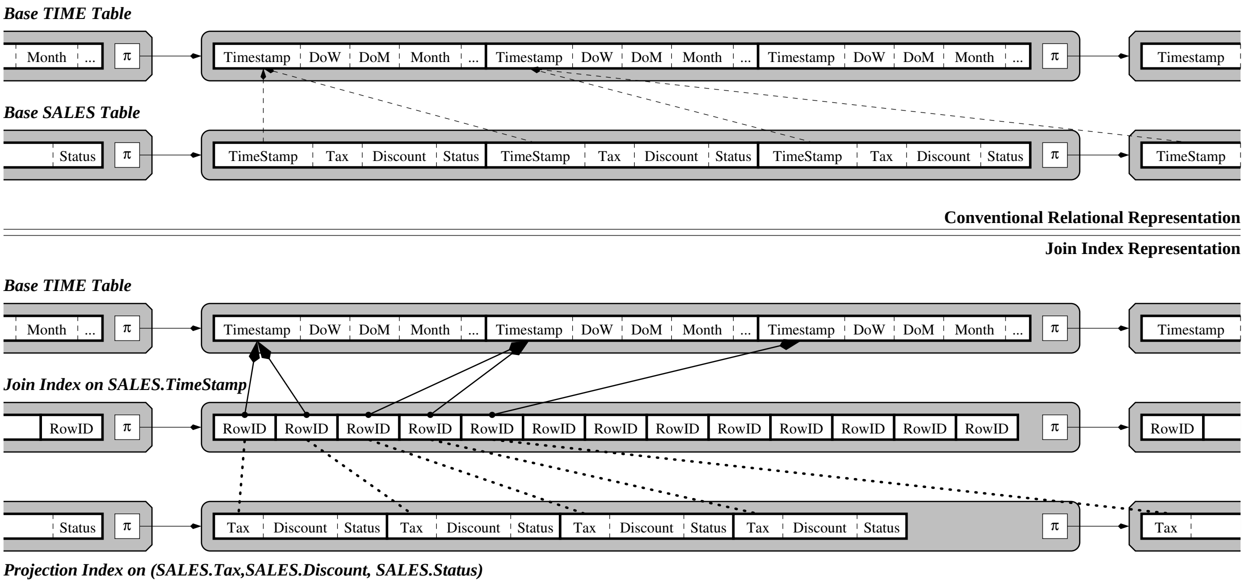

Figure 3: The Join Index

As can be seen in this figure, instead of storing the data corresponding to the `ShipDate` column, the JI provides a *direct* mapping between individual tuples of the `SALES` and `TIME` tables. Because of this, the join required to answer Query 2 can thus be performed in a single scan of the JI. This property of JIs 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 above ideas to a data warehouse results in a physical design that exploits parallelism. This design principle requires the storage of each foreign key column in the fact table as JIs and the rest of the columns in the star scheme (for both dimension as well as fact tables) as projection indexes.

In summary, the data warehouse structure discussed in this section takes the best aspects of vertical partitioning, projection indexes, and join indexes and integrates them such that, as shown in the next section, an effective parallel join algorithm can be developed. Performance studies, discussed in Section 8, show that this join algorithm enhances the performance of star-join queries in parallel environments compared to traditionally used approaches.

3 The Parallel Star Join

In this section, we describe a data placement strategy based on the physical design strategy outlined in the previous section assuming a shared-nothing architecture with $\mathcal{N}$ processors. Subsequently, we present an efficient Parallel Star Join Algorithm that exploits this placement strategy. In Table 1, we show the various notations used throughout the paper.

3.1 Data Placement Strategy

Assume a d-dimensional data warehouse physically designed according to the strategy outlined in the previous section. The basic approach to data placement is as follows: partition the $\mathcal{N}$ processors into $d+1$ (potentially mutually non-exclusive) processor groups. Assign, to processor group j, the dimension table j, i.e., D_j, and $\mathcal{J}_j$, the fact table JI corresponding to the key attribute of D_j. Inside processor group j, a hybrid strategy is used

[1]Throughout the paper, the descriptions for the storage structures and the algorithms assume that referential integrity is maintained. With simple extensions, our approach can be made to tolerate acceptable violations of referential integrity.

Symbol	Description	Symbol	Description
$\mathcal{N}$	Number of processors	d	Number of dimensions
D_j	Dimension j	F	Fact Table
$\mathcal{J}_j$	Join Index on D_j	$\mathcal{P}_{\texttt{cname}}$	Projection Index on attribute `cname`
$\mathcal{S}_{md}$	Aggregate size of metric data	$\mathcal{S}_j$	Aggregate size of D_j
m	Memory per processor (bytes)	G_j	Processor group for D_j
P_{ij}	Processor i of G_j	A_P^m	set of projection predicates on metric data
A_P^d	Set of projection predicates on dimensions	P_σ	Set of restriction predicates
$P_{\bowtie}$	Set of join predicates	$\mathcal{R}_{global}$	Global join rowset
$\mathcal{R}_{dim,i}$	Dimension restriction rowset for D_i	$w(D)$	Width of a tuple in a dimension D
B	Size of a disk block (bytes)	R	Size of a RID
a_{ij}	Attribute j of dimension i	A_R	Set of restriction attributes
a_{Fj}	Metric attribute j of the fact table	A_J	Set of join attributes
$W_{a_{ij}}$	Width of attribute a_{ij}, in bytes	A_P	Set of projection attributes
W_{JI}	Width of JI, in bytes	D_J	Number of Dimensions participating in join
$\mathcal{N}_i$	Number of processors in group i	D_P	Number of Dimensions participating in output
$\mathcal{N}_F$	Number of processors in the metric group	S_m	Size of available memory
SF	Scale Factor (affects size of database)	P	Order of BI or BJI
V	Number of distinct values indexed in BI or BJI	f	Bitmap compression factor
K	Number of search key values per index node	B	Block Size
ς	Selectivity of dimension	T_B	Tuples per block

Table 1: Table of Notation for Cost Model

to allocate records to individual processors. The metric PIs (that is, PIs of columns not associated with foreign keys) are allocated to group $d+1$.

There are three fundamental motivations behind this approach. (1) 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 in a fact table can be the most appropriate candidates for the partitioning attributes, because they are expected to be used as join attributes frequently. (2) The use of JIs 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 JI and partitioning them by the same strategy. (In general, with a traditional horizontal partitioning method, a relation can be co-located with only one other relation.) Therefore, the join of a dimension table and a fact table can be computed efficiently without data redistribution, and completely independent of other join computations that involve different dimension tables and the same fact table. (3) 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 of a particular processor group, 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 JI.

Now we describe our strategy in detail. Essentially there are two phases: (a) a processor group partitioning phase, in which 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.

3.1.1 Processor Group Partitioning

The first phase computes for each dimension j, the composition of the processor group (i.e., the physical processors assigned to each group) where the j^{th} dimension table and its associated JI are stored. Since every JI has the same cardinality as the fact table and consequently identical data volumes, the size ($\mathcal{S}_j$) of the j^{th} dimension table (in bytes) is used to determine the size of its corresponding processor group. This is not just to balance the data distribution across available processors but also to minimize the data accesses required to process a join computation between a dimension table and its JI and thereby improve response times.

Group $d+1$, i.e, the group that houses the metric PIs uses a different criterion than the dimensional groups for determining its composition. There are two main reasons for this. First, the metric attributes do not appear in any join predicates. (However, they may appear in restrictions). Second, the volume of the metric data is largely independent of those of dimension tables. Thus, we choose to use the metric data volume, $\mathcal{S}_{md}$, relative to the volume of the entire data warehouse, to determine the composition of the metric group.

Before delving into the precise details of our approach, we first enumerate a number of issues that need to be considered in tackling this problem. Note that the optimization strategy described below considers the size of dimensions in forming processor groups. Clearly, this could easily be extended to include constraints based on knowledge of the query workloads on each dimension in addition to dimension size, further improving the grouping.

(1) We first make a remark regarding the computation of the sizes of, i.e., the number of processors in, the

different dimensional groups, denoted by $\mathcal{N}_1, \ldots, \mathcal{N}_d$. The fundamental goal here is to have group sizes large enough such that the entire dimension table fits into the aggregate memory of the processors comprising this group. Intuitively then, if the dimension table can be loaded in the aggregate memory of its processor group, the join computation can be done with only a single scan of the JI. It is, of course, possible that this goal cannot be achieved, given the available total number of processors. Then, based on the above mentioned criteria, the minimum size of the j^{th} processor group (i.e., G_j), $1 \leq j \leq d$, is given by $\mathcal{N}_j = \min\left(\mathcal{N}, \lceil \mathcal{S}_j/m \rceil\right)$, where m is the size of the main memory attached to each processor.[2] This assumes that all the processors have an equal amount of memory.

(2) Next we comment on the minimum size of the metric group, for which we use a different logic, as it does not participate in joins, unlike the dimensional groups. We choose to use the metric data volume relative to the entire data warehouse in order to determine the minimum size of G_{d+1}. In other words, $\mathcal{N}_{d+1} = \mathcal{N} \times \frac{\text{Aggregate Size of Metric PIs}}{\text{Total Size of the Data Warehouse}}$.

(3) Note that a processor may participate in more than one group. There may be many reasons for this. A trivial case would be when there are more dimensions than processors. A more likely case would be when the data sizes (dimensional, metric or both) are significant enough that the sum of the sizes of different groups (given the criteria outlined in items (1) and (2) above) may exceed the number of processors available, which would mandate the assignment of the same processor to different groups. This phenomenon adds the following requirement to the problem – the overlap of the processor groups must be minimized. We have developed an optimal solution to the processor group partitioning problem by formulating it as a constrained optimization problem solvable as a linear integer mathematical program. Due to space limitations, we are unable to include it in this paper; however, readers are referred to [7] for full details.

In the context of a data warehousing environment, where star schemas are common, the above-mentioned optimization strategy results typically results in the assignment of each processor to two groups, (a) a single dimension D_i, and (b) the fact table. This scenario is depicted graphically in Figure 4, and occurs because of the

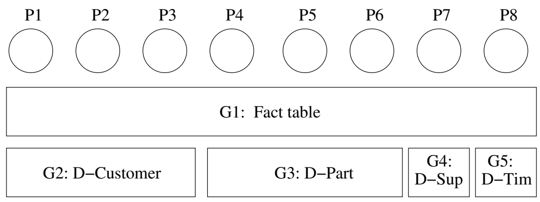

Figure 4: Example Processor Grouping for a Star Schema

relative difference in the sizes of dimensions versus the size of the fact table – the fact table is much larger than all the dimensions combined. The processor allocation shown in Figure 4 allocates processors to dimensions based on the relative size of the dimension; however, if query workload is known, processor allocation for dimensions could take this into account. Note, however, that even in this case, the fact table would still be allocated a full set of processors, since it is not only the largest table (by far), but it also participates in every join (all join paths lead through it).

3.1.2 Physical Data Placement

In this phase, the actual data are placed on the individual processors in the groups. To state our approach for data placement we will simply discuss the approach in the context of a single group. Consider processor group j, denoted by G_j, consisting of $\mathcal{N}_j$ processors, $P_1, P_2, \ldots P_{\mathcal{N}_j}$. The exact processor to group assignment is done by solving the optimization problem described previously. Clearly the contents of G_j include the PIs corresponding to the j^{th} dimension table and the associated JI, denoted by $\mathcal{J}_j$, from the fact table.

We first horizontally partition the JI in round robin fashion among the $\mathcal{N}_i$ processors. Our rationale for adopting the above-mentioned strategy has two salient features: (a) the JI partitioning strategy and (b) the dimensional replication strategy. The dimensional strategy is easy to understand. Replicating PIs across all processors in a group ensures that all JI records and their associated primary keys are co-located. Thus, all matching records satisfying any join predicate will be found on the same processor, ensuring that joins can be performed and outputs created efficiently in parallel. Note that this allocation scheme is not based on fixed sizes,

[2]To be precise, the memory should have space for loading (at least) one block of the JI. This " + 1" factor is not included here so as to avoid clutter.

but rather on relative sizes. Even though table sizes may change, tables in data warehouses tend to retain similar sizes relative to one another, thus allowing grouping on this basis.

An alternative dimensional strategy, partitioning the PIs across the processors of a group, was considered. However, this strategy presented two problems. First, since many (not necessarily co-located) JI records may point to the same PI record, some replication of PI data across processors would be required to ensure co-location of matching PI and JI records. Second, the RIDs in the PI records would change with partitioning, which would require updating the JI records to reflect the new RIDs they point to.

The objective of the JI partitioning strategy is to preserve the ordinal position mapping property which is threatened by partitioning JIs across different processors. It is easy to see that when the records are partitioned, it is important to regenerate the original ordinal position of a record, i.e., given a partitioned JI record i in processor j for group k, we want to be able to say that this record occupied the o^{th} ordinal position in the original, unpartitioned JI. This is important for several reasons, e.g., to form the final output of a join by putting together the output from various processor groups. Of the well known horizontal partitioning mechanisms (such as hash, range and round-robin), only the round robin policy is capable of naturally ensuring this mapping.

3.2 The Parallel Star Join Algorithm

In this section we present our algorithm to perform star joins in parallel. We assume a physical design strategy as described in Section 2 and a partitioning strategy as described in Section 3.1. We represent a general k-dimensional star-join query as follows.

Query 3 `SELECT` A_P^d, A_P^m `FROM` F, D_1, ..., D_k `WHERE` $P_{\bowtie}$ `AND` P_σ

Here $D_1, \ldots, D_k$ are the k dimensional tables participating in the join. P_σ and $P_{\bowtie}$ denote a set of restriction and join predicates respectively. 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.

Based on the partitioning strategy described earlier, a join query such as the one above will reduce to a number of one dimensional joins in each processor group. These can be performed in parallel. These smaller joins will produce *local groupwise rowsets* that will be processed to generate a *global rowset* which will be used to produce the final output. Accordingly, to describe our *Parallel Star Join* (PSJ) algorithm we will subdivide it into three phases: (a) The *Local Rowset Generation* (LRG) phase, (b) The *Global Rowset Synthesis* (GRS) phase, and (c) The *Output Preparation* (OP) phase.

3.2.1 Local Rowset Generation

In the LRG phase, each dimensional processor group generates a rowset (with fact table cardinality) representing the fact table rows that qualify based on the restrictions and join relevant to that group. This proceeds as follows. Consider dimensional processor group i, which consists of c processors and houses the PIs corresponding to D_i and the associated JI from the fact table, $\mathcal{J}_{\mathtt{i}}$. The restriction and join predicates that apply to dimension i, will be shipped to this group for evaluation. Let us assume, for simplicity, that group i receives, in addition to a i^{th} dimensional join predicate, a restriction predicate for a dimensional PI (note that more than one restriction predicate may be received in reality).

The first step of the LRG phase, *Load PI Fragments*, performed at each participating group i generates a dimensional rowset, $\mathcal{R}_{\mathtt{dim,i}}$, based on the restriction(s). This rowset is a bit vector of cardinality of D_i in which bits are set corresponding to rows in D_i that meet the restriction criterion. This rowset is developed in the following manner. First, each processor is allocated a range of the PI amounting to $\frac{1}{c}^{th}$ of the dimensional PI. For example, the first processor in the group loads records 1 to $\frac{|D_i|}{c}$, the second loads $\frac{|D_i|}{c} + 1$ to $\frac{2|D_i|}{c}$ and processor c loads $\frac{(c-1)|D_i|}{c} + 1$ to $|D_i|$. Then, each processor scans the fragment allotted to it, setting the corresponding bit(s) in the rowset for rows meeting the restriction. This process can easily be expanded to handle more than one restriction predicate by considering all the restrictions during the PI scan, and setting the corresponding rowset bit only for those records meeting all the restriction conditions.

The second step of the local rowset generation process, *Merge Dimension Rowset Fragments*, involves the merging of the restriction rowsets generated on each processor.

The restriction rowsets are merged in parallel via transmission through a binary tree structure of c leaf nodes, one for each processor of the group. The restriction rowsets are first merged pairwise, then the pairwise results are merged, and so on, until a final merged rowset is generated. We now describe the actual merging operation. This operation takes as input two rowsets of cardinality of the dimensional PI. Since each processor is allotted a non-overlapping set of PI records to examine for the restriction condition(s), each processor is responsible for a non-overlapping set of bits in the final restriction rowset. Thus, merging two rowsets involves simply OR-ing them together.

When the final restriction rowset has been generated, the next step, *Distribute Dimension Rowset*, takes place. Here, the final dimension rowset is transmitted back to the individual processors of the group through the same binary tree transmission mechanism through which the merging process took place.

Once the dimensional restriction rowset has been constructed and distributed, the next step, *Load JI Fragments*, loads the JI fragments allocated to group i in preparation for the creation of the local fact rowsets, $\mathcal{R}_{\texttt{fact,i}}$. This is a bit vector of cardinality of the fact table, where a bit is set if the corresponding row of the fact table satisfies the join condition for this dimension. The precise logic to set the bits in $\mathcal{R}_{\texttt{fact,i}}$ is given below. This procedure assumes that the rowset structure is already defined and initialized (i.e., the insertion pointer points to the first bit of $\mathcal{R}_{\texttt{fact,i}}$). The above discussion, for expository ease, assumes a centralized join in group i. In

Algorithm 1 Load JI Fragments Algorithm

start scanning $\mathcal{J}_{\texttt{i}}$ from the top
for each row j in $\mathcal{J}_{\texttt{i}}$ $(1 \leq j \leq |F|)$ do
  read the value of the current element, which yields a RID
  map this RID to an ordinal position in D_i, say k
  if the k^{th} bit in $\mathcal{R}_{\texttt{dim,i}}$ is set then
    set the j^{th} bit of $\mathcal{R}_{\texttt{fact,i}}$

reality, a segment of $\mathcal{R}_{\texttt{fact,i}}$ is generated at each processor of group i and then merged. Note that in order to perform this merging, the system needs to map ordinal positions of fragments at each physical processor into ordinal positions at the unpartitioned tables. This is done by simple arithmetic transformations - the details of this are given later in this section.

Finally, a note regarding group $d+1$, i.e., the metric group. If there exists one (or more) metric restriction(s) in the submitted query, then these are evaluated at this group and a rowset, i.e., $\mathcal{R}_{\texttt{fact,(d+1)}}$ constructed. Clearly, no join takes place here.

In the final step of the first phase, *Merge Partial Fact Rowsets*, the partial rowsets created on each processor are merged to form a local rowset, i.e., a rowset that represents the result of the *Local Rowset Generation* phase of the algorithm, i.e., local to each group.

3.2.2 Global Rowset Synthesis

In first step of the GRS phase, *Merge Local Fact Rowsets*, a global rowset, denoted by $\mathcal{R}_{\texttt{global}}$, is constructed by combining the rowsets $\mathcal{R}_{\texttt{fact,i}}$, for all i, generated in the LRG phase by each group. We remind the reader that each such rowset is simply a bit vector of fact table cardinality in which bits are set corresponding to records meeting the local restriction and join conditions. For a record to participate in the output of the query, it must meet all the restriction and join conditions, i.e., the corresponding bit must be set in *all* the rowsets $\mathcal{R}_{\texttt{fact,i}}$. Thus, the global rowset is simply the bitwise AND of all the local rowsets.

We generate the global rowset in a manner similar to the generation of the local restriction rowsets. The local rowsets are transmitted and merged through a binary tree construct in which the number of leaf nodes is equal to the number of dimensions participating in the join. The transmission portion of this operation is virtually the same as that of the local rowset generation operation, but the merge operation consists of a bitwise AND operation. The final rowset contains bits set only for records that meet all the join and restriction conditions, and should thus participate in the output of the query.

Once the global rowset has been generated, it is transmitted to those processor groups that participate in the output phase of the query in the second and final step of the GRS phase, *Distribute Global Rowset to Groups*. Each such group houses a dimension which contributes to the final output. For example, if `Customer.name` is an output column (identified by its presence in the SELECT clause of the query), then the group housing the customer dimension will participate in the OP phase.

3.2.3 Output Preparation

The OP phase is performed by each *participating processor group* (to be simply referred to as *participating group* henceforth) contributing a column of output and the eventual "racking" of these individual columns to produce the final output. These groups receive the global rowset computed in the GRS phase and in conjunction with the dimensional rowset already computed in the LRG phase and the PI(s) that contribute to the output, construct the final output column. For instance, consider the previous example where `Customer.name` is an output column. The corresponding participating group houses the PIs for the Customer dimension as well as the Customer JI from the fact table, denoted by $\mathcal{J}_{\mathtt{customer}}$. In this group, there will exist a PI on the `Customer.name` column, denoted by $\mathcal{P}_{\mathtt{cname}}$. Furthermore, assume there exists a dimensional rowset, denoted by $\mathcal{R}_{\mathtt{Customer}}$ that was computed in the LRG phase[3].

The first step in the OP phase, *Distribute Global Rowset to Processors*, involves the transmission of the global rowset to all processors of a participating group. In the next phase, *Load PIs*, the PI columns necessary for output are loaded.

When the global rowset, $\mathcal{R}_{\mathtt{global}}$, has been shipped to the customer dimensional group and all necessary data loaded, the following procedure, *Merge Output*, is executed, to construct the final output column. In describing this procedure we assume that the final output column will be encapsulated in a structure called *cust_name*. The procedure also assumes that the *cust_name* structure is already defined and initialized (i.e., the insertion pointer points to the first slot (or row) in the structure).

Algorithm 2 Merge Output Algorithm

start scanning $\mathcal{R}_{\mathtt{global}}$ from the top, i.e., the first bit
for each bit in $\mathcal{R}_{\mathtt{global}}$ do
  let the ordinal position of current bit in $\mathcal{R}_{\mathtt{global}}$ be denoted by i
  if the i^{th} bit (i.e., the current bit) in $\mathcal{R}_{\mathtt{global}}$ is set then
    read the i^{th} element of $\mathcal{J}_{\mathtt{customer}}$, which yields a RID of the primary key PI of the customer dimension
    map this RID to an ordinal position, say j
    read the element in the j^{th} position in $\mathcal{P}_{\mathtt{cname}}$
    insert this element into the *cust_name* structure
    move the insertion pointer of *cust_name* to the next insertion position

Again, note that the above description assumes (for ease of explanation), a "centralized" structure. In reality, however, each physical processor in a participating group would execute the above procedure and produce a fragment of the output column, which would then be merged to produce the "real" final output column. Thus, from a cost perspective (for details, see Section 6), one can think of the following phases: (a) *distributing global rowset to all processors*, (b) *loading PIs and JIs* at each processor in every participating group to produce the output (as indicated in the procedure above), (c) *merging the output fragments* produced by each individual processor in a participating group to produce a local output, i.e., one column of final output, and (d) *merging local outputs* to produce final output.

In a centralized system, a query is executed as a series of disk accesses – which load the relevant portions of the database to memory – interleaved with bursts of CPU activity, when the loaded data is operated upon. Mapping functions are required to determine the specific disk block that needs to be accessed and these depend on the index structure used. With this strategy, in most cases, the delays associated with the mapping computations will be negligible compared to the much slower storage access times [6]. This expectation is corroborated by other studies [22], which have shown that I/O related costs (disk access plus I/O related CPU costs) are several orders of magnitude more than other CPU costs relating to query processing. Based on these findings, in a centralized system one can focus on analyzing the query performance with respect to disk access delays.

In a parallel system, while the focus is still on the delays in obtaining needed data blocks, the difference (from centralized systems) arises from the fact that the required data can come from other nodes/processors as well as from the disk. Hence, in this paper, we are interested in response time (as measured in disk I/O delays as well as delays in obtaining the data from other nodes) and the volume of data transmitted between processors.

To aid the reader in understanding how the response time is computed, we provide a pictorial example of the PSJ algorithm at work. Consider a 2-dimensional join query that will be executed across two processor groups, G_1 and G_2, consisting of 2 processors each as shown in Figure 5 below. Essentially, this figure shows the various stages that occur in PSJ and the associated operations and time instants when each operation starts and ends. Note that we assume all processors start execution at the same time (time t_0 in the figure). Note further that we only consider those operations which result in data blocks arriving at a processor (either from disk or other processors) or leaving a processor.

Let us examine this process in detail by considering a specific processor in the figure, say P_3. P_3 first loads its PI fragment (denoted by LPF_3) – this activity ends at time t_1. Then it performs some CPU activity, as

[3]This is a valid assumption. If there were no restriction clauses in the submitted query based on the customer dimension, one can assume that all bits in $\mathcal{R}_{\mathtt{customer}}$ are set.

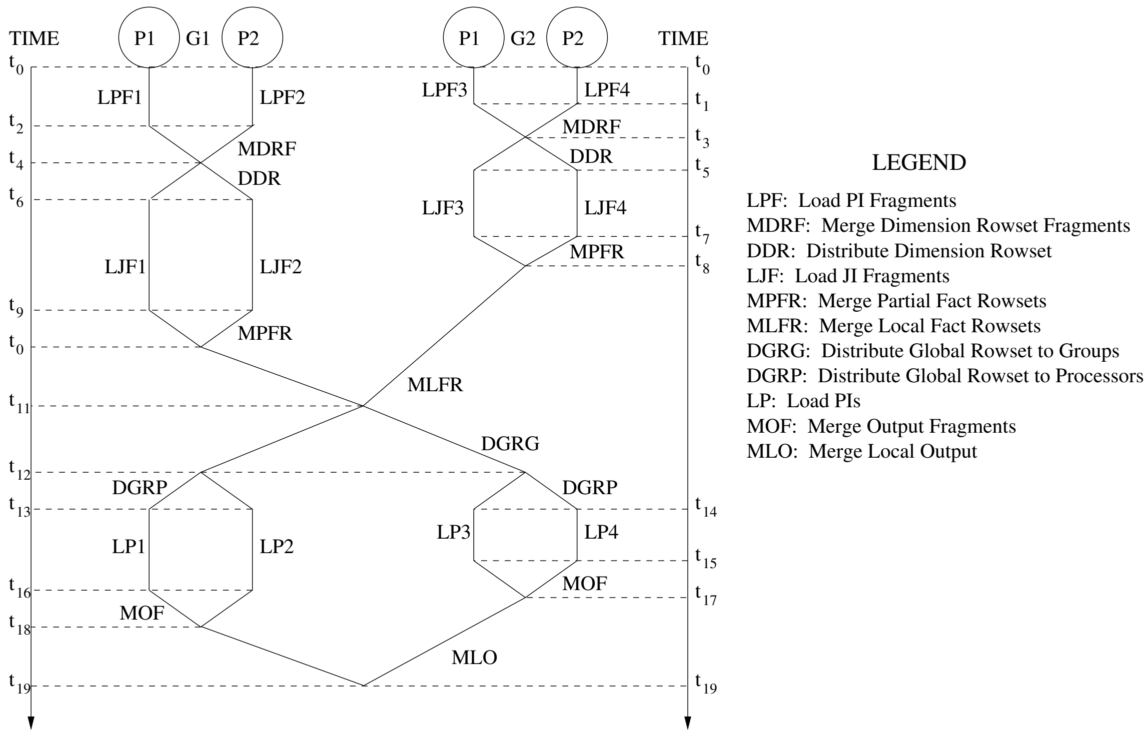

Figure 5: Response Time Computation Example for the PSJ Strategy

described before in the algorithm, to produce a dimensional rowset fragment based on the PI fragment fetched in the LPF step. This CPU activity does not show up in the figure for reasons already explained. The next cost phase consists of the merging the dimensional rowset fragments (MDRF) and subsequently distributing the full dimensional rowset (DDR) to all processors in G_2. This ends at t_5. Upon receipt of the full dimensional rowset, P_3 loads the JI fragment allocated to it (LJF_3). This step, finishing at t_7, is used to produce a partial fact table rowset (cost ignored as this is CPU activity) which is then merged with the other partial fact table rowsets produced by the other member of G_2, namely P_4. This shows up in the figure as the transmission cost MPFR. Note that until this time, P_3 was continuously busy either fetching data from disk or receiving/sending transmissions. At this point though, according to the scheme of figure 5 it must wait until the other group, namely G_1 finishes producing its local fact rowset. This occurs at time t_{10}, which signals the end of the *Local Rowset Generation* phase for this query. Now starts the *Global Rowset Synthesis* (GRS) phase, which requires the merging of the different local fact table rowsets (MLFR) into a global rowset and the subsequent distribution of this global rowset (DGRG) to the different groups. The DGRG phase ends at t_{12}, and indicates the end of the GRS phase. The *Output preparation* (OP) phase starts now where P_3 goes through the specific steps outlined in the algorithm. This consists of loading dimensional PI (LP_3), producing an output fragment which is then merged with the output fragments produced by other group members, producing a complete output column. The only non-CPU cost for this step is a transmission cost for merging the output fragments (MOF). Finally, all the individual output columns produced by the different participating groups are racked together to produce the final output, which requires a transmission step (MLO). The query finishes at t_{19}, which is the response time for the query.

We simulate this exact process in the performance experiments reported later (for varying number of dimensions and processors, of course). In order to extract the total I/O cost of the query we need to compute the costs for the various steps outlined (e.g., LPF_i, LJF_i, etc.). We have developed cost models for these steps, which are detailed in Section 6.

4 The Parallel BJI Join

We now consider a warehouse environment utilizing *Bitmapped Join Indexes* (BJIs). Join processing with BJIs is described in [22], and an overview is provided in [7]. Due to space limitations, we must refer the reader to this paper for a description of BJI join processing. Of interest in this area, however, is the memory requirement for BJI join processing. Here, all relevant columns and rows from the dimension tables (including the primary key column) are extracted from the dimension tables and pinned in memory. In [6], we have shown that the memory requirement for BJI, in blocks, is$\mathcal{M}_{JOIN}(BJI) = 1+|\mathcal{D}|+\Sigma_{D\in\mathcal{D}} \left\lceil \frac{|D|\times w(D)}{B} \right\rceil$ where $\mathcal{D}$ is the set of dimension table participating in the join, $w(D)$ refers the width of a tuple in a dimension table, and B is the size of a disk block. The first term in this expression corresponds to a block of memory for the fact table, the second term corresponds to a block of memory for each dimension table, and the third term corresponds to the memory required for pinning the relevant dimension tables in memory. Note that this imposes a significant memory requirement in the context of a data warehouse, leading to the potential for losses in join efficiency (see [7] for details).

In terms of processor grouping, the scheme is exactly the same as the PSJ scheme. This results in G_{d+1} processor groups, one for each dimension, plus one for the fact table. In terms of data placement, the following are loaded on each disk in processor group G_j: (1) For dimension D_j: $\frac{1}{N_j}^{th} of D_j$ (horizontal partition), BI for the resident fragment for each non-primary key attribute in D_j, Dimension fragment to fact table BJI, and B+Tree index for the primary key attribute of D_j. (2) For the fact table: The entire fact table (needed for OP phase) and BI for each attribute in the fact table.

The Parallel BJI Join algorithm has the same general structure as the PSJ algorithm; only the *Local Rowset Generation* phase, and the *Generate Partial Output Fragments* step of the *Output Preparation* phase differ. In the *Local Rowset Generation* phase, to generate the restriction rowset fragments, the BJI algorithm uses the BIs to perform restrictions, rather than PIs, as in the PSJ algorithm. For each (dimension or fact) table on which there is a restriction predicate, the following is done on each processor of each processor group. For each restriction predicate, the BI for the attribute referenced in the predicate is traversed, and a partial restriction rowset constructed. For each distinct attribute value meeting the restriction, the bitmap pointed to by the leaf node that represents that attribute value is loaded. All loaded bitmaps are bitwise ORed. The result is a bitmap of size $|D_j|$ (for dimensions) or F (for the fact table), where each set bit represents an attribute value that meets the restriction for attribute values in the resident fragment. Subsequently, D_j is joined to the fact table as follows. For each bit set in the dimension restriction rowset of D_j, the corresponding value in the BJI is found, and the bitmap pointed to by that leaf node is loaded into memory. All loaded bitmaps are merged into a single partial fact rowset using bitwise OR. This results in a bitmap of size $|F|$, where each bit set represents a tuple in the fact table that meets the restriction predicates on attributes in D_j for the resident fragment. Within the *Output Preparation* phase, the only step that is different for the BJI approach is the *Generate Partial Output Fragments* step. Recall that each bit set in the global rowset translates to a row of output. Output for participating dimensions is produced as follows. Space is allocated for all output rows, with an appropriate width for holding the projected attribute values from D_j. The blocks containing the fact table tuples corresponding to the set bits in the global rowset are loaded[4] and the needed dimensional key values are projected from the loaded tuples. For each key value, the appropriate disk block is accessed next using the index. Here, we assume a B+tree index, since it performs better for dense indexes. Once this rowset is produced, processing proceeds in the same fashion as in PSJ.

Effectively, using the approach outlined above, a one-dimensional join is performed at each processor. In this context, the BJI memory requirement equation above may be restated as follows: at a processor housing dimension i (D_i), the amount of memory required to perform the BJI-based join as outlined previously is given by the following equation: $\mathcal{M}_{JOIN}(BJI) = 1 + 1 + \left\lceil \frac{|D_i|\times w(D_i)}{B} \right\rceil$.

5 The Parallel Hash Join

Here, we describe a parallel hash join strategy (based on work in [4, 5]), a well-known join strategy based on the conventional relational data model using segmented right-deep trees and pipelining, and apply the technique in a data warehousing environment. In the remainder of the paper, we will refer to this algorithm as the HASH algorithm.

The HASH algorithm provides a means of performing multi-way hash joins, using pipelining techniques to improve performance. This technique assumes a shared-disk architecture, whereas we assume a shared-nothing architecture for the other algorithms in this paper. Rather than embarking on a discussion of optimal data placement policies for the HASH algorithm (the authors of [4] note the difficulty of this problem; it is well beyond

[4]To obtain a simpler cost model, but one that affords BJI join a significant advantage when we evaluate its performance, we make a simplifying assumption for the dimension output generation phase - we assume that the bits set in the fact table are clustered, i.e., the tuples are co-located in sequential disk blocks.

the scope of this paper), we retain the shared-disk assumption for the HASH algorithm in our discussions and performance comparisons. *Note that this confers a significant advantage to the HASH algorithm, by removing the need for additional data transfer phases.* It also removes the need for an a priori data placement and processor allocation phase; since each processor can access data on any disk, data placement is not an issue, and processor allocation can be computed on the fly for each query plan. We note that we assume an ideal hash function for each stage, such that the load is distributed evenly across all processors allocated to a stage. Due to space considerations, we refer the reader to [4] for the full details of the HASH algorithm (an overview is also provided in [7]), and move on to discuss the cost models for each algorithm.

6 Cost Model

In this section, we develop cost models for the PSJ, BJI, and HASH algorithms. Throughout this discussion, multiplication and division by the constant value 8 denotes conversion from bits (the unit of transmission measurement) to bytes (the unit of disk access measurement) and from bytes to bits, respectively. The notation used in this paper is summarized in Table 1.

COST MODEL FOR PSJ

(1) *Local Rowset Generation (LRG)*

(1a) *Load PI Fragments (LPF)*: Disk access cost to load $\frac{1}{\mathcal{N}_i}^{th}$ of the PIs needed for restriction in a single processor group (in blocks) is $\left\lceil \frac{\frac{|D_i|}{\mathcal{N}_i} \times \sum W_{a_{ij}}}{B} \right\rceil$, where $a_{ij} \in A_R$. Disk access cost to load $\frac{1}{\mathcal{N}_i}^{th}$ of the PIs needed for restriction in the metric data group (in blocks) is $\left\lceil \frac{\frac{|F|}{\mathcal{N}_F} \times \sum W_{a_{Fj}}}{B} \right\rceil$, where $a_{Fj} \in A_R$.

(1b) *Merge Dimension Rowset Fragments (MDRF)*: Transmission cost to merge dimension restriction rowset fragments into a single dimension restriction rowset in a single processor group is $log_2\mathcal{N}_i \times |D_i|$. Transmission cost to merge metric restriction fragments into a single metric group restriction rowset is $log_2\mathcal{N}_F \times |F|$.

(1c) *Distribute Dimension Rowset (DDR)*: Transmission cost to distribute dimension rowset to all members of a processor group is $log_2\mathcal{N}_i \times |D_i|$. Transmission cost to distribute metric restriction rowset to all members of a processor group is $log_2\mathcal{N}_i \times |F|$

(1d) *Load JI Fragments (LJF)*: Disk access cost to load $\frac{1}{\mathcal{N}_i}^{th}$ of the JI in a single processor group (in blocks) is $\left\lceil \frac{\frac{|F|}{\mathcal{N}_i} \times W_{JI}}{B} \right\rceil$.

(1e) *Merge Partial Fact Rowsets (MPFR)* Transmission cost to merge partial fact rowsets in a single processor group is $log_2\mathcal{N}_i \times |F|$.

(2) *Global Rowset Synthesis*

(2a) *Merge Local Fact Rowsets (MLFR)*: Transmission cost to merge all local fact rowsets is $log_2 D_J \times |F|$.

(2b) *Distribute Global Rowset to Groups (DGRG)*: Transmission cost to distribute the global rowset to all groups participating in the output phase is $log_2 D_P \times |F|$.

(3) *Output Preparation (OP)*

(3a) *Distribute Global Rowset to Processors (DGRP)*: Transmission cost to distribute the global rowset to all processors in a group (in bits) is $log_2 N_i \times |F|$. Transmission cost to distribute the global rowset to all processors in the metric group (in bits) is $log_2 N_F \times |F|$.

(3b) *Load PIs (LP)*: Disk access cost to load all PIs involved in output for a single dimension. Let B represent the PI cost $\left\lceil \frac{|D_i| \times \sum W_{a_{ij}}}{B} \right\rceil$, where $a_{ij} \in A_P$, J represent the JI cost $\left\lceil \frac{\frac{|F|}{\mathcal{N}_F} \times \sum W_{JI}}{B} \right\rceil$, R represent the Rowset cost $\left\lceil \frac{\frac{|F|}{\mathcal{N}_F \times 8} \times \sum W_{JI}}{B} \right\rceil$, and OU represent the Output cost $\left\lceil \frac{|O| \times \sum W_{a_{ij}}}{B} \right\rceil$, where $a_{ij} \in A_P$. If either $B < \frac{S_m}{\mathcal{N}_i}$ or $J < \frac{S_m}{\mathcal{N}_i}$, i.e., if either the PI or the JI fragment fits in memory, then the cost is $B + J + R + OU$. Otherwise, either the JI or the PI must be loaded multiple times to perform the join. If $B > J$, the cost is $(B \times J) + J + R + OU$. Otherwise, when $B < J$, the cost is $B + (J \times B) + R + OU$. Disk access cost to load all PIs involved in output for the metric table (in blocks) is $\left\lceil \frac{\frac{|F|}{\mathcal{N}_F} \times \sum W_{a_{Fj}}}{B} \right\rceil + \left\lceil \frac{\frac{|F|}{\mathcal{N}_F \times 8}}{B} \right\rceil a_{Fj} \in A_P$.

(3c) *Merge Output Fragments (MOF)*: Let O represent the output relation, and $|O|$ be its cardinality. Transmission cost to merge output from single processors into a local group output on a single processor group is $\frac{|O|}{\mathcal{N}_i} \times \sum(W_{a_{ij}} \times 8)$, where $a_{ij} \in A_P$. Transmission cost to merge output within the metric processor group is $\frac{|O|}{\mathcal{N}_F} \times \sum(W_{a_{Fj}} \times 8)$, where $a_{Fj} \in A_P$.

(3c) *Merge Local Output (MLO)*: Transmission cost to merge output from local groups into a single final output. Like the costs for merging the outputs within a processor group, the costs here assume serial transmission to a single target $|O| \times (\sum(W_{a_{ij}} \times 8) + \sum(W_{a_F j} \times 8))$, where $a_{ij}, a_{Fj} \in A_P$.

COST MODEL FOR BJI

(1) *Local Rowset Generation (LRG)*

(1a) *Load BI Fragments (LBF)*: Disk access cost to load index and bitmaps for $\frac{1}{\mathcal{N}_i}^{th}$ of dimension D needed for processing a single restriction predicate in a single processor group (in blocks) is $\lceil log_{P_i} V_i - 1 \rceil + \left(\frac{V_i}{K_i N_i}\right) + f\left(\frac{\varsigma |V_i|}{8BN_i}\right)$. Disk access cost to load index and bitmaps for $\frac{1}{\mathcal{N}_i}^{th}$ of the fact table needed for processing a single restriction predicate in a single processor group (in blocks) is $\lceil log_{P_F} V_F - 1 \rceil + \left(\frac{V_F}{K_F N_F}\right) + f\left(\frac{\varsigma |F|}{8BN_F}\right)$. Here, the costs for index access are taken from [6]

(1b) *Merge Dimension Rowset Fragments (MDRF)*: Transmission cost to merge dimension restriction rowset fragments into a single dimension restriction rowset in a single processor group is $log_2 \mathcal{N}_i \times |D_i|$. Transmission cost to merge metric restriction fragments into a single metric group restriction rowset is $log_2 \mathcal{N}_F \times |F|$.

(1c) *Distribute Dimension Rowset (DDR)*: Transmission cost to distribute dimension rowset to all members of a processor group is $log_2 \mathcal{N}_i \times |D_i|$. Transmission cost to distribute metric restriction rowset to all members of a processor group is $log_2 \mathcal{N}_F \times |F|$

(1d) *Generate Partial Fact Rowsets (GPFR)*: Disk access cost to traverse BJI and load appropriate RIDs for $\frac{1}{\mathcal{N}_i}^{th}$ of a dimension in a single processor group (in blocks) is $\left(\frac{\varsigma |D_i|}{N_i} \lceil log_{P_i} V_i - 1 \rceil\right) + \left(R \frac{\varsigma |D_i|}{N_i}\right)$. Here, the costs for index access are taken from [6]

(1e) *Merge Partial Fact Rowsets (MPFR)*: Transmission cost to merge partial fact rowsets in a single processor group is $log_2 \mathcal{N}_i \times |F|$.

(2) *Global Rowset Synthesis*

(2a) *Merge Local Fact Rowsets (MLFR)*: Transmission cost to merge all local fact rowsets is $log_2 D_J \times |F|$.

(2b) *Distribute Global Rowset to Groups (DGRG)*: Transmission cost to distribute the global rowset to all groups participating in the output phase is $log_2 D_P \times |F|$.

(3) *Output Preparation (OP)*

(3a) *Distribute Global Rowset to Processors (DGRP)*: Transmission cost to distribute the global rowset to all processors in a group (in bits) is $log_2 N_i \times |F|$. Transmission cost to distribute the global rowset to all processors in the metric group (in bits) is $log_2 N_F \times |F|$.

(3b) *Generate Partial Output Fragments (GPOF)*: Let O represent the output relation, and $|O|$ be its cardinality. Generate dimensional output fragment by loading the fact table blocks for dimension keys, traversing the B+tree for each key value, and loading the blocks containing the dimension tuples): $\left(|O| \times \frac{|F|}{T_{BF}}\right) + \left(|O| \times \left\lceil log_{P_i} \frac{V_i}{N_i} - 1 \right\rceil\right) + min\left(\frac{|O|}{N_i}, \frac{|D_i|}{T_{Bi} N_i}\right)$. Disk access cost to load metric data for metric output fragments: $min\left(\frac{|O|}{N_F}, \frac{|F|}{T_{BF} N_F}\right)$. Here, the costs for index access are taken from [6]

(3c) *Merge Output Fragments (MOF)*: Transmission cost to merge output from single processors into a local group output on a single processor group is $\frac{|O|}{\mathcal{N}_i} \times \sum(W_{a_{ij}} \times 8)$, where $a_{ij} \in A_P$. Transmission cost to merge output within the metric processor group is $\frac{|O|}{\mathcal{N}_F} \times \sum(W_{a_{Fj}} \times 8)$, where $a_{Fj} \in A_P$.

(3d) *Merge Local Output (MLO)*: Transmission cost to merge output from local groups into a single final output. Like the costs for merging the outputs within a processor group, the costs here assume serial transmission to a single target $|O| \times (\sum(W_{a_{ij}} \times 8) + \sum(W_{a_F j} \times 8))$, where $a_{ij}, a_{Fj} \in A_P$.

COST MODEL FOR HASH[5]

(1) *Table Building*

(1a) *Load dimension data for segment (LDD)*: Disk access cost to load dimension data for a segment is the *max* across all dimensions (where data is loaded in parallel), given the processor allocation for the segment, and assuming a B+ tree index on restriction attributes: $max_i \left(\lceil log_{\mathcal{N}_i} V_i - 1 \rceil + \left\lceil \frac{|D_i| \times \sum W_{a_i}}{\mathcal{N}_i B} \right\rceil\right)$, where D_i is in the segment.

(1b) *Transmit dimensional data (TDD)*: Transmit dimensional data to appropriate processor, according to

[5]We begin our discussion of the costs with the table building phase, and ignore the costs of the preliminary phase, where the segments and processor allocation are determined. Since the HASH algorithm repeats over segments, we model the cost of executing a segment in phases (1) and (2), while phase (3) considers the cost of generating the final output.

the partitioning function. Assumes a uniform distribution of data across disks[6], where $\frac{1}{\mathcal{N}_i^{th}}$ of a group's data is loaded on the appropriate processor in the *LDD* phase: $\Sigma_{j=1}^{\mathcal{N}_i} \left\lceil \varsigma|D_i| \times (\sum W_{a_{RP}} \times 8) - \frac{\varsigma|D_i| \times (\sum W_{a_{RP}} \times 8)}{\mathcal{N}_i} \right\rceil$, where $W_{a_{RP}} \in A_P$ or $W_{a_{RP}} \in A_R$.

(2) *Tuple Probing*

(2) *Load probing table (LPT)*: Let Q be the probing table, i.e., the fact table for the first segment, or the intermediate result from the preceding segment. Let W_Q be the width of Q in bytes. Then the cost to load the probing table from disk is as follows: $\left\lceil \frac{|Q| \times W_Q}{\mathcal{N}B} \right\rceil$.

(2b) *Transmit probing data (TPD)*: The cost to transmit probing table data to the appropriate processor, where $frac1\mathcal{N}^{th}$ of a data is loaded on the appropriate processor in the *LDD* phase: $\left\lceil (|Q| \times W_Q \times 8) - \frac{|Q| \times W_Q \times 8}{\mathcal{N}} \right\rceil$

(2c) *Save intermediate results (SIR)*: Let I be the set of intermediate results of the segment to be saved to disk for use in the succeeding segment. Let W_I be the width of I in bytes. The cost to save I to disk is as follows: $\left\lceil \frac{|I| \times W_I}{\mathcal{N}B} \right\rceil$

(3) *Generate output*

(3a) *Transmit final output (TFO)* Let O represent the set of final results. The cost to transmit the final output to a single processor, where $\frac{1}{\mathcal{N}^{th}}$ of a data is located on the appropriate processor after the *TPD* phase, is as follows: $\left\lceil (|O| \times W_I \times 8) - \frac{|O| \times W_I \times 8}{\mathcal{N}} \right\rceil$

7 System Model for Performance Comparison

In this section, we describe our performance analysis model, focusing on the three major components: the database, the database server, and the query.

Database Model: Our database is organized in a star schema, i.e., there is a single fact table and *NumDimensions* dimension tables. For the purposes of these experiments, we assume the database schema depicted in Figure 1B. We chose to base our experiments on the TPC-D benchmark specifically because it is a well-known and well-accepted benchmark, designed to represent a "generic" data warehouse and serve as the basis for comparison tests. As such, we believe that it provides a sufficient basis for comparison tests, and that adding other databases to the empirical study would increase the length of the paper, without significant benefit.

We assume a static dataset, i.e., there are no updates to the database during the experiments, and that referential integrity exists from the fact table to each dimension table. The size of the data in the database is increased or decreased through the use of a *ScaleFactor* parameter, as is done in the TPC-D benchmark. A *ScaleFactor* of s corresponds to a database size of 86s MB.

Database Server Model: Our database server has a shared-nothing architecture consisting of *NumProcessors* processors. Each processor is associated with *MemorySize* bytes of RAM. The processor group assignment and data placement are static, i.e., the assignments are made prior to the start of the experiments (according to the scheme outlined for each type of physical design) and remain constant over the course of the experiments.

Query Model: Queries, in our model, are characterized in terms of four basic parameters: *NumJoinDimension,* $Selectivity_i$, *NumRestrictions* and *NumProjections*. The *NumJoinDimension* parameter denotes the number of dimensions that will participate in the star join query. The actual dimension instances that will be picked are chosen randomly from among the set of dimensions. The selectivity of a dimension i in the set of join dimensions of a query, i.e., the fraction of records returned based on an equality based restriction predicate, is denoted by $Selectivity_i$, and is drawn from a uniform distribution in the range *[MinSelectivity,MaxSelectivity]*. The selectivity of the metric table, denoted by $Selectivity_F$, is generated in a similar fashion. The number of restriction predicates in a query, denoted by *NumRestrictions*, is generated from a truncated normal distribution in the range *[MinNumRestrictions,MaxNumRestrictions]*, where *MinNumRestrictions* is 0 and *MaxNumRestrictions* is the total number of attributes in the dimensions involved in a given query, with a mean of *MeanNumRestrictions* and a standard deviation of *StdNumRestrictions*. A set of *NumRestrictions* attributes, denoted by *RestrictionSet*, is then selected randomly from among the attributes of the set of join dimensions and the metric attributes. The number of projection attributes, denoted by *NumProjections*, is generated from a truncated normal distribution in the range *[MinNumProjections,MaxNumProjections]*, where *MinNumProjections* is 1 and *MaxNumProjections* is the total number of attributes in the dimensions involved in a given query, with a mean of *MeanNumProjections* and a standard deviation of *StdNumProjections*. A set of *NumProjections* attributes, denoted by *ProjectionSet*, is then selected randomly from among the attributes of the set of join dimensions and the metric attributes.

[6]This represents a best case scenario, where data loading in parallel minimizes I/O costs.

8 Results of Performance Comparison

After discussing the performance metrics, we present the results of the experimental study of the Parallel Star Join algorithm, the Bitmapped Join Index Algorithm, and the Pipelined Hash Join algorithm. We begin with a discussion of our two primary performance metrics:

Response Time in Block Access (RTBA): RTBA represents the response time of a query measured in units of block accesses. As explained in Section 6, I/O costs overwhelmingly dominate overall query processing costs, and so we have chosen to characterize response times with the RTBA measure. RTBA is computed using the cost model for the PSJ algorithm as outlined in detail in Section 6 and demonstrated pictorially in Figure 5 (in this figure the RTBA of the query is $t_{19} - t_0$). The same figure also illustrates the cost model for the BJI algorithm, though the steps have different cost attributes. The costs incurred by the various steps of the three phases of the PSJ and BJI algorithms as well as by the two phases (per segment) of the HASH algorithm are described in Section 6. Note that whereas RTBA provides a good indication of the overall response times, it does not take into account the effects of sequential vs. random I/O.

Aggregate Data Transmission (ADT) cost: ADT represents the total number of blocks of data transmitted between all the processors in the course of performing a query. We initially considered a combined metric that would consider both disk costs as well as network costs in a single metric. Such a metric would be possible if I/O and transmission steps were guaranteed not to overlap at all. However, our algorithm allows for the overlap of I/O and data transmission, e.g., when a processor completes the Load PI Fragments step of the PSJ algorithm, it need not wait for all other processors is the system to complete this step before sending its results on for the Merge Dimension Rowset Fragments phase.

All curves presented in this section exhibit mean values that have relative half-widths about the mean of less than 10% at the 90% confidence level. Each experiment was run for 400 queries. We only discuss statistically significant differences. Table 2 shows the values of the parameters used in our experiments. We next present the results of scalabilty experiments comparing PSJ, BJI, and HASH. Due to space considerations, we are unable to include the results of the corresponding speedup experiments. For these results, the reader is referred to [7].

Parameter	**Value**	**Parameter**	**Value**
Num Dimensions	4	*MaxSelectivity*	0.5
ScaleFactor	20 to 1280	*MeanNumRestrictions*	0.25
NumProcessors	8 to 1024	*StdNumRestrictions*	0.01
MemorySize	8MB to 1024 MB	*MeanNumProjections*	0.5
BlockSize	512KB	*StdNumProjections*	0.01
MinSelectivity	0.1		

Table 2: Table of Parameter Values

8.1 Scalability Experiments

We explore the scalability properties of PSJ, BJI and HASH. As is common in such experiments, we explore how the cost curves scale as system resources are increased, with a concomitant increase in warehouse size. We report on scalability, in this paper, with respect to increasing processor counts. For each experiment, we measure how RTBA and ADT costs scale as more processors are added to the system. Simultaneously, the size of the warehouse is increased at the same rate by increasing the scale factor of the database. Specifically, we vary the number of processors between 8 and 1024 with associated scale factor variations from 10 through 1280. Note that a scale factor of 10 represents a warehouse of size 860 MB while scale factor of 1280 represents a 1.4 TB warehouse.

Each plot for PSJ and HASH in the next two figures corresponds to an experiment assuming a specific amount of memory per processor – the specific values of the memory size parameter are shown in the caption of the figure. The case for BJI is somewhat different. As discussed earlier in Section 4, the BJI algorithm needs a certain minimum amount of memory to work well. This minimum memory size is given in Section 4. For the BJI experiments, we have assumed that each processor has the requisite memory available to perform its local join operation using BJIs. In the experiments conducted here, this memory requirement varies between 1.36 MB (for joining the `TIME` dimension to the fact table with a *ScaleFactor* of 20 and a *Processor Count* of 1024) and 807 MB (for joining the `CUSTOMER` dimension to the the fact table with a*ScaleFactor* of 1280 and a *Processor Count* of 64).

RTBA Results: Figure 6[A] shows RTBA curves for PSJ and HASH corresponding to memory sizes of 16 MB and 64 MB, as well as a curve for BJI join. As explained in the previous paragraph, BJI join needs a requisite amount of memory to work. In the schema used in these experiments (outlined in Figure 1B), the largest dimension is the `CUSTOMER` table. At each processor housing the `CUSTOMER` dimension, the maximum amount of memory required turns out to be 807 MB. Note that this memory requirement is much larger than the amounts provided to the PSJ and HASH algorithms.

We first remark on the general properties of the curves. The curves rise as the scale factor and the number of processors increase. In other words, PSJ, BJI, and HASH have less than perfect scaling. The reasons for this behavior are easy to identify. Let us first examine the PSJ algorithm. In PSJ, as can be easily seen from the cost model, the *output preparation* (OP) phase cost dominates the other costs combined. In this phase, for a given output column (i.e., in the context of a single participating group), the relevant projection index (PI) is scanned at each processor in the group. Recall that the guiding principle of the PSJ algorithm was that joins should be performed with at most a *single* scan of the JI across all the processors in a group. The motivation for this of course is that JIs are much, much larger than PIs and we wanted to minimize the number of JI scans required. To achieve this though, one would potentially need to scan the PI several times (unless the entire PI could be pinned in memory). It can be easily shown that in the worst case, a PI may need to be scanned $\frac{S_j}{M}$ number of times at each processor in the group, where S_j is the size of the JI fragment at each processor and M is the memory size. This translates to an I/O cost of $\frac{S_j}{M} \times S_b$ where S_b is the size of the PI. As the number of processors is increased, and the size of the warehouse increases proportionally, S_j remains constant while S_b increases in proportion to the size increase. Put another way, since the PIs are replicated at all processors, adding processors has no impact on PI scan cost, which keeps on increasing. As a result, there is an increase in access cost when processors are added.

The increase in BJI cost is due to the output preparation costs that do not scale linearly. Each processor in a dimension group must scan all the qualifying tuples of the fact table (i.e., all the tuples indicated by the global fact table rowset) to find the dimension keys needed to produce dimensional output fragments, regardless of the number of processors in the group. Since this cost cannot be distributed over all processors in a group, it results in sub-linear scaling.

The increase in HASH cost is also simple to explain. Because the sum of the sizes of participating dimension relations is larger than the aggregate size of memory, the query must be divided into multiple pipeline segments. At the end of each segment, the intermediate results must be saved to disk, and then reloaded (as the probing table) at the beginning of the subsequent segment. These results must be written out to disk from the processors serving the final relation of the segment, i.e., this I/O cost cannot be spread across all processors in the system. This results in the sub-linear scaling behavior we see for HASH in Figure 6[A].

Now we turn our attention to comparing the PSJ, BJI, and HASH cost curves. The first observation, of course, is that PSJ costs are an order of magnitude lower than BJI and HASH costs (note that the Y-axis is plotted in log scale). This is due to the fact that our data placement exploits the natural partitionability of star schemas and we only access data that is needed.

In contrast, both BJI and HASH must access entire tuples, even though only a portion of the attributes may be needed. This leads to higher costs for BJI and HASH over PSJ. BJI costs are higher than HASH because each processor must scan the entire fact table for dimension keys in the output preparation phase (as mentioned at the beginning of this section), while HASH spreads the cost of scanning the fact table (as well as the cost of loading the new probing table for each segment) across all processors.

Another notable observation is that at larger memory sizes, PSJ approaches “near-perfect” scalability. For instance, at memory size of 1024 MB, the PSJ curve is almost horizontal. This is due to the fact that greater the memory size, larger the PI that can be pinned in memory (as explained a few paragraphs ago), and consequently lower the PI scan cost in the OP phase.

ADT Results: Because for PSJ, BJI, and HASH, ADT costs are independent of the memory size (denoted by S_m in the cost functions in Section 6) parameter, we only report the results of a single ADT experiment. Figure 6[B] shows curves for transmission costs for both HASH and PSJ/BJI. We note that the transmission costs for BJI are exactly the same as those of PSJ, so the curves overlap. Hence, the remainder of this section, we will compare PSJ and HASH transmission costs, with the understanding that PSJ and BJI costs are exactly the same.

We first note that both HASH and PSJ exhibit imperfect scalability, i.e., transmission volumes increase at a greater rate than the rate of increase of the warehouse volume. In PSJ, this occurs as increasing the size of the warehouse results in greater amounts of data that must be transmitted across groups. As the size of the warehouse and the number of processors increase, the number of groups remain the same but processors per group increase. Since the processors in a group exchange rowsets in order to generate a group result (as described in Section 3, more rowsets must be transmitted to generate a result than would be the case with fewer processors, resulting in

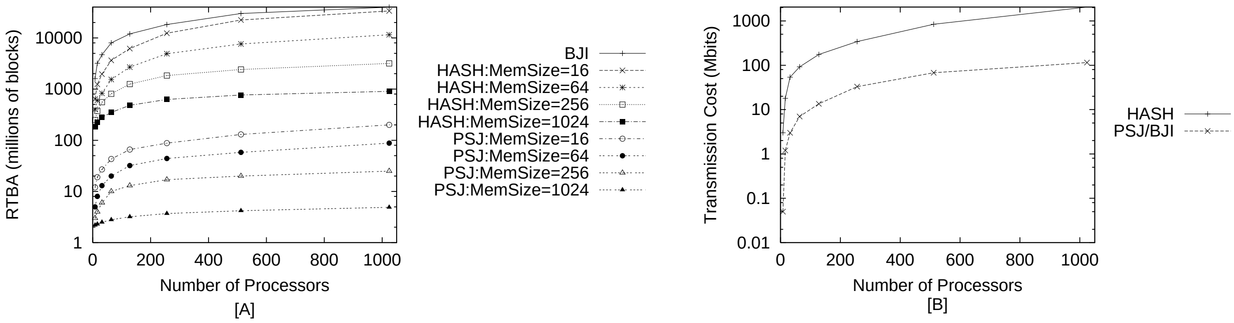

Figure 6: [A]Disk Access cost for PSJ, BJI, and HASH Scalability as CountProc and ScaleFactor Increase; [B] Transmission cost for PSJ, BJI, and HASH Scalability as CountProc and ScaleFactor Increase

sub-linear transmission costs.

In HASH, the reasons for the sub-linear scalability of transmission costs are similar to those of PSJ. As in PSJ, as the size of the warehouse and the number of processors increase, the number of joins per segment remain the same but number of processors allocated to processing a particular group within a segment increases. In the table-building phase, each processor within a group of $\mathcal{N}_i$ processors loads $\frac{1}{\mathcal{N}_i^{th}}$ of the data for dimension D_i, hashes each tuple, and transmits it to the appropriate processor. Here, only $\frac{1}{\mathcal{N}_i}$ of those tuples will be co-located on the appropriate processor at load time, i.e., $\frac{\mathcal{N}_i-1}{\mathcal{N}_i}$ of the tuples must be transmitted to other processors. Here, as the number of processors increases, more tuples must be transmitted to other processors at table-building time. This results in sub-linear scaling of HASH transmission costs.

We now compare the relative magnitudes of the HASH and PSJ curves. HASH has substantially higher transmission costs than PSJ – this is because HASH needs to exchange actual data tuples within groups at table-building time, and across groups within a segment during the probing phase. In contrast, PSJ represents intermediate results with a smaller rowset structure, resulting in lower overall transmission costs.

9 Related Work

The research related to our work includes the areas of data warehousing/OLAP and parallel database processing. For this reason, we review related work in both areas, as well as the work that combines the two fields. We will briefly review the work related to query performance since it is most relevant to the work in this paper. Two main approaches have been proposed to improve 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* [3]. 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 [14]. The work in indexing strategies includes traditional approaches, such as tree-based indexing, as well as non-traditional approaches, such as positional indexing, which has been proposed in [22, 6]. As stated earlier, the physical design strategy underlying PSJ exploits many of these approaches in the context of the Star Schema.

A large body of work exists in applying parallel processing techniques to relational database systems (e.g., [8, 26, 28, 25, 19, 1]). From this work has emerged the notion that highly-parallel, shared-nothing architectures can yield much better performance than equivalent closely-coupled systems [24, 17, 9]. Indeed, may commercial database vendors have capitalized on this fact [10]. Our focus on shared-nothing systems is also motivated by this fact. Various methods have been developed over the years to distribute data across sites, including *hash-* or *range-partitioning* based on a single key. This approach is supported by a number of database vendors (e.g., Oracle, Informix and NCR).

A related body of work considers shared disk or shared memory architectures, e.g., [4, 18, 5]. This work considers hash-based join techniques, and the effects of pipelining and segmenting right-deep trees. While this work has produced several good techniques, these schemes are unlikely to be widely used in practice due primarily to two considerations. The first consideration is cost: shared-resource architectures require expensive, specially designed parallel machines. Adding new components (e.g., processors) to such machines requires purchasing components specifically designed for the machine, rather than less-expensive commodity components. The second consideration is scalability and resource contention. Here, all processing units access the same pool of shared resources – additional processing units will clearly increase contention for the pool of shared resources. With the exception of [16], whose major thrust is efficient parallel hardware to support warehousing, we were able to find very little in the published (academic) literature related to parallelism in data warehouses. However, there are several DBMS vendors that claim to support parallel warehousing products to various degrees. We now focus on some of these commercial products. In [2], the architecture and design of the Teradata system is outlined. This is the most directly related work to our own that we were able to find, and there are two significant differences. First, Teradata relies on a variant of the hash join algorithm, to which we compare our algorithms in this paper. Second, Teradata assumes a shared-memory environment, while we consider a shared-nothing architecture. In [11], the parallel processing techniques used in the Red Brick Warehouse product are described. Intended for symmetric multiprocessing platforms (SMP), Red Brick Warehouse utilizes *STAR indexes* to perform single-pass joins of 2 to 10 tables. A STAR index is a proprietary multidimensional join index that allows more than two tables to be joined in a single operation [23]. A parallel version of this algorithm performs horizontal partitioning so that a single star-join is allocated among 2 to 32 processors. More recently, Red Brick has introduced the Red Brick Warehouse xPP. This product is designed to take advantage of multiple nodes in massively parallel processing (MPP) architectures. In [13], the parallel processing model used in the Oracle Parallel Warehouse Server is described. This product is based on a processing model that incorporates both horizontal and pipelined parallelism.

A potential limitation of the techniques mentioned above is that they rely primarily on traditional horizontal partitioning methods to achieve scalability. Whereas such partitioning does not fully exploit the dimensionality of the data in a data warehouse, the approach outlined in this paper attempts to exploit the multidimensionality inherent in a warehouse environment. It exploits the “natural” task partitioning made possible by the star schema while allowing traditional horizontal partitioning to be used to achieve much greater efficiency in query processing than current techniques can provide.

10 Conclusion

In this paper we have presented a framework for applying parallel processing to OLAP systems. Our physical design strategy takes advantage of the efficient partitioning suggested by the star schema representation of a data warehouse. Specifically, we proposed a declustering strategy which incorporates both task and data partitioning. We 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. Based on a detailed cost model we determined the response times achievable for star join queries and using a simulated environment evaluated the performance of our Parallel Star Join algorithm.

We compare the performance of both PSJ to approaches based on bitmapped join indexes, as well as hash structures. We develop cost models for I/O usage and transmission costs for all three algorithms. Comparative results show that the overall I/O costs achievable with the Parallel Star Join (PSJ) are dramatically lower than those achieved with the Pipelined Hash Join (HASH), which in turn outperforms the Bitmapped Join Index (BJI) based join algorithm. In terms of transmission costs, PSJ and BJI, which have identical transmission costs, outperform HASH by an order of magnitude. We believe that this indicates that physical design strategy advocated in this paper and the associated join algorithm have a big part to play in warehouses, in spite of the fact that the TPC-D benchmark explicitly disallows vertical partitioning. To highlight this fact, our examples and experiments in this paper were run based on the TPC-D schema.

The approach we have presented is just the first of many steps, leaving open a number of issues including further refinement of the Parallel Star Join Algorithm, implementation of the algorithm, and the development of algorithms for other OLAP operations such as slice, dice, roll-up, and drill-down to exploit parallelism, as well as experiments in commercial environments.

References

[1] M. Abdelguerfi and K-F Wong, editors. *Parallel Database Techniques.* IEEE Computer Society, 1998.

[2] C. Ballinger and R. Fryer. Born to be parallel: Why parallel origins give teradata an enduring performance edge. In *Proc 8th International Hong Kong Computer Society Database Workshop*, pages 29–31, 1997.

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

[4] M-S Chen, M. Lo, P.S. Yu, and H.C. Young. Applying segmented right-deep trees to pipelining multiple hash joins. *IEEE Transactions on Knowledge and Data Engineering*, 7(4), August 1995.

[5] M-S Chen, P.S. Yu, and K-L Wu. Optimization of parallel execution for multi-join queries. *IEEE Transactions on Knowledge and Data Engineering*, 8(3), June 1996.

[6] A. Datta, B. Moon, K. Ramamritham, H. Thomas, and I. Viguier. Have your data and index it, too: Efficient storage and indexing for data warehouses (tr98-07). Technical report, University of Arizona, 1998.

[7] A. Datta, D. VanderMeer, and K. Ramamritham. Tr2001-15: Applying parallel processing techniques in data warehousing and olap. Technical report, Chutney Technologies, 2001.

[8] 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.

[9] D. DeWitt, J. Naughton, D. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proc VLDB*, pages 27–40, 1992.

[10] 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.

[11] P.M. Fernandez. Red brick warehouse: A read-mostly rdbms for open smp platforms. In *Proc. ACM SIGMOD*, page 492, Minneapolis, MN, May 1994.

[12] C.D. French. Teaching an OLTP database kernel advanced datawarehousing techniques. In *Proc. 13th ICDE*, pages 194–198, Birmingham, UK, April 7-11 1997.

[13] G. Hallmark. Oracle parallel warehouse server. In *Proc. Thirteenth Intl. Conf. on Data Engineering*, pages 314–320, Birmingham, UK, April 7-11 1997. IEEE.

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

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

[16] M. Kitsuregawa, H. Imai, and W. Loe. Introduction to the super database computer, sdc-ii. In *Proc JSPS-NUS Seminar on Computing*, pages 101–113, 1995.

[17] C. Lee and Z. Chang. Workload balance and page access scheduling for parallel joins in shared-nothing systems. In *Proc ICDE*, pages 411–418, 1993.

[18] M-L Lo, M-S Chen, C.V. Ravishankar, and P.S. Yu. On optimal processor allocation to support pipelined hash joins. In *Proceedings of the 1993 ACM SIGMOD*, 1993.

[19] H. Lu, B.C. Ooi, and K.L. Tan, editors. *Query Processing in Parallel Relational Database Systems.* IEEE Computer Society, 1994.

[20] K. Lyons. Hosting massive databases on the legion cluster. Technical report, AT&T Research Labs, Florham Park, NJ 07932, February 1997. Internal Report.

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

[22] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 38–49, Tucson, AZ, May 13-15 1997.

[23] Red Brick Systems. Star schemas and STARjoin technology. White Paper, September 1995.

[24] D. Schneider and D. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proc. ACM SIGMOD*, Portland, OR, June 1989.

[25] M. Seetha and P. Yu. Effectiveness of parallel joins. *Transactions on Knowledge and Data Engineering*, 2(4):410–424, 1990.

[26] A. Shatdal and J. Naughton. Using shared virtual memory for parallel join processing. In *Proc ACM SIGMOD*, pages 119–128, June 1993.

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

[28] J. Wolf, D. Dias, P. Yu, and J. Turek. Comparative performance of parallel join algorithms. In *Proc PDIS*, pages 78–88, 1991.