Applying Parallel Processing Techniques in Data Warehousing and OLAP

Anindya Datta[*], Debra VanderMeer [†], Krithi Ramamritham[‡], Bongki Moon[§]

Abstract

We explore the applicability of parallel processing techniques for OLAP query processing. In particular, we exploit the natural partitionability of a star schema and render it even more efficient by applying a storage and access structure that serves both as an index as well as data and lends itself naturally to vertical partitioning of the data. 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 this algorithm with two parallel query processing strategies. The first strategy 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 from the OLTP domain, namely the Hybrid 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 PSJ algorithm leads to dramatically better performance than both the BJI as well as the hybrid hash approaches, with regard to disk access costs, in both speedup and scaleup experiments. Even though the hybrid hash approach exhibits the best performance with regard to transmission overhead, since it is typically the case that network costs are much lower than disk access costs, the overall conclusion is that the PSJ approach and the storage and access structures it is based on are the very well suited for parallel OLAP systems.

Contact Author: Anindya Datta, email: adatta@cc.gatech.edu, Phone: (404)894-4364, FAX: (404)894-6030

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 [11]. Data warehouses tend to be extremely large - it is quite possible for a data warehouse to be hundreds of gigabytes to terabytes in size [2]. 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. processing is critical in such an environment, response times with OLAP queries is still largely an open issue. In this paper, we explore the applicability of parallel processing techniques for OLAP query processing. Performance in a parallel system is typically measured using these two key properties:

[*]Georgia Institute of Technology, Atlanta, GA 30332

[†]University of Arizona, Tucson, AZ 85721

[‡]University of Massachusetts, Amherst, MA 01003 and IIT Bombay

[§]University of Arizona, Tucson, AZ 85721

- *Linear scale-up* is the property that 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 [14]. Thus, a *scalable* architecture is crucial in a warehouse environment.

- *Linear speedup* is the property that 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*s, significantly improving query response times.

The first property is obvious, 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 [21] (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.

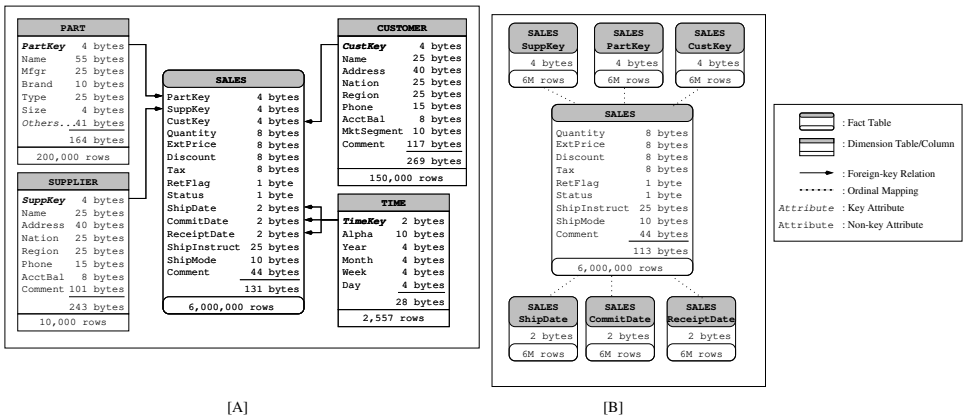

Figure 1: A Sample Warehouse Star Schema and Projection Index

This schema suggests an efficient data partitioning as we will soon show.

A common type of query in OLAP systems is the *star-join* query. In a star-join, one or more dimension tables are joined with the fact table. For example, the following query is a three-dimensional star-join that identifies the volumes sold locally by suppliers in the United States for the period between 1996 and 1998 [21]:

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 have developed 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 *Hybrid Hash* strategy [18], one of the best performing parallel query processing strategy from the traditional OLTP literature. NCR is using a variant of this algorithm on its Teradata system, a special purpose parallel warehousing system [1]. [1]

Our performance results indicate that both approaches lead to performance gains. However, the PSJ approach leads to dramatically better performance than the hybrid hash approach, with regard to disk access costs, in both speedup and scaleup experiments. The hybrid hash approach, on the other hand, exhibits better performance with regard to transmission overhead. Since it is typically the case that network costs are much lower than disk access costs, the overall conclusion is that the PSJ based approach is much better.

Shared-nothing architectures have been shown to achieve near linear speedups and scale-ups in OLTP environments as well as on complex relational queries [6]. Some of the major efforts at developing parallel data warehousing systems (such as the LEGION project at AT&T [14] attempting to aid decision support based on *call detail information*), are based on shared nothing architectures as well. So, we believe it is worth developing and evaluating query processing techniques for data warehouses implemented on shared nothing platforms. That is the goal of this paper.

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 7 and contrast these to our work.

The remainder of the paper is organized as follows. In Section 2 we introduce an approach to structure data warehouses by exploiting many proposed indexing strategies. The associated physical design and star join processing strategies are discussed in Section 3. This is followed by a description of the hybrid hash approach in Section 8. We then present a cost model of the disk access and transmission costs of the two approaches in Section 3.3 and compare the performance of these approaches in Section 6. Finally, in Section 7 we discuss related work, and in Section 8 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 [16] to be particularly appropriate for OLAP systems: B^+*-trees* , indexes based on *bitmaps* [16, 15], *projection indexes* and *bit-sliced indexes* [16]. 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 [8, 16].

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 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 `Discount, Tax, RetFlag, Status` and two projection indices are constructed, one on the `RetFlag` column, and another

[1] Note that the Teradata system runs on a shared-memory architecture, while we focus on a shared-nothing architecture in this work.

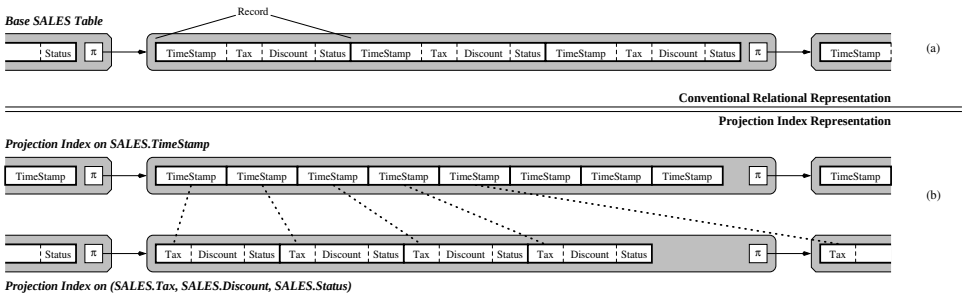

Figure 2: Projection Index

on the `Discount, Tax` 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 [15, 3].

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 [15]. 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 [15] introduced the idea of a bitmapped join index (BJI) for efficiently supporting multi-table joins. A BJI associates related rows from two tables [15], 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[2] .

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.

[2]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.

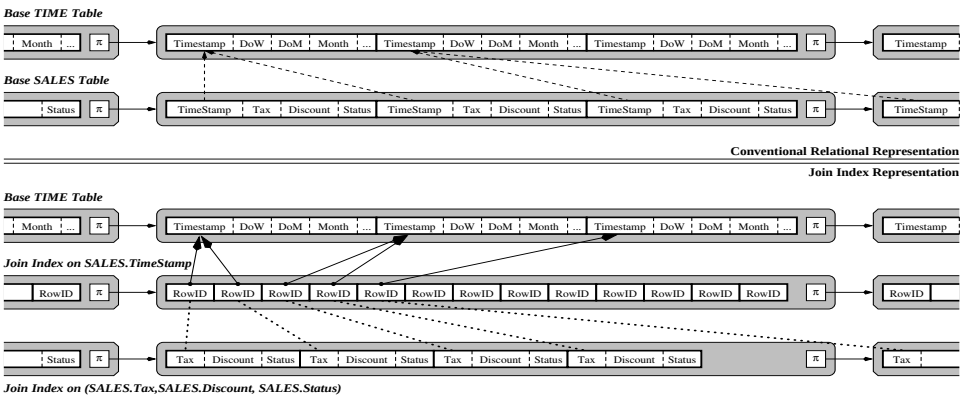

Figure 3: The Join Index

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 6 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. For a table of notation used throughout this paper, refer to Appendix A.

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 $\mathcal{J}_\mathrm{j}$, the fact table JI corresponding to the key attribute of D_i. Inside processor group j, a hybrid strategy is used 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. First, the task of data placement can be *hinted* by the structure of the star schema. For example, the primary key of a dimension table and its associated foreign key 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.

Second, 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.

Third, it is generally the case that the size of a dimension table is much smaller than that of a fact table, and often small enough to be fit in main memory. Thus, given the number of available processors and aggregate main memory capacity 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 where we partition the set of given processors into $d+1$ groups and (b) a physical data placement phase where we allocate data fragments to individual processors.

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. Each processor has m bytes of memory. 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.

(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 abovementioned criteria, the minimum size of the j^{th} processor group (i.e., G_j), $1 \leq j \leq d$, is given by $\mathcal{N}_j = \min(\mathcal{N}, \lceil \mathcal{S}_j/m \rceil)$[3] where m is the size of the main memory attached to each processor. This assumes that all the processors have 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 – it is generally the case that most OLAP queries involve joins. It is therefore crucial that the overlap of the processor groups be minimized to allow concurrent processing of multiple join computations in a star join query without intervention. In other words, we would like to minimize, for each processor, the number of groups it participates in.

Below we outline an *optimal* solution to the processor group partitioning problem by formulating it as a constrained optimization problem solvable as a linear integer mathematical program. Our formulation assumes the following variable, in addition to the parameters outlined above.

- w_{ij} : A (0,1) variable denoting the assignment of processor i to group j. This variable takes on a value of 1 if the assignment is valid and 0 otherwise.

[3] 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.

$$
\begin{array}{rll}
\text{Minimize} & \sum_i \sum_j w_{ij} & \\
\text{such that} & & \\
\forall j, 1 \le j \le d+1 & 1 \le \sum_i w_{ij} \le \mathcal{N} & \\
\forall j, 1 \le j \le d & \sum_i w_{ij} & \ge \min(\mathcal{N}, \lceil \mathcal{S}_j/m \rceil) \\
 & \sum_i w_{i(d+1)} & \ge \mathcal{N} \times \dfrac{\text{Aggregate Size of Metric PIs}}{\text{Total Size of the Data Warehouse}}
\end{array}
$$

The objective function attempts to minimize the number of groups each processor should be assigned to and thus attempts to satisfy item 3.

The first constraint models the two degenerate requirements that the group sizes must be at least 1, and that there are no more than $\mathcal{N}$ processors available for each group. The second and third constraints model requirements expressed by items (1) and (2).

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 $\mathcal{J}_\mathtt{j}$, from the fact table. We first horizontally partition the JI in a round robin (RR) fashion among the $\mathcal{N}_i$ processors. We now provide the rationale for adopting the above mentioned strategy, which 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 colocated. 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.

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 colocated) JI records may point to the same PI record, some replication of PI data across processors would be required to ensure colocation 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 (RR)), only the RR 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_⋈ 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., $=, \le, \ge$). 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 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 was 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, $\mathcal{R}_{\mathtt{fact,i}}$. 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 $\mathcal{R}_{\mathtt{fact,i}}$ is given below. This procedure assumes that the $\mathcal{R}_{\mathtt{fact,i}}$ structure is already defined and initialized (i.e., the insertion pointer points to the first bit of $\mathcal{R}_{\mathtt{fact,i}}$).

```
start scanning J_i from the top
foreach row j in J_i, (1 ≤ j ≤ |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 R_dim,i is set
        set the j^th bit of R_fact,i
    endif
enddo
```

The above discussion, for expository ease, assumes a centralized join in group i. In reality, a segment of $\mathcal{R}_{\mathtt{fact,i}}$ 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. (Some details, necessary to compute the cost, are given in Section 3.3.)

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 $|\mathcal{R}_{\texttt{fact}}|$ 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* from this point onwards) 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 $\mathcal{J}_{\texttt{customer}}$. Based on our physical design, in this group, there will exist a PI on the *Customer.name* column, denoted by $\mathcal{P}_{\texttt{cname}}$. Furthermore, assume there exists a dimensional rowset, denoted by $\mathcal{R}_{\texttt{Customer}}$ that was computed in the LRG phase[4].

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, $\mathcal{R}_{\texttt{global}}$, has been shipped to the customer dimensional group and all necessary data loaded, the following procedure 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).

```
start scanning R_global from the top, i.e., the first bit
do until no more bits left to be scanned in R_global
  let the ordinal position of current bit in R_global be denoted by i
  if the i^th bit (i.e., the current bit) in R_global is set
```

[4]This is a valid assumption. In case, there were no restriction clauses in the submitted query based on the customer dimension, one can assume that $\mathcal{R}_{\texttt{customer}}$ consists of all 1's.

read the i^{th} element of $\mathcal{J}_{\texttt{customer}}$, 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 $\mathcal{P}_{\texttt{cname}}$
insert this element into the *cust_name* structure
move the insertion pointer of *cust_name* to the next insertion position
endif
scan the next bit in $\mathcal{R}_{\texttt{global}}$
enddo

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 3.3), 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.

3.3 Cost model for the PSJ Algorithm

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 DataIndexes, in most cases, the delays associated with the mapping computations will be negligible compared to the much slower storage access times [3]. This expectation is corroborated by other studies [16], 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 4 below. Essentially, this figure shows

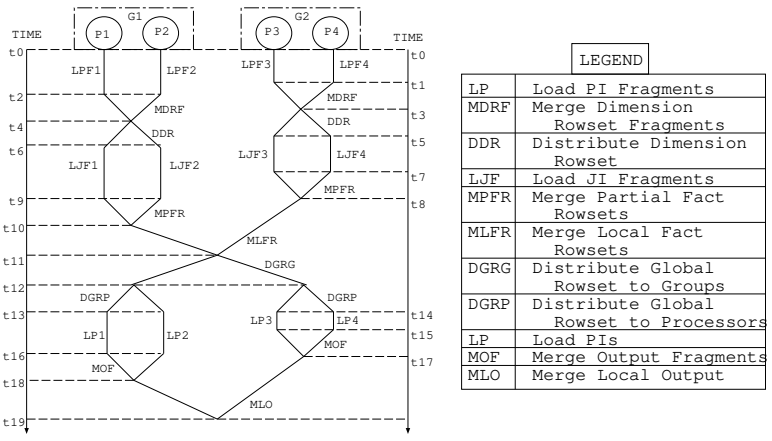

Figure 4: Response Time Computation Example for the PSJ Strategy

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. Finally, we remark on the use of two time lines on either side of the figure. Both refer to the same time progression and sequence. We could have used only one time line and marked all the time instants on it, but this would have eroded the readability of the figure. 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 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 of receiving/sending transmissions. At this point though, according to the scheme of figure 4 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 the Appendix C attached at the end of this paper.

4 The Parallel BJI Join

We now consider a warehouse environment utilizing *Bitmapped Join Indexes* (BJIs). This environment will require BJIs to perform joins and *Bitmapped Indexes* (BIs) to perform restrictions. Each BI is a B+tree with leaf level pointers to bit vectors, rather than RIDs. As in a B+tree, each node in a BI contains values and pointers. A BJI connects a foreign key column in a fact table to a column in a dimension table. Figure 5 shows a value-list BJI [16] on `PURCHASE.Customer_ID` where attribute `Customer_ID` connects two tables, `PURCHASE` and `CUSTOMER`. As can be seen from the figure, the values in a BJI node are RIDs corresponding to the `CUSTOMER` table, and the leaf-level pointers point to bit vectors corresponding to the `PURCHASE` table. The set bits in each bit vector indicate the tuples in the `PURCHASE` table that are join candidates. Based on BJIs, a star join algorithm proceeds roughly as follows, inspired by [16]. The dimensional rowsets (i.e., $\mathcal{R}_{\texttt{CUSTOMER}}$) computed during the predicate selection phase are used to determine the set of matching records in the fact table with the corresponding BJI. In other words, for a dimension table D_i, the leaf-level of the corresponding BJI is scanned, and the rowsets associated with the rows that appear in a dimensional rowset $\mathcal{R}_i$ are loaded and bitwise ORed together. When these operations have been applied to all participating dimension tables, the ANDed result is the *join rowset* $\mathcal{R}_{join}$ which indicates which records of the fact table appear in the join result. 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 [3], we have shown that

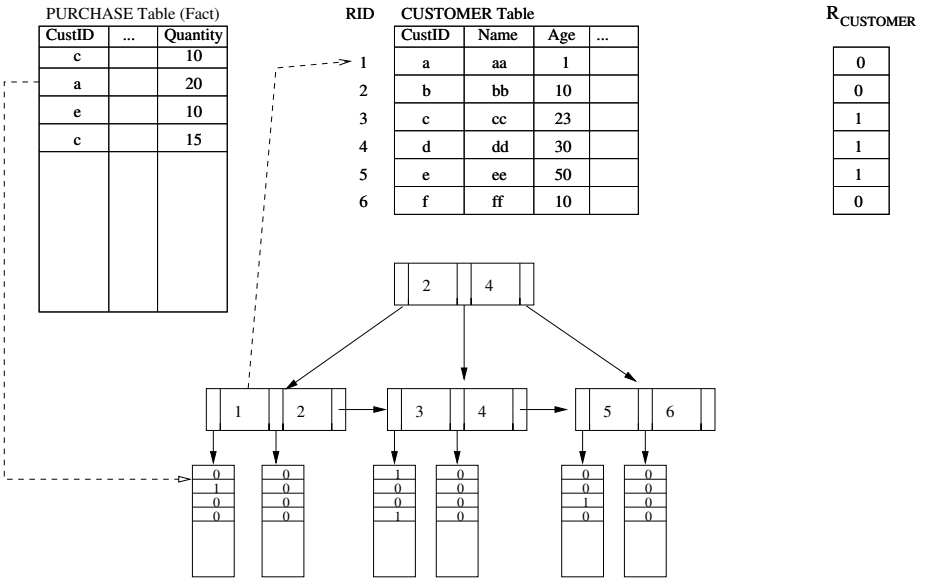

Figure 5: Bitmapped Join Index on PURCHASE.Customer_ID

the memory requirement for BJI, in blocks, is

$$\mathcal{M}_{JOIN}(BJI) = 1 + |\mathcal{D}| + \Sigma_{D \in \mathcal{D}} \left\lceil \frac{|D| \times w(D)}{B} \right\rceil \quad (1)$$

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. This expressions corresponds to the BJI join strategy outlined above.

If enough memory is not available to load all necessary dimensional columns into memory, a pairwise hash join technique may be applied in conjunction with BJIs. In this approach, once $\mathcal{R}_{JOIN}$ has been determined, the relevant values from the different dimensions are extracted from the source tables and stored in temporary files. These temporary files are then joined pairwise until the final join result is computed, thereby requiring $|\mathcal{D}|$ individual two-way joins. Since it is well known [16] that pairwise joins do not perform well in a data warehouse environment, we do not consider it any further.

Now, we describe a strategy for applying the BJI join algorithm in a data warehousing environment in a parallel fashion. Like the PSJ algorithm, this strategy consists of three parts: (a) a processor grouping strategy: (b) a data placement strategy; and (c) a BJI join algorithm for multidimensional star joins. We describe each part in turn.

4.1 Processor Grouping

The processor grouping 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.

4.2 Data Placement

The following are loaded on each disk in processor group G_j:

- 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.

- For the Fact table: The entire fact table (needed for OP phase) and BI for each attribute in the fact table,

4.3 The BJI based Join Algorithm

The Parallel BJI Join algorithm has the same general structure as the PSJ algorithm. So, only the parts that are different are highlighted below.

In the *Local Rowset Generation* phase, to generate the restriction rowset fragments, the BJI algorithm uses the BIs to perform restrictions, rather than BDIs, 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.

The *Global Rowset Synthesis* phase is exaclty the same as for PSJ.

Considering 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[5] 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. The attribute values in the selection predicates are projected out and placed in the appropriate rows of the output fragment structure.

Output for the fact table is produced as follows. Space is allocated for all output rows, with an appropriate width for holding the projected attribute values from the fact table. For each bit set in the global rowset the appropriate tuple is loaded. The needed attribute values are projected out and the resulting output fragment is placed in the appropriate row of the output fragment structure.

Finally, the *Merge Partial Output Fragments by Group* and the *Merge Group Output Fragments* are performed exactly as for PSJ.

Effectively, using the approach outlined above, a one-dimensional join is performed at each processor. In this context, equation 1 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 equation 2.

$$\mathcal{M}_{JOIN}(BJI) = 1 + 1 + \left\lceil \frac{|D_i| \times w(D_i)}{B} \right\rceil \tag{2}$$

In addition to the PSJ and BJI algorithms, we also evaluated the performance of the Hybrid Hash Join (referred to in this paper as the HASH algorithm). Details of this evaluation are located in the Appendix B for space considerations.

5 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. We first describe these components for the PSJ strategy and

[5]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.

then highlight the differences for modeling Hash.

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. In addition, 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 4 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 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 Hybrid Hash Join algorithm.

6.1 Performance Metrics

We focus on two primary performance metrics:

Response Time measured in Block Access (RTBA): RTBA represents the response time of a query measured in units of block accesses. As explained in Section 3.3, 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 3.3 and demonstrated pictorially in figure 4 (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, however, note that 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 five phases of the HASH algorithm are described in Appendix B.

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.

All the 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 1 shows the values of the parameters used in our experiments.

Parameter	Value	Parameter	Value
Num Dimensions	4	*MaxSelectivity*	100%
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 1: Table of Parameter Values

6.2 Scalability Experiments

We first 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 by equation 2 earlier. 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 shows two plots, labeled [A] and [B]. Figure 6A shows RTBA curves for PSJ and HASH corresponding to memory sizes of 16 MB and 64 MB. Figure 6A also shows 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.

Figure 6B shows the PSJ and HASH curves for memory sizes of 256 MB and 1024 MB, as well as a BJI curve (with memory requirements as explained above).

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, we had to, potentially, 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 primarily to the output preparation costs that do not scale linearly. In particular, 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. Even though we accord the HASH algorithm an advantage by discounting the cost of initial hashing and data placement, the algorithm has to rehash its intermediate results. There is significant costs involved in this, as this is usually done in a "hashing server" which must receive intermediate result fragments from the other nodes (the intermediate results could be huge), rehash and re-distribute.

Now we turn our attention to comparing the PSJ, BJI, and HASH cost curves. The first observation, of course, is that PSJ costs are orders 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 absolutely needed. In contrast, HASH works on straight horizontal partitioning, and entire schemas are accessed, regardless of whether certain columns are needed or not. Although the BJI schema is also horizontally partitioned, it fares much better than HASH because data is accessed via smart structures, such as B+trees, and the high cost of rehashing is absent. BJI's costs are still much higher than PSJ costs due to the fact that horizontal partitioning requires accessing entire tuples, even though only a portion of the attributes may be needed. In addition, each processor must scan the entire fact table for dimension keys in the output preparation phase, as mentioned at the beginning of this section.

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 two paragraphs ago), and consequently lower the PI scan cost in the OP phase.

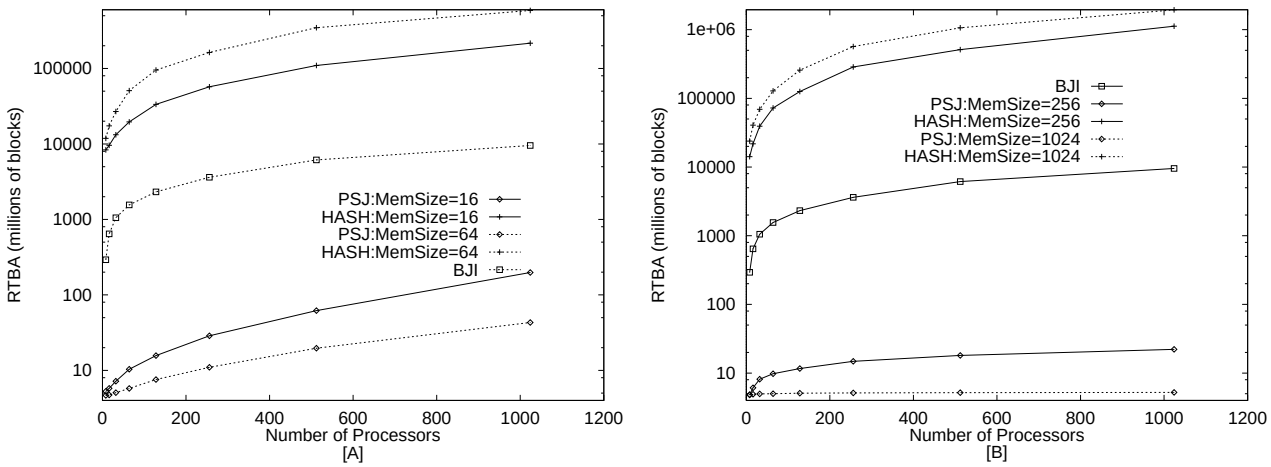

Figure 6: Disk Access cost for PSJ, BJI, and HASH Scalability as CountProc and ScaleFactor Increase

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 Appendix C) parameter, we only report the results of a single ADT experiment. In Figure 7A, 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.

HASH has substantially lower transmission costs than PSJ – this is because PSJ needs to exchange intermediate results several times during the course of its execution (at the end of each phase), whereas HASH needs to pay transmisison costs only twice – first for rehashing and then at the end, to produce the final output.

Moreover, while both HASH and PSJ exhibit imperfect scalability, PSJ scales worse than HASH as the transmission volumes increase at a greater rate for PSJ. This occurs as increasing the size of the warehouse results in greater amounts of data that need to be transmitted across groups (note that as the size of the warehouse and the number of processors increase, the number of groups remain the same but processors per group increase). HASH, on the other hand, only has the additional transmission costs of the rehash step.

It is generally considered that network bandwidth costs are much lower than disk access costs - thus, we believe that this RTBA vs. ADT tradeoff works in favor of PSJ. This is also corroborated by the RTBA results.

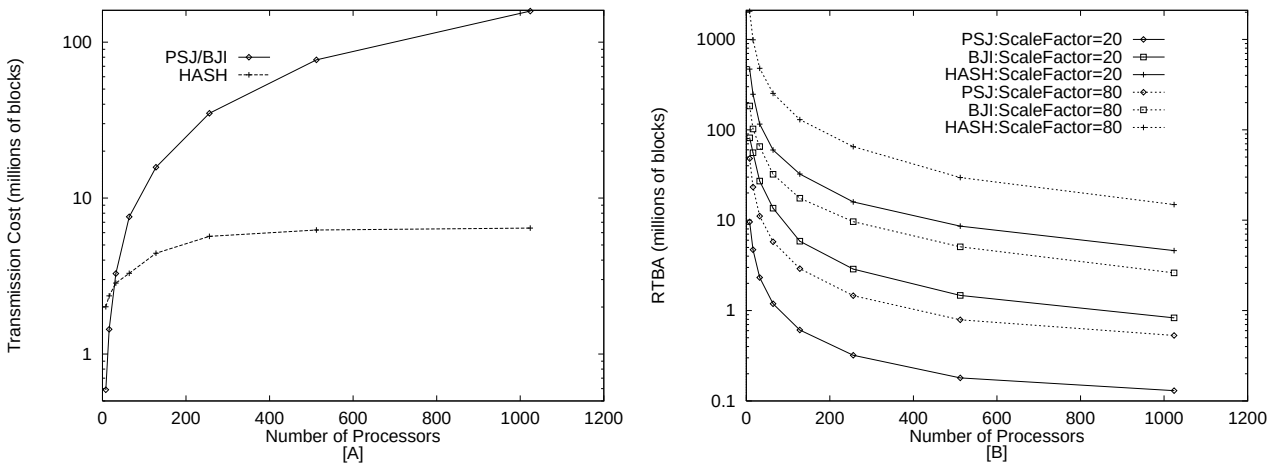

Figure 7: [A]: Transmission cost for PSJ, BJI, and HASH Scalability as CountProc and ScaleFactor Increase; [B]: Response time for PSJ, BJI, and HASH as the Number of Processors Increases

6.3 Speedup Experiments

Having discussed the scaleup properties, we now turn our attention to the speedup performance of our algorithm by adding processors to the parallel system, but holding the database size constant.

RTBA Results: The curves in Figure 7B show the RTBA costs for various database sizes and a constant memory size of 64 MB per processor as the number of available processors increases. However, unlike the scaleup experiments the database sizes are held constant in each experiment. In paricular, Figure 7B plots RTBA speedup curves for PSJ and HASH corresponding to scale factors of 20 and 80. We also performed speedup experiments for scale factors of 320 and 1280. The results show similar relative performance to the curves for scale factors of 20 and 80. In the interests of brevity, we omit these curves.

PSJ, BJI, and HASH exhibit positive but sublinear speedups. In general, disk access costs decrease as the number of processors increases. This occurs because much of the data can be spread across multiple processors, lowering the disk access cost on individual processors.

In all algorithms, as the number of processors increases, the slope of the curves decreases. In PSJ, this sublinearity occurs because the full PI must be loaded on each processor of a group in order to generate output, i.e., it is not possible to spread the disk access cost of loading the PI across several processors.

This was also explained earlier when describing scaleup. In BJI, the sub-linearity is a result of scanning all qualifying tuples of the fact table for finding dimension keys in the output preparation phase. In HASH, as with the other algorithms, the slope of the curve decreases sharply at low processor counts, and flattens out at higher processor counts. This occurs because the difference between the amount of data loaded per processor for low processor counts is much greater than the amount of data loaded per processor at high processor counts.

Having discussed the general shape of the curves, we now consider the individual curves in relation to one another. We observe PSJ having consistently significantly lower costs than HASH. The reader has already been exposed to the causes of these lower costs.

ADT Results: We ran experiments for ADT costs for PSJ, BJI, and HASH for scale factors of 160 and 640. The results (plots not shown for space considerations) showed constant costs for all rage for processor values, i.e., the curves turned out to be parallel to the X-axis. In particular, PSJ and BJI showed a uniform cost of 16.5 million blocks and 42.6 million blocks at scale factors of 160 and 640 respectively. HASH showed costs of 4.3 million blocks and 6.4 million blocks at scale factors of 160 and 640 respectively.

The notable feature of these results is that adding processors does not contribute to transmission costs. This makes sense for both PSJ and HASH. In PSJ, adding more processors simply means that the size of groups increase – note that the number of groups still remains the same. Thus, the overall intergroup communication remains constant. Inside a group, adding more processors results in the JI being distributed accross more nodes, meaning the local rowsets generated at each node are smaller. However, the *aggregate* communication remains the same, as they have to be merged to produce a dimensional rowset of the same size. Similarly, for HASH, adding more processors results in smaller buckets at each processor. In the intermediate rehashing step, exactly the same amount of data need to be transferred to and from the hashing server, regardless of the number of nodes involved. Similarly the final output production step also needs the same amount of transmission involved.

Also, note that PSJ has substantially higher ADT than HASH, for reasons already identified when discussing the ADT scaleup results.

7 Related Work

The research related to our work includes the areas of data warehousing/OLAP and parallel database processing, which until recently, have largely been considered distinct areas of research. 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* [2]. 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 [10]. 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 [16, 3]. 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., [4, 20, 22, 19]). From this work has emerged the notion that highly-parallel, shared-nothing architectures can yield much better performance than equivalent closely-coupled systems [18, 13, 5]. Indeed, may commercial database vendors have capitalized on this fact [6]. Our focus on shared-nothing systems is also motivavted 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).

As mentioned previously, we were able to find very little in the published (academic) literature related to parallelism in data warehouses. One exceptions is [12], whose major thrust is efficient parallel hardware to support warehousing. 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 [1], 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 [7], 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 [17]. 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 [9], the parallel processing model used in the Oracle Parallel Warehouse Server is described. This product is based on a parallel processing model that incorporates both horizontal and pipelined parallelism.

A potential limitation of the products 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.

8 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.

As a comparison and to serve as a contribution on its own, this paper also showed how the traditional parallel hybrid hash join can be tailored for OLAP systems. As another point of comparison, we showed how a Bitmapped Join Index based physical design strategy can be exploited in a parallel environment. In the process of performing these comparisons, we developed cost models for PSJ and BJI.

Comparative results show that even though the hybrid hash join based approach incurs lesser communication, the overall response times achievable with the Parallel Star Join (PSJ) and Bitmapped Join Index (BJI) based join algorithm are typically orders of magnitude lower. PSJ is in fact between five times and orders of magnitude better than BJI. We believe that this indicates that physical design strategy advocated in this paper and the associated join algorithm, namely PSJ, have a big part to play in warehouses, in spite of the fact that the TPC-D benchmark explicitly disallows vertical partitioning. In fact, 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 many issues. Future work in this area includes 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.

References

[1] 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.

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

[3] 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.

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

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

[6] 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.

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

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

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

[10] 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.

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

[12] 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.

[13] 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.

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

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

[16] 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.

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

[18] 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.

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

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

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

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

Appendix A: Table of Notation

In this appendix we show the various notations used in the paper, including the subsequent appendices. throughout the paper.

Symbol	Description	Symbol	Description
$\mathcal{N}$	Number of processors	d	Number of dimensions
D_j	Dimension j	F	Fact Table
$\mathcal{J}_\mathrm{j}$	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 fact table tuple
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 2: Table of Notation for Cost Model

Appendix B: The Hybrid Hash Join

Here, we describe the Parallel Hybrid Hash Join [18] strategy, a well-known join strategy based on the conventional relational model, 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.

8.1 Overview of the Parallel Hybrid Hash Join Algorithm

Here, we provide a brief review of the HASH algorithm in the context of a standard OLTP, parallel environment (for a detailed treatment of this algorithm, see [18]). The algorithm provides a means of performing two-relation joins. This is important to note, as we had to extend the algorithm to handle multi-way joins in an OLAP environment (this will be described in section 8.2 below). Let R and S be the relations to be joined, and let R be smaller than S. The HASH algorithm for joining R and S consists of five steps.

1. First Hash: Hash the smaller relation, R into N logical buckets, such that each logical bucket is the size of the aggregate memory available across all available processors. Hash S the same way.

2. Second Hash: Hash the buckets resulting from the first hash of R such that the size of the resulting buckets is the memory size of an individual processor, save two blocks (one for loading a block of S at a time to perform the join, and one for writing the output, one block at a time). Hash S using the same hash function. This guarantees that the join tuples of R and S will be colocated in corresponding buckets to be processed on the same processor.

3. Select and Project: Each bucket of a relation containing restriction and/or projection predicates is scanned to perform the selection and projection. For tuples meeting the restriction predicates (all tuples, if no restriction predicates have been specified), only the attributes specified for projection are returned (all attributes, if no projection is specified).

4. Join: Join corresponding buckets of R and S on each processor to obtain the result buckets. This is accomplished by loading a bucket of R into memory, and one block of the corresponding S bucket at a time to perform the join.

5. Merge Results: Transmit the resulting buckets to a single node for output processing.

8.2 Applying Parallel HASH strategy to OLAP

In this section, we describe our strategy for adapting the HASH algorithm in a data warehousing environment. This strategy consists of two parts: (a) a data placement strategy, and (b) an extension of the HASH algorithm for multi-dimensional star joins. Even though this adaptation is fairly straightforward, we consider this a contribution of this paper since the application of Parallel Hybrid Hash Join to OLAP processing, to the best of our knowledge, has not been reported in the literature.

8.2.1 Physical data placement for HASH

In order to perform a join of two relations, the HASH algorithm places multiple disjoint buckets of data for each relation on each processor using hash functions, such that the buckets of one relation on a given processor are guaranteed to join with the corresponding buckets of the other relation on the same processor.

In order to provide a comparison between the PSJ and HASH algorithms that is favorable to HASH, we assume that the hashing portion of the HASH algorithm has already been completed, and that the necessary buckets are available as queries arrive. In doing so, we essentially choose to ignore the *substantial* upfront hashing cost incurred by the HASH algorithm. Moreover, we assume that for every hashing step required, an *ideal* hash function is always applied that partitions the buckets perfectly and uniformly. In reality this is unlikely to occur, but in making this assumption we ascribe even more advantages to the HASH strategy. The intent is to subject PSJ to the most stringent comparative tests feasible.

The HASH algorithm uses all available processors for each join step performed. Therefore, there is no processor grouping strategy corresponding to the PSJ processor grouping strategy.

8.2.2 The HASH Algorithm

We now present a brief overview of the HASH algorithm for star joins. It consists of five steps: (a) Selection and Projection, (b) Join Dimension and Fact tables to create intermediate tables, and (c) Rehash Intermediate Tables , (d) Join intermediate tables to obtain the result, and (e) transmitting the results to a single node for output.

In the *Selection and Projection* phase, the buckets of each dimension relation to be joined are scanned, in parallel at each processor (recall that all relations are spread across all processors), once for each restriction predicate on that relation, maintaining only those tuples that meet the restriction predicates. For each tuple meeting the restriction predicate, attributes that are not necessary for the join or output of the query are discarded from further consideration. The results of the selection and projection phase are then written back to disk.

In the *Join Dimension and Fact Table* phase, each dimension table that participates in the join is joined with the fact table to create an intermediate result relation. This is accomplished by scanning the buckets resulting from the selection and projection on all processors for the dimension and fact tables to be joined, performing the join, and writing the resulting buckets back to disk. In other words, for a n-dimensional join, n intermediate relations are created, each one based on joining a single dimension table with the fact table. These joins are performed sequentially, with each individual join being performed in parallel across all processors.

For joins of a dimensionality greater than 1, phases (c) and (d), i.e., *Rehash Intermediate Tables* and *Join Intermediate Tables* are performed. Here, the intermediate results are joined pairwise in a manner similar to that of phase (b), until a final result is obtained. However, this phase is complicated by the need to rehash the intermediate relations so that tuples to be joined are appropriately colocated. The HASH algorithm specifies that the hash be applied to the smaller of two relations to be joined. In the context of a star join, that means that the hash in our data placement scheme is applied to the dimension tables. In

order to join intermediate relations, therefore, it is necessary to rehash intermediate results to guarantee colocation of join tuples. This adds not only disk access costs, but also transmission costs.

In phase (e), the *Merge Join Results* phase, the join results from each processor are transmitted to a single node to produce the final join output.

The reader will note that the (a), (b) and (e) phases of the extended hash algorithm specified above correspond to the standard 2-way parallel HASH algorithm described in prior literature (and summarized in Section 8.1). Our extension is limited to phases (c) and (d) which were added to incorporate multiway star joins.

Appendix C: Cost Models

In this section, we the develop the cost models for PSJ, BJI, and HASH.

C.1 Cost Model for PSJ

1. *Local Rowset Generation (LRG)*

   (a) *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 $a_{Fj} \in A_R$.

   (b) *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 $log_2\mathcal{N}_F \times |F|$.

   (c) *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 $log_2\mathcal{N}_F \times |F|$

   (d) *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 $\left\lceil \frac{\frac{|F|}{\mathcal{N}_i} \times W_{JI}}{B} \right\rceil$.

   (e) *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*

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

   (b) *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)*

   (a) *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 $log_2 N_F \times |F|$.

(b) *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 $\left\lceil \frac{\frac{|F|}{\mathcal{N}_F} \times \sum W_{a_{Fj}}}{B} \right\rceil +$ $\left\lceil \frac{\frac{|F|}{\mathcal{N}_F \times 8}}{B} \right\rceil a_{Fj} \in A_P$.

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

(d) *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
$|O| \times (\sum W_{a_{ij}} + \sum W_{a_{Fj}})$, where $a_{ij}, a_{Fj} \in A_P$.

C.2 Cost Model for BJI

1. *Local Rowset Generation (LRG)*

(a) *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 $\left\lceil log_{P_i} \frac{V_i}{N_i} - 1 \right\rceil + \left(\frac{V_i}{K_i N_i} \right) + f\left(\frac{\varsigma |V_i|}{8BN_i} \right)$.
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 $\left\lceil log_{P_F} \frac{V_F}{N_F} - 1 \right\rceil + \left(\frac{V_F}{K_F N_F} \right) +$ $f\left(\frac{\varsigma |F|}{8BN_F} \right)$. Here, the costs for index access are taken from [3]

(b) *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 $log_2 \mathcal{N}_F \times |F|$.

(c) *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 $log_2 \mathcal{N}_F \times |F|$

(d) *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 $\left(\frac{\varsigma |D_i|}{N_i} \left\lceil log_{P_i} \frac{V_i}{N_i} - 1 \right\rceil \right) +$ $+ \left(R \frac{\varsigma |D_i|}{N_i} \right)$. Here, the costs for index access are taken from [3]

(e) *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*

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

   (b) *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)*

   (a) *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 $log_2 N_F \times |F|$.

   (b) *Generate Parital Output Fragments (GPOF)* 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 [3]

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

   (d) *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
   $|O| \times (\sum W_{a_{ij}} + \sum W_{a_{Fj}})$, where $a_{ij}, a_{Fj} \in A_P$.

C.3 Cost Model for HASH

1. *Selection and Projection*: Disk access cost for performing a single restriction (with projection) on a dimension i (in blocks) is $\left\lceil \frac{\frac{|D_i|}{\mathcal{N}} \times \sum W_{a_{ij}}}{B} \right\rceil \forall a_{ij}$. Disk access cost for performing a restriction on a metric table (in blocks) is $\left\lceil \frac{\frac{|D_i|}{\mathcal{N}} \times \sum W_{a_{Fj}}}{B} \right\rceil \forall a_{Fj}$.

2. *Join Dimension and Fact Table*: Disk access cost for perfoming a join of a dimension i and the fact table (in blocks) is $\left\lceil \frac{\frac{|D_i|}{\mathcal{N}} \times \sum W_{a_{ij}}}{B} \right\rceil, a_{ij} \in A_P + \left\lceil \frac{\frac{|F|}{\mathcal{N}} \times \sum W_{a_{Fj}}}{B} \right\rceil, a_{Fj} \in A_P$.

3. *Join Intermediate Tables*: Let R_1 and R_2 represent any two intermediate tables to be joined. The cost of this join will be $\frac{R_1}{\mathcal{N}} + \frac{R_2}{\mathcal{N}}$.

4. *Rehash Intermediate Table*: Disk access cost for the rehash of any intermediate relation R_x (in blocks) is $\left\lceil \frac{\frac{|R_x|}{\mathcal{N}} \times \sum W_{a_{ij}}}{B} \right\rceil \forall a_{ij}$. Transmission cost for the rehash of any intermediate relation R_x is $\left\lceil \frac{\frac{|R_x| \times 8}{\mathcal{N}} \times \sum W_{a_{ij}}}{B} \right\rceil, a_{ij} \in A_P$.

5. *Merge Join Results*: Transmission cost to send all output buckets to a single node
   $|O| \times (\sum W_{a_{ij}})$, where $a_{ij} \in A_P$.