“Have your Data and Index it, too” Efficient Storage and Indexing for Data Warehouses

Anindya Datta* Bongki Moon† Krithi Ramamritham‡
Helen Thomas* Igor Viguier*

Technical Report 98-7

Abstract

Two possible strategies may be utilized to enhance the efficiency of processing OLAP queries: (a) precomputation strategies (e.g., view materialization, realizing data cubes), and (b) ad-hoc strategies. While a significant amount of work has been done in developing precomputation strategies, it is generally recognized that it is difficult to materialize the answers to all possible queries. Thus, ad-hoc querying must be supported in data warehouses. This realization has sparked an interest in exploring indexing strategies suitable for OLAP queries. There appears to have been relatively little work done in ad-hoc query support for data warehouses [45, 46, 55, 39].

In this paper we propose *DataIndexes* as a new paradigm for storing the base data. *An attractive feature of DataIndexes is that they serve as indexes as well as the store of base data.* Thus, DataIndexes actually define a physical design strategy for a data warehouse where the indexing, for all intents and purposes, comes for “free”. We also present two efficient algorithms for performing star-joins with DataIndexes. In addition, we present a mathematical analysis of all the indexes presented by O’Neil and Quass as well as our DataIndexes and present analytical expressions categorizing the cost of query evaluation using these structures for range selections and star-joins, two common classes of queries in OLAP. These aid in performing an analysis yielding precise “break-even” points for comparing these indexing alternatives. Overall, it turns out that DataIndexes are very attractive in a wide variety of cases in terms of enhancing the performance of range and star-join queries in data warehouses.

August, 1998

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

*Georgia Institute of Technology, College of Computing, Atlanta, GA 30332
†University of Arizona, Department of Computer Science, Tucson, AZ 85721
‡University of Massachusetts, Department of Computer Science, Amherst, MA 01003

Contents

1 Introduction

Data warehousing and *On-Line Analytical Processing* (OLAP) are becoming critical components of decision support as advances in technology are improving the ability to manage and retrieve large volumes of data. *Data warehousing* refers to "a collection of decision support technologies aimed at enabling the knowledge worker (executive, manager, analyst) to make better and faster decisions" [11]. Thus, loosely speaking, a data warehouse is a "very large" repository of historical data pertaining to an organization [35]. OLAP refers to the technique of performing complex analysis over the information stored in a data warehouse [13]. The significance of data warehousing is evidenced by the recent growth in the number of related products and services offered - the market for data warehousing, including hardware, database software, and tools, is projected to be $8 billion in 1998, up from $2 billion in 1995 [11]. These technologies are gaining widespread acceptance in a multitude of industries including retail sales (supermarkets, department stores, etc), telecommunications, and financial services.

Data warehousing/OLAP systems are best understood by comparing them to traditional *On-Line Transaction Processing* (OLTP) systems. OLTP systems are designed to automate clerical data processing tasks (e.g., order entry), which are structured and repetitive, tasks that operate on detailed data. Therefore, the emphasis in such systems is placed on maximizing transaction throughput. In contrast to OLTP systems, data warehouses are designed for decision support purposes and contain long periods of historical data. For this reason, data warehouses tend to be extremely large - it is not uncommon for a data warehouse to be hundreds of gigabytes to terabytes in size [11]. OLAP applications are characterized by the rendering of enterprise data into multidimensional perspectives, which is achieved through complex, ad-hoc queries that frequently aggregate and consolidate data, often using statistical formulae [13]. Thus, OLAP environments are query intensive, where aggregated and summarized data are much more important than detailed records. Typical OLAP queries require 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 environments is on efficient query processing. This area is starting to receive the attention it deserves. A number of "conventional" relational query processing approaches have been applied to or extended for answering OLAP queries. Some of this work has concentrated on efficiently performing `GROUP BY` [8, 9, 20], aggregation [10, 23, 33, 30, 50, 68, 69], join or range queries [32, 60, 64], or supporting incomplete query answers [6, 29, 66]. Several approaches have been proposed for supporting the SQL `CUBE` operator, including [2, 17, 23, 42, 53, 58].

Yet another facet of query processing that has received attention in the literature is that of efficiency. Fast query evaluation is critical in OLAP environments given the interactive nature of most OLAP sessions. There are two basic approaches for quickly evaluating OLAP queries:

1. *Precomputation Strategies.* This approach relies on *summary tables*, derived tables that house precomputed or "ready-made" answers to queries [11]. This has been, by far, the most explored area in the context of data warehouses [1, 14, 22, 24, 25, 26, 28, 40, 41, 48, 49, 70]. The basic premise underlying this work is that data warehouses can achieve faster response times by pre-aggregating (i.e., materializing) the answers to frequently asked queries. It is recognized however, that such anticipation only works up to a point [11, 46], and a considerable fraction of the workload in OLAP applications will consist of *ad-hoc* queries which will need to be computed on demand [3]. This has led to work on strategies for ad-hoc query processing.

2. *Ad-hoc Strategies.* This approach to fast OLAP query processing supports ad-hoc querying by using fast

access structures on the base data. Database systems use indexes to improve efficiency of access to data. Various general purpose indexing techniques have been proposed and are utilized in OLTP systems, including hashing [54], B trees [12, 15], and multidimensional trees such as the R-tree [27], the K-D-B tree [52], and the BV tree [18]. There exists another class of multidimensional structures, namely *grid files* [43], that allows for very fast access to multidimensional data. However, in these and other index structures proposed for OLAP, one envisions a set of relations or table structures, and a separate set of indices or access structures. That is, thus far, databases have considered index and data separately. Given the large size of data warehouses, storage is a non-trivial cost, and so is the additional storage requirement due to the index structures. This is especially true given that data and storage maintenance costs are often up to seven times as high per year as the original purchase cost [59]. Hence, a terabyte-sized system, with an initial media cost of $100,000, could cost an additional $700,000 for every year it is operational. This cost is certainly non-trivial. Indexess, obviously, add to this cost and hence it is essential to minimize these additional costs. Unfortunately, as we show in Section 2, even the simplest index structure used today incurs substantial increase in total storage requirements, both in absolute and percentage terms. This, in turn, translates into higher media and maintenance costs. More importantly though, intuition dictates that an increased overall database size should result in lower performance. This prompts us to ask the following question: "*is it possible to reduce storage requirements, without sacrificing the efficiency obtained from indexing?*"

In this paper we answer the above question in the affirmative by proposing *DataIndexes* as a novel paradigm for storing the base data *as well as* serving as access structures to this data in warehouses. Because DataIndexes are both storage and access structures, substantial space savings are realized.

As we shall see, DataIndexes combine and extend, in an effective way, ideas embedded in other well-known database structuring techniques, specifically *vertical partitioning* and *transposed files* [57], as well as indexing techniques, specifically projection indexes [46] and bit-mapped join indexes [45].

As a second contribution, we propose two algorithms for efficiently performing star joins and show that these algorithms outperform existing approaches in a vast majority of situations.

A third contribution of this paper is an extensive mathematical analysis of the performance of OLAP queries based on DataIndexes as well as other indexes and query processing approaches. This analysis as well as a quantitative performance study show that DataIndexes are preferable to other popular index types in a large number of cases.

The rest of this paper is organized as follows. In Section 2 we motivate the need to reduce the additional storage costs introduced by the presence of indexes and in Section 3 we introduce the notion of DataIndexing and describe two types of DataIndexes. Also, we compare it with related work motivated by similar goals. In Section 4, we propose efficient algorithms for performing star-join operations with DataIndexes and in Section 5, we present a cost framework to model the performance of the proposed algorithms. In Section 6 we present the query performance costs for other selected indexing structures, in Section 7, we analyze the costs of performing star-join queries using the various indexing structures under study, and finally, in Section 8, we conclude the paper.

2 Motivation behind DataIndexes

DataIndexes are motivated by the desire to reduce the additional storage costs due to index structures. To illustrate these costs we refer to the star schema [37] presented in Figure 1, which was derived from the TPC-D benchmark database [62] with a scale factor of 1. This schema models the activities of a world-wide wholesale supplier over a period of 7 years, and will be used as a running example throughout this paper. The central *fact* table is the `SALES` table, and the dimensions of the data are captured through the `PART`, `SUPPLIER`, `CUSTOMER` and `TIME` tables. Each dimension table has a primary key. The fact table is associated, through foreign-key reference, to the four dimension tables. Note that some applications do not enforce referential integrity between the fact and dimension tables. However, we assume throughout the paper that *referential integrity is strictly enforced in all star schemas.*

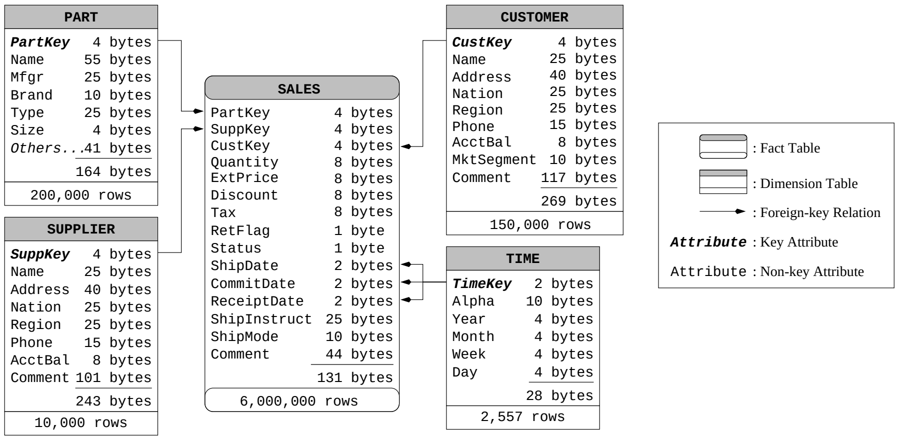

Figure 1: A Sample Warehouse Star Schema

We first compute the storage requirements for the data, based on a conventional relational implementation, where the 5 tables would be represented as a series of records partitioned over a set of data blocks. To simplify our analysis, we assume that each record is small enough to fit within a data block. Given the usually large sizes ($\geq$ 8 KB) of warehouse data blocks, this assumption is very realistic. We also assume that each block contains some header information (e.g., version number, pointers to other blocks, etc.) which makes its *effective size*, B, smaller than its actual size, B_{act}. Given a particular table, T, with a record width $w(T)$ and cardinality $|T|$, one can compute the number of records that fit in a data block, i.e., the *blocking factor* for T, as $\rho(T) = \left\lfloor \frac{B}{w(T)} \right\rfloor$. In turn, we determine the size of the table to be $B_{act} \times \left\lfloor \frac{|T|}{\rho(T)} \right\rfloor$. Let us further assume that the implementation platform uses a block size (B_{act}) of 8192 bytes and an effective block size (B) of 8000 bytes. From this we compute the size of each table as shown in Table 1.

Table	**Table Size** (bytes)
`SALES`	805,773,312
`PART`	34,136,064
`SUPPLIER`	2,564,096
`CUSTOMER`	42,377,216
`TIME`	73,728
all tables	884,924,416

Table 1: Table Sizes For Example Star Schema

We next compute the total storage requirements of the sample database based on the exact expressions for the size of the various index types derived in [65]. We have selected four popular index structures for data warehousing : *B-trees* [15], *bitmapped indexes* [44, 47], *bit-sliced indexes* [46, 34], and *projection indexes* [46, 61]. It is assumed that only the `PartKey`, `SuppKey`, `CustKey`, `ShipDate`, `CommitDate` and `ReceiptDate` columns of the `SALES` table are indexed. Table 2 summarizes the storage requirements of each indexing scheme for the sample database[1]. Table 2 indicates that the the indexing overhead for projection indexes is the least; the

Indexed Column	B^+-tree	Bitmapped	Projection	Bit-sliced
`PartKey`	1,675,288,576	1,640,816,640	24,576,000	24,649,728
`SuppKey`	118,800,384	82,780,160	24,576,000	24,649,728
`CustKey`	1,265,680,384	1,230,807,040	24,576,000	24,649,728
`ShipDate or CommitDate or ReceiptDate`	36,896,768	21,733,376	12,288,000	12,324,960
total	3,170,459,648	3,019,603,969	110,592,000	110,936,064

Table 2: Indexing Costs for Example Star Schema

other access structures yield higher storage overheads. But even projection indexes incur a more than 12.5% increase in size of the indexed database over the unindexed database, assuming the index structures are stored in addition to the data. (Some implementations of projection indexes, such as Sybase IQ [61], do not store both the index and data.) We also emphasize that projection indexes are not very effective for many OLAP queries. While they may perform well for simple queries involving a single table (e.g., restrictions, counting), they do not offer any improvement over conventional pairwise join techniques [21]. Typically, additional indexes are required along with projection indexes to improve join performance, incurring additional overhead.

Clearly, there is a need to minimize the additional overhead incurred by index structures. This is the motivation behind DataIndexes, introduced next.

3 The DataIndex

In this section, we propose the *DataIndex*, which is a storage structure that serves both as an index as well as data. Specifically, we examine two types of DataIndexes, both based on the same basic idea of vertical partitioning. We refer to these as *Basic DataIndexes* and *Join DataIndexes*. We then compare Dataindexes to existing indexing approaches and then discuss physical database design based on DataIndexes.

3.1 Basic DataIndex (BDI)

A DataIndex can be simply created as a vertical partition of a relational table. 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 ordinal position as its corresponding base record. The isolated partitions can then be used for fast access to data in the table. We call this partition a *Basic DataIndex* (BDI). A graphical representation of this structure is shown in figure 2.

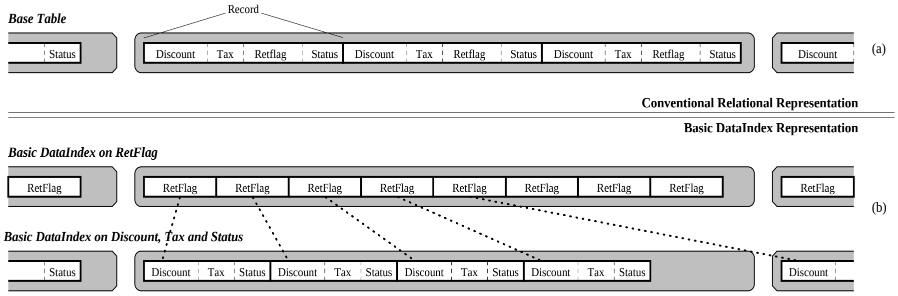

Figure 2: The Basic DataIndex

[1]Note that the sizes of both the standard and bitmapped B-trees depend on the distribution of data values. The numbers presented in table 2 correspond to a perfectly uniform distribution of values.

In this figure, we show the actual storage configurations of the two cases: a base table (Fig. 2a) and the corresponding BDIs (Fig. 2b). The base table consists of the attributes Discount, Tax, RetFlag, Status and the two BDIs are constructed on the RetFlag column, and on the Discount, Tax and Status columns.

As indicated by the dotted lines joining records from the two BDIs, the order of the records in the base table is conserved in both DataIndexes. This allows for an easy mapping between entries in the two BDIs. This mapping relies on the fact that, for a given BDI, there exists a fixed number of "slots" for holding records in each data block. Denoting by $w(\beta)$ the width of a record of BDI β, the number of slots per page in the BDI is given by $\rho(\beta) = \left\lfloor \frac{B}{w(\beta)} \right\rfloor$.

Based on this value, one can simply associate the elements of a record in the two BDIs through a simple arithmetic mapping. Assume for instance that we need to access the record in the base table corresponding to some RetFlag value. This translates to associating the corresponding RetFlag BDI record r with its matching record in the other BDI. Denoting the BDI on RetFlag by β_R and the BDI on Discount, Tax, and Status by β_{DTS}, we first compute the RID of record r in β_R, i.e., we determine that r is located in the $(b_R)^{\text{th}}$ block of β_R, at slot number s_R. This can be done by simply keeping track of the number of data blocks loaded during the scan operation and seeing the position of the matching record within its data block. The ordinal position, Ω, of this record in the base table is simply given by $\Omega = b_R \times \rho(R) + s_R$. From this number, we can determine the RID of the corresponding entry in the other BDI (β_{DTS}). This RID is characterized by b_{DTS}, the ordinal number of the β_{DTS} data block in which the record is located, and s_{DTS}, the slot in block b_{DTS} corresponding to record r. This RID can be expressed as $b_{DTS} = \left\lfloor \frac{\Omega}{\rho(R)} \right\rfloor$, and the slot number as $s_R = \Omega \bmod \rho(R)$.

It is the *ordinal mapping* that makes this basic approach more efficient than existing vertical partitioning methods such as the Decomposition Storage Model (DSM) [16, 36, 63]. Indeed, DSM utilizes surrogate keys to map individual attributes together, hence requiring a surrogate key to be associated with each attribute of each record in the database.

The resulting database size is essentially the same as the size of the raw data in the original database configuration. However, we can now utilize the separate dimensional columns of the partitioned fact table as both elements of and indexes onto that table. Through this simple technique, we can store the data and index for the same storage cost as for the data alone. Hence, in terms of storage, the indexing is free.

Turning to our running example, we can divide the `SALES` table in Fig. 1, into five smaller tables, as shown in Fig. 3. The new schema is then composed of 5 vertical partitions: one for each of the `SuppKey`, `PartKey`,

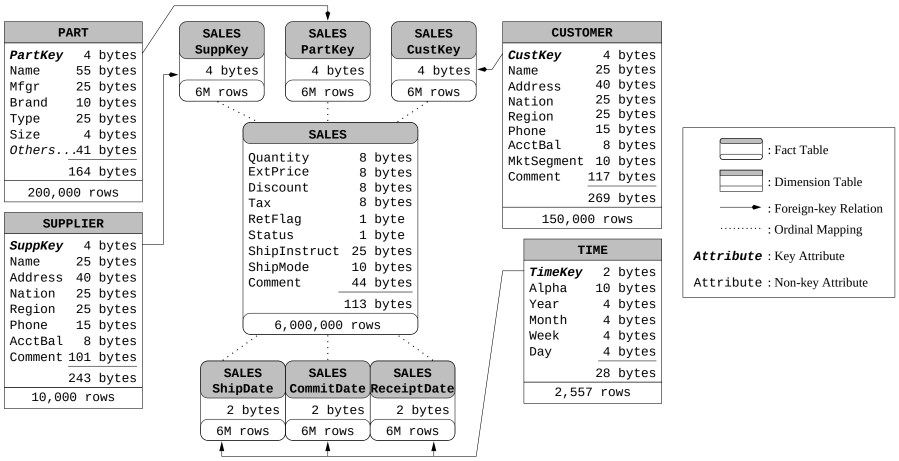

Figure 3: Example Warehouse Schema with DataIndex

and CustKey dimensional attributes, one for the combination of ShipDate, CommitDate, and ReceiptDate dimensional attributes and one for the remaining columns from the original SALES table. A record in the original SALES table is now partitioned into 5 individual records, one in each of the resulting tables. Any such record can easily be re-built from these, since its component rows in the 5 resulting tables all share the same *ordinal position*. In the example, each of the 5 new tables is a DataIndex.

We note that a potential problem arises in the presence of variable-length attributes (e.g., those of type VARCHAR). In such cases, the number of records can vary from one page to the next. To solve this problem, one can define a maximum number of records per page, as is done in the model 204 database system [44]. In this case, a few ordinal position numbers (Ω) may not actually correspond to actual records. Alternatively, one can "encode" each unique value to a fixed length surrogate. To simplify our analysis, in this paper we thus assume that field lengths are fixed with no loss of generality.

3.2 Join DataIndex (JDI)

In decision support databases, a large portion of the workload consists of queries that operate on multiple tables. Many queries on the star schema of Fig. 1 would access one or more dimension tables and the central SALES table. For instance, a marketing analyst might want to identify the part type most often purchased by different customer groups identified by their nation and market segment. The PART, CUSTOMER and SALES table must be joined to answer this query. Access methods that efficiently support join operations thus become crucial in decision support environments [45, 51]. The idea of a BDI presented in the previous section can very easily be extended to support such operations. Consider for instance, an analyst who is interested in possible trends or seasonalities in discounts offered to customers. This analysis would be based on the following query:

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

Using the conventional relational approach, the association between the two tables TIME and SALES in Fig. 1 is implemented through the primary key/foreign key relationship linking the columns ShipDate and TimeKey. To perform a join operation on these two tables, the two columns must be accessed to determine the records that are join candidates. There exist relatively fast algorithms (e.g., merge and hash joins) for evaluating joins. However, approaches that use pointers to the underlying data, instead of the actual records, tend to give a better performance than other join strategies [21]. Thus, if a DataIndex relied on pointers to records to both store and index the underlying data, it would perhaps have a good join performance.

Indeed, one can significantly reduce the number of data blocks to be accessed while processing a join by storing the RIDs of the matching records in the corresponding dimension table – instead of the corresponding key values – in a BDI for a foreign key column. This structure is a *Join DataIndex* (JDI). The JDI on SALES.ShipDate would then consist of a list of RIDs on the TIME table. Such a JDI is shown in Fig. 4. As before, we show both the conventional relational and our proposed representations. In the conventional approach, we show referential integrity links between the SALES and TIME tables as dashed arrows. For our proposed 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 JDI and the SALES BDI is preserved from the base table.

As can be seen in this figure, instead of storing the data corresponding to the ShipDate column, the JDI provides a *direct* mapping between individual tuples of the SALES and TIME tables. The join required to answer the above query can thus be performed in a single scan of the JDI (more details in section 4). This property

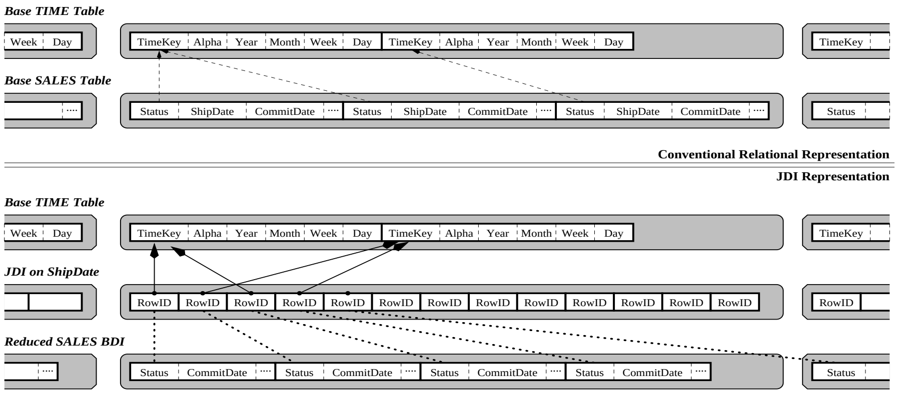

Figure 4: The Join DataIndex

of JDIs is indeed attractive, since the size of this index is, of course, proportional to the number of tuples in the table from which it was derived. In our example schema, for instance, the JDI on `ShipDate` contains 6 million entries. A join operation could thus be performed by examining each one of these entries in turn. This approach should be significantly faster than with conventional join algorithms, which typically perform joins between two or more tables in a pairwise fashion. Such algorithms include nested-loop joins, merge joins [4], hash-joins [5], or any derivative of these techniques [38, 56] (see [21] for a survey). In fact, the exact number of block accesses needed to scan a JDI is simply the number of data blocks occupied by this structure. This is given by $\left\lceil \frac{|\texttt{SALES}|}{\rho(r)} \right\rceil$, where r is the size of a RID (6 bytes). This results in $\left\lceil \frac{6{,}000{,}000}{\left\lfloor \frac{8000}{6} \right\rfloor} \right\rceil = 4502$ block accesses.

However, a JDI does not contain any data values. It might thus make the evaluation of queries where these values are needed more difficult. For instance, consider the following query:

```
SELECT ShipDate FROM SALES
```

If the `ShipDate` column is stored as a JDI, this query requires access to both the `SALES` and `TIME` tables, even though the latter one is not explicitly specified. We can thus compute the number of block accesses necessary to evaluate the above query as $4502 + \left\lceil \frac{|\mathrm{TIME}|}{\rho(\texttt{TIME})} \right\rceil = 4511$. This figure is somewhat larger than would be required with a BDI. With a BDI, evaluating the above query would require only $\left\lceil \frac{|\texttt{SALES}|}{\rho(\mathrm{ShipDate})} \right\rceil = 1500$ block accesses.

However, this would not always be the case: if the foreign table is small and the width of the indexed column is large, then scanning it to obtain data values will be more efficient than actually scanning a corresponding BDI. For instance, consider the (somewhat unlikely) case where a TimeKey value is 8-bytes wide (i.e., $w(\mathrm{TimeKey}) = 8$). Evaluating the above query with a JDI would require $4502 + \left\lceil \frac{|\mathrm{TIME}|}{\rho(\texttt{TIME})} \right\rceil = 4513$ block accesses, where $\rho(\texttt{TIME})$ is now $\left\lfloor \frac{8000}{34} \right\rfloor$, while a BDI would require $\left\lceil \frac{|\texttt{SALES}|}{\rho(\mathrm{TimeKey})} \right\rceil = 6000$ I/O operations, where $\rho(\texttt{TimeKey})$ is now $\left\lfloor \frac{8000}{8} \right\rfloor$. There would thus appear to be situations where a JDI is, in fact, preferable to a BDI even when no join is explicitly involved[2]. Hence, even though a JDI is useful for a column storing foreign keys, it is also useful when the column is wide and the number of distinct values in the column is small. In this case, it is preferable to realize the column as a JDI with the addition of another (small) lookup column storing the distinct values in the original columns. When it pays to do this is precisely characterized in section 6.1.5.

[2]Interestingly, we can significantly reduce the cost of accessing the `TIME` table in this sort of query by storing its primary key column (i.e., TimeKey) as a BDI. In this case, the number of I/Os required to evaluate the above query with a JDI is equivalent to the number of blocks of *both* the JDI on SALES.ShipDate and the BDI on TimeKey. As it turns out, the `TIME` table is so small that only 3 blocks are required to store the BDI on TimeKey. Hence, in this particular example, the total number of block accesses required to evaluate the above query is only 4505, which is again smaller than would be required with a regular BDI (whose size, in this case, was computed above to be 6000 blocks).

In Sections 6.1.1 and 6.1.5 we perform an analysis of the performance of the different indexing schemes in order to characterize exactly when a particular indexing scheme is to be preferred. Now we provide a qualitative summary of the features embedded in DataIndexes.

3.3 Comparison of BDIs and JDIs with existing Indexing Approaches

The Basic DataIndex is closely related to the idea of the Projection Index. A projection index is simply a mirror image of the column being indexed. 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 the starting point of the proposed DataIndex scheme and is how Sybase IQ stores data [19, 46]. Furthermore, with DataIndexes, each BDI of a table is stored separately – with ordinal positon based mapping providing more efficient access to individual record fields compared to other vertical partitioning based methods. Because BDIs are stored separately, only columns of interest need to be loaded in memory when joins are performed. Another attractive aspect of BDIs, and a point of departure from pure projection indexes, is that each BDI can contain any number of columns from the original table, unlike projection indexes which are restricted to single columns. Finally, as mentioned previously in Section 2, projection indexes do not improve join performance. We introduce the notion of *Join DataIndexes* (JDI) for this purpose.

O'Neil and Graefe [45] briefly introduced the idea of a bitmapped join index for efficiently supporting multi-table joins. JDIs capitalize on this idea in the context of the Basic DataIndex. A bitmapped join index (BJI) associates related rows from two tables [45], as follows. Consider two tables, T_1 and T_2, related by a one-to-many relationship (i.e., one record of T_1 is referenced by many records of T_2). A BJI from T_1 to T_2 can be seen as a bitmapped index that uses RIDs of T_1 instead of search-key values to index the records of T_2. Using a similar basic philosophy, a JDI stores the RIDs of the matching records in the corresponding dimension table instead of the corresponding key values. There are however, two important differences between JDIs and BJIs implemented in commercial systems:

1. Commercial implementations of BJI (such as in INFORMIX) are tree structured and exist as separate index structures. JDIs are flat structures and *do not exist* as auxiliary index structures. Rather, JDIs are a representation of the base data itself.

2. As we saw, JDIs are useful even when no joins are performed, specifically, when a column stores foreign keys, it is useful when the column is wide and the number of distinct values in the column is small. This implies that with DataIndexes, *the amount of storage required may even be smaller than the storage required for the base tables.*

We discussed above how DataIndexes are different from the indexes proposed for warehouses so far. In this context we must also mention that the way *we use* these structures is also radically different from how current structures are employed. More specifically, we design a number of query processing algorithms that use DataIndexes in novel ways to deliver much improved performances of star-join queries in a large number of cases.

In summary, DataIndexes take the best aspects of vertical partitioning, projection indexes, and Bitmapped Join Indexes and integrates as well as extends them in (what we will show to be) an effective manner.

Before we conclude this section, it is important to point out that a number of *variant* indexes are supported in commercial products such as Sybase IQ [61], Oracle 8 [47], Informix Universal Server [34], and Red Brick Warehouse [51]. In addition to projection indexes[46] and bitmapped join indexes [45] mentioned already, such index structures include bitmapped indexes [44], bit-sliced indexes[46]. Wu and Buchmann [67] presented an efficient encoding scheme for reducing the size of bitmapped indexes, and Chan and Ioannidis [7] proposed a framework for the design and evaluation of bitmap indexing schemes. An analysis of three index structures along with B$^+$-trees is presented in [46], which indicates that these four structures are particularly appropriate for warehousing/OLAP environments. In Section 6, we present the results of extensive analysis of the previously proposed index structures along with DataIndexes.

3.4 The DataIndex Physical Design Strategy

Having introduced the two types of DataIndexes, we now briefly describe a physical design strategy based on these indexing structures. In general, we propose that a JDI be established for each foreign column in the fact table, and single-column BDIs be established for all other fact table columns and for all dimension table columns. Thus every column in a given star schema is represented as either a single-column BDI or as a JDI.

To illustrate, consider again the star schema of Figure 1. The foreign columns in the `SALES` fact table are `SALES.PartKey`, `SALES.SuppKey`, `SALES.CustKey`, `SALES.ShipDate`, `SALES.CommitDate`, and `SALES.ReceiptDate`. If we let $j_{\mathtt{A}}$ denote a JDI on attribute A in the `SALES` fact table, then our physical design strategy assumes the following JDIs are defined: $j_{\mathtt{PartKey}}$ corresponding to the `PART` dimension table, $j_{\mathtt{SuppKey}}$ corresponding to the `SUPPLIER` dimension table, $j_{\mathtt{CustKey}}$ corresponding to the `CUSTOMER` dimension table, $j_{\mathtt{ShipDate}}$, $j_{\mathtt{CommitDate}}$, and $j_{\mathtt{ReceiptDate}}$ all corresponding to the `TIME` dimension table. The remaining columns in the fact table would be stored as BDIs. For instance, single-column BDIs would be defined for `SALES.Quantity`, `SALES.ExtPrice`, and `SALES.Discount`. Likewise, single-column BDIs would be defined for all dimension table columns. These would include, for instance, `TIME.TimeKey`, `TIME.Alpha`, `TIME.Year`, `TIME.Month`, `TIME.Week`, and `TIME.Day` for the `TIME` dimension table. The physical design strategy we have described will be assumed throughout this paper.

4 Fast Star-Join Algorithms Based on DataIndexes

A common operation in OLAP applications is the *star-join* query. In a star-join, the fact table is joined with a set of dimension tables. Due to the large size of most data warehouses, a star-join is typically an extremely expensive operation. As mentioned previously, response time is critical in OLAP applications. Therefore, it is imperative to have algorithms that can perform star-join queries very quickly. Such algorithms must ensure that the appropriate access structures are utilized. In this section, we present efficient algorithms for performing star-join operations with DataIndexes.

A typical OLAP query is of the form

$$\texttt{SELECT}\ \text{column list}\ \texttt{FROM}\ F,\ D_1,\ \ldots,\ D_m\ \texttt{WHERE}\ P_{\bowtie}\ \texttt{AND}\ P_{\sigma}$$

where F is the central fact table, $D_1, \ldots, D_m$ are the m dimensional tables participating in the join, P_σ is a set of *selection* predicates (i.e., each individual predicate only concerns one table), and $P_{\bowtie}$ is a set of *join* predicates (i.e., each predicate is of the form $F.A_1 = D_i.A_2$). To illustrate, consider the following query, based on our example from section 3, which lists the prices for sales made locally by suppliers in the United States:

```
SELECT U.Name, 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
```

In this query, the selection predicates (i.e., P_σ) are "`T.Year BETWEEN 1996 AND 1998 AND U.Nation='United States' AND C.Nation='United States'`"; the joining predicates ($P_{\bowtie}$) on the other hand are "`S.ShipDate = T.TimeKey AND S.CustKey=C.CustKey AND S.SuppKey = U.SuppKey`". We will utilize this query example through the remainder of our discussion.

Before presenting the algorithms, we first briefly describe the notion of a *rowset*, an important underlying concept used throughout this analysis. A rowset is simply a representation of selected tuples from a table. Evaluation of a star-join query consists of two phases: creating access structures to identify which tuples are to be retrieved to answer the query, and retrieving the actual data for the selected tuples. A rowset is the access structure used in the first phase. Two approaches to representing a rowset would be to represent it as a list of *row identifiers* (RIDs) or a bit vector [46]. In a RID-list representation, a rowset can be thought of as a list structure containing a set of RIDs for selected tuples, and so the rowset cardinality is the number of selected tuples. In a bit vector representation, a rowset is a vector of bits (having cardinality of the table itself), where bits are set only for selected tuples.

The size of a rowset R, in either form, can easily be computed. For an RID-list representation, the size of the rowset is governed by the number of rows in the set, $|R|$. In this case, the minimum size of the rowset representation is $r \times |R|$, where r is the size of a RID. For a bit vector representation, the size of a rowset is governed by the number of records present in the table. It is given by $\left\lceil \frac{|T|}{8} \right\rceil \approx \frac{|T|}{8}$. Thus, from the point of view of storage requirements, a RID-list representation is better than a bit vector if the following condition holds.

$$r \times |R| < \frac{|T|}{8} \Longleftrightarrow \frac{|R|}{|T|} < \frac{1}{8r} \qquad (1)$$

In other words, a RID-list is only smaller when the *selectivity* ($\frac{|R|}{|T|}$) of the rowset is less than $\frac{1}{8r}$. In the example presented in section 3, where r is 6 bytes, a RID-list representation of a rowset would only be better if this rowset corresponds to less than 2% of the number of records in the underlying table. In decision support environments, many queries access significant portions of the underlying database [11]. In addition, many operations on bitmaps are much faster than on RID lists[46]. For these reasons, in the remainder of this paper we assume that the rowsets used in evaluating a selection predicate are implemented as bit vectors.

Now we turn our attention to analyzing how a star-join query is evaluated in a data warehouse.

Essentially, star-join query is evaluated in two phases: the range selection phase and the join phase. In the range selection phase, the selection predicates (P_σ) are applied individually to each table that participates in the join. This results in a set of rowsets that indicate which tuples from each table are candidates for inclusion in the join result. In the join phase, the rowsets are used in conjunction with index structures to retrieve the data for tuples appearing in the join result.

To illustrate this approach, consider again our sample query that was presented earlier in this section. The set of dimension tables participating in the join is $\mathfrak{D} = \{\texttt{TIME}, \texttt{CUSTOMER}, \texttt{SUPPLIER}\}$, the set of dimension table columns that contribute to the join result is $\mathfrak{C}_{\mathfrak{D}} = \{\texttt{SUPPLIER.Name}\}$, and the set of fact-table columns that appear in the result is $\mathfrak{C}_F = \{\texttt{ExtPrice}\}$.

To answer this query, we would begin the range selection phase by first applying the predicates `T.Year BETWEEN 1996 AND 1998`, `U.Nation='United States'` and `C.Nation='United States'` to the corresponding dimension tables (i.e., `TIME`, `CUSTOMER` and `SUPPLIER`). These selections would result in a set of rowsets, $\mathfrak{R}$, one for each of the dimension tables involved (i.e., $\mathfrak{R} = \{R_{\texttt{TIME}}, R_{\texttt{CUSTOMER}}, R_{\texttt{SUPPLIER}}\}$). Since no predicates were applied on the fact table, F, the corresponding rowset, R_F, corresponds to all tuples of F.

Once the range selection phase completes, the join phase would commence. We now discuss the execution of these two phases separately. Once we have completed describing the two phases, we will present our analysis which computes the costs of performing each phase.

4.1 The Range Selection Phase Using DataIndexes

In this phase, rowsets are computed based on the restriction (selection) criteria applied in the query under consideration. Performing this phase using BDIs and JDIs is quite simple.

To evaluate a selection using a BDI, it is necessary to scan the entire BDI and evaluate the selection predicate on each value in the BDI. For example, to evaluate the predicate `T.Year BETWEEN 1996 AND 1998`, it would be necessary to scan the `T.Year` BDI and generate a rowset where a set bit corresponds to an ordinal position such that the record at this position in the BDI satisfies the predicate.

A JDI is fundamentally different from a BDI in that none of the search-key values are present in the index. Rather, the RIDs corresponding to foreign records that hold these values are stored in the index. So, evaluating a predicate-based selection operation on a JDI can not be done by only accessing the JDI. Rather, the foreign column is first scanned, and a rowset of all matching entries is generated and kept in memory. The JDI is then scanned and a second rowset is constructed by determining which entries in the JDI are present in the first rowset.

For instance, consider the TPC-D schema shown in Figure 1. Assume a JDI exists on `SALES.SuppKey` and a BDI exists on `SUPPLIER.Nation`. To evaluate the predicate "`SUPPLIER.Nation = 'United States'` " requires first scanning the `SUPPLIER.Nation` BDI and determining which values have '`United States`' in the `Nation` field. A rowset (having cardinality equal to that of the `SUPPLIER` table) indicating which entries in the `SUPPLIER` table meet this condition is then created and kept in memory. Next the JDI on `SALES.SuppKey` is scanned and compared to the first rowset to determine which entries in the `SALES` fact table meet the predicate condition. The result of this comparison is a second rowset (having cardinality equal to that of the `SALES` table) indicating the requested `SALES` records. Having described the range selection phase, we now turn our attention to describing the *join phase* evaluation of a star-join query.

4.2 The Join Phase

A star-join can be evaluated in a variety of ways using DataIndexes. We propose two such approaches. Each one should be used depending on the amount of available memory. We call the first approach *Star-Join with Large memory* (SJL). It is the more efficient of the two in terms of response time, but may require significant amounts of memory in certain cases. The second one is somewhat less efficient but has negligible memory requirements. We refer to it as *Star-Join with Small memory* (SJS).

4.2.1 SJL algorithm

The basic idea behind SJL is to perform a star join with a single pass over each table participating in the join. Clearly the performance afforded by such an algorithm would be difficult to improve upon. The SJL algorithm is shown in Algorithm 1.

To illustrate this approach, consider again our sample query that was presented earlier in this section. The set of dimension tables participating in the join is $\mathfrak{D} = \{\texttt{TIME}, \texttt{CUSTOMER}, \texttt{SUPPLIER}\}$, the set of dimension table columns that contribute to the join result is $\mathfrak{C}_{\mathfrak{D}} = \{\texttt{SUPPLIER.Name}\}$, and the set of fact-table columns that appear in the result is $\mathfrak{C}_F = \{\texttt{ExtPrice}\}$.

Algorithm 1 SJL (Star Join with Large memory)

Note: Needs enough memory to hold all dimension BDIs used in the join result.

Input:

$\mathfrak{D}$: set of dimension tables involved in the join.

$\mathfrak{C}_{\mathfrak{D}}$: set of dimension table columns that contribute to the join result.

$\mathfrak{C}_F$: of fact-table columns that contribute to the join result.

$\mathfrak{R}$: set of rowsets, one for each table in $\mathfrak{D}$ and one for the fact table F ($\mathfrak{R} = \{R_1, \ldots R_{|\mathfrak{D}|}, R_F\}$). These are computed through the range selection phase, before SJL starts. Note that $R_1, \ldots R_{|\mathfrak{D}|}$ are already loaded into memory, whereas R_F is not.

1: **for each** column $c_i \in \mathfrak{C}_{\mathfrak{D}}$ **do**
2:    **for each** row $r \in R_i$ **do**
3:       **if** the block of BDI on c_i where r is located is not loaded **then**
4:          Load this block into memory array a_{c_i} and pin in memory
5: **for each** row $r \in R_F$ **do**
6:    **for each** JDI j on a table of $\mathfrak{D}$ **do**
7:       **if** $j(r) \notin R_j$ **then**
8:          **goto** 5
9:    **for each** $c_i \in \mathfrak{C}_{\mathfrak{D}}$ **do**
10:       $s[c_i] \leftarrow a_{c_i}(j_{c_i}(r))$
11:    **for each** $c_j \in \mathfrak{C}_F$ **do**
12:       $s[c_j] \leftarrow r[c_j]$
13:    Output s.

Now assume that the range selection phase is complete for this query, using methods outlined in section 4.1. The output of this phase is a set of rowsets, $\mathfrak{R}$, one for each of the dimension tables involved (i.e., $\mathfrak{R} = \{R_{\texttt{TIME}}, R_{\texttt{CUSTOMER}}, R_{\texttt{SUPPLIER}}\}$). Since no predicates were applied on the fact table, F, the corresponding rowset, R_F, corresponds to all tuples of F.

The SJL algorithm would then execute the join phase, beginning by loading all blocks of the dimensional BDIs where it is known that some record of interest occurs that appear in the join result (i.e., all columns in $\mathfrak{C}_{\mathfrak{D}}$). This is done by steps 1 through 4 in Alg. 1, by scanning rowset $R_{\texttt{SUPPLIER}}$. The result of this operation is that the appropriate blocks of the `SUPPLIER.Name` BDI would be in memory, along with the three rowsets generated during predicate selection (i.e., $R_{\texttt{TIME}}$, $R_{\texttt{CUSTOMER}}$, and $R_{\texttt{SUPPLIER}}$).

After this, SJL would begin scanning the appropriate `SALES` JDIs to determine which `SALES` records should appear in the join result (steps 5-8). This would proceed as follows. For each record in the fact table, the JDIs linking F to elements of $\mathfrak{D}$ are examined (step 6). We will denote these JDIs as $j_{\texttt{TIME}}$, $j_{\texttt{CUSTOMER}}$ and $j_{\texttt{SUPPLIER}}$, corresponding to the three elements of $\mathfrak{D}$. SJL uses these JDIs to "look up" matching entries in the corresponding rowsets (step 7). To illustrate, consider two successive fact-table rows, r_1 and r_2. SJL would first examine the entry corresponding to r_1 in $j_{\texttt{TIME}}$ (step 6). This entry, noted $j_{\texttt{TIME}}(r_1)$, is a RID onto the `TIME` dimension table and it can thus be used to access the corresponding bit in $R_{\texttt{TIME}}$ through a simple array look-up (step 7). Assume that this bit is cleared (i.e., set to 0); SJL would then simply skip r_1 and examine the next record (step 8). This next record, r_2, would undergo a similar set of operations. First of all, the corresponding entry in $j_{\texttt{TIME}}$ (step 6) (i.e., $j_{\texttt{TIME}}(r_2)$) would be checked against $R_{\texttt{TIME}}$ (step 7); assuming that r_2 corresponds to a sales in the years 1996 to 1998, then the next JDI, i.e., $j_{\texttt{CUSTOMER}}$ would be checked (step 6). If the corresponding bit in $R_{\texttt{CUSTOMER}}$ is set to 1 (step 7), then the last JDI (i.e., $j_{\texttt{SUPPLIER}}$) would be checked as well (step 6). Assuming that the corresponding bit in $R_{\texttt{SUPPLIER}}$ is also set to 1 (step 7), then this would indicate that r_2 does indeed appear in the join result.

Once a fact-table row r has been identified as contributing to the join result, SJL builds the corresponding join record (steps 9-12) prior to output (step 13). To do this, the corresponding entry in the in-memory BDI for `SUPPLIER.Name` is accessed and used to construct the output record s (step 9-10). To access the correct entry, the RID of the `SUPPLIER` record referenced by r is simply obtained from $j_{\texttt{SUPPLIER}}$ (step 10). This RID is mapped to an ordinal position and used to access the in-memory BDI on `SUPPLIER.Name`. Since this BDI is represented as an array ($a_{\texttt{SUPPLIER.Name}}$), this step simply consists of a lookup into the array, based on the ordinal position in `SUPPLIER` of $j_{\texttt{SUPPLIER}}(r)$ (step 10).

Finally, the attribute values corresponding to fact-table columns are loaded from disk to complete the output record (step 11-12). In our example, the only such column is `ExtPrice`. The appropriate page from the `ExtPrice` BDI would then simply be loaded from disk and the corresponding attribute (i.e., r[`ExtPrice`]) would be used to finish constructing output record s (step 12).

4.2.2 The SJS algorithm

Recall that in SJL (steps 1-4 of Algorithm 1), the dimension table BDI's for columns appearing in the join result are loaded and pinned in memory. Thus, SJL assumes that all relevant columns from dimension tables are small enough to fit in memory. Clearly, in some cases, this assumption is not realistic. In Algorithm 2, we present the SJS algorithm for performing a star-join on a DataIndexed warehouse, when the combined size of all relevant columns from dimension tables cannot fit in memory. Like SJL, the SJS algorithm assumes that all restrictions on participating tables have been computed and the results stored in a set of rowsets, $\mathfrak{R} = \{R_1, \ldots, R_{|\mathfrak{D}|}, R_F\}$, with the dimensional rowsets $\{R_1, \ldots, R_{|\mathfrak{D}|}\}$ loaded into memory prior to the start of the algorithm. Thus, it is assumed that enough memory exists to load all dimensional rowsets. However, it is *not* assumed that sufficient memory exists to load all dimension table BDIs. Rather, the join is performed on smaller subsets of the dimensional BDIs by loading as many blocks of these BDIs as will fit into the available memory. Temporary structures and merge techniques are required to perform these operations. Clearly memory plays a critical role in the performance of this algorithm, which we will analyze in more detail in a later section. For now, we describe the SJS algorithm in more detail.

Algorithm 2 SJS (Star Join with Small memory)

Note: Has negligible memory requirements.
Input: Same as in Algorithm 1.
for each JDI j on a table of $\mathfrak{D}$ **do**
  for each row $r \in R_F$ **do**
    if $j(r) \notin R_j$ **then**
      $R_F \leftarrow R_F - \{r\}$ `/* turn corresponding bit off */`
for each JDI j_t on a table $t \in \mathfrak{D}$ **do**
  for each row $r \in R_F$ **do**
    write $j_t(r)$ to temporary JDI $j_{t,\mathrm{temp}}$ on disk
for each BDI b_i on a column $c_i \in \mathfrak{C}_{\mathfrak{D}}$ **do**
  Create output BDI $b_i(\mathrm{out})$ on disk
  $k \leftarrow 1$
  while $\exists$ unloaded blocks of b_i **do**
    Load as many blocks of b_i as possible into in-memory array a_i
    for each row r in $j_{i,\mathrm{temp}}$ **do**
      if $b_i(r) \in a_i$ **then**
        $b_i(\mathrm{out}) \leftarrow b_i(r)$ `/* write matching entry to output BDI */`
    $k \leftarrow k + 1$
for each row $r \in R_F$ **do**
  for each column $c_i \in \mathfrak{C}_{\mathfrak{D}}$ **do**
    $s[c_i] \leftarrow b_i(r)$
  for each column $c_j \in \mathfrak{C}_F$ **do**
    $s[c_j] \leftarrow r[c_j]$
  Output s

We describe the algorithm in four main phases. We refer to the first phase (steps 1-4) as the *fact table rowset* (R_F) *restriction phase.* This phase restricts the initial rowset on the fact table, R_F, so that it only indicates records that will appear in the join result. This is done by accessing the JDIs of the fact table corresponding to all participating dimensional tables. This process is similar to scanning the JDIs in the SJL algorithm (steps 5-7 of Algorithm 1), except that R_F is updated. We refer to the second phase (steps 5-7) as the *JDI restriction phase.* This phase restricts the JDIs to those rows of the fact table that appear in the join result. This is done by simply scanning the restricted fact table rowset created in the previous phase. The resulting restricted JDIs are stored in temporary structures on disk. These first two phases basically prepare the data for the actual join, which we describe in the next two phases.

We refer to the third phase (steps 8-16) as the *output BDI creation phase.* This phase constructs an output BDI for each dimension table column appearing in the join result (i.e., all columns in $\mathfrak{C}_{\mathfrak{D}}$). This is done by loading into memory as many blocks of the dimension table BDI as will fit; i.e., loading some fraction of the BDI. Then for this loaded portion of the BDI, the restricted JDI is scanned to find matching BDI entries, which are then written to the output BDI. Each value is written to the output BDI in the order corresponding to the restricted JDI, so the output BDI will have the same cardinality as the restricted JDI. The JDI and output BDI are processed sequentially, one block at a time. This processing is repeated as many times as necessary to load all dimension table BDIs. The same processing is done for all columns in $\mathfrak{C}_{\mathfrak{D}}$.

Referring back to our example query, recall that $\mathfrak{C}_{\mathfrak{D}} = \{\texttt{SUPPLIER.Name}\}$. Suppose the total memory required for the entire `SUPPLIER.Name` BDI is 250 MB, yet only 64 MB of memory is available. In other words, we assume here that 64 MB of memory is available after accounting for the memory already occupied by the JDI and output BDI blocks that are currently loaded for this phase. Clearly we cannot load all dimensional BDIs at once, but rather, can load up to 64 MB at a time. If we assume an effective block size of 8 KB, then we can load 8,000 blocks of this BDI at once ($\lceil \frac{64\ MB}{8\ KB} \rceil$), which is roughly one-fourth of the `SUPPLIER.Name` BDI ($\lceil \frac{250\ MB}{64\ MB} \rceil$). Once we have 8,000 blocks of the `SUPPLIER.Name` BDI loaded, the JDI for the `SUPPLIER` dimension, $j_{\texttt{SUPPLIER}}$, is scanned for RIDs that point to one of the BDI entries in memory. Matching entries are then written to the output BDI. This process is repeated until all blocks for the BDI have been loaded, 4 times in our example. The result is the output BDI for `SUPPLIER.Name`, which contains the corresponding output values in fact table order based on the restricted JDI.

We refer to the fourth phase (steps 17-22) as the *final output merge phase.* This phase creates the final output by scanning R_F and merging the dimension table output BDIs (created in the third phase) with the fact table output BDIs (i.e., all columns $\mathfrak{C}_{\mathfrak{F}}$). In our example, for each record in R_F, the corresponding value from the output BDI for `SUPPLIER.Name` is obtained along with the corresponding value from the output BDI for `ExtPrice`. Note the simplicity of this phase since all output BDIs are in fact table order.

We now proceed to estimate the cost of performing star-joins using the algorithms outlined above.

5 Cost Analysis of the Star-Join Algorithms

In a relational system, a query is generally first translated from its original format (e.g., SQL) into some internal format (often an extension of SPJ relational algebra) which is then used by the query optimizer to determine a query execution plan, i.e., a sequence of data and index accesses and manipulations. This plan is then executed as a series of disk accesses – which load the relevant portions of the database to main 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. The mapping operations needed for DataIndexes are similar to the ones used in other systems, which utilize bitmap vectors (in bitmapped indexes or for bitmapped rowsets). Depending on the system in use, the *logical* block numbers discussed in the previous section are translated into physical block IDs either by the operating system or by low-level routines of the storage manager of the DBMS itself. The logical block numbers allow the system to work as if the files were allocated contiguous storage when in fact they are not. In all cases the mapping operations can be implemented through a few integer operations and are thus quite fast. In fact, in most cases, we believe that the delays associated with these computations will be negligible compared to the much slower storage access times. This belief is strengthened by other studies [46, 31], 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 the rest of the paper, we will focus on analyzing the index structure performance with respect to disk access. Specifically, we characterize the performance of a query as the number of data blocks, $\mathcal{N}$, that are accessed during the execution of that query.

Overall, the number of block accesses necessary to perform a star-join, $\mathcal{N}_{\text{star}}$, can be expressed in terms of the cost of creating all initial rowsets and that of joining the corresponding rows together to compute the final result. More specifically, we define $\mathcal{N}_{\texttt{ROWSET}}$to be the number of data block accesses required to construct a rowset corresponding to a particular selection predicate. We also define $\mathcal{N}_{\texttt{JOIN}}$to be the number of block accesses required to join theses rowsets to form the final query result. Putting these two costs together gives the following expression for the cost (in terms of number of block accesses) to perform a star-join:

$$\mathcal{N}_{\text{star}} = \sum \mathcal{N}_{\texttt{ROWSET}} + \mathcal{N}_{\texttt{JOIN}} \tag{2}$$

Following this model, we now present an analysis of the cost of performing rowset constructions based on the DataIndexes and then present two separate analyses of the cost of performing the actual join based upon the two different algorithms outlined in the previous section.

First, we summarize in table 3 the notation used throughout this paper. Note that some of the notation in this table has not yet been used.

Notation	**Description**
B	Effective Size, in bytes, of a data block
π	Size, in bytes, of a pointer to a data block
r	Size, in bytes, of a RID
$\|T\|$	Number of records present in table T
V	Number of distinct values present in the column being indexed
ς_T	Selectivity factor on table T ($0 \leq \varsigma_T \leq 1$)
c	Distinctness factor of range selection ($0 \leq c \leq 1$)
V_{range}	Number of distinct search-key values referenced by a particular range selection (i.e., number of all such v_k values present in the table such that $k_1 \leq v_k \leq k_2$). Note that $V_{\text{range}} = \varsigma_T\|T\|c$
$w(C)$	Width, in bytes, of a particular column C
$w(T)$	Width, in bytes, of a table T
K	Number of search key values per node of B$^+$-tree
P	Order of B$^+$-tree , i.e., $P = K + 1$
f	Compression factor such that $0 < f \leq 1$ where $f = 1$ indicates no compression
$\mathfrak{D}$	Set of dimension tables involved in a join
$\mathfrak{C}_{\mathfrak{D}}$	Set of dimension table columns that contribute to the join result
$\mathfrak{C}_F$	Set of fact-table columns that contribute to the join result
R_i	Rowset corresponding to dimension i ($i = 1, 2, \ldots, \mathfrak{D}$)
R_F	Rowset corresponding to the fact table
$\mathfrak{R}$	Set of rowsets, one for each table in $\mathfrak{D}$ and one for the fact table F ($\mathfrak{R} = \{R_1, \ldots R_{\|\mathfrak{D}\|}, R_F\}$)
M_a	Number of blocks allocated to input BDI

Table 3: Notation used in this paper

5.1 Cost for Constructing Rowsets Using DataIndexes ($\mathcal{N}_{\texttt{ROWSET}}$)

We now examine the first component of query cost, $\mathcal{N}_{\texttt{ROWSET}}$, for BDIs and JDIs. Most OLAP selection operations will consist of range predicates, i.e., having the form $k_1 \leq C \leq k_2$, where k_1 and k_2 are (possibly equal) constants and C is the column being inspected. Our analysis can easily be extended to more complex predicates.

First, to evaluate a selection using a BDI, it is necessary to scan the entire list and evaluate the selection predicate on each value in the list. Hence, the cost of evaluating a selection on a column C indexed by a BDI is simply the number of block accesses required to scan the index:

$$\mathcal{N}_{\texttt{ROWSET}}(\text{BDI}) = \left\lceil \frac{|T| \times w(C)}{B} \right\rceil \tag{3}$$

where $|T|$ represents the number of records in the table, $w(C)$ represents the width of the column being indexed, and B is the effective size of a data block in bytes.

To simplify the analysis, we drop the ceiling ($\lceil\cdot\rceil$) function and approximate the cost of building a rowset using a BDI to be:

$$\mathcal{N}_{\texttt{ROWSET}}(\text{BDI}) \approx \frac{|T| \times w(C)}{B} \ . \tag{4}$$

Second, as mentioned previously, a JDI is fundamentally different from a BDI in that none of the search-key values are present in the index. Rather, the RIDs corresponding to foreign records that hold these values are stored in the index. So, evaluating a predicate-based selection operation on a JDI can not be done by only accessing the JDI. Rather, the foreign column is first scanned, and a rowset of all matching entries is generated and kept in memory. The JDI is then scanned and a second rowset is constructed by determining which entries in the JDI are present in the first rowset.

The cost to construct a rowset for the `SALES` fact table using this method yields the following expression, where the first term corresponds to scanning the BDI and the second term corresponds to scanning the JDI:

$$\mathcal{N}_{\texttt{ROWSET}}(\text{JDI}) = \left\lceil \frac{V\ w(C)}{B} \right\rceil + \left\lceil \frac{|T|r}{B} \right\rceil \tag{5}$$

Here V represents the number of distinct values present in the column being indexed. It is assumed that the foreign column is a primary key of the dimension table and is stored as a BDI. Thus the cardinality of the referenced BDI is V. Note that V is usually much smaller than $|T|$. In the second term, r represents the size of a RID in bytes. Applying the same simplifications as before results in the following expression for the cost to construct a rowset using a JDI:

$$\mathcal{N}_{\texttt{ROWSET}}(\text{JDI}) \approx \frac{V\ w(C) + |T|r}{B} \tag{6}$$

5.2 Cost for Joining Tables ($\mathcal{N}_{\texttt{JOIN}}$)

In this section we examine the cost to perform a join using both the SJL algorithm, $\mathcal{N}_{\texttt{JOIN}}(\text{SJL})$, and the SJS algorithm, $\mathcal{N}_{\texttt{JOIN}}(\text{SJS})$. In this analysis we assume that disk fragmentation is neglible. Unlike transactional processing systems, in a data warehouse, the emphasis is on querying rather than updating. Updates in a data warehouse typically occur in bulk, so records will be packed. It is therefore reasonable to assume that the degree of fragmentation will be insignificant and so we assume packed records throughout this analysis.

5.2.1 Cost of the SJL Algorithm

It should be clear from the discussion in Section 4.2.1 that SJL performs a single scan of the central SALES fact table, and that only a limited number of columns is ever examined. During this scan, only pages that correspond to records in R_F are actually considered (step 5 of Algorithm 1), and often, only a subset of the corresponding JDI pages will need to be loaded and examined (steps 6-8 of Algorithm 1). Indeed, a query optimizer should determine the order in which these JDIs should be examined, so as to minimize the number of page accesses. A simple rule of thumb for this type of optimization would be to select the JDI whose corresponding rowset has the smallest selectivity. Finally, once a record is known to participate in the join, only a subset of the columns from the different tables is ever accessed (steps 9-12 of Algorithm 1).

This simple approach allows for very efficient star-join evaluation. Indeed, the cost of a query on a fact table F and a set of dimension tables $\mathfrak{D}$ can be expressed as follows:

$$\mathcal{N}_{\mathtt{JOIN}}(\mathrm{SJL}) = \mathcal{N}_{\mathfrak{D}}(\mathrm{BDI}) + \mathcal{N}_F(\mathrm{JDI}) + \mathcal{N}_F(\mathrm{BDI}) \tag{7}$$

where $\mathcal{N}_{\mathfrak{D}}(\mathrm{BDI})$represents the cost of retrieving all blocks containing relevant records from dimensional BDIs, $\mathcal{N}_F(\mathrm{JDI})$, the cost of scanning each of the relevant JDIs from the fact table, and $\mathcal{N}_F(\mathrm{BDI})$, the cost of scanning all relevant records from fact table BDIs. We now derive expressions for each of these terms.

The cost to retrieve the relevant blocks for dimensional BDIs requires scanning the dimensional rowset for each dimension table column involved in the join result. This cost can be expressed as follows:

$$\mathcal{N}_{\mathfrak{D}}(\mathrm{BDI}) = \sum_{D\in\mathfrak{D}} \sum_{C\in\mathfrak{C}_D} \left\lceil \frac{|D| \times w(C)}{B} \right\rceil \tag{8}$$

where $\mathfrak{D}$ represents the set of dimension tables involved in the join, $|D|$ represents the cardinality of the rowset for dimension table D, and $\mathfrak{C}_D$ represents the set of columns from a table D that appear in the join result.

The cost to scan the relevant JDIs from the fact table is given by:

$$\mathcal{N}_F(\mathrm{JDI}) = |\mathfrak{D}| \left\lceil \frac{r|F|}{B} \right\rceil \tag{9}$$

Here $\left\lceil \frac{r|F|}{B} \right\rceil$ represents the number of block accesses required for a particular JDI and $|\mathfrak{D}|$ the number of dimension tables participating in the join. In the worst case, all foreign colums satisfy the restriction conditions and so all $|\mathfrak{D}|$ JDIs must be examined.

Finally, the last part of the algorithm involves loading columns from the fact table to complete the output record. The cost to scan the relevant records from the fact table is given by:

$$\mathcal{N}_F(\mathrm{BDI}) = \sum_{C\in\mathfrak{C}_F} \min\left(\varsigma_F|F|, \left\lceil \frac{|F| \times w(C)}{B} \right\rceil\right) \tag{10}$$

Here ς_F denotes the final selectivity on the fact table (i.e., the number of records in the join is $\varsigma_F|F|$ and $0 \le \varsigma_F \le 1$). The cost of loading the fact table columns depends on the selectivity. This cost will be the lesser of the number of records in the join (i.e., the first term in (10) by random access), and the total number of blocks required for all relevant columns in F (i.e., the second term in (10) by sequential scan of the entire index).

By inspection, we can thus establish that the dominant factor in these equations is $|F| \times |\mathfrak{D}|$, which indicates that the worst-case performance of this algorithm will be $O(|F| \times |\mathfrak{D}|)$ or simply $O(|F|)$ since $|\mathfrak{D}|$ is bounded by a small constant for a given star schema.

We note again that this efficient approach can only be utilized if enough memory can be allocated to the query. This memory requirement is given by

$$\mathcal{M}_{\texttt{JOIN}}(\mathrm{SJL}) = 1 + |\mathfrak{D}| + |\mathfrak{C}_F| + \sum_{D\in\mathfrak{D}} \sum_{C\in\mathfrak{C}_D} \left\lceil \frac{|D| \times w(C)}{B} \right\rceil + |\mathfrak{D}| \sum_{D\in\mathfrak{D}} \left\lceil \frac{|D|}{8B} \right\rceil . \tag{11}$$

Here the first term corresponds to one block of memory for the fact table, the second term corresponds to $|\mathfrak{D}|$ blocks for the JDIs, and the third term corresponds to $|\mathfrak{C}_F|$ blocks for the fact table BDIs. Thus we assume that the algorithm proceeds by accessing the fact table rowset, each JDI, and each fact table display column one block at a time. The fourth term corresponds to the memory requirements for the dimension table BDIs, and the last term corresponds to the memory requirements for the dimension table rowsets, both of which are loaded into memory for the duration of the algorithm.

From 11, we can conclude the following:

Result 1 *The memory requirements for the SJL algorithm are independent of the size of the fact table.*

This is an interesting result because it allows us to see that the SJL algorithm often does not require much memory, and that the memory requirements do not increase as the size of the fact table increases..

To illustrate, consider again the star-join query from Section 4, which joins the `SALES` fact table with the `TIME`, `CUSTOMER`, and `SUPPLIER` dimension tables. Using 11, it is easily shown that a total of 52 blocks of memory are required to answer the query using SJL. While this is by no means a "monster" query, it is certainly respectable. Yet, on any given system, it would only require about 416KB of memory, regardless of the size of the fact table. This amount of memory is small enough as to be even available on low-end personal computers. In addition, the corresponding columns could be used concurrently by other queries, thereby reducing the effective memory requirements of each query. This can be done because SJL always loads the entire columns "as-is", without pre-performing selections or reordering the data. Overall, thus, it would appear that the memory requirements of SJL are indeed acceptable for many OLAP-type queries.

5.2.2 Cost of the SJS Algorithm

We now present the cost to perform a join using the SJS approach. This cost is given by:

$$\mathcal{N}_{\texttt{JOIN}}(\mathrm{SJS}) = \mathcal{N}_{R_F} + \mathcal{N}_{\mathrm{JDI}} + \mathcal{N}_{\mathrm{OBDI}} + \mathcal{N}_{\mathrm{Merge}} \tag{12}$$

where $\mathcal{N}_{R_F}$represents the cost of the R_F restriction phase, (steps 1-4 of Algorithm 2), $\mathcal{N}_{\mathrm{JDI}}$represents the cost of the JDI restriction phase (steps 5-7), $\mathcal{N}_{\mathrm{OBDI}}$represents the cost of the output BDI creation phase (steps 8-16), and $\mathcal{N}_{\mathrm{Merge}}$represents the cost of the final output merge phase (steps 17-22). The expressions for each of these terms is given below. The cost to restrict R_F is given by:

$$\mathcal{N}_{R_F} = 2 \left\lceil \frac{|F|}{8B} \right\rceil + \mathcal{N}_F(\mathrm{JDI}) \tag{13}$$

The term $\left\lceil \frac{|F|}{8B} \right\rceil$ in (13) represents the total number of blocks required to store R_F. Since the entire rowset must be loaded and written back to disk, the first term thus represents the cost both to load and write the rowset. The second term represents the cost to load the JDIs and is given by (9).

The cost to restrict the JDIs is given by:

$$\mathcal{N}_{\text{JDI}} = \left\lceil \frac{|F|}{8B} \right\rceil + \mathcal{N}_F(\text{JDI}) + |\mathfrak{D}|\mathcal{N}_R(\text{JDI}) \tag{14}$$

where the first term represents the cost to load R_F and the second term represents the cost to load the JDIs and is given by (9). The third term represents the cost to write the new restricted JDIs where $\mathcal{N}_R(\text{JDI})$ is given by $\left\lceil \frac{r|F'|}{B} \right\rceil$ and $|F'| = \varsigma_F \varsigma_D^{|\mathfrak{D}|}|F|$. Here ς_F represents selectivity on the fact table and similarly, ς_D represents selectivity on dimension table D. For simplicity, it is assumed that selectivity is the same for all dimension tables. The effect of this restriction is to reduce the number of relevant tuples based on ς_F and ς_D. Hence the cardinality of each JDI is reduced to $|F'|$.

The cost to create the output BDIs can be expressed as:

$$\mathcal{N}_{\text{OBDI}} = \sum_{D \in \mathfrak{D}} \sum_{C \in \mathfrak{C}_D} L_C \left(\mathcal{N}_R(\text{JDI}) + \frac{\mathcal{N}_{\mathtt{i}}(C)}{L_C} + \mathcal{N}_{\mathtt{o}}(C) \right) \tag{15}$$

For each participating dimension table and for each column appearing in the join result, several passes may have to be made in order to scan the restricted JDI, locate the associated dimension table column value, and then write the value to the output BDI. We let L_C represent the number of passes required, which is given by $\left\lceil \frac{\mathcal{N}_{\mathtt{i}}(C)}{M_a} \right\rceil$, where $\mathcal{N}_{\mathtt{i}}(C)$ is the total number of blocks required for the input BDI and is given by $\left\lceil \frac{|D| \times w(C)}{B} \right\rceil$. M_a is the number of blocks allocated to the input BDI and depends on the *available* memory, which is the total memory less what is already loaded. We discuss memory requirements of SJS in more detail later in this section. $\mathcal{N}_{\mathtt{o}}(C)$ represents the total number of blocks required for the output BDI and is given by $\left\lceil \frac{|F'| \times w(C)}{B} \right\rceil$. Returning to the actual processing, for each pass, all blocks of the corresponding JDI must be loaded (the first term in (15)), the number of blocks of the input BDI that fit in memory must be loaded (the second term in (15)), and all blocks of the output BDI must be accessed (the third term in (15)).

The expression in (15) can be simplified to give the following:

$$\mathcal{N}_{\text{OBDI}} = \mathcal{N}_R(\text{JDI}) \sum_{D \in \mathfrak{D}} \sum_{C \in \mathfrak{C}_D} L_C + \mathcal{N}_{\mathfrak{D}}(\text{BDI}) + \sum_{D \in \mathfrak{D}} \sum_{C \in \mathfrak{C}_D} L_C \mathcal{N}_{\mathtt{o}}(C) \tag{16}$$

where $\mathcal{N}_{\mathfrak{D}}(\text{BDI})$is the cost to retrieve all blocks containing relevant dimension table BDIs and is given by (8).

The cost to create the final output can be expressed as:

$$\mathcal{N}_{\text{Merge}} = \left\lceil \frac{|F|}{8B} \right\rceil + \mathcal{N}_F(\text{BDI}) + \sum_{D \in \mathfrak{D}} \sum_{C \in \mathfrak{C}_D} L_C \mathcal{N}_{\mathtt{o}}(C) \tag{17}$$

where the first term corresponds to the cost to load R_F, the second term corresponds to the cost to load the fact table output BDIs and is given by (10), and the third term corresponds to the cost to load the dimension table output BDIs created in the previous phase.

The minimum memory requirements for SJS are quite small and are given by:

$$\mathcal{M}_{\mathtt{JOIN}}(\text{SJS}) = 1 + |\mathfrak{C}_{\mathfrak{D}}| + |\mathfrak{C}_F| + \sum_{D \in \mathfrak{D}} \left\lceil \frac{|D|}{8B} \right\rceil \ . \tag{18}$$

Since each phase in SJS has different memory requirements, (18) is based on the memory requirements for the final output merge phase, which requires the most memory of all phases. Therefore, the first term corresponds to one block of memory for R_F, the second term corresponds to one block for each of the dimension table output columns, and the third term corresponds to one block for each of the fact table output columns. Finally, the last term corresponds to the number of blocks required for the dimensional rowsets.

6 Comparative Analyses

Having analyzed the query processing cost based on DataIndexes, we now turn our attention to different types of indexing structures. This cost, as expressed in equation (2), is the sum of the cost to construct rowsets corresponding to all selection predicates ($\sum \mathcal{N}_{\texttt{ROWSET}}$), and the cost to join these rowsets ($\mathcal{N}_{\texttt{JOIN}}$). Following this model again, we analyze the cost of evaluating star join processing using different techniques and index structures. In the ensuing analysis we consider virtually all the state-of-the-art access structures used in datawarehouses currently. For rowset selections, we use *B-Tree* indexes, Bitmap indexes, *Bit-sliced* indexes and *Projection Indexes*, while for joins we use the *Bitmap Join Index*. In each case, we first derive expressions that yield the *best case expected performance* of each approach and then compare these results to determine which approach is the most promising, and under what conditions.

6.1 Comparative Anaysis of $\mathcal{N}_{\texttt{ROWSET}}$

6.1.1 $\mathcal{N}_{\texttt{ROWSET}}$ for B-tree Index

Possibly the most common indexing scheme available is the B^+-tree . This structure consists of a balanced tree whose nodes occupy each a single data block. The data blocks in the leaf level make up a sorted list of the V distinct search-key values in the column being indexed (in our sample query above, this would match, for instance, the number of unique values for the `Nation` field in the `Customer` table). Attached to each one of the unique values is a list of the RIDs of the records corresponding to that value. B^+-trees are often implemented so as to reduce the number of tree reorganizations necessary when the underlying data is updated. This translates to an average utilization of about 69% for the nodes in the tree. While this is indeed useful in transaction processing systems, this overhead is not needed in the read-mostly environment of data warehouses. In this study, thus, we assume that the tree is optimally filled (i.e., almost all nodes are full). We also assume that all values obtained in a range query are contiguous.

The cost to construct a selection predicate rowset with a B-tree can be expressed as follows:

$$\mathcal{N}_{\texttt{ROWSET}}(\text{B-tree}) = \mathcal{N}_{\text{descent}}(\text{B-tree}) + \mathcal{N}_{\text{leaf}}(\text{B-tree}) + \mathcal{N}_{\text{RID-list}}(\text{B-tree}) \tag{19}$$

where $\mathcal{N}_{\text{descent}}$ is the cost of descending the tree, $\mathcal{N}_{\text{leaf}}$ is the cost of scanning the leaf-level for all matching entries, and $\mathcal{N}_{\text{RID-list}}$ is the cost of actually accessing all RID-lists. We now derive expressions for each of these components.

The cost to descend a B^+-tree depends on the number of levels in the tree. The number of levels is given by $\lceil \log_P V \rceil$, where P is the *order* of the tree and V represents the number of distinct values present in the column being indexed. The order of the tree is $K+1$ where K is the number of search key values per node and K is given by $\left\lceil \frac{B}{w(C)+\pi} \right\rceil$. This expression determines the number of key values that can fit in a node given the size of each key value, pointer pair $(w(C)+\pi)$, and the effective blocksize, B. Since we do not need to include the leaf level, the cost to descend the tree is then as follows:

$$\mathcal{N}_{\text{descent}}(\text{B-tree}) = \lceil \log_P V - 1 \rceil \tag{20}$$

The cost of scanning the leaf-level is the number of blocks accessed at the leaf level and is given by:

$$\mathcal{N}_{\text{leaf}}(\text{B-tree}) = \left\lceil \frac{V_{\text{range}}}{K} \right\rceil \tag{21}$$

where V_{range} is the number of distinct search-key values referenced by the range selection and K is as defined previously. This expression follows from the fact that there are V_{range} distinct search-key values in the range and K key values per node. Note that in the best case, this expression evaluates to one since only a single block access is required.

The cost of accessing the RID-lists is given by the following expression:

$$\mathcal{N}_{\text{RID-list}}(\text{B-tree}) = \left\lceil \frac{r|T|}{V \times B} \right\rceil V_{\text{range}} \tag{22}$$

where r is the size of a RID in bytes, $|T|$ is the number of records present in the table, V_{range} is the number of distinct search-key values referenced by the range selection, and B and V are as defined previously. In the sample query from the previous section, V_{range} for the predicate "`T.Year BETWEEN 1996 AND 1998`" is 3, since the range covers 3 years. In deriving this expression, we assume that the distribution of distinct values is uniform. Thus the average number of RIDs per RID-list is given by $\frac{|T|}{V}$. The size of a RID-list, in bytes, is then $\frac{r|T|}{V}$. Dividing by the effective blocksize then gives the number of blocks per RID-list, $\left\lceil \frac{r|T|}{V \times B} \right\rceil$. Finally, multiplying by the number of distinct search-key values in the range results in the number of blocks required to access the RID-lists, as given in equation (22).

To simplify the analysis, we drop all ceiling ($\lceil \cdot \rceil$) functions and approximate the cost of building a rowset using a B-tree to be:

$$\mathcal{N}_{\texttt{ROWSET}}(\text{B-tree}) \approx \log_P V - 1 + \frac{V_{\text{range}}}{K} + \left(\frac{r|T|}{V \times B} \right) V_{\text{range}} \tag{23}$$

6.1.2 $\mathcal{N}_{\texttt{ROWSET}}$ for a Bitmap Index

A bitmapped index is identical to a conventional B-tree except that the rowsets corresponding to each unique search-key value are represented as bit vectors instead of RID lists [46]. As in the case of the B$^+$-tree , the cost of performing a range selection with a bitmapped index can be expressed as follows:

$$\mathcal{N}_{\texttt{ROWSET}}(\text{Bitmap}) = \mathcal{N}_{\text{descent}}(\text{Bitmap}) + \mathcal{N}_{\text{leaf}}(\text{Bitmap}) + \mathcal{N}_{\text{RID-list}}(\text{Bitmap}) \tag{24}$$

where $\mathcal{N}_{\text{descent}}$ and $\mathcal{N}_{\text{leaf}}$ are exactly the same as for the B$^+$-tree . The difference in these two structures appears in the third term, the cost to access the RID-lists, since these are stored differently in the two structures. In practice, in a bitmapped index, a certain amount of compression is typically employed in storing the bitmaps. We thus assume a compression factor f ($0 < f \leq 1$), which is a percentage indicating the compression level ($f = 1$ indicates no compression). We can then express $\mathcal{N}_{\text{RID-list}}$ as follows:

$$\mathcal{N}_{\text{RID-list}}(\text{Bitmap}) = f \left\lceil \frac{|T|}{8\ B} \right\rceil V_{\text{range}} \tag{25}$$

where $\frac{|T|}{8}$ is the size of a bit vector in bytes, and so $\left\lceil \frac{|T|}{8\ B} \right\rceil$ represents the average number of blocks per bit vector. Multiplying this expression by V_{range} gives the total number of blocks accessed, which is then weighted by the compression factor. Applying the same simplifications as in (23), the rowset construction cost using a bitmapped index can be expressed as:

$$\mathcal{N}_{\texttt{ROWSET}}(\text{Bitmap}) \approx \log_P V - 1 + \frac{V_{\text{range}}}{K} + f\left(\frac{|T|}{8\ B}\right) V_{\text{range}} \tag{26}$$

From equations 23 and 26, there appears to be a tradeoff between B$^+$-tree and bitmapped indexes. However, in practice, bitmapped indexes almost always require less storage than B$^+$-tree as significant compression can usually be achieved on the bitmaps. We now provide an illustrative example. A simple compression technique used in bitmapped indexes [46] is to represent the rowsets as bitmaps *only when the bitmap representation is smaller than a RID list representation.* It is easily seen that a bitmapped index constructed according to this method *can never require more storage than a B$^+$-tree* . Of course, this compression technique is quite simple and more effective compression mechanisms can be used. Thus we conclude the following result:

Result 2 *The performance of a bitmapped index is never worse than that of a conventional B-tree index.*

Because of this result, in the remainder of this paper, we do not consider the B$^+$-tree but rather concentrate on the bitmapped index.

6.1.3 $\mathcal{N}_{\texttt{ROWSET}}$ for a Projection Index

A projection index corresponds to a mirror copy of the column being indexed [46]. Like a single column BDI, to evaluate a selection using a projection index, it is necessary to scan the entire list and evaluate the selection predicate on each value in the list.

6.1.4 $\mathcal{N}_{\texttt{ROWSET}}$ for a Bit-sliced Index

Like the projection index and BDI, the bit-sliced index[3] also scans the entire index. In addition, each slice must be accessed in turn, which requires that a small header that points to each of the bit-slices will need to be accessed. Since the cost to access each slice is usually quite small, we can disregard this cost. Thus the actual cost incurred with these two techniques can be expressed as:

$$\mathcal{N}_{\texttt{ROWSET}}(\text{Bit-sliced}) = \mathcal{N}_{\texttt{ROWSET}}(\text{Projection}) = \left\lceil \frac{|T| \times w(C)}{B} \right\rceil \tag{27}$$

Applying the usual simplifications, the cost of evaluating a selection on a column C indexed by either of the above two methods is simply given by:

$$\mathcal{N}_{\texttt{ROWSET}}(\text{Bit-sliced}) \approx \mathcal{N}_{\texttt{ROWSET}}(\text{Projection}) \approx \frac{|T| \times w(C)}{B} \tag{28}$$

Note that equations (27) and (28) are equivalent to (3) and (4), respectively. Thus we conclude that the cost to construct a rowset using either a projection index, bit-sliced index, or BDI are essentially the same.

6.1.5 Cost comparisons

Having determined the best-case performance of rowset evaluation with different techniques, we can now compare these performances to understand the conditions under which a particular scheme performs the best. To lend some structure to these comparisons, we classify the indexing mechanisms into two classes: (a) *Indexing*

[3] [46] gives an efficient algorithm for performing ranging queries on bit-sliced indexes. This algorithm uses multiple bit vectors to compute the final rowset. These intermediary bit vectors are generated by scanning each bit slice in the index.

techniques based on ordinal positions, which include projection indexes, bit-sliced indexes, BDIs and JDIs, and (b) *Tree Based Indexing Techniques*, which include B-Trees and Bitmapped indexes.

Comparison of Indexing Techniques based on Ordinal Positions

From expressions (4) and (28), it can be seen that the performances of projection indexes, BDIs and bit-sliced indexes is equivalent. We thus turn our attention to comparing BDIs and JDIs. Using (4) and (6), we can determine that the performance of a BDI is better than that of a JDI if

$$\frac{r}{w(C)} \geq 1 - \frac{V}{|T|} \ . \tag{29}$$

This result provides an easy decision guideline for the physical design of a table in a data warehouse. The right-hand side of this inequality can never be greater than 1. Thus, we immediately note the following result.

Result 3 *Contrary to popular belief, a BDI does not always perform better than a JDI. In fact, a JDI performs better if $r < w(C)$, that is, if the size of a RID is smaller than the width of the column.*

More precisely, the higher the ratio $\frac{V}{|T|}$ of the column of interest, the smaller the right hand side of the above equation and hence the larger the value of $w(C)$ for which BDI is preferable (for a given r).

Recall that we proposed the JDI only for foreign key columns. An alternative approach using a JDI would consist of a BDI containing each of the V distinct values taken by the column and an associated JDI onto that BDI, to map these V values to records in the table. To illustrate, let us consider the `SALES.ShipMode` column from the example in section 3; the TPC-D benchmark indicates that this can take one of 9 different values. Equation 29 clearly indicates that `ShipMode` would be better represented as a JDI with an associated look-up BDI (e.g., `ShipMode_Type`) than as a simple BDI; indeed, (29) indicates $\frac{r}{w(C)} = 0.6 \leq 1 - \frac{V}{|T|} \approx 1$. In this case, the JDI/BDI combination would require only 4503 blocks, whereas a simple BDI would need 7500 blocks of storage. Hence, it may be more efficient to implement a wide, repeating column as a JDI with an associated look-up BDI than as a simple BDI.

Comparison of Tree-Based Techniques and Ordinal-Position Based Techniques

Since we have shown already that a bitmapped index can never cost more than a B-tree index, we begin by comparing bitmapped indexes with BDIs. By comparing the last term of (26) (the first two terms are usually quite small and can thus be disregarded to simplify the analysis) and (4), it is easily seen that a BDI will perform better if:

$$V_{\text{range}} \geq \frac{8\ w(C)}{f} \ . \tag{30}$$

This simply states that, for a given compression factor, a BDI performs better than a bitmapped index when the selection range of a query is 'large'. For example, suppose f is 0.2 and the indexed column is `CUSTOMER.AcctBal` from the star schema of Figure 1, which has a width $w(C)$ of 8 bytes. This column appears to be a reasonable candidate for indexing since it is likely that account balances would be queried frequently. The above relationship indicates that a BDI will perform better if the number of distinct values in the selection range (i.e., V_{range}) is at least 320. Note that even though we claim that a BDI performs better for 'large' values of V_{range}, 320 is not really that large when compared to the cardinality of the `CUSTOMER` table, 150,000. Larger values of f would imply that a BDI is preferable for even smaller values of V_{range}. Thus we can conclude the following:

Result 4 *A BDI outperforms a bitmapped index if the number of search-key values in the predicate range is large (with respect to the ratio of the width of the column, in bits, to the compression factor).*

Note that BDIs, like bitmapped indexes, can also be compressed, which would improve their performance. However, in this analysis, we compare uncompressed BDIs to compressed bitmapped indexes. In other words, we compare the worst case BDI to the best case bitmapped index.

A similar analysis can be performed to compare the performance of bitmapped indexes and JDIs. From (26) and (6), it is easily seen that a JDI will perform better if:

$$V_{\text{range}} \geq 8 \left(\frac{V\ w(C) + r|T|}{f|T|} \right) . \tag{31}$$

Similar to the results obtained for the BDI, we conclude the following:

Result 5 *A JDI outperforms a bitmapped index if the number of search-key values in the predicate range is relatively large.*

These results conclude our study of the rowset construction performance of the different indexing schemes under study. The results of this analysis can be summarized as follows. Of the indexes that rely on the ordinal position of records, the best performing ones are the JDI or the BDI (or the projection index). The cutoff point between the two occurs when $\frac{r}{w(C)} = 1 - \frac{V}{|T|}$. However, the bitmapped index sometimes performs better than each one of these approaches, but this would only occur when V_{range} is relatively small. We summarize the above results in table 4, where **BI** denotes the bitmapped index.

	Better than BI if	**Better than BDI if**	**Better than JDI if**
BI	–	$V_{\text{range}} < \frac{8\ w(C)}{f}$	$V_{\text{range}} < 8 \left(\frac{V\ w(C)+r\|T\|}{f\|T\|} \right)$
BDI	$V_{\text{range}} \geq \frac{8\ w(C)}{f}$	–	$\frac{r}{w(C)} \geq 1 - \frac{V}{\|T\|}$
JDI	$V_{\text{range}} \geq 8 \left(\frac{V\ w(C)+r\|T\|}{f\|T\|} \right)$	$\frac{r}{w(C)} < 1 - \frac{V}{\|T\|}$	–

Table 4: Comparison of the rowset-construction performance of the different indexes under study

Having examined the rowset construction performance of different access schemes, we now turn our attention to the second part of the cost of performing star joins, namely $\mathcal{N}_{\texttt{JOIN}}$, the cost of performing a join on restricted tables.

6.2 Comparative Analysis of $\mathcal{N}_{\texttt{JOIN}}$

Bitmapped indexes are used in Oracle 8 and BJIs in the Informix Universal Server. Though we would have liked to compare the performance of our algorithms with that of Red Brick's STARjoin approach, we were unable to obtain enough information to model STARjoins with any level of detail. However, as will become clear later on in this section, the high-performance of our algorithms is strongly tied to the fact that DataIndexes do not store table rows contiguously. The Red Brick system, however, relies on conventional storage approaches; hence we believe that our proposed algorithms also outperform the STARjoin approach for the majority of OLAP-type queries. It would, however, be interesting to verify –or refute– this belief.

Also, we note that some of the indexes analyzed in section 6.1.1 (namely projection indexes and bit-sliced indexes) seem to provide little help in computing joins. Our analysis in section 6.1.1 also indicates that bitmapped indexes will never perform worse than B-trees. We thus do not investigate the performance of star joins using projection indexes, bit-sliced indexes or B-trees. Instead, we concentrate on the approaches based on bitmapped indexes, BJIs and DataIndexes.

6.2.1 Bitmapped-Join Indexes

Based on BJIs, a star join algorithm proceeds roughly as follows. The dimensional rowsets (i.e., $R_1, \ldots R_{|\mathfrak{D}|}$) 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, the leaf-level of the corresponding BJI is scanned, and the rowsets associated with rows that appear in rowset R_D are loaded and bitwise `OR`'ed together. When these operations have been applied to all participating dimension tables, the result is the *join rowset* $R_{\mathtt{JOIN}}$ which indicates which records of the fact table appear in the join result. $R_{\mathtt{JOIN}}$ can then be used in one of two ways, depending on the amount of available memory.

In the first approach, all relevant columns and rows from the dimension tables (including the primary key column) are extracted from the dimension tables and pinned in memory. The algorithm then proceeds similarly to the SJL algorithm we proposed: the fact table F is scanned in $R_{\mathtt{JOIN}}$ order, and the result rows are constructed from the in-memory structures, then output.

Hence, we can compute that the overall *best case* cost of performing the join with bitmapped indexes, given that enough memory is available, and assuming that all rowsets are packed (for reasons stated previously), is:

$$\mathcal{N}_{\mathtt{JOIN}}(\mathrm{BJI}) = \mathcal{N}_{R_{\mathtt{JOIN}}} + \mathcal{N}_{\mathrm{Dim}} + \mathcal{N}_{F\text{-scan}} \ , \tag{32}$$

where $\mathcal{N}_{R_{\mathtt{JOIN}}}$represents the cost of forming the join-rowset ($R_{\mathtt{JOIN}}$), $\mathcal{N}_{\mathrm{Dim}}$the cost of loading all dimensional tuples of interest, and $\mathcal{N}_{F\text{-scan}}$the cost of scanning the fact-table itself. We now derive expressions for each of these components. To form $R_{\mathtt{JOIN}}$ requires descending the tree, scanning the leaf level, and then loading the blocks having rowsets that appear in the join. Thus the cost to form $R_{\mathtt{JOIN}}$, can be expressed as follows:

$$\mathcal{N}_{R_{\mathtt{JOIN}}} = \sum_{D \in \mathfrak{D}} \left(\lceil \log_{P_D} V_D - 1 \rceil + \min\left(\varsigma_D |D|, \left\lceil \frac{|D|}{K_D} \right\rceil \right) + f\left(\left\lceil \frac{|F|}{8\ B} \right\rceil \varsigma_D |D| \right) \right) \tag{33}$$

where the first term corresponds to the cost to descend the tree, the second term corresponds to the cost to scan the leaf level, and the third term corresponds to the cost to load the relevant blocks of rowsets. In the first term, P_D represents the order of the tree for dimension table D and V_D represents the number of distinct search key values in D. Both of these terms are as defined in section 6.1.1, with the exception that the width of each column, pointer pair is different. Specifically, a RID is contained in each node rather than a column value. Thus, the expression for P_D is then $K_D + 1$, where K_D represents the number of search keys per node in the index for D and $K_D = \left\lceil \frac{B}{r+\pi} \right\rceil$. In the second term, $|D|$ represents the cardinality of dimension table D and ς_D represents the selectivity on D. In the best case, only $\varsigma_D |D|$ blocks must be accessed at the leaf level, whereas in the worst case, all blocks must be accessed.

The cost to load dimensional tuples, $\mathcal{N}_{\mathrm{Dim}}$, is given by:

$$\mathcal{N}_{\mathrm{Dim}} = \sum_{T \in \mathfrak{D}} \left\lceil \frac{|T| w(T)}{B} \right\rceil \tag{34}$$

where $w(T)$ represents the width of dimension table T. Note that in this case the entire tuple (all columns) must be loaded.

Finally, the cost to scan the fact table, $\mathcal{N}_{F\text{-scan}}$, is given by:

$$\mathcal{N}_{F\text{-scan}} = \min\left(\varsigma_F|F|, \left\lceil \frac{|F|w(F)}{B} \right\rceil\right) \tag{35}$$

where $w(F)$ represents the width of the fact table. This cost depends on the selectivity of F. In the best case, only $\varsigma_F|F|$ blocks must be accessed, whereas in the worst case, all blocks must be accessed.

The total memory requirements for this first approach can be expressed as:

$$\mathcal{M}_{\mathtt{JOIN}}(\mathrm{BJI}) = 1 + |\mathfrak{D}| + \sum_{D \in \mathfrak{D}} \left\lceil \frac{|D|w(D)}{B} \right\rceil \tag{36}$$

where the first term 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.

The second approach for evaluating star-joins with BJIs is based on the pairwise, hash-join technique. It should be applied when there is not enough memory to load all necessary dimensional columns into memory. Once $R_{\mathtt{JOIN}}$ has been determined, the relevant values from the different tables are extracted from the source tables and stored in temporary files. These temporary files are then joined pair-wise until the final join result is computed, thereby requiring $|\mathfrak{D}|$ individual two-way joins. Since it is well known [46] that pairwise joins do not perform well in a data warehouse environment, we do not include the cost analysis of this approach.

6.2.2 Bitmapped Indexes

Bitmapped indexes may also be used to perform a star-join similarly to BJIs. However, since bitmapped indexes are single-table structures, more operations are required during a join. Specifically, while a BJI contains RIDS and thus allows direct access to a particular dimension table, a bitmapped index contains values and therefore requires accessing the primary key values from the participating dimensional table tuples. These tuples are then used to scan the bitmapped indexes on the corresponding fact table columns, resulting in additional accesses in creating the join rowset $R_{\mathtt{JOIN}}$. These additional accesses result in approximately the same number of block accesses as a pairwise join between the fact table and each of the dimension tables. In addition, the size of the tree structure of each index might be slightly different from those used in BJIs. For instance, the values in bitmapped indexes may vary in size, while the RIDS in BJIs are typically rather small and constant in size. Based on these differences, the following result can easily be shown:

Result 6 *Bitmapped join indexes outperform bitmapped indexes for evaluating star joins.*

6.2.3 Cost comparison of Bitmapped-Join and DataIndexes

We now compare the performance of DataIndexes and bitmapped join indexes under packed conditions. To do so, we compare the worst-case performance expressions for SJL to the best case expressions for bitmapped join indexes. By comparing (and making the usual simplifications) to (7) and (32), it can be shown that SJL can outperform BJI and other approaches if the following condition holds true:

$$\frac{\sum_{D \in \mathfrak{D}} \varsigma_D |D|}{|\mathfrak{D}|} \geq \frac{8r}{f} \ . \tag{37}$$

To derive this condition, we have assumed that all columns from the fact table appear in the output (i.e., $\sum_{C \in \mathfrak{C}_F} w(C) = w(F)$) and that all dimension table columns for participating dimension tables also appear in the output (i.e., $\sum_{C \in \mathfrak{C}_D} w(C) = w(D)$). These assumptions are strongly biased towards traditional approaches. Clearly, the fewer the number of output columns, the better dataindexes will perform, as only the relevant columns will need to fetched, unlike traditional approaches where all columns will be fetched, regardless of the desired output. Thus, by making the assumption that all fact table columns are needed for output, we nullify a strong advantage of dataindexes. Even then, SJL outperforms the other approaches in a number of cases. Indeed the above condition can be understood as follows:

Result 7 *SJL outperforms other star-join approaches if the selectivity on the dimension tables is relatively large (with respect to the ratio of the RID size, in bits, to the compression factor).*

In other words, SJL outperforms other star-join approaches if the average number of tuples accessed from each dimension table is large. Referring again to the star schema of Figure 1, if we assume a RID size r of 6 bytes and a compression factor f of 0.2, then at least 240 tuples must be selected, on average, from a dimension table for SJL to outperform the other approaches. For less compressed representations (i.e., larger values of f), SJL can outperform other approaches for even smaller dimension table selectivities.

7 Overall Cost Comparison of Star Join Performance

In this section, we perform a comparative analysis of the star-join query costs ($\mathcal{N}_{\text{star}}$) associated with the different index structures under study. We showed in the last two sections that a DataIndex-based approach and the bitmapped index/BJI approach are both among the most efficient known approaches for evaluating star-joins. In this section, thus, we only consider these approaches and will refer to them as the SJL/DataIndex (SJL), SJS/DataIndex (SJS), and the bitmapped index/BJI (BJI) approaches. Also, to simplify the analysis, *we only consider the best performance obtainable with the BJI approach with the worst-case performance of DataIndexes*, which should be sufficient to demonstrate the superiority of DataIndexes.

The cost comparisons were generated based on the expressions we have presented in this paper for the worst- or best-case performance achievable with the different algorithms under study. The query utilized to perform the analysis is the query presented in Section 4 and joins the `TIME`, `CUSTOMER`, `SUPPLIER` and `SALES` tables of our sample star-schema. The query is repeated below for convenience.

```
SELECT U.Name, 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
```

The corresponding selection predicates occur on the `TIME.Year`, `CUSTOMER.Nation` and `SUPPLIER.Nation` columns, and the columns displayed in the result are `SUPPLIER.Name` and `SALES.ExtPrice`. This query is thus similar to the "Volume Shipping Query" in [62], which identifies sales volumes between different nations. Such a query is relatively typical of OLAP environments.

Finally, we continue to use the same metric as used previously, the number of blocks accessed to evaluate the query, $\mathcal{N}_{\text{star}}$. Using this metric, we first examine a baseline case where only the overall size of the warehouse is varied. We then analyze the corresponding performance sensitivities with respect to query selectivity, compression levels, and, for the SJS approach, available memory. Table 5 lists the parameter values used in the baseline analysis.

Parameter	Description	Value
$\lvert\mathfrak{D}\rvert$	Number of dimensions tables involved in join	3
$\lvert\mathfrak{C}_F\rvert$	Number of fact-table columns that contribute to the join result	1
B	Effective Size, in bytes, of a data block	8,000
π	Size, in bytes, of a pointer to a data block	4
r	Size, in bytes, of a RID	6
$\lvert\texttt{SALES}\rvert$	Number of records in `SALES` fact table	$6,000,000\times$ scale factor
ς_F	Selectivity factor on fact table	0.01
$\lvert\texttt{TIME}\rvert$	Number of records in `TIME` dimension table	$2,557$
$\lvert\texttt{CUSTOMER}\rvert$	Number of records in `CUSTOMER` dimension table	$150,000\times$ scale factor
$\lvert\texttt{SUPPLIER}\rvert$	Number of records in `SUPPLIER` dimension table	$10,000\times$ scale factor
ς_D	Selectivity factor on dimension table D	0.05
c	Distinctness factor of range selection	0.2
V_{range}	Number of distinct search-key values referenced by a particular range selection	$\varsigma_T\lvert T\rvert c$
$w(\texttt{T.Year})$	Column width of `TIME.Year`, in bytes	4
$w(\texttt{C.Nation})$	Column width of `CUSTOMER.Nation`, in bytes	25
$w(\texttt{U.Nation})$	Column width of `SUPPLIER.Nation`, in bytes	25
$w(\texttt{U.Name})$	Column width of `SUPPLIER.Name`, in bytes	25
$w(\texttt{SALES})$	Table width, in bytes	131
$w(\texttt{TIME})$	Table width, in bytes	28
$w(\texttt{CUSTOMER})$	Table width, in bytes	269
$w(\texttt{SUPPLIER})$	Table width, in bytes	243
f	Compression factor	0.2
M_a	Number of blocks allocated to input BDI	8,000

Table 5: Parameters used in the Baseline Analysis

7.1 Baseline Case

For the baseline experiment, we assume the following parameter values:

- Selectivity on the fact table, ς_F, is 0.01 (i.e., 1% of fact table rows appear in the join result).
- Selectivity on each dimension table, ς_D, is 0.05 (i.e., 5% of the rows of each dimension table appear in the join result).
- Selectivity on each range predicate, V_{range}, is computed as $\varsigma_T|T|c$, where $\varsigma_T|T|$ represents the number of rows appearing in the join result for table T and c is the *distinctness factor* of the range selection, which we assume to be 0.2. For instance, consider the base case for the `SUPPLIER.Nation` selection predicate. If $\varsigma_D = 0.05$ and $|\texttt{SUPPLIER}| = 10,000$, then 500 rows from this table appear in the join result. Multiplying this number by the distinctness factor of 0.2 results in a V_{range}value of 100. Thus there are 100 distinct values in this range selection.

Holding the above values constant, we then vary the size of the database by varying the *scale factor* from 0.1 to 1000. This results in overall database sizes ranging from about 86 MB to about 860 GB. As we will soon show, the expressions derived for the memory requirements for BJI in (36), for SJL in (11), and for SJS in (18), indicate that BJI requires more memory than either SJL or SJS. Thus we assume in this analysis that, for a given database size, the system is equipped with sufficient memory to perform a star-join using the BJI approach. For instance, from (36), we know that a star-join using BJI for an 860 GB database requires approximately 2.5 GB of memory. We then assume that for an 860 GB database, there exists 2.5 GB of main memory. For larger databases (e.g., scale factor greater than 1000), this assumption may not be valid and so different techniques must be used, such as SJS for DataIndexes and hash-join for BJI. However, we do not consider such cases and instead focus our analysis on databases that are 860 GB or less in size. We include the performance of SJS in this range for comparison purposes. In the baseline case, we assume that 64 MB of memory is available to be allocated to the input BDI (i.e., $M_a = 8,000$ blocks).

The resulting plots for the baseline case are presented in Fig. 5 (Note that Fig. 5 as well as the sensitivity plots are displayed using a log scale for both axes). All three approaches exhibit a similar pattern - the cost or number of required block accesses increases as the size of the database increases. However, it is quite clear from Figure 5 that both SJL and SJS outperform the BJI approach over the entire range. In addition, the cost of the BJI approach increases much more quickly than does the cost of either SJL or SJS. This is primarily due to the fact that BDIs are maintained separately with DataIndexes and so only columns of interest need to be brought from disk. For a relatively small database, e.g., 86 MB or scale factor 0.1, SJL requires only $2,007$ block accesses, SJS requires $3,403$ accesses, and BJI requires $7,810$ such accesses. For larger databases, this difference is at least an order of magnitude. For instance, an 86 GB database (scale factor 100) requires approximately 2 million accesses for SJL and approximately 3 million for SJS, compared to 1.5 billion for BJI. The weak performance of BJI, especially with large database sizes, is largely due to the last term in (33), the expression for the cost to form the join-rowset $R_{\texttt{JOIN}}$. From this expression, we can see that BJI performs in $O(|F| \times |D|)$.

Overall, the base case curves clearly show that DataIndexes outperform the BJI approach. As we shall see, this pattern is repeated throughout the rest of our experiments.

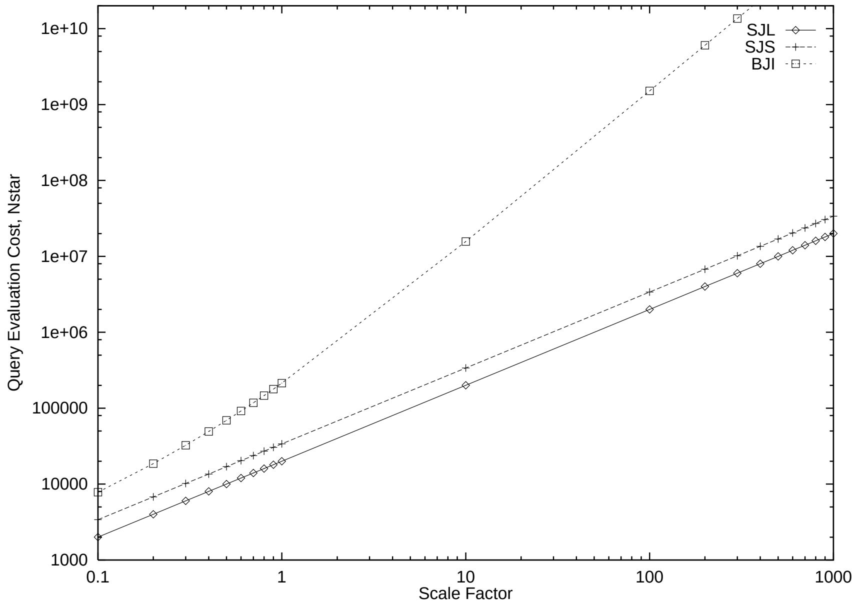

Figure 5: Base Case Performance ($\varsigma_F = 1\%$, $\varsigma_D = 5\%$ and $f = 20\%$)

7.2 Sensitivity to Query Selectivity

Query or fact table selectivity, ς_F, is an important factor in peforming a star-join for all three approaches. For SJL, ς_F impacts the cost of scanning the fact table to create the final query output, as shown in (10). For SJS, ς_F impacts the cost to restrict the JDIs (14), the cost to create the output BDIs (16), and the cost to create the final output (17). For BJI, ς_F also impacts the cost to construct the final output, as shown in (35). Based on these expressions, we would expect a decrease in ς_F (i.e., higher selectivity) to improve the performance of all approaches. In order to study this sensitivity of the different approaches to query selectivity, we repeated the baseline experiments using several fact table selectivities ranging from 0.0003 to 0.1. From this analysis, it became clear that all approaches are more sensitive to relatively small values of ς_F (e.g., less than 0.0005) and that BJI is most sensitive to changes in ς_F. We therefore chose to include results for the following values of ς_F: 0.03%, 0.1%, and 2%.

The resulting plots are shown in Fig. 6.

As expected, the overall shape of the curves in Fig. 6 remains the same as in the base case, and all approaches do in fact benefit from higher selectivity. Overall, SJL still outperforms BJI over the entire range, as shown by the two lower-most curves in Fig. 6, which represent the cost of SJL for 1% selectivity (i.e., the baseline case) and 0.03% selectivity. These are the only two curves displayed for SJL because the performance of SJL is largely insensitive to changes in ς_F, except for very small values of ς_F. BJI, on the other hand, exhibits a significant improvement from lower values of ς_F, but these improvements only occur for small to medium sized databases. In fact, for very small values of ς_F and small database size, the performance of BJI approaches that of SJL. However, SJL still performs better. For instance, an 86 MB database (scale factor 0.1) and selectivity of 0.03% requires $1,587$ block accesses for SJL and $1,990$ accesses for BJI. However, the same selectivity with an 86 GB database (scale factor 100) requires about 1.5 million block accesses for SJL and about 1.5 billion accesses for BJI.

We now examine the curves for the SJS approach (the two curves just above the SJL curves). Like SJL, SJS is also rather insensitive to changes in ς_F. It is interesting to note, however, that for very small values of ς_F and

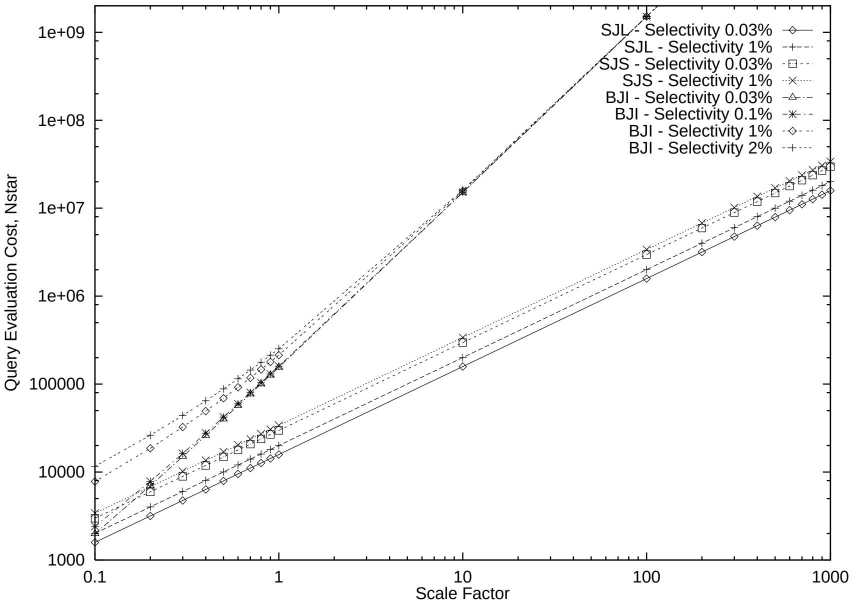

Figure 6: Sensitivity to Query Selectivity

small database size, BJI outperforms SJS. For example, an 86 MB database (scale factor 0.1) and selectivity of 0.03% requires only 1,990 accesses for BJI compared to 2,983 block accesses for SJS. As the database size increases, however, the performance of SJS surpasses that of BJI. For instance, an 86 GB database (scale factor 100) and selectivity of 0.03% requires about 2.9 million accesses for SJS compared to about 1.5 billion accesses for BJI.

From this analysis, it appears that BJI is somewhat more sensitive to changes in ς_F than either SJL or SJS. This phenomenon can be explained by examining the expressions for the cost to scan the fact table to create the final query result for SJL in (10) and for BJI in (35). Both expressions are similar in that they take the minimum of either the number of fact table rows appearing in the join result (i.e., $\varsigma_F|F|$) or the number of blocks required for all relevant fact table columns. The difference is that for BJI, the number of blocks required for fact table columns will be much greater than for SJL, since BJI requires that the entire fact table tuple be loaded for each tuple appearing in the join result. Thus the term representing the number of fact table rows will usually be the minimum term for BJI and so selectivity will typically have a greater impact.

7.3 Sensitivity to Compression Factor

Another important factor in performing a star-join with the approaches in this study is compression. As mentioned previously, some degree of compression is typically used in data warehouses, so it is worth examining the impact of such compression. Recall that we define the compression factor f to be a percentage representing the degree of compression, where $f = 1$ indicates no compression. Also recall that we assume no compression of BDIs, therefore, this analysis affects only the BJI approach. The level of compression appears to be a significant factor in the BJI approach, as f appears in both the $\mathcal{N}_{\texttt{ROWSET}}$ and $\mathcal{N}_{\texttt{JOIN}}$ expressions in (26) and (33), respectively. By examining these expressions, we would expect an increase in the amount of compression achieved (i.e., a decrease in f) to improve the performance of BJI. In order to study this sensitivity of BJI to varying degrees of compression, we repeated the baseline experiments with compression factors of 1% and 3%. The results are displayed in Fig. 7.

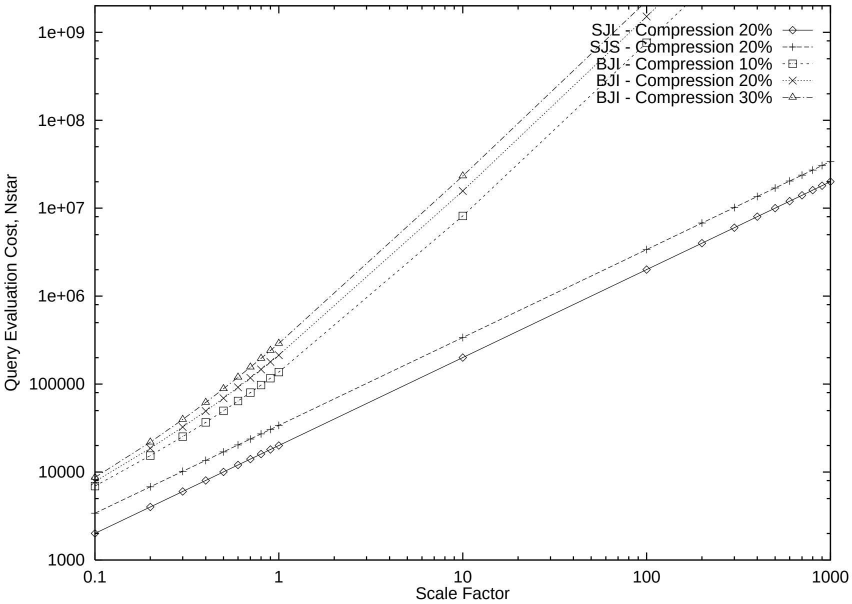

Figure 7: Sensitivity to Compression Factor

Here again the overall shape of the curves in Fig. 7 remains the same as in the base case. As expected, the performance of BJI does in fact improve when more compression is used. For instance, for a database size of 86 GB (scale factor 100) in the baseline case (20% as shown in Fig. 7), 1.5 billion block accesses are required for BJI. When more compression is applied, or f is reduced to 0.1, the number of accesses decreases to approximately 759 million. For lower compression levels (i.e., higher f), as expected, the number of block accesses increases for BJI (from 1.5 billion to 2.2 billion).

7.4 Memory Requirements

Finally, we compare the memory requirements for SJL and BJI. As shown in Figure 8, SJL requires significantly less memory than BJI on average. For instance, a database of 86 GB (i.e., scale factor 100) requires only 31 MB of memory for SJL, yet requires 243 MB for BJI. As the database size increases, the memory requirements for BJI increase much more quickly than for SJL. For example, a 344 GB database (i.e., scale factor 400) requires 125 MB of memory for SJL, yet requires 972 MB for BJI. This result is largely due to the fact that SJL loads only the columns that are relevant for the join, whereas BJI loads the entire dimension table for such columns. For larger databases, the SJS approach would be used, which requires only a small amount of memory, 64 KB for our example query. For smaller databases, SJL still requires less memory than BJI. For instance, a database of 86 MB (i.e., scale factor 0.1) requires 0.14 MB of memory for SJL and 0.28 MB for BJI.

8 Conclusions

In this paper, we proposed a new type of storage and indexing structure specifically geared towards data warehouses: the *DataIndex.* One of the main advantages of this structure is that it essentially provides free indexing. This is achieved by vertically partitioning the tables that form the relational schema and maintaining the ordinal mapping among the columns. Many of these partitions can be stored as is, in *basic DataIndexes* (BDIs). Some others - especially foreign-key columns - should instead be represented as lists of RIDs of foreign

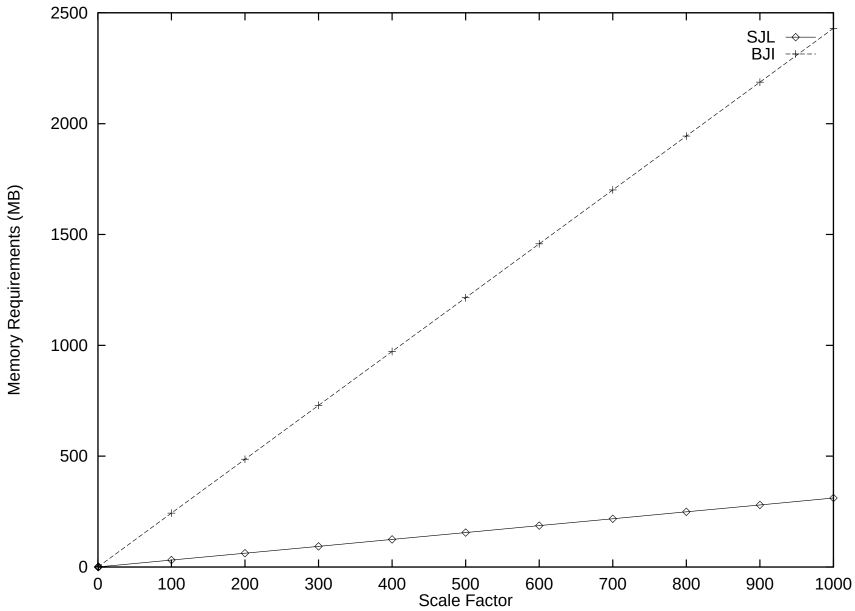

Figure 8: Memory Requirements for SJL and BJI

column records, or *join DataIndexes* (JDIs). We also proposed two fast algorithms for performing star-join operations on DataIndexed data warehouses.

We have derived expressions that show the expected performance of DataIndexes and a number of other indexing approaches that have been proposed and utilized in data warehousing[45, 46]. Based on these expressions, we have analytically shown that the performance obtainable with DataIndexes is most often superior to that of even the best conventional approach. Specifically, we have analyzed the performance of DataIndexes with respect to *range selections* and *star joins*, two of the most common operations in OLAP. The corresponding results are summarized in table 6.

Structure	**Best for Range Selections When ...**
Bitmapped Indexes	Small Ranges
BDI	Medium to Large Ranges and Narrow Columns
JDI	Medium to Large Ranges and Wide Columns

Structure	**Best for Star-Joins When ...**
Bitmapped Indexes & Bitmapped Join Indexes	Very High Selectivities
DataIndexes	Otherwise

Table 6: Summary of Results

Evidently the main contributions of this paper are the development of the DataIndexes and fast star-join algorithms based on them, and the extensive performance study that shows the clear superiority of the proposed methods. There are, however, a number of other advantages of this approach.

Bulk Loading: Data warehouses are typically "refreshed" periodically from production systems. During these refreshes, a large amount of new data is appended to the existing base. Our intuition is that DataIndexes are particularly amenable to these bulk append operations, due to the fact that they can be done by simple additions of new data pages to existing BDIs instead of repeated insertions of records into sophisticated index structures such as B-trees. Bulk-append operations on JDIs would occur in much the same way. However, since JDIs utilize RIDs on foreign tables, these foreign tables should be updated prior to the JDI. This imposes a partial ordering on the sequence of operations to follow in refreshing a data warehouse. Also, it adds the additional constraint that the foreign table (or at least the column(s) composing its primary key) be available to the loading program during the creation of the JDI (to map from the actual column values to RIDs). These two problems, however, are likely to be negligible when compared with the overhead incurred by expensive index update operations.

Compressibility: It should be clear that DataIndexes can be compressed much more readily than conventional tables, since the range of values each DataIndex covers is much smaller [19]. Some compression techniques allow operations on the underlying data to be performed without decompression. Thus, we can expect that compressed DataIndexes can provide even better performance than those studied in this paper.

Other OLAP queries: The results in [46], lead us to believe that DataIndexes would yield relatively low evaluation costs for other types of warehousing queries (such as group-bys and aggregations). Of course, analysis and experimentation similar to those conducted for rowset construction and star join queries must also be done for these other types of queries to verify this belief.

Buffer Utilization: Queries often access the same column a number of times (e.g., when this column appears both in the `SELECT` and the `WHERE` clause). When these columns are small (which will often be true when compression is used), or the size of main memory is large, query cost can be significantly reduced by keeping one or more of these columns in memory for the duration of the query evaluation. For instance, if the `SALES.ShipDate` attribute of our earlier example appears in both the `SELECT` and `WHERE` clauses of an SQL query, then one can save 1500 block accesses if the BDI for that attribute is kept in memory after being loaded. Similarly, since the overall size of the database is smaller with DataIndexes than with conventional structures, it is likely that the number of memory faults will be smaller with DataIndexes.

References

[1] B. Adelberg, H. Garcia-Molina, and J. Widom. The STRIP rule system for efficiently maintaining derived data. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 147–158, Tucson, AZ, May 13-15 1997.

[2] S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. 22nd VLDB Conf.*, Mumbai, India, 1996.

[3] R. Armstrong. Data warehousing: Dealing with the growing pains. In *Proc. Thirteenth Intl. Conf. on Data Engineering*, pages 199–205, Birmingham, UK, April 7-11 1997. IEEE.

[4] M.W. Blasgen and K.P. Eswaran. On the evaluation of queries in a database system. Technical Report RJ-1745, IBM Corp., San Jose, CA, April 1976.

[5] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proc. 10th VLDB*, Singapore, August 1984.

[6] M.J. Carey and D. Kossman. On saying "enough already" in SQL. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 219–230, Tucson, AZ, May 13-15 1997.

[7] C.Y. Chan and Y. Ioannidis. Bitmap index design and evaluation. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 355–366, Seattle, WA, June 1-4 1998.

[8] D. Chatziantoniou and K.A. Ross. Querying multiple features of groups in relational databases. In *Proc. 22nd VLDB Conf.*, Mumbai, India, 1996.

[9] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proc. 20th Intl. Conf. on Very Large Databases*, pages 131–139, Santiago, Chile, September 1994.

[10] S. Chaudhuri and K. Shim. Optimizing queries with aggregate views. In P. Apers, M. Bouzeghoub, and G. Gardaring, editors, *Advances in Database Technology – EDBT'96 5th Intl. Conf. on Extending Database Technology*, volume 1057 of *Lecture Notes in Computer Science*, pages 167–182. Springer-Verlag, New York, 1996.

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

[12] J-H. Chu and G. Knott. An analysis of B-trees and their variants. *Information Systems*, 14(5), 1989.

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

[14] L. Colby, T. Griffin, L. Libkin, I. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Montreal, Quebec, Canada, June 1996.

[15] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–138, June 1979.

[16] G.P. Copeland and S. Khoshafian. A decomposition storage model. In *Proc. ACM SIGMOD*, pages 268–279, 1985.

[17] C. Dyreson. Information retrieval from an incomplete data cube. In *Proc. 22nd VLDB Conf.*, Mumbai, India, 1996.

[18] M. Freeston. A general solution to the n-dimensional B-tree problem. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, San Jose, CA, 1995.

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

[20] P. Goel and B. Iyer. Sql query optimization: Reordering for a general class of queries. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 47–56, Montreal, Quebec, Canada, June 4-6 1996.

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

[22] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, San Jose, CA, May 23-25 1995.

[23] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proc. 21st VLDB Conf.*, Zurich, Switzerland, 1995.

[24] A. Gupta, H. Jagadish, and I. Mumick. Data integration using self-maintainable views. In *Proc. Fifth Intl. Conf. on Extending Database Technology*, Avignon, France, March 1996.

[25] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 26–28, Washington, DC, May 26-28 1993.

[26] H. Gupta, V. Harinarayan, A. Rajaraman, and J.D. Ullman. Index selection for OLAP. In *Proc. Thirteenth Intl. Conf. on Data Engineering*, pages 208–219, Birmingham, UK, April 7-11 1997. IEEE.

[27] A. Guttman. R-trees: a dynamic index structure for spatial searching. In M. Stonebraker, editor, *Readings in Database Systems*, pages 599–609. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1988.

[28] 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.

[29] J.M. Hellerstein, P.J. Haas, and H.J. Wang. Online aggregation. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 171–182, Tucson, AZ, May 13-15 1997.

[30] J.M. Hellerstein and J.F. Naughton. Query execution techniques for caching expensive methods. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 423–424, Montreal, Quebec, Canada, June 1996.

[31] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 2nd edition, 1996.

[32] C-T. Ho, R. Agrawal, N. Meggido, and R. Srikant. Range queries in OLAP data cubes. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 73–88, Tucson, AZ, May 13-15 1997.

[33] C-T. Ho, J. Bruck, and R. Agrawal. Partial-sum queries in OLAP data cubes using covering codes. In *Proc. 16th ACM Symposium on Principles of Database Systems*, Tucson, AZ, May 1997.

[34] Informix Software. INFORMIX-online extended parallel server and INFORMIX-universal server: A new generation of decision-support indexing for enterprise data warehouses. White Paper, 1997.

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

[36] S. Khoshafian, G.P. Copeland, T. Jagodis, H. Boral, and P. Valduriez. A query processing strategy for the decomposed storage model. In *Proc. ICDE*, pages 636–643, 1987.

[37] R. Kimball. *The Data Warehouse Toolkit*. J. Wiley & Sons, Inc., first edition, 1996.

[38] M. Kitsuregawa, M. Nakayama, and M. Takagi. The effect of bucket size tuning in the dynamic hybrid GRACE hash join method. In *Proc. 15th VLDB*, Amsterdam, August 1989.

[39] Y. Kotidis and N. Roussopoulos. An alternative storage organization for rolap aggregate views based on cubetrees. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 249–258, Seattle, WA, June 1-4 1998.

[40] D. Lomet, ed. Special issue on materialized views and data warehousing. *IEEE Data Engineering Bulletin*, 18(2), 1995.

[41] I.S. Mumick and A. Gupta, editors. *Proc. Workshop on Materialized Views: Techniques and Applications*, Montreal, Canada, June 7 1996.

[42] I.S. Mumick, D. Quass, and B.S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 100–111, Tucson, AZ, May 13-15 1997.

[43] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. In M. Stonebraker, editor, *Readings in Database Systems*, pages 582–598. Morgan-Kaufmann Publishers, Inc., San Mateo, CA, 1988.

[44] P. O'Neil. Model 204 architecture and performance. In *2nd Intl. Workshop on High Performance Transaction Systems (HPTS)*, volume 359 of *Springer-Verlag Lecture Notes on Computer Science*, pages 40–59. Springer-Verlag, Asilomar, CA, 1987.

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

[46] 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.

[47] Oracle Corp. Star queries in Oracle8. White Paper, June 1997.

[48] D. Quass. Maintenance expressions for views with aggregations. In *Proc. Workshop on Materialized Views: Techniques and Applications*, Montreal, Canada, June 7 1996.

[49] D. Quass and J. Widom. On-line warehouse view maintenance. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 393–404, Tucson, AZ, May 13-15 1997.

[50] S.G. Rao, A. Badia, and D. Van Gucht. Providing better support for a class of decision support queries. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 217–227, Montreal, Quebec, Canada, June 4-6 1996.

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

[52] J.T. Robinson. The K-D-B-tree: A search structure for large multi-dimensional dynamic indexes. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 10–18, New York, NY, 1981.

[53] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and bulk updates on the data cube. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 89–99, Tucson, AZ, May 13-15 1997.

[54] B. Salzberg. Access methods. *ACM Computing Surveys*, 28(1):117–120, March 1996.

[55] S. Sarawagi. Indexing OLAP data. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, pages 36–43, 1997.

[56] L.D. Shapiro. Join processing in database systems with large main memories. *ACM TODS*, 11(3), October 1986.

[57] A. Shoshani. OLAP and statistical databases: Similarities and differences. *ACM TODS*, 22, 1997.

[58] A. Shukla, P.M. Deshpande, J.F. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. In *Proc. 22nd VLDB Conf.*, Mumbai, India, 1996.

[59] D. Simpson. Corral your storage management costs. *Datamation*, pages 88–93, April 1997.

[60] M. Spiliopoulou, M. Hatzopoulos, and Y. Cotronis. Parallel optimization of large join queries with set operators and aggregates in a parallel environment supporting pipeline. *IEEE TKDE*, 8(3):429–45, June 1996.

[61] Sybase, Inc. Sybase IQ – optimizing interactive performance for the data warehouse. White Paper, 1997.

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

[63] P. Valduriez, S. Khoshafian, and G.P. Copeland. Implementation techniques of complex objects. In *Proc. VLDB*, pages 101–110, 1986.

[64] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 35–46, Montreal, Quebec, Canada, June 4-6 1996.

[65] I.R. Viguier, A. Datta, and K. Ramamritham. Exact performance expressions for olap queries. Technical Report GOOD-TR-9709, U. of Arizona, 1997. Available from `http://loochi.bpa.arizona.edu`.

[66] S.V. Vrbsky and J.W.S. Liu. APPROXIMATE – a query processor that produces monotonically improving approximate answers. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):1056–1068, 1993.

[67] M-C. Wu and A. Buchmann. Encoded bitmap indexing for data warehouses. In *Proc. 14th ICDE*, pages 220–230, Orlando, Florida, February 1998.

[68] W.P. Yan and P.A. Larson. Eager aggregation and lazy aggregation. In *Proc. 21st Intl. Conf. on Very Large Databases*, pages 345–357, Zurich, Switzerland, September 1995.

[69] Y. Zhao, P.M. Deshpande, and J.F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 159–170, Tucson, AZ, May 13-15 1997.

[70] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 316–327, San Jose, CA, May 23-25 1995.