Curio: A Novel Solution for Efficient Storage and Indexing in Data Warehouses

Anindya Datta*, Krithi Ramamritham†, Helen Thomas*

Abstract

Efficient query processing is a critical requirement for data warehousing systems as decision support applications often require interactive response times to answer complex, ad-hoc queries (e.g., aggregations, multi-way joins) over vast repositories of data (e.g., hundreds of gigabytes to terabytes in size). The most common approach used to improve On-Line Analytical Processing (OLAP) query performance is to utilize indexes or access structures to quickly access the base data. A major drawback to this approach is that it often incurs significant overhead as the access structures must be stored in addition to the base data. In this paper, we present Curio, a data repository and OLAP query server, which provides drastically improved performance for ad-hoc queries, while simultaneously reducing the storage costs associated with warehousing. Curio, a data storage and access technology for data warehouses recently developed by Muninn Technologies, LLC, is based on a novel paradigm that allows fast access to data, yet *does not require indexes*. The basic idea underlying the Curio technology is related to the notion of *positional indexing*. After briefly describing positional indexing we provide an overview of the techniques incorporated in Curio. We then compare the performance of Curio with that of several leading commercial relational warehousing products on a set of typical OLAP queries. Our results indicate that Curio significantly and consistently outperforms these products in terms of query response times, storage requirements, and loading times.

*Muninn Technologies, LLC and Georgia Institute of Technology, Atlanta, GA 30332
†University of Massachusetts, Amherst, MA 01003

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" [5]. Thus, loosely speaking, a data warehouse is a "very large" repository of historical data pertaining to an organization [18]. OLAP refers to the technique of performing complex analysis over the information stored in a data warehouse [7]. It is often used by management analysts and decision makers in a variety of functional areas such as sales and marketing planning. Typically, OLAP queries look for specific trends and anomalies in the base information by aggregating, ranging, filtering and grouping data in many different ways [22]. Efficient query processing is a critical requirement for OLAP because the underlying data warehouse is very large, queries are often quite complex, and decision support applications typically require interactive response times.

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 large amounts 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 [5]. 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 [7]. 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.

Two main approaches for fast OLAP query processing have emerged:

1. *Precomputation Strategies.* This approach relies on *summary tables*, derived tables that house precomputed or "ready-made" answers to queries [5]. This has been, by far, the most explored area in the context of data warehouses [1, 12, 14]. 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. This approach is commonly referred to as the multi-dimensional OLAP (MOLAP) or multi-dimensional database (MDDB) approach. There are currently numerous commercial products available based on the MDDB approach, including Hyperion's Essbase [15], Oracle's Express [23], and MicroStrategy's DSS Agent [20], to name a few. It is recognized however, that such pre-aggregation only works up to a point [22], and precomputing a data cube is often infeasible, as the number of tables to form the cube grows exponentially with the number of dimensions [14]. Moreover, a considerable fraction of the workload in OLAP applications will consist of *ad-hoc* queries which will need to be computed on demand [2]. 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 B^+-tree [6, 8] and multidimensional trees such as the R-tree [13]. For OLAP queries, the index structures that appear to provide the best performance so far are *B^+-tree indexes*, *bitmapped indexes* [21], *bit-sliced indexes* [22] and *projection indexes* [22]. A thorough analysis and discussion of these index structures is presented in [22]. In addition, a number of these indexes are supported in commercial products such as Sybase IQ [29], Oracle 8 [24], and Informix Universal Server [17].

However, in these and other index structures proposed for OLAP, a separate set of indices or access structures is typically maintained in addition to the base data. That is, thus far, databases have mostly 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 [28]. 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.

In this paper we present *Curio*, a data repository and OLAP query server, as a solution to the aforementioned problems with existing technologies. Curio, recently developed by Muninn Technologies, LLC, is based on a novel design technique that allows fast access to data, yet *does not require indexes*. Hence, Curio provides drastically improved performance for ad-hoc queries, while *simultaneously* reducing the storage costs associated with warehousing. The basic idea behind the Curio solution is related to other well-known database techniques such as *vertical partitioning* and *transposed files* [27]. Our objectives in this paper are to present an overview of some of the techniques underlying Curio and demonstrate their potential through a comprehensive performance analysis. We do so by comparing the performance of Curio with that of several leading relational OLAP (ROLAP) products over a range of dimensions including query response times, database loading times, and storage requirements. Our results indicate that Curio significantly and consistently outperforms the indexing structures and algorithms used in these products on all these dimensions. Finally, we highlight some of the key properties and advantages of Curio. Another auxiliary, though noteworthy contribution of this paper is a comparative performance analysis of several leading ROLAP products against one another along the above-mentioned dimensions.

This paper is organized as follows. In Section 2, we motivate the need for an efficient warehousing solution through an example, and in Section 3, we describe positional indexing and its application to Curio. In Section 4, we present the results of a performance evaluation study, and in Section 5, we discuss additional properties and advantages of Curio. We conclude in Section 6.

2 Data and Indexes: A Motivating Example

In this section, we motivate the need for efficient storage and retrieval in data warehousing environments. Before doing so, we first provide a brief description of the *star schema* [19], a key design technique used in relational OLAP.

Conventional relational database design dictates highly normalized schemas. This allows a large number of transactions to access and update the data. However, querying large portions of the database becomes very costly, since potentially large tables must be joined. Since data warehouses face "read-mostly" environments and very large repository sizes, normalization is not desirable for these systems. In addition, analysts often prefer a multidimensional view of data [7], i.e., one where individual data items are seen as points in a multidimensional data space.

The star schema is a relational design paradigm that seeks to encapsulate multidimensionality and denormalization, and is now almost universally acknowledged as the preferred modeling strategy for data warehouses and OLAP. In a star schema, there are two sets of tables: *dimension* tables and *fact* tables. To illustrate this concept, consider the simple warehouse star schema presented in figure 1. It

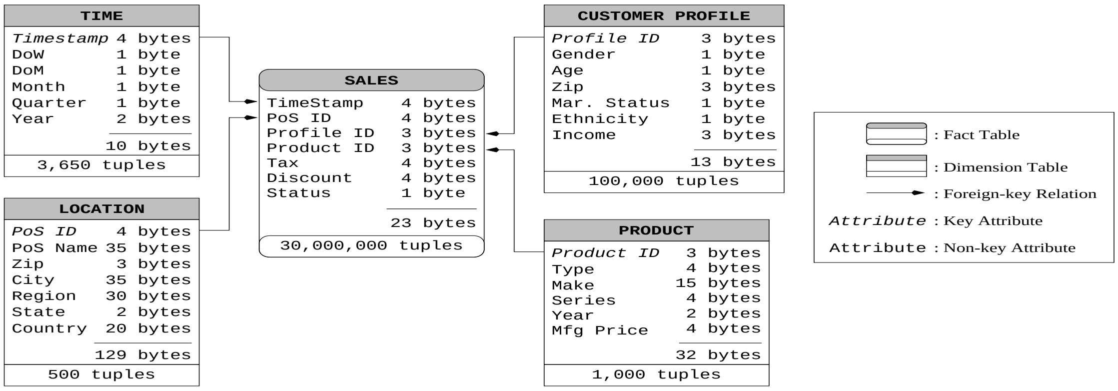

Figure 1: A Simple Warehouse Star Schema

models the 10-year history of the sales of an automobile manufacturer with dealerships located around the globe and is intended to support marketing and strategic decision-making.

There is a central *fact* table, `SALES`, and the dimensions of the data are captured through the `TIME`, `LOCATION`, `CUSTOMER PROFILE` and `PRODUCT` tables. Note that the fact table is nothing but a set of *business metrics* (in this example, the attributes tax, discount and status) associated with the primary keys (PK) of the dimension tables, namely Timestamp (The PK of the `TIME` table), PoS ID (the PK of the `LOCATION` table), Profile ID (the PK of the `CUSTOMER PROFILE` table) and Product ID (the PK of the `PRODUCT` table). We will use this example throughout the remainder of this paper.

Database systems, thus far, have mostly considered index and data separately, i.e., in conventional database design, one envisions a set of relations or table structures, and a separate set of indices or access structures.

In a relational system, tables would be stored "as-is": each table would be represented as a series of records partitioned over a chain of data blocks linked by *block pointers.* In this paper we make the assumption that entire records are stored contiguously in each data block. Given a particular table, T, whose record length is R_T and cardinality (i.e., the number of records in T) is $|T|$, it is easy to compute the number of records that fit on a data block as $\lfloor (B-\pi)/R_T \rfloor$ where B is the block size and π is the size of the page pointer. In turn, we determine the size of that table as $B \times \left\lceil \frac{|T|}{\lfloor (B-\pi)/R_T \rfloor} \right\rceil$. Let us further assume that the implementation platform uses a block size (B) of 8192 bytes and a block pointer size (π) of 4 bytes. Given this information, one can easily compute the total size of the database for the schema in Figure 1 to be approximately 700 MB.

In addition to the base tables, for retrieval efficiency, index structures would typically be defined (especially in data warehouses, where fact tables are very large). More specifically, the `SALES` table will likely be indexed on each of its four dimensional attributes.

Most database systems (including commercial DBMSs) provide a referencing primitive for individual records known as *Row Identifiers* (RID)[1]. Using RIDs, base records are pointed to by indexes. For instance, the leaf level pointers in a B^+-Tree point to a set of RIDs, which allows the base records to be retrieved from the disk blocks where they are stored. In general, a RID consists of a unique block identifier (internally generated by the database) concatenated with the number of the slot in which the row appears. For instance, the value of the RID for the second record in block 1234 would be 123402. We will use the notion of RIDs to clarify the concepts presented in this paper.

A large number of indexing schemes have been proposed in the literature. Among these, four index types have been shown in [22] to be particularly appropriate for data warehousing/OLAP and they are indeed used in both commercial and research systems. These are:

- *"standard" B^+-tree :* B^+-trees and their derivatives have been successfully used in relational systems for a number of years. Each entry in the leaf-level of the index points to a list of RIDs, which, in turn, point to the base data. For any given *search-key value*, that is for each distinct value that the field contains in the table, this list is stored in a chain of data blocks.
- *Bitmapped B^+-tree :* The only difference between bitmapped and standard B^+-trees is that the list of RIDs associated with each distinct search-key value is replaced, in bitmapped trees, by a vector of $|T|$ bits, each representing a row in the table. A bit is set to 1 if the corresponding row matches the search-key value associated with that vector.
- *Projection:* As its name implies, a projection index simply consists of a mirror-copy of a specific column in a table. Entries in the index appear in the same order as the corresponding search-key values in the underlying table, which provides an efficient mapping from search-key values to table records. The projected column, which may or may not be duplicated, is stored as a chain of one or more data blocks. Some implementations of projection indexes, such as Sybase IQ [29], do *not* duplicate the columns.
- *Bit-sliced:* The bit-sliced index follows the same principle as the projection index: the value of the indexed column of each row in the table is represented in that order in the index. However, each value

[1] Variously referred to in the literature as *Record Identifiers*, *Tuple Identifiers* etc.

is "sliced" instead of being stored as is. In other words, each bit position in the column being indexed is stored in a separate chain of data blocks.

While the above-mentioned structures have indeed improved query performance, they have imposed a significant space penalty. Certain schemes, such as the standard B^{+}-tree and bitmapped B^{+}-tree , incur much more storage overhead than others due to the nature of their structure. It is not uncommon for a standard or bitmapped B^{+}-tree index on a single column of a table to be 50 to 100% of the size of the table itself. In fact, even for moderate indexed databases, where only a few columns are indexed, storage requirements can increase by factors of 3 to 10. These numbers will be corroborated in our performance study in Section 4. This cost is substantial, both in absolute as well as percentage terms. In response, we ask the following question: "*is it possible to reduce this cost, without sacrificing the efficiency obtained by indexing?*"

It turns out that the answer to the above question is in the affirmative. Recall that a projection index is simply a mirror image of the column being indexed. In particular, when indexing columns of a fact table, storing both the index and the corresponding column in the fact table results in a duplication of data. In such situations, it might be advisable to only store the index if original table records can be reconstructed easily from the index itself. This is one of the basic ideas behind the Curio solution.

By applying this idea, we can divide the SALES table in figure 1, into five smaller tables, as shown in figure 2. The new schema is then composed of 5 vertical partitions: one for each of the dimensional

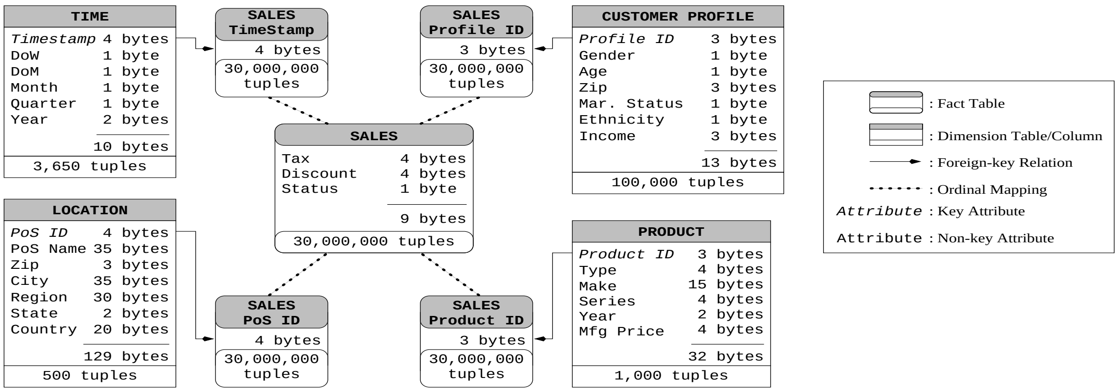

Figure 2: Example Warehouse Schema with Vertical Partitioning Index Scheme

attributes (i.e., SALES.TimeStamp, SALES.PoS ID, SALES.Profile ID and SALES.Product ID), and one for the remaining columns from the original, i.e, the business metrics of the SALES table (i.e., Tax, Discount and Status). A record in the original SALES table is now partitioned into 5 individual records, one in each of the resulting tables. Any original record can easily be re-built from these, since its component rows in the 5 resulting tables all share the same ordinal position.

The resulting database size can be computed to be approximately 700 MB. This 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 yet powerful technique, thus, we can store the data and index for the same storage cost as for the data alone. In terms of storage cost, the indexing is free.

Each of the four tables `SALES.Timestamp`, `SALES.Profile ID`, `SALES.PoS ID` and `SALES.Product ID` can be considered a positional index. Broadly speaking, an index is a storage structure that allows easy access to particular tuples from a given search-key value. A positional index is an extension of this basic idea: in addition to providing a mapping from column values to tuples, it is also the sole repository of these column values.

We now proceed to describe the positional indexing approach in more detail.

3 The Positional Indexing Approach

As discussed in the previous section, a positional index can be very simply created as a vertical partition of a relational table: the column being indexed is removed from its original table, and stored separately, with each entry in the same position as its corresponding base record. This isolated column can then be used as a projection index for fast access to data in the table. A graphical representation of this structure is shown in figure 3. In this figure, we show the configurations of the two cases: a base table

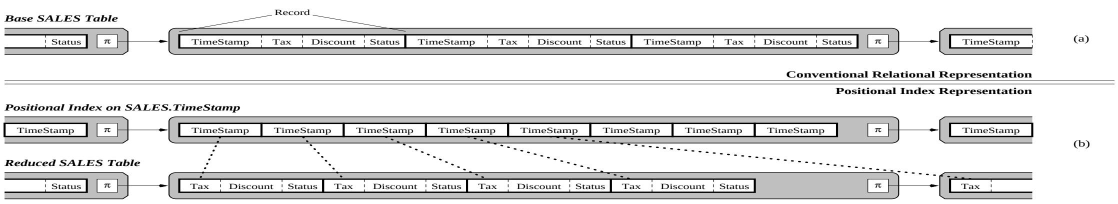

Figure 3: A Positional Index

(shown in figure 3a) and the corresponding positional index and reduced table (shown in figure 3b). The base table consists of the attributes TimeStamp, Tax, Discount, Status and a positional index is constructed on the TimeStamp column. As can be seen, the positional index is simply a chain of TimeStamp attribute values, whereas the reduced table contains the remaining fields from the base table, namely Tax, Discount and Status.

As indicated by the dotted lines joining the positional index records with the corresponding records in the reduced table, the order of the records in the base table is conserved in both the positional index and the reduced table. This property allows for the easy mapping between `TimeStamp` values and records in the reduced tables (and vice-versa). This mapping relies on the fact that there exists a fixed number of "slots" for holding records in each data block. Denoting by $w(A)$ the width of an attribute A, the number of slots per page in the positional index of figure 3 is given by $\mathrm{SPP}_{\mathrm{BDI}} = \left\lfloor \frac{B-\pi}{w(\mathtt{TimeStamp})} \right\rfloor$. The corresponding value for the reduced table is $\mathrm{SPP}_{\mathrm{Reduced}} = \left\lfloor \frac{B-\pi}{w(\mathsf{Tax})+w(\mathsf{Discount})+w(\mathsf{Status})} \right\rfloor$.

Based on these values, one can easily associate the elements of a record in the positional index and the reduced table by using a simple arithmetic mapping. Assume for instance that we need to access the record in the base table corresponding to some search-key value K. This translates to associating the positional index record K (assuming, of course, that there exists a positional index on the column of which K is a value) with its associated record in the reduced table. We first compute the RID of the K record in the positional index, i.e., we determine that K is located in the $(b_{\mathtt{TimeStamp}})^{\mathrm{th}}$ block of the positional index, at slot number $s_{\mathtt{TimeStamp}}$. K is found through a scan operation. This can be done easily by simply keeping track of the number of data blocks loaded during the scan operation and noting 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 $\Omega = b_{\mathtt{TimeStamp}} \times \mathrm{SPP}_{\mathtt{TimeStamp}} + s_{\mathtt{TimeStamp}}$. From this number, we can determine the RID of the corresponding entry in the reduced table. This RID is characterized by b_{Reduced}, the number of the data block in the block chain of the reduced table in which the record is located, and s_{Reduced}, the slot in that block corresponding to the desired record. b_{Reduced} is given by $b_{\mathrm{Reduced}} = \left\lfloor \frac{\Omega}{\mathrm{SPP}_{\mathrm{Reduced}}} \right\rfloor$; and $s_{\mathrm{Reduced}} = \Omega \;\mathtt{rem}\; \mathrm{SPP}_{\mathrm{Reduced}}$, where "`rem`" is the remainder operator (i.e., the "%" operator in C).

These mapping operations can easily be implemented through a few integer operations and will thus be fast on modern architectures. In fact, in most cases, the delays associated with these computations will be negligible compared to the much slower storage access times. The vertical partitioning of tables associated with positional indexing thus appears to carry little computational overhead for accessing a table.

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 [21]. 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.

The ideas presented in this section form the basis of the positional indexing techniques underlying Curio. These ideas have been extended to develop several additional proprietary data structures and algorithms that allow efficient join and aggregation operations. For instance, another noteworthy feature of Curio is the way it performs joins. Essentially there exists a mapping between individual elements of the foreign key columns (i.e., dimensional key values) of the fact table and the ordinal positions of these key values in the appropriate dimension table. This mapping allows for fast evaluation of join operations. These relatively simple, yet elegant data structures and algorithms greatly reduce the number of disk accesses required for query operations, thereby achieving fast response times. We now proceed to demonstrate the performance of these techniques.

4 Performance Evaluation of Curio

In this section, we present the results of our performance study, which compares the query processing speeds, storage requirements, and loading times of Curio with those of several existing RDWMS products. We have done benchmarking against virtually all the ROLAP vendors. However, due to space and readability concerns (we refer the reader to the plots that appear later in this section to justfiy this concern), we only report the results for 3 specific products. The relative performance of Curio is essentially uniform when compared to all the ROLAP products that we considered. The products selected for the study are Oracle (version 8.0.5) [25], Red Brick Warehouse (version 5.1.5) [26], and DB2 Universal Database (version 5.0) [16]. These products were selected because their respective vendors have been identified as the market leaders in ROLAP-based data warehousing solutions. In a recent study [9], Oracle has the largest market share (31.8%), followed closely by IBM (21.7%). We also felt that Red Brick (now acquired by Informix) would be an interesting comparison as this product is a "warehouse mostly" solution (like Curio).

We first present the query processing results, since we believe query performance to be the most important aspect of this analysis. All tests were performed on a Windows NT machine having a single 300 MHz Intel Pentium processor and 64 MB of RAM. The queries used in this experiment are based on the star schema shown in Figure 4. This schema has 3 dimensions: `PRODUCT`, `CUSTOMER`, and `TIME`. The fact table is PURCHASE, which has a single foreign key column corresponding to each of the 3 dimensions. The numbers displayed next to each column indicate the size of the column in bytes. For each RDWMS product, indexes were built on the following columns: PURCHASE.Customer_ID, PURCHASE.Product_ID, PURCHASE.TimeKey, PURCHASE.Quantity, and CUSTOMER.Age. This scheme indexed *only those columns used in the test queries*, thus providing a significant advantage for the above mentioned products in terms of storage requirements. Typically, in a warehouse environment, indexes would be built on most (if not all) attributes, incurring significant additional storage overhead. Where possible, we used specialized indexes to give the other products as much of an advantage as possible. For example, bitmapped indexes were used in Oracle and star indexes in Red Brick.

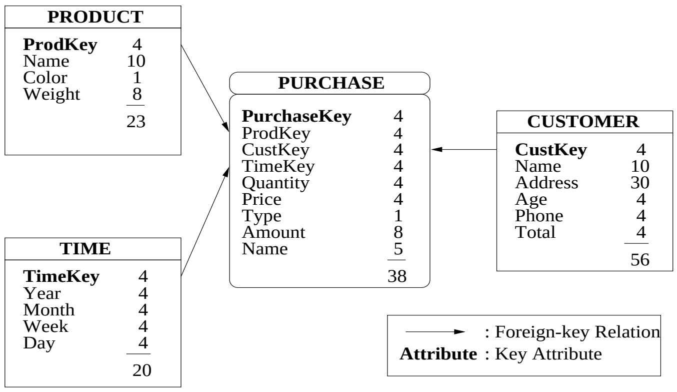

Figure 4: Warehouse Schema Used in Performance Evaluation

The data for the schema were randomly generated and loaded into each RDWMS using their respective bulk load utilities. Data loading and index creation were treated as separate steps so that loading times could be compared to Curio (since Curio does not have an index creation step). Three database sizes were considered in this experiment: 0.25 GB, 0.5 GB, and 1 GB. Database size here refers to the size of the raw data, and thus does not include any overhead that may be added once the data is loaded. Table 1 lists the number of tuples in each table for each of the sizes considered.

Table	0.25 GB	0.5 GB	1 GB
PURCHASE	5,000,000	14,000,000	22,000,000
CUSTOMER	10,000	20,000	40,000
PRODUCT	100,000	200,000	400,000
TIME	2,500	5,000	10,000

Table 1: Table Cardinalities for Performance Evaluation

The queries used in this experiment are described in Table 2. For each query, Table 2 includes a description of the query, the SQL formulation, and the *characteristics* of the query. The characteristics indicate the complexity of the query in terms of number of tables joined, number of restrictions, and number of grouping columns for aggregation queries. For instance, Query 9 joins 4 tables, has 1 fact table restriction and 2 dimension table restrictions, computes a summation, and groups by 2 columns. Note that fact table restrictions are much more costly than dimension table restrictions, since many more tuples must be evaluated.

For each of the queries, we provide a plot of the response times for each RWDMS at each of the 3 data size levels. Response time, for the purposes of this experiment, is defined as the elapsed time (in seconds) from the start of the query until the first record is displayed. This metric was selected because the display of the first record marks the point at which an analyst may begin processing the retrieved information. The most unproductive time of a query session is the time an analyst spends waiting for the first record to appear, and so that is what we measure in this analysis [4]. Note that the queries were run under controlled conditions to eliminate the effects of extraneous events. In particular, the machine was run in a "stand-alone" mode, thus rendering unnecessary the need for running any network processes. Additionally, the only processes running on the test machine other than the relevant RDWMS software processes were core OS processes. Also, to make the test fair, we eliminated the effects of caching by restarting the machine between each successive query.

The results for all tests are presented using bar charts. The numbers above each bar for Red Brick, DB2, and Oracle are ratios of the performance compared to that of Curio. We first present the results of the queries. We then follow with a comparison of storage requirements, and then a comparison of loading times.

4.1 Query Results

The first 4 queries are the least complex in the sense that they do not involve any aggregation. Figures 5A through 5D show the response times for each of the selected products on these 4 queries. In each of these figures, it is clear that Curio outperforms the other products, and this performance advantage increases as the raw data size increases. For Query 1 at the 0.25 GB level (Figure 5A), Red Brick appears to come close to the 16 second response time of Curio. However, Curio is still nearly twice as fast as Red Brick (29 seconds). For the large data size (1 GB), Curio (54 seconds) is approximately 10 times faster than Oracle (548 seconds) on the same query. Note that Query 1 is a single table query. For more complex queries involving joins, Curio performs especially well as shown in Figures 5B, 5C,

ID	Description	SQL Formulation	Characteristics
1	List high volume purchases for a particular customer.	SELECT Purchase_ID, Type, Quantity FROM PURCHASE WHERE Customer_ID = x AND Quantity > y	single table query 2 restrictions 3 columns projected
2	List high volume purchases and the associated customers.	SELECT C.Name, Quantity FROM PURCHASE P, CUSTOMER C WHERE C.Customer_ID = P.Customer_ID AND C.Customer_ID = x AND Quantity > y	2-way join 2 restrictions: 1 fact, 1 dimension 2 columns projected
3	Find customers having high volume purchases of a particular product.	SELECT Purchase_ID, C.Name, P2.Name, Quantity FROM PURCHASE P1, PRODUCT P2, CUSTOMER C WHERE P1.Product_ID = P2.Product_ID AND P1.Customer_ID = C.Customer_ID AND P2.Product_ID = x AND Quantity > y	3-way join 2 restrictions: 1 fact, 1 dimension 4 columns projected
4	Find customers having high volume purchases of a particular product and the month of these purchases.	SELECT C.Name, P2.Name, Month, Quantity FROM PURCHASE P1, PRODUCT P2, CUSTOMER C, TIME T WHERE P1.Product_ID = P2.Product_ID AND P1.Customer_ID = C.Customer_ID AND P1.TimeKey = T.TimeKey AND P2.Product_ID = x AND Quantity > y	4-way join 2 restrictions: 1 fact, 1 dimension 4 columns projected
5	Find the total number of purchases.	SELECT count(*) FROM PURCHASE	single table query no restrictions aggregation type = COUNT no GROUP BY columns
6	Find the total quantity purchased for each product having high volume purchases.	SELECT P2.Name, SUM(Quantity) FROM PURCHASE P1, PRODUCT P2 WHERE P1.Product_ID = P2.Product_ID AND P2.Product_ID = x AND Quantity > y GROUP BY P2.Name	2-way join 2 restrictions: 1 fact, 1 dimension 2 columns projected aggregation type = SUM 1 GROUP BY column
7	For a particular product, find the maximum quantity purchased by elderly customers.	SELECT C.Name, MAX(Quantity) FROM PURCHASE P1, PRODUCT P2, CUSTOMER C WHERE P1.Product_ID = P2.Product_ID AND P1.Customer_ID = C.Customer_ID AND P2.Product_ID = x AND C.Age > y GROUP BY C.Name	3-way join 2 dimension restrictions 2 columns projected aggregation type = MAX 1 GROUP BY column
8	For a range of products, find the maximum quantity purchased by elderly customers.	SELECT C.Name, P.Name, MAX(Quantity) FROM PURCHASE P1, PRODUCT P2, CUSTOMER C WHERE P1.Product_ID = P2.Product_ID AND P1.Customer_ID = C.Customer_ID AND P2.Product_ID < x AND C.Age > y GROUP BY C.Name, P.Name	3-way join 2 dimension restrictions 3 columns projected aggregation type = MAX 2 GROUP BY columns
9	For elderly customers, find the total quantity of high volume purchases for a range of products. List the month of purchase.	SELECT C.Name, Month, SUM(Quantity) FROM PURCHASE P1, PRODUCT P2, CUSTOMER C, TIME T WHERE P1.Product_ID = P2.Product_ID AND P1.Customer_ID = C.Customer_ID AND P1.TimeKey = T.TimeKey AND P2.Product_ID < x AND Quantity > y AND C.Age > z GROUP BY C.Name, Month	4-way join 3 restrictions: 1 fact, 2 dimension 3 columns projected aggregation type = SUM 2 GROUP BY columns
10	For elderly customers, find the total quantity of high volume purchases for a range of products. List the product name and month of purchase.	SELECT C.Name, P.Name, Month, SUM(Quantity) FROM PURCHASE P1, PRODUCT P2, CUSTOMER C, TIME T WHERE P1.Product_ID = P2.Product_ID AND P1.Customer_ID = C.Customer_ID AND P1.TimeKey = T.TimeKey AND P2.Product_ID < x AND Quantity > y AND C.Age > z GROUP BY C.Name, P.Name, Month	4-way join 3 restrictions: 1 fact, 2 dimension 4 columns projected aggregation type = SUM 3 GROUP BY columns

Table 2: Queries Used in Performance Evaluation

and 5D, which are 2-way, 3-way, and 4-way joins, respectively. For instance, a 4-way join on a 1 GB database (Query 4 in Figure 5D), Curio's response time (37 seconds) is approximately 14 times faster than that of Oracle (546 seconds).

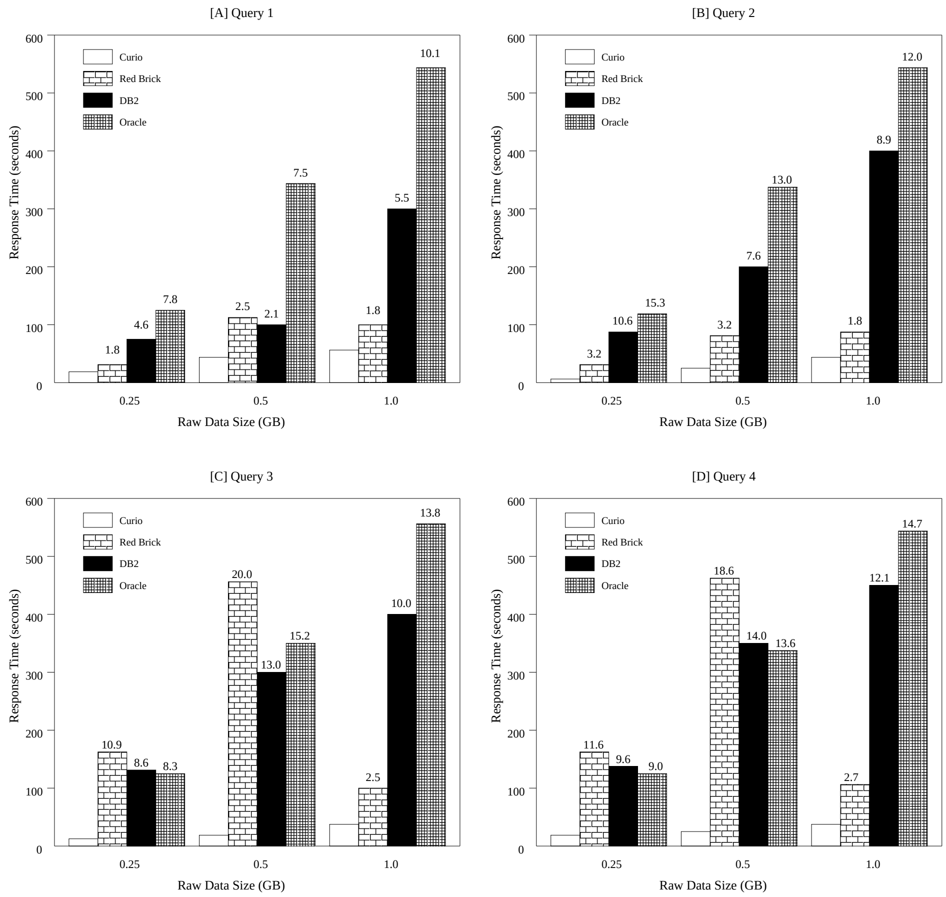

Figure 5: Response Times for Non-Aggregation Queries (Queries 1 - 4)

We now consider the more complex aggregation queries, Queries 5 - 10. Performance results for these queries are displayed in Figures 6A through 6F. These results are similar to those obtained for the non-aggregation queries, except that the magnitude of Curio's performance advantage is much greater in most cases. Query 5, shown in Figure 6A, is the simplest of the aggregation queries. It consists of a count involving only a single table and has no restrictions. One would therefore expect this query to be evaluated quickly by most systems. Both Curio and Red Brick do so by answering the query in less than 1 second for all 3 data sizes. Oracle and DB2, on the other hand, require 957 and 750 seconds, respectively, on a 1 GB database.

The remaining queries increase in complexity. For example, Query 6 is a 2-way join having 2 restrictions and 1 grouping column, whereas Query 10 is a 4-way join having 3 restrictions and 3 grouping columns. For all queries and all data sizes, Curio *significantly* outperforms the other products. To get an idea of the magnitude of Curio's advantage, consider Query 9 in Figure 6E, a 4-way join having 3 restrictions and 2 grouping columns. On a 1 GB database, Curio performs this query in 118 seconds, approximately 83 times faster than DB2 (9,900 seconds), 97 times faster than Oracle, (11,490 seconds), and 119 times faster than Red Brick (14,100 seconds).

4.2 Storage Requirements Results

We now compare the storage requirements of each product for loading the 3 sizes of raw data. The resulting database sizes for the unindexed data and the indexed data are provided in Figures 7A and 7B, respectively. From these figures, it is clear that in general, Curio is much more efficient in terms of storage requirements. For example, 1 GB of unindexed data requires nearly twice as much disk space in Oracle, as shown in Figure 7A, while 1 GB of indexed data requires 3 times the space, as shown in Figure 7B. Red Brick is clearly the second best performer in terms of storage requirements, especially for unindexed data (Figure 7A). However, once indexes are created, storage requirements increase significantly (e.g., more than 1.5 times for 1 GB of data for Red Brick), as shown in Figures 7A and 7B.

Such increases in storage requirements can become quite costly in dollar terms. There has been a recent trend towards centralization of storage, driven mostly by the prohibitive cost of transmitting and receiving data over clogged networks. As a result, overall storage expenses have increased despite a significant decrease in hard disk prices. According to a recent study, 35% of all IT corporate budgets are now being spent on storage costs as opposed to 30% in 1994 [10]. These costs include the maintenance and purchase of storage media. It is also estimated that for every dollar spent on storage media, seven dollars are spent on annual maintenance [28]. These costs can be significant, and it has been our experience that for a fully indexed data warehouse, Curio offers a tremendous savings in storage costs (at least an order of magnitude). Many IT managers view this feature of Curio as more attractive than the query performance gains.

4.3 Loading Time Results

Finally, we examine the time to load the data and build the indexes for each product. Comparisons of these times are provided in Figures 8A and 8B, respectively. From these figures, it is clear that

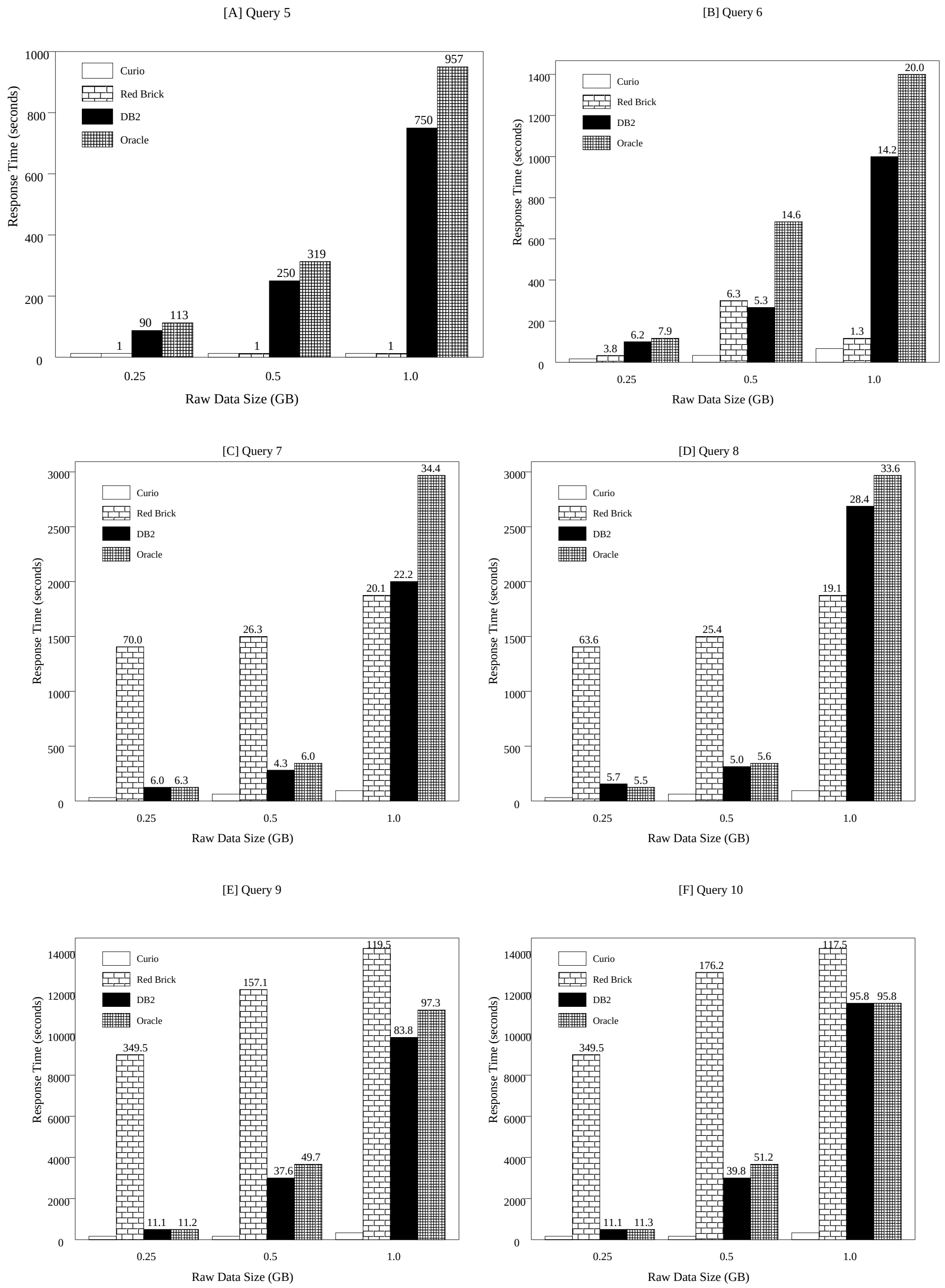

Figure 6: Response Times for Aggregation Queries (Queries 5 - 10)

an additional advantage of the Curio design is its superior performance in loading. Without even considering the fact that Curio requires no index creation, Curio is still significantly faster than the other products. For example, Curio loaded 1 GB of data in 508 seconds, 10 times faster than Oracle (5,535 seconds, or just over 1.5 hours) as shown in Figure 8A. In Figure 8B, index creation times are

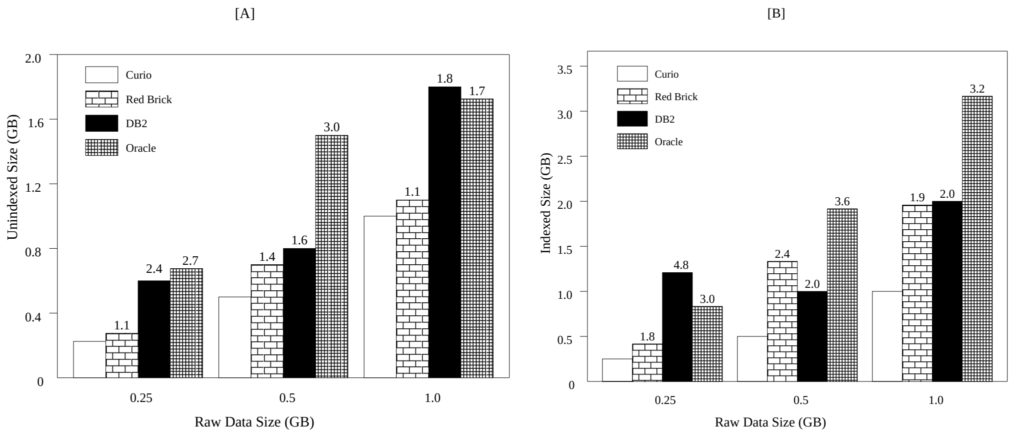

Figure 7: Comparison of Database Sizes

displayed. Although Curio essentially has no index creation time, we have repeated the loading times for Curio on this plot. This figure shows the magnitude of the costs associated with building indexes. Of the 3 other products, the fastest creation time for 1 GB of data is Red Brick, 14,940 seconds or just over 4 hours, while the slowest time is Oracle, 45,738 seconds or over 12.5 hours.

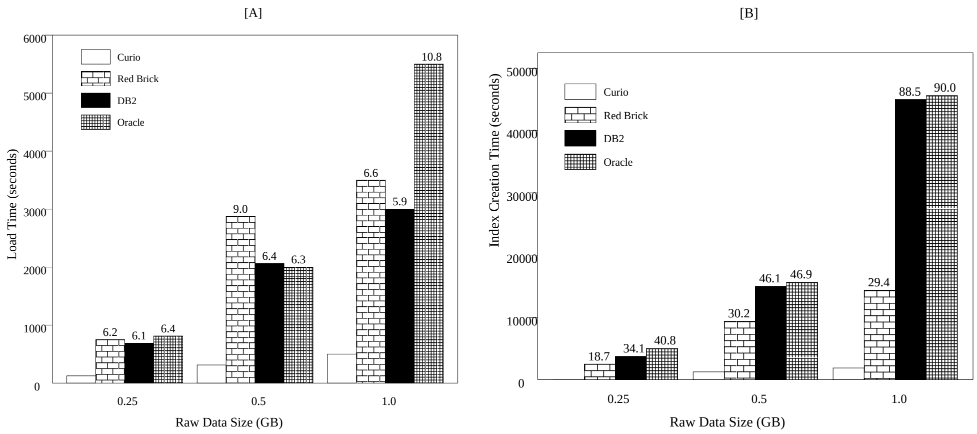

Figure 8: Comparison of Loading Times

5 Other Advantages of Curio

Thus far we have described Curio mainly in terms of query processing cost, storage requirements, and to some extent, loading time. In section 4, we showed that Curio is significantly faster in processing OLAP queries, less costly in terms of storage requirements, and much faster in loading data when compared to several of the leading existing relational RDWMS solutions. However, there are several properties of the Curio approach that provide additional advantages. We now discuss some of these advantages.

•**Reduced Number of Block Accesses:** The vertical partitioning employed by Curio brings about an additional advantage. A large portion of OLAP queries access only a subset of the attributes of the tables examined. In a conventional RDWMS, even if the value of a single attribute is needed, entire records must be loaded from disk, because attribute values inside unindexed records are stored contiguously. For instance, in a conventional implementation of our sample schema from section 2, all attributes of the `SALES` table will always be loaded from disk, even when only the PoS ID and Tax attributes are needed. In other words, conventional RDWMSs most often load extraneous data. Because the Curio approach partitions warehouse data by column, only the relevant columns are loaded to answer a query.

• **High Compressibility:** The storage requirements of Curio can be further reduced by compressing each index in a given schema. Though this approach could also potentially be applied to conventional RDWMSs, storage size reductions are likely to be much higher with Curio. Indeed, most compression algorithms will yield higher compression ratios if the set of values being compressed ranges over a small domain (and thus displays repetitions) [11]. This is more likely to happen with a vertical partitioning approach than with a conventional table, where the data is stored row by row. Moreover, while it will sometimes be necessary to decompress the data, there exist query-processing algorithms that can operate on compressed data. We can hence expect that compressed Curio indexes can also provide higher performance gains than compressed versions of the other indexes described in this paper.

• **Better Memory Usage:** 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.TimeStamp` attribute appears in both the `SELECT` and `WHERE` clauses of an SQL query, then one can save several thousand block accesses if the index for that attribute is kept in memory after being loaded. Similarly, since the overall size of the database is smaller with Curio than with conventional structures, it is likely that the number of memory faults will be smaller with Curio. For instance, consider a database that is 700 MB when implemented in Curio and 1.1 GB when implemented using a conventional indexing scheme, such as the B^+-tree . Further assume that the size of main memory is 256 Mbytes. We can then expect the Curio approach to find the required column in memory approximately $\frac{256\text{Mbytes}}{700\text{Mbytes}} \approx 36\%$ of the time. On the other hand, the conventional approach will only yield a block hit ratio of about $\frac{256\text{Mbytes}}{1.1\text{Gbytes}} \approx 23\%$. Thus, the Curio approach should yield significant performance advantages by reducing the number of times that a data block actually needs to be loaded from disk.

• **Reduced Memory Requirements:** The algorithm used by Curio to perform star-join queries has the property that the *memory requirements are independent of the size of the fact table.* Given the extreme size of most warehouses, this is a valuable property, as it allows for the rapid growth that typically occurs in warehouses.

• **Easy Warehouse Refresh:** In order to remain up-to-date, a data warehouse must be periodically *refreshed.* That is, its tables and indexes must be updated to reflect changes in the real world. Since a warehouse stores a history, a refresh consists of a series of append operations to the fact table[2] and updates to the corresponding indexes. Although appending data to existing tables can be performed relatively efficiently, updating the corresponding indexes can be quite costly since these indexes might need to be completely reorganized. Since Curio does not utilize indexes, it should be clear that updating a Curio warehouse would not incur this additional index updating cost. The Curio refresh process assumes that data is simply appended; i.e., no sorting or other operations are performed. As shown in Section 4, Curio performs loads extremely fast when compared to other products. This property is indeed desirable since, in many application arenas, the time window available for warehouse refreshes is shrinking [3].

6 Conclusion

In this paper, we presented Curio, an efficient OLAP query engine and data repository based on a simple data storage and indexing scheme. Through an example, we showed the clear superiority of the positional indexing approach underlying Curio from the point of view of storage costs. Clearly, one of the main advantages of this approach lies in the fact that essentially indexing is provided "for free". We also showed that Curio outperforms other leading RDWMS products (often by orders of magnitude) in terms of query speed and storage requirements. Finally, we discussed some additional advantages that can be gained from using Curio. Briefly, query processing with Curio is likely to occur faster than with a conventional implementation because a) only the relevant attributes need to be loaded, b) these attributes are highly compressible, and c) Curio allows for a much more efficient use of main memory.

[2]Changes to dimension tables occur much less frequently, so we assume that they are not updated during a normal warehouse refresh.

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] 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.

[3] J. Byard and D. Schneider. The ins and outs (and everything in between) of data warehousing. Tutorial in ACM SIGMOD Intl. Conf. on Management of Data, Montreal, Quebec, Canada, June 4-6 1996.

[4] 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.

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

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

[7] 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.

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

[9] International Data Corp. Data warehousing tools: 1998 worldwide markets and trends, report 17622, 1998.

[10] B. Francis. Managing storage silos. In *IT Outlook Supplement of PC Week*, pages 2–19, November 16 1998.

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

[12] 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.

[13] 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.

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

[15] Hyperion Solutions Corp. Hyperion essbase olap server.

[16] IBM Corp. Db2 universal database version 5.0 for windows nt, 1997.

[17] 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.

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

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

[20] MicroStrategy, Inc. Dss agent.

[21] 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.

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

[23] Oracle Corp. Oracle express.

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

[25] Oracle Corp. Oracle8 enterprise edition release 8.0.5 for windows nt, 1998.

[26] Red Brick Systems. Red brick warehouse version 5.1 for windows nt, 1998.

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

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

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