C-Store: A Column-oriented DBMS

Mike Stonebraker[*], Daniel J. Abadi[*], Adam Batkin[+], Xuedong Chen[†], Mitch Cherniack[+], Miguel Ferreira[*], Edmond Lau[*], Amerson Lin[*], Sam Madden[*], Elizabeth O'Neil[†], Pat O'Neil[†], Alex Rasin[‡], Nga Tran[+], Stan Zdonik[‡]

[*]MIT CSAIL, Cambridge, MA
[+]Brandeis University, Waltham, MA
[†]UMass Boston, Boston, MA
[‡]Brown University, Providence, RI

Abstract

This paper presents the design of a read-optimized relational DBMS that contrasts sharply with most current systems, which are write-optimized. Among the many differences in its design are: storage of data by column rather than by row, careful coding and packing of objects into storage including main memory during query processing, storing an overlapping collection of column-oriented projections, rather than the current fare of tables and indexes, a non-traditional implementation of transactions which includes high availability and snapshot isolation for read-only transactions, and the extensive use of bitmap indexes to complement B-tree structures.

We present preliminary performance data on a subset of TPC-H and show that the system we are building, C-Store, is substantially faster than popular commercial products. Hence, the architecture looks very encouraging.

1. Introduction

Most major DBMS vendors implement record-oriented storage systems, where the attributes of a record (or tuple) are placed contiguously in storage. With this *row store* architecture, a single disk write suffices to push all of the fields of a single record out to disk. Hence, high performance writes are achieved, and we call a DBMS with a row store architecture a *write-optimized* system. These are especially effective on OLTP-style applications.

In contrast, systems oriented toward ad-hoc querying of large amounts of data should be *read-optimized*. Data warehouses represent one class of read-optimized system, in which periodically a bulk load of new data is performed, followed by a relatively long period of ad-hoc queries. Other read-mostly applications include customer relationship management (CRM) systems, electronic library card catalogs, and other ad-hoc inquiry systems. In such environments, a *column store* architecture, in which the values for each single column (or attribute) are stored contiguously, should be more efficient. This efficiency has been demonstrated in the warehouse marketplace by products like Sybase IQ [FREN95, SYBA04], Addamark [ADDA04], and KDB [KDB04]. In this paper, we discuss the design of a column store called C-Store that includes a number of novel features relative to existing systems.

With a column store architecture, a DBMS need only read the values of columns required for processing a given query, and can avoid bringing into memory irrelevant attributes. In warehouse environments where typical queries involve aggregates performed over large numbers of data items, a column store has a sizeable performance advantage. However, there are several other major distinctions that can be drawn between an architecture that is read-optimized and one that is write-optimized.

Current relational DBMSs were designed to pad attributes to byte or word boundaries and to store values in their native data format. It was thought that it was too expensive to shift data values onto byte or word boundaries in main memory for processing. However, CPUs are getting faster at a much greater rate than disk bandwidth is increasing. Hence, it makes sense to trade CPU cycles, which are abundant, for disk bandwidth, which is not. This tradeoff appears especially profitable in a read-mostly environment.

There are two ways a column store can use CPU cycles to save disk bandwidth. First, it can code data elements into a more compact form. For example, if one is storing an attribute that is a customer's state of residence, then US states can be coded into six bits, whereas the two-character abbreviation requires 16 bits and a variable length character string for the name of the state requires many more. Second, one should *densepack* values in storage. For example, in a column store it is straightforward to pack N values, each K bits long, into N * K bits. The coding and compressibility advantages of a

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

column store over a row store have been previously pointed out in [FREN95]. Of course, it is also desirable to have the DBMS query executor operate on the compressed representation whenever possible to avoid the cost of decompression, at least until values need to be presented to an application.

Commercial relational DBMSs store complete tuples of tabular data along with auxiliary B-tree indexes on attributes in the table. Such indexes can be *primary*, whereby the rows of the table are stored in as close to sorted order on the specified attribute as possible, or *secondary*, in which case no attempt is made to keep the underlying records in order on the indexed attribute. Such indexes are effective in an OLTP write-optimized environment but do not perform well in a read-optimized world. In the latter case, other data structures are advantageous, including bit map indexes [ONEI97], cross table indexes [ORAC04], and materialized views [CERI91]. In a read-optimized DBMS one can explore storing data using only these read-optimized structures, and not support write-optimized ones at all.

Hence, C-Store physically stores a collection of columns, each sorted on some attribute(s). Groups of columns sorted on the same attribute are referred to as "projections"; the same column may exist in multiple projections, possibly sorted on a different attribute in each. We expect that our aggressive compression techniques will allow us to support many column sort-orders without an explosion in space. The existence of multiple sort-orders opens opportunities for optimization.

Clearly, collections of off-the-shelf "blade" or "grid" computers will be the cheapest hardware architecture for computing and storage intensive applications such as DBMSs [DEWI92]. Hence, any new DBMS architecture should assume a grid environment in which there are G nodes (computers), each with private disk and private memory. We propose to horizontally partition data across the disks of the various nodes in a "shared nothing" architecture [STON86]. Grid computers in the near future may have tens to hundreds of nodes, and any new system should be architected for grids of this size. Of course, the nodes of a grid computer may be physically co-located or divided into clusters of co-located nodes. Since database administrators are hard pressed to optimize a grid environment, it is essential to allocate data structures to grid nodes automatically. In addition, intra-query parallelism is facilitated by horizontal partitioning of stored data structures, and we follow the lead of Gamma [DEWI90] in implementing this construct.

Many warehouse systems (e.g. Walmart [WEST00]) maintain two copies of their data because the cost of recovery via DBMS log processing on a very large (terabyte) data set is prohibitive. This option is rendered increasingly attractive by the declining cost per byte of disks. A grid environment allows one to store such replicas on different processing nodes, thereby supporting a Tandem-style highly-available system [TAND89]. However, there is no requirement that one store multiple copies in the exact same way. C-Store allows redundant objects to be stored in different sort orders providing higher retrieval performance in addition to high availability. In general, storing overlapping projections further improves performance, as long as redundancy is crafted so that all data can be accessed even if one of the G sites fails. We call a system that tolerates K failures *K-safe*. C-Store will be configurable to support a range of values of K.

It is clearly essential to perform transactional updates, even in a read-mostly environment. Warehouses have a need to perform on-line updates to correct errors. As well, there is an increasing push toward real-time warehouses, where the delay to data visibility shrinks toward zero. The ultimate desire is on-line update to data warehouses. Obviously, in read-mostly worlds like CRM, one needs to perform general on-line updates.

There is a tension between providing updates and optimizing data structures for reading. For example, in KDB and Addamark, columns of data are maintained in entry sequence order. This allows efficient insertion of new data items, either in batch or transactionally, at the end of the column. However, the cost is a less-than-optimal retrieval structure, because most query workloads will run faster with the data in some other order. However, storing columns in non-entry sequence will make insertions very difficult and expensive.

C-Store approaches this dilemma from a fresh perspective. Specifically, we combine in a single piece of system software, both a read-optimized column store and an update/insert-oriented writeable store, connected by a *tuple mover***,** as noted in Figure 1**.** At the top level, there is a small Writeable Store (WS) component, which is architected to support high performance inserts and updates. There is also a much larger component called the Read-optimized Store (RS), which is capable of supporting very large amounts of information. RS, as the name implies, is optimized for read and supports only a very restricted form of insert, namely the batch movement of records from WS to RS, a task that is performed by the tuple mover of Figure 1.

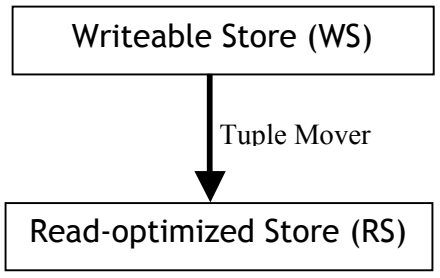

Figure 1. Architecture of C-Store

Of course, queries must access data in both storage systems. Inserts are sent to WS, while deletes must be

marked in RS for later purging by the tuple mover. Updates are implemented as an insert and a delete. In order to support a high-speed tuple mover, we use a variant of the LSM-tree concept [ONEI96], which supports a *merge out* process that moves tuples from WS to RS in bulk by an efficient method of merging ordered WS data objects with large RS blocks, resulting in a new copy of RS that is installed when the operation completes.

The architecture of Figure 1 must support transactions in an environment of many large ad-hoc queries, smaller update transactions, and perhaps continuous inserts. Obviously, blindly supporting dynamic locking will result in substantial read-write conflict and performance degradation due to blocking and deadlocks.

Instead, we expect read-only queries to be run in *historical* mode. In this mode, the query selects a timestamp, T, less than the one of the most recently committed transactions, and the query is semantically guaranteed to produce the correct answer as of that point in history. Providing such *snapshot isolation* [BERE95] requires C-Store to timestamp data elements as they are inserted and to have careful programming of the runtime system to ignore elements with timestamps later than T.

Lastly, most commercial optimizers and executors are row-oriented, obviously built for the prevalent row stores in the marketplace. Since both RS and WS are column-oriented, it makes sense to build a column-oriented optimizer and executor. As will be seen, this software looks nothing like the traditional designs prevalent today.

In this paper, we sketch the design of our updatable column store, C-Store, that can simultaneously achieve very high performance on warehouse-style queries and achieve reasonable speed on OLTP-style transactions. C-Store is a column-oriented DBMS that is architected to reduce the number of disk accesses per query. The innovative features of C-Store include:

1. A hybrid architecture with a WS component optimized for frequent insert and update and an RS component optimized for query performance.
2. Redundant storage of elements of a table in several overlapping projections in different orders, so that a query can be solved using the most advantageous projection.
3. Heavily compressed columns using one of several coding schemes.
4. A column-oriented optimizer and executor, with different primitives than in a row-oriented system.
5. High availability and improved performance through K-safety using a sufficient number of overlapping projections.
6. The use of snapshot isolation to avoid 2PC and locking for queries.

It should be emphasized that while many of these topics have parallels with things that have been studied in isolation in the past, it is their combination in a real system that make C-Store interesting and unique.

The rest of this paper is organized as follows. In Section 2 we present the data model implemented by C-Store. We explore in Section 3 the design of the RS portion of C-Store, followed in Section 4 by the WS component. In Section 5 we consider the allocation of C-Store data structures to nodes in a grid, followed by a presentation of C-Store updates and transactions in Section 6. Section 7 treats the tuple mover component of C-Store, and Section 8 presents the query optimizer and executor. In Section 9 we present a comparison of C-Store performance to that achieved by both a popular commercial row store and a popular commercial column store. On TPC-H style queries, C-Store is significantly faster than either alternate system. However, it must be noted that the performance comparison is not fully completed; we have not fully integrated the WS and tuple mover, whose overhead may be significant. Finally, Sections 10 and 11 discuss related previous work and our conclusions.

2. Data Model

C-Store supports the standard relational *logical data model*, where a database consists of a collection of named tables, each with a named collection of attributes (columns). As in most relational systems, attributes (or collections of attributes) in C-Store tables can form a unique *primary key* or be a *foreign key* that references a primary key in another table. The C-Store query language is assumed to be SQL, with standard SQL semantics. Data in C-Store is not physically stored using this logical data model. Whereas most row stores implement physical tables directly and then add various indexes to speed access, C-Store implements only *projections*. Specifically, a C-Store projection is *anchored* on a given logical table, T, and contains one or more attributes from this table. In addition, a projection can contain any number of other attributes from other tables, as long as there is a sequence of n:1 (*i.e.*, foreign key) relationships from the anchor table to the table containing an attribute.

To form a projection, we project the attributes of interest from T, retaining any duplicate rows, and perform the appropriate sequence of value-based foreign-key joins to obtain the attributes from the non-anchor table(s). Hence, a projection has the same number of rows as its anchor table. Of course, much more elaborate projections could be allowed, but we believe this simple scheme will meet our needs while ensuring high performance. We note that we use the term projection slightly differently than is common practice, as we do not store the base table(s) from which the projection is derived.

Name	Age	Dept	Salary
Bob	25	Math	10K
Bill	27	EECS	50K
Jill	24	Biology	80K

Table 1: Sample EMP data

We denote the *ith* projection over table *t* as *ti*, followed by the names of the fields in the projection. Attributes from other tables are prepended with the name of the logical table they come from. In this section, we consider an example for the standard EMP(name, age, salary, dept) and DEPT(dname, floor) relations. Sample EMP data is shown in Table 1. One possible set of projections for these tables could be as shown in Example 1.

```
EMP1 (name, age)
EMP2 (dept, age, DEPT.floor)
EMP3 (name, salary)
DEPT1(dname, floor)
```

Example 1: Possible projections for EMP and DEPT

Tuples in a projection are stored column-wise. Hence, if there are K attributes in a projection, there will be K data structures, each storing a single column, each of which is sorted on the same *sort key*. The sort key can be any column or columns in the projection. Tuples in a projection are sorted on the key(s) in left to right order.

We indicate the sort order of a projection by appending the sort key to the projection separated by a vertical bar. A possible ordering for the above projections would be:

```
EMP1(name, age| age)
EMP2(dept, age, DEPT.floor| DEPT.floor)
EMP3(name, salary| salary)
DEPT1(dname, floor| floor)
```

Example 2: Projections in Example 1 with sort orders

Lastly, every projection is *horizontally partitioned* into 1 or more *segments***,** which are given a *segment identifier***,** Sid, where Sid > 0**.** C-Store supports only value-based partitioning on the sort key of a projection. Hence, each segment of a given projection is associated with a *key range* of the sort key for the projection. Moreover, the set of all key ranges *partitions* the key space.

Clearly, to answer any SQL query in C-Store, there must be a *covering set* of projections for every table in the database such that every column in every table is stored in at least one projection. However, C-Store must also be able to reconstruct complete rows of tables from the collection of stored segments. To do this, it will need to join segments from different projections, which we accomplish using *storage keys* and *join indexes*.

Storage Keys. Each segment associates every data value of every column with a storage key, SK. Values from different columns in the same segment with matching storage keys belong to the same logical row. We refer to a row of a segment using the term *record* or *tuple***.** Storage keys are numbered 1, 2, 3, … in RS and are not physically stored, but are inferred from a tuple's physical position in the column (see Section 3 below.) Storage keys are physically present in WS and are represented as integers, larger than the largest integer storage key for any segment in RS.

Join Indices. To reconstruct all of the records in a table T from its various projections, C-Store uses *join indexes*. If *T1* and *T2* are two projections that cover a table *T*, a join index from the M segments in *T1* to the N segments in *T2* is logically a collection of M tables, one per segment, S, of *T1* consisting of rows of the form:

```
(s: SID in T2, k: Storage Key in Segment s)
```

Here, an entry in the join index for a given tuple in a segment of *T1* contains the segment ID and storage key of the corresponding (joining) tuple in *T2*. Since all join indexes are between projections anchored at the same table, this is always a one-to-one mapping. An alternative view of a join index is that it takes *T1*, sorted in some order O, and logically resorts it into the order, O' of *T2*.

In order to reconstruct *T* from the segments of *T1*, …, *Tk* it must be possible to find a *path* through a set of join indices that maps each attribute of *T* into some sort order O^*. A path is a collection of join indexes originating with a sort order specified by some projection, *Ti* , that passes through zero or more intermediate join indices and ends with a projection sorted in order O^*. For example, to be able to reconstruct the EMP table from projections in Example 2, we need at least two join indices. If we choose `age` as a common sort order, we could build two indices that map EMP2 and EMP3 to the ordering of EMP1. Alternatively, we could create a join index that maps EMP2 to EMP3 and one that maps EMP3 to EMP1. Figure 2 shows a simple example of a join index that maps `EMP3` to `EMP1`, assuming a single segment (SID = 1) for each projection. For example, the first entry of `EMP3`, `(Bob, 10K)`, corresponds to the second entry of `EMP1`, and thus the first entry of the join index has storage key 2.

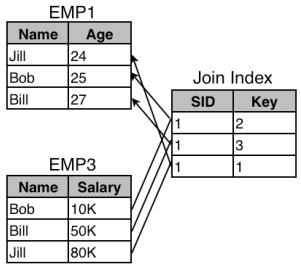

Figure 2: A join index from EMP3 to EMP1.

In practice, we expect to store each column in several projections, thereby allowing us to maintain relatively few join indices. This is because join indexes are very expensive to store and maintain in the presence of updates, since each modification to a projection requires every join index that points into or out of it to be updated as well.

The segments of the projections in a database and their connecting join indexes must be allocated to the various nodes in a C-Store system. The C-Store administrator can optionally specify that the tables in a database must be *K-safe*. In this case, the loss of K nodes in the grid will still allow all tables in a database to be reconstructed (i.e., despite the K failed sites, there must exist a covering set of projections and a set of join indices that map to some common sort order.) When a failure occurs, C-Store simply continues with K-1 safety until the failure is

repaired and the node is brought back up to speed. We are currently working on fast algorithms to accomplish this.

Thus, the C-Store physical DBMS design problem is to determine the collection of projections, segments, sort keys, and join indices to create for the collection of logical tables in a database. This physical schema must give K-safety as well as the best overall performance for a given *training workload*, provided by the C-Store administrator, subject to requiring no more than a given *space budget*, B. Additionally, C-Store can be instructed to keep a log of all queries to be used periodically as the training workload. Because there are not enough skilled DBAs to go around, we are writing an automatic schema design tool. Similar issues are addressed in [PAPA04]

We now turn to the representation of projections, segments, storage keys, and join indexes in C-Store.

3. RS

RS is a read-optimized column store. Hence any segment of any projection is broken into its constituent columns, and each column is stored in order of the sort key for the projection. The storage key for each tuple in RS is the ordinal number of the record in the segment. This storage key is not stored but calculated as needed.

3.1 Encoding Schemes

Columns in the RS are compressed using one of 4 encodings. The encoding chosen for a column depends on its *ordering* (i.e., is the column ordered by values in that column (self-order) or by corresponding values of some other column in the same projection (foreign-order), and the proportion of *distinct values* it contains. We describe these encodings below.

Type 1: Self-order, few distinct values: A column encoded using Type 1 encoding is represented by a sequence of triples, *(v, f, n)* such that *v* is a value stored in the column, *f* is the position in the column where *v* first appears, and *n* is the number of times *v* appears in the column. For example, if a group of 4's appears in positions 12-18, this is captured by the entry, (4, 12, 7). For columns that are self-ordered, this requires one triple for each distinct value in the column. To support search queries over values in such columns, Type 1-encoded columns have clustered B-tree indexes over their value fields. Since there are no online updates to RS, we can *densepack* the index leaving no empty space. Further, with large disk blocks (e.g., 64-128K), the height of this index can be kept small (e.g., 2 or less).

Type 2: Foreign-order, few distinct values: A column encoded using Type 2 encoding is represented by a sequence of tuples, *(v, b)* such that *v* is a value stored in the column and *b* is a bitmap indicating the positions in which the value is stored. For example, given a column of integers 0,0,1,1,2,1,0,2,1, we can Type 2-encode this as three pairs: (0, 110000100), (1, 001101001), and (2,000010010). Since each bitmap is sparse, it is run length encoded to save space. To efficiently find the i-th value of a type 2-encoded column, we include "offset indexes": B-trees that map positions in a column to the values contained in that column.

Type 3: Self-order, many distinct values: The idea for this scheme is to represent every value in the column as a delta from the previous value in the column. Thus, for example, a column consisting of values 1,4,7,7,8,12 would be represented by the sequence: 1,3,3,0,1,4, such that the first entry in the sequence is the first value in the column, and every subsequent entry is a delta from the previous value. Type-3 encoding is a block-oriented form of this compression scheme, such that the first entry of every block is a value in the column and its associated storage key, and every subsequent value is a delta from the previous value. This scheme is reminiscent of the way VSAM codes B-tree index keys [VSAM04]. Again, a densepack B-tree tree at the block-level can be used to index these coded objects.

Type 4: Foreign-order, many distinct values: If there are a large number of values, then it probably makes sense to leave the values unencoded. However, we are still investigating possible compression techniques for this situation. A densepack B-tree can still be used for the indexing.

3.2 Join Indexes

Join indexes must be used to connect the various projections anchored at the same table. As noted earlier, a join index is a collection of (sid, storage_key) pairs. Each of these two fields can be stored as normal columns.

There are physical database design implications concerning where to store join indexes, and we address these in the next section. In addition, join indexes must integrate RS and WS; hence, we revisit their design in the next section as well.

4. WS

In order to avoid writing two optimizers, WS is also a column store and implements the identical physical DBMS design as RS. Hence, the same projections and join indexes are present in WS. However, the storage representation is drastically different because WS must be efficiently updatable transactionally.

The storage key, SK, for each record is explicitly stored in each WS segment. A unique SK is given to each insert of a logical tuple in a table T. The execution engine must ensure that this SK is recorded in each projection that stores data for the logical tuple. This SK is an integer, larger than the number of records in the largest segment in the database.

For simplicity and scalability, WS is horizontally partitioned in the same way as RS. Hence, there is a 1:1 mapping between RS segments and WS segments. A (sid,

storage_key) pair identifies a record in either of these containers.

Since we assume that WS is trivial in size relative to RS, we make no effort to compress data values; instead we represent all data directly. Therefore, each projection uses B-tree indexing to maintain a logical sort-key order.

Every column in a WS projection is represented as a collection of pairs, *(v, sk)*, such that *v* is a value in the column and *sk* is its corresponding storage key. Each pair is represented in a conventional B-tree on the second field. The sort key(s) of each projection is additionally represented by pairs *(s, sk)* such that *s* is a sort key value and *sk* is the storage key describing where *s* first appears. Again, this structure is represented as a conventional B-tree on the sort key field(s). To perform searches using the sort key, one uses the latter B-tree to find the storage keys of interest, and then uses the former collection of B-trees to find the other fields in the record.

Join indexes can now be fully described. Every projection is represented as a collection of pairs of segments, one in WS and one in RS. For each record in the "sender," we must store the sid and storage key of a corresponding record in the "receiver." It will be useful to horizontally partition the join index in the same way as the "sending" projection and then to co-locate join index partitions with the sending segment they are associated with. In effect, each (sid, storage key) pair is a pointer to a record which can be in either the RS or WS.

5. Storage Management

The storage management issue is the allocation of segments to nodes in a grid system; C-Store will perform this operation automatically using a *storage allocator*. It seems clear that all columns in a single segment of a projection should be co-located. As noted above, join indexes should be co-located with their "sender" segments. Also, each WS segment will be co-located with the RS segments that contain the same key range.

Using these constraints, we are working on an allocator. This system will perform initial allocation, as well as reallocation when load becomes unbalanced. The details of this software are beyond the scope of this paper.

Since everything is a column, storage is simply the persistence of a collection of columns. Our analysis shows that a raw device offers little benefit relative to today's file systems. Hence, big columns (megabytes) are stored in individual files in the underlying operating system.

6. Updates and Transactions

An insert is represented as a collection of new objects in WS, one per column per projection, plus the sort key data structure. All inserts corresponding to a single logical record have the same storage key. The storage key is allocated at the site where the update is received. To prevent C-Store nodes from needing to synchronize with each other to assign storage keys, each node maintains a locally unique counter to which it appends its local site id to generate a globally unique storage key. Keys in the WS will be consistent with RS storage keys because we set the initial value of this counter to be one larger than the largest key in RS.

We are building WS on top of BerkeleyDB [SLEE04]; we use the B-tree structures in that package to support our data structures. Hence, every insert to a projection results in a collection of physical inserts on different disk pages, one per column per projection. To avoid poor performance, we plan to utilize a very large main memory buffer pool, made affordable by the plummeting cost per byte of primary storage. As such, we expect "hot" WS data structures to be largely main memory resident.

C-Store's processing of deletes is influenced by our locking strategy. Specifically, C-Store expects large numbers of ad-hoc queries with large read sets interspersed with a smaller number of OLTP transactions covering few records. If C-Store used conventional locking, then substantial lock contention would likely be observed, leading to very poor performance.

Instead, in C-Store, we isolate read-only transactions using *snapshot isolation*. Snapshot isolation works by allowing read-only transactions to access the database as of some time in the recent past, before which we can guarantee that there are no uncommitted transactions. For this reason, when using snapshot isolation, we do not need to set any locks. We call the most recent time in the past at which snapshot isolation can run the *high water mark* (HWM) and introduce a low-overhead mechanism for keeping track of its value in our multi-site environment. If we let read-only transactions set their effective time arbitrarily, then we would have to support general time travel, an onerously expensive task. Hence, there is also a *low water mark* (LWM) which is the earliest effective time at which a read-only transaction can run. Update transactions continue to set read and write locks and obey strict two-phase locking, as described in Section 6.2.

6.1 Providing Snapshot Isolation

The key problem in snapshot isolation is determining which of the records in WS and RS should be visible to a read-only transaction running at effective time ET. To provide snapshot isolation, we cannot perform updates in place. Instead, an update is turned into an insert and a delete. Hence, a record is visible if it was inserted before ET and deleted after ET. To make this determination without requiring a large space budget, we use coarse granularity "epochs," to be described in Section 6.1.1, as the unit for timestamps. Hence, we maintain an *insertion vector (IV)* for each projection segment in WS, which contains for each record the epoch in which the record was inserted. We program the tuple mover (described in Section 7) to ensure that no records in RS were inserted after the LWM. Hence, RS need not maintain an insertion

vector. In addition, we maintain a *deleted record vector (DRV)* for each projection, which has one entry per projection record, containing a 0 if the tuple has not been deleted; otherwise, the entry contains the epoch in which the tuple was deleted. Since the DRV is very sparse (mostly zeros), it can be compactly coded using the type 2 algorithm described earlier. We store the DRV in the WS, since it must be updatable. The runtime system can now consult *IV* and *DRV* to make the visibility calculation for each query on a record-by-record basis.

6.1.1 Maintaining the High Water Mark

To maintain the HWM, we designate one site the *timestamp authority* (TA) with the responsibility of allocating timestamps to other sites. The idea is to divide time into a number of *epochs*; we define the *epoch number* to be the number of epochs that have elapsed since the beginning of time. We anticipate epochs being relatively long – e.g., many seconds each, but the exact duration may vary from deployment to deployment. We define the initial HWM to be epoch 0 and start *current epoch* at 1. Periodically, the TA decides to move the system to the next epoch; it sends a *end of epoch* message to each site, each of which increments *current epoch* from *e* to *e*+1, thus causing new transactions that arrive to be run with a timestamp *e*+1. Each site waits for all the transactions that began in epoch *e* (or an earlier epoch) to complete and then sends an *epoch complete* message to the TA. Once the TA has received *epoch complete* messages from all sites for epoch *e*, it sets the HWM to be *e*, and sends this value to each site. Figure 3 illustrates this process.

After the TA has broadcast the new HWM with value *e*, read-only transactions can begin reading data from epoch *e* or earlier and be assured that this data has been committed. To allow users to refer to a particular real-world time when their query should start, we maintain a table mapping epoch numbers to times, and start the query as of the epoch nearest to the user-specified time.

To avoid epoch numbers from growing without bound and consuming extra space, we plan to "reclaim" epochs that are no longer needed. We will do this by "wrapping" timestamps, allowing us to reuse old epoch numbers as in other protocols, e.g., TCP. In most warehouse applications, records are kept for a specific amount of time, say 2 years. Hence, we merely keep track of the oldest epoch in any DRV, and ensure that wrapping epochs through zero does not overrun.

To deal with environments for which epochs cannot effectively wrap, we have little choice but to enlarge the "wrap length" of epochs or the size of an epoch.

6.2 Locking-based Concurrency Control

Read-write transactions use strict two-phase locking for concurrency control [GRAY92]. Each site sets locks on data objects that the runtime system reads or writes, thereby implementing a distributed lock table as in most distributed databases. Standard write-ahead logging is employed for recovery purposes; we use a NO-FORCE, STEAL policy [GRAY92] but differ from the traditional implementation of logging and locking in that we only log UNDO records, performing REDO as described in Section 6.3, and we do not use strict two-phase commit, avoiding the PREPARE phase as described in Section 6.2.1 below.

Locking can, of course, result in deadlock. We resolve deadlock via timeouts through the standard technique of aborting one of the deadlocked transactions.

6.2.1 Distributed COMMIT Processing

In C-Store, each transaction has a *master* that is responsible for assigning units of work corresponding to a transaction to the appropriate sites and determining the ultimate commit state of each transaction. The protocol differs from two-phase commit (2PC) in that no PREPARE messages are sent. When the master receives a COMMIT statement for the transaction, it waits until all workers have completed all outstanding actions and then issues a *commit* (or *abort*) message to each site. Once a site has received a commit message, it can release all locks related to the transaction and delete the UNDO log for the transaction. This protocol differs from 2PC because the master does not PREPARE the worker sites. This means it is possible for a site the master has told to commit to crash before writing any updates or log records related to a transaction to stable storage. In such cases, the failed site will recover its state, which will reflect updates from the committed transaction, from other projections on other sites in the system during recovery.

6.2.2 Transaction Rollback

When a transaction is aborted by the user or the C-

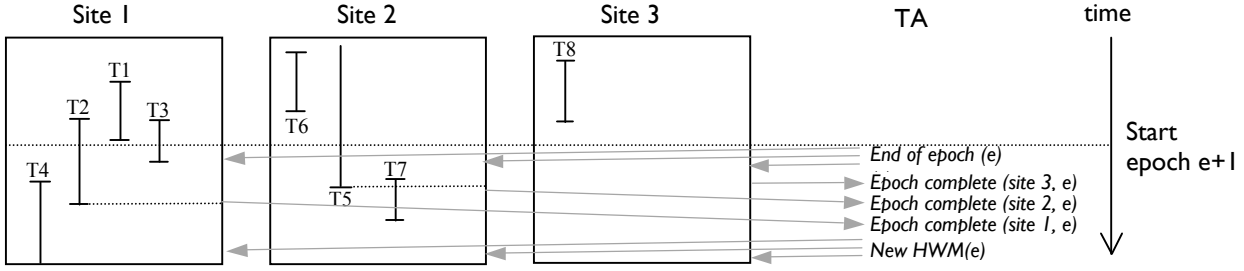

Figure 3. Illustration showing how the HWM selection algorithm works. Gray arrows indicate messages from the TA to the sites or vice versa. We can begin reading tuples with timestamp *e* when all transactions from epoch *e* have committed. Note that although T4 is still executing when the HWM is incremented, read-only transactions will not see its updates because it is running in epoch *e*+1.

Store system, it is undone by scanning backwards in the UNDO log, which contains one entry for each logical update to a segment. We use logical logging (as in ARIES [MOHA92]), since physical logging would result in many log records, due to the nature of the data structures in WS.

6.3 Recovery

As mentioned above, a crashed site recovers by running a query (copying state) from other projections. Recall that C-Store maintains K-safety; i.e. sufficient projections and join indexes are maintained, so that K sites can fail within *t,* the time to recover, and the system will be able to maintain transactional consistency. There are three cases to consider. If the failed site suffered no data loss, then we can bring it up to date by executing updates that will be queued for it elsewhere in the network. Since we anticipate read-mostly environments, this roll forward operation should not be onerous. Hence, recovery from the most common type of crash is straightforward. The second case to consider is a catastrophic failure which destroys both the RS and WS. In this case, we have no choice but to reconstruct both segments from other projections and join indexes in the system. The only needed functionality is the ability to retrieve auxiliary data structures (IV, DRV) from remote sites. After restoration, the queued updates must be run as above. The third case occurs if WS is damaged but RS is intact. Since RS is written only by the tuple mover, we expect it will typically escape damage. Hence, we discuss this common case in detail below.

6.3.1 Efficiently Recovering the WS

Consider a WS segment, *Sr*, of a projection with a sort key *K* and a key range *R* on a recovering site *r* along with a collection *C* of other projections, *M1, ..., Mb* which contain the sort key of *Sr*. The tuple mover guarantees that each WS segment, *S*, contains all tuples with an insertion timestamp later than some time $t_{lastmove}(S)$, which represents the most recent insertion time of any record in *S'*s corresponding RS segment.

To recover, the recovering site first inspects every projection in *C* for a collection of columns that covers the key range *K* with each segment having $t_{lastmove}(S) \leq t_{lastmove}(Sr)$. If it succeeds, it can run a collection of queries of the form:

```
SELECT desired_fields,
       insertion_epoch,
       deletion_epoch
FROM recovery_segment
WHERE insertion_epoch > t_lastmove(Sr)
      AND insertion_epoch <= HWM
      AND deletion_epoch = 0
          OR deletion_epoch >= LWM
      AND sort_key in K
```

As long as the above queries return a storage key, other fields in the segment can be found by following appropriate join indexes. As long as there is a collection of segments that cover the key range of *Sr*, this technique will restore *Sr* to the current HWM. Executing queued updates will then complete the task.

On the other hand, if there is no cover with the desired property, then some of the tuples in *Sr* have already been moved to RS on the remote site. Although we can still query the remote site, it is challenging to identify the desired tuples without retrieving everything in RS and differencing against the local RS segment, which is obviously an expensive operation.

To efficiently handle this case, if it becomes common, we can force the tuple mover to log, for each tuple it moves, the storage key in RS that corresponds to the storage key and epoch number of the tuple before it was moved from WS. This log can be truncated to the timestamp of the oldest tuple still in the WS on any site, since no tuples before that will ever need to be recovered. In this case, the recovering site can use a remote WS segment, *S*, plus the tuple mover log to solve the query above, even though $t_{lastmove}(S)$ comes after $t_{lastmove}(Sr)$.

At *r*, we must also reconstruct the WS portion of any join indexes that are stored locally, i.e. for which *Sr* is a "sender." This merely entails querying remote "receivers," which can then compute the join index as they generate tuples, transferring the WS partition of the join index along with the recovered columns.

7. Tuple Mover

The job of the tuple mover is to move blocks of tuples in a WS segment to the corresponding RS segment, updating any join indexes in the process. It operates as a background task looking for *worthy* segment pairs. When it finds one, it performs a *merge-out process*, *MOP* on this (RS, WS) segment pair.

MOP will find all records in the chosen WS segment with an insertion time at or before the LWM, and then divides them into two groups:

- Ones deleted at or before LWM. These are discarded, because the user cannot run queries as of a time when they existed.
- Ones that were not deleted, or deleted after LWM. These are moved to RS.

MOP will create a new RS segment that we name RS'. Then, it reads in blocks from columns of the RS segment, deletes any RS items with a value in the DRV less than or equal to the LWM, and merges in column values from WS. The merged data is then written out to the new RS' segment, which grows as the merge progresses. The most recent insertion time of a record in RS' becomes the segment's new $t_{lastmove}$ and is always less than or equal to the LWM. This old-master/new-master approach will be more efficient than an update-in-place strategy, since essentially all data objects will move. Also, notice that records receive new storage keys in RS', thereby requiring join index maintenance. Since RS items may also be deleted, maintenance of the DRV is also mandatory. Once

RS' contains all the WS data and join indexes are modified on RS', the system cuts over from RS to RS'. The disk space used by the old RS can now be freed.

Periodically the timestamp authority sends out to each site a new LWM epoch number. Hence, LWM "chases" HWM, and the delta between them is chosen to mediate between the needs of users who want historical access and the WS space constraints.

8. C-Store Query Execution

The query optimizer will accept a SQL query and construct a query plan of execution nodes. In this section, we describe the nodes that can appear in a plan and then the architecture of the optimizer itself.

8.1 Query Operators and Plan Format

There are 10 node types and each accepts operands or produces results of type projection (`Proj`), column (`Col`), or bitstring (`Bits`). A projection is simply a set of columns with the same cardinality and ordering. A bitstring is a list of zeros and ones indicating whether the associated values are present in the record subset being described. In addition, C-Store query operators accept predicates (`Pred`), join indexes (`JI`), attribute names (`Att`), and expressions (`Exp`) as arguments.

Join indexes and bitstrings are simply special types of columns. Thus, they also can be included in projections and used as inputs to operators where appropriate.

We briefly summarize each operator below.

`1. Decompress` converts a compressed column to an uncompressed (Type 4) representation.

`2. Select` is equivalent to the selection operator of the relational algebra (σ), but rather than producing a restriction of its input, instead produces a bitstring representation of the result.

`3. Mask` accepts a bitstring `B` and projection `Cs`, and restricts `Cs` by emitting only those values whose corresponding bits in `B` are 1.

`4. Project` equivalent to the projection operator of the relational algebra (π).

`5. Sort` sorts all columns in a projection by some subset of those columns (the *sort* columns).

`6. Aggregation Operators` compute SQL-like aggregates over a named column, and for each group identified by the values in a projection.

`7. Concat` combines one or more projections sorted in the same order into a single projection

`8. Permute` permutes a projection according to the ordering defined by a join index.

`9. Join` joins two projections according to a predicate that correlates them.

`10. Bitstring Operators` `BAnd` produces the bitwise AND of two bitstrings. `BOr` produces a bitwise OR. `BNot` produces the complement of a bitstring.

A C-Store query plan consists of a tree of the operators listed above, with access methods at the leaves and iterators serving as the interface between connected nodes. Each non-leaf plan node consumes the data produced by its children via a modified version of the standard iterator interface [GRAE93] via calls of "get_next." To reduce communication overhead (i.e., number of calls of "get_next") between plan nodes, C-Store iterators return 64K blocks from a single column. This approach preserves the benefit of using iterators (coupling data flow with control flow), while changing the granularity of data flow to better match the column-based model.

8.2 Query Optimization

We plan to use a Selinger-style [SELI79] optimizer that uses cost-based estimation for plan construction. We anticipate using a two-phase optimizer [HONG92] to limit the complexity of the plan search space. Note that query optimization in this setting differs from traditional query optimization in at least two respects: the need to consider compressed representations of data and the decisions about when to mask a projection using a bitstring.

C-Store operators have the capability to operate on both compressed and uncompressed input. As will be shown in Section 9, the ability to process compressed data is the key to the performance benefits of C-Store. An operator's execution cost (both in terms of I/O and memory buffer requirements) is dependent on the compression type of the input. For example, a `Select` over Type 2 data (foreign order/few values, stored as a delta-encoded bitmaps, with one bitmap per value) can be performed by reading only those bitmaps from disk whose values match the predicate (despite the column itself not being sorted). However, operators that take Type 2 data as input require much larger memory buffer space (one page of memory for each possible value in the column) than any of the other three types of compression. Thus, the cost model must be sensitive to the representations of input and output columns.

The major optimizer decision is which set of projections to use for a given query. Obviously, it will be time consuming to construct a plan for each possibility, and then select the best one. Our focus will be on pruning this search space. In addition, the optimizer must decide where in the plan to mask a projection according to a bitstring. For example, in some cases it is desirable to push the `Mask` early in the plan (e.g, to avoid producing a bitstring while performing selection over Type 2 compressed data) while in other cases it is best to delay masking until a point where it is possible to feed a bitstring to the next operator in the plan (e.g., `COUNT`) that can produce results solely by processing the bitstring.

9. Performance Comparison

At the present time, we have a storage engine and the executor for RS running. We have an early

implementation of the WS and tuple mover; however they are not at the point where we can run experiments on them. Hence, our performance analysis is limited to read-only queries, and we are not yet in a position to report on updates. Moreover, RS does not yet support segments or multiple grid nodes. As such, we report single-site numbers. A more comprehensive performance study will be done once the other pieces of the system have been built.

Our benchmarking system is a 3.0 Ghz Pentium, running RedHat Linux, with 2 Gbytes of memory and 750 Gbytes of disk.

In the decision support (warehouse) market TPC-H is the gold standard, and we use a simplified version of this benchmark, which our current engine is capable of running. Specifically, we implement the **lineitem**, **order**, and **customer** tables as follows:

```
CREATE TABLE LINEITEM (
L_ORDERKEY  INTEGER NOT NULL,
L_PARTKEY   INTEGER NOT NULL,
L_SUPPKEY   INTEGER NOT NULL,
L_LINENUMBER       INTEGER NOT NULL,
L_QUANTITY  INTEGER NOT NULL,
L_EXTENDEDPRICE    INTEGER NOT NULL,
L_RETURNFLAG       CHAR(1) NOT NULL,
L_SHIPDATE  INTEGER NOT NULL);

CREATE TABLE ORDERS  (
O_ORDERKEY  INTEGER NOT NULL,
O_CUSTKEY   INTEGER NOT NULL,
O_ORDERDATE INTEGER NOT NULL);

CREATE TABLE CUSTOMER (
C_CUSTKEY   INTEGER NOT NULL,
C_NATIONKEY INTEGER NOT NULL);
```

We chose columns of type INTEGER and CHAR(1) to simplify the implementation. The standard data for the above table schema for TPC-H scale_10 totals 60,000,000 line items (1.8GB), and was generated by the data generator available from the TPC website.

We tested three systems and gave each of them a storage budget of 2.7 GB (roughly 1.5 times the raw data size) for all data plus indices. The three systems were C-Store as described above and two popular commercial relational DBMS systems, one that implements a row store and another that implements a column store. In both of these systems, we turned off locking and logging. We designed the schemas for the three systems in a way to achieve the best possible performance given the above storage budget. The row-store was unable to operate within the space constraint so we gave it 4.5 GB which is what it needed to store its tables plus indices. The actual disk usage numbers are shown below.

C-Store	Row Store	Column Store
1.987 GB	4.480 GB	2.650 GB

Obviously, C-Store uses 40% of the space of the row store, even though it uses redundancy and the row store does not. The main reasons are C-Store compression and absence of padding to word or block boundaries. The column store requires 30% more space than C-Store. Again, C-Store can store a redundant schema in less space because of superior compression and absence of padding.

We ran the following seven queries on each system:

Q1. *Determine the total number of lineitems shipped for each day after day D.*

```
SELECT l_shipdate, COUNT (*)
FROM lineitem
WHERE l_shipdate > D
GROUP BY l_shipdate
```

Q2. *Determine the total number of lineitems shipped for each supplier on day D.*

```
SELECT l_suppkey, COUNT (*)
FROM lineitem
WHERE l_shipdate = D
GROUP BY l_suppkey
```

Q3. *Determine the total number of lineitems shipped for each supplier after day D.*

```
SELECT l_suppkey, COUNT (*)
FROM lineitem
WHERE l_shipdate > D
GROUP BY l_suppkey
```

Q4. *For every day after D, determine the latest shipdate of all items ordered on that day.*

```
SELECT o_orderdate, MAX (l_shipdate)
FROM lineitem, orders
WHERE l_orderkey = o_orderkey AND
      o_orderdate > D
GROUP BY o_orderdate
```

Q5. *For each supplier, determine the latest shipdate of an item from an order that was made on some date, D.*

```
SELECT l_suppkey, MAX (l_shipdate)
FROM lineitem, orders
WHERE l_orderkey = o_orderkey AND
      o_orderdate = D
GROUP BY l_suppkey
```

Q6. *For each supplier, determine the latest shipdate of an item from an order made after some date, D.*

```
SELECT l_suppkey, MAX (l_shipdate)
FROM lineitem, orders
WHERE l_orderkey = o_orderkey AND
       o_orderdate > D
GROUP BY l_suppkey
```

Q7. *Return a list of identifiers for all nations represented by customers along with their total lost revenue for the parts they have returned.* This is a simplified version of query 10 (Q10) of TPC-H.

```
SELECT c_nationkey, sum(l_extendedprice)
FROM lineitem, orders, customers
WHERE l_orderkey=o_orderkey AND
       o_custkey=c_custkey AND
       l_returnflag='R'
GROUP BY c_nationkey
```

We constructed schemas for each of the three systems that best matched our seven-query workload. These schema were tuned individually for the capabilities of each system. For C-Store, we used the following schema:

```
D1: (l_orderkey, l_partkey, l_suppkey,
     l_linenumber, l_quantity,
     l_extendedprice, l_returnflag, l_shipdate
     | l_shipdate, l_suppkey)
```

```
D2: (o_orderdate, l_shipdate, l_suppkey |
    o_orderdate, l_suppkey)
D3: (o_orderdate, o_custkey, o_orderkey |
    o_orderdate)
D4: (l_returnflag, l_extendedprice,
    c_nationkey | l_returnflag)
D5: (c_custkey, c_nationkey | c_custkey)
```

D2 and D4 are materialized (join) views. D3 and D5 are added for completeness since we don't use them in any of the seven queries. They are included so that we can answer arbitrary queries on this schema as is true for the product schemas.

On the commercial row-store DBMS, we used the common relational schema given above with a collection of system-specific tuning parameters. We also used system-specific tuning parameters for the commercial column-store DBMS. Although we believe we chose good values for the commercial systems, obviously, we cannot guarantee they are optimal.

The following table indicates the performance that we observed. All measurements are in seconds and are taken on a dedicated machine.

Query	C-Store	Row Store	Column Store
Q1	0.03	6.80	2.24
Q2	0.36	1.09	0.83
Q3	4.90	93.26	29.54
Q4	2.09	722.90	22.23
Q5	0.31	116.56	0.93
Q6	8.50	652.90	32.83
Q7	2.54	265.80	33.24

As can be seen, C-Store is much faster than either commercial product. The main reasons are:

- *Column representation* – avoids reads of unused attributes (same as competing column store).
- *Storing overlapping projections, rather than the whole table* – allows storage of multiple orderings of a column as appropriate.
- *Better compression of data* – allows more orderings in the same space.
- *Query operators operate on compressed representation* – mitigates the storage barrier problem of current processors.

In order to give the other systems every possible advantage, we tried running them with the materialized views that correspond to the projections we used with C-Store. This time, the systems used space as follows (C-Store numbers, which did not change, are included as a reference):

C-Store	Row Store	Column Store
1.987 GB	11.900 GB	4.090 GB

The relative performance numbers in seconds are as follows:

Query	C-Store	Row Store	Column Store
Q1	0.03	0.22	2.34
Q2	0.36	0.81	0.83
Q3	4.90	49.38	29.10
Q4	2.09	21.76	22.23
Q5	0.31	0.70	0.63
Q6	8.50	47.38	25.46
Q7	2.54	18.47	6.28

As can be seen, the performance gap closes, but at the same time, the amount of storage needed by the two commercial systems grows quite large.

In summary, for this seven query benchmark, C-Store is on average 164 times faster than the commercial row-store and 21 times faster than the commercial column-store in the space-constrained case. For the case of unconstrained space, C-Store is 6.4 times faster than the commercial row-store, but the row-store takes 6 times the space. C-Store is on average 16.5 times faster than the commercial column-store, but the column-store requires 1.83 times the space.

Of course, this performance data is very preliminary. Once we get WS running and write a tuple mover, we will be in a better position to do an exhaustive study.

10. Related Work

One of the thrusts in the warehouse market is in maintaining so-called "data cubes." This work dates from Essbase by Arbor software in the early 1990's, which was effective at "slicing and dicing" large data sets [GRAY97]. Efficiently building and maintaining specific aggregates on stored data sets has been widely studied [KOTI99, ZHAO97]. Precomputation of such aggregates as well as more general materialized views [STAU96] is especially effective when a prespecified set of queries is run at regular intervals. On the other hand, when the workload cannot be anticipated in advance, it is difficult to decide what to precompute. C-Store is aimed entirely at this latter problem.

Including two differently architected DBMSs in a single system has been studied before in data mirrors [RAMA02]. However, the goal of data mirrors was to achieve better query performance than could be achieved by either of the two underlying systems alone in a warehouse environment. In contrast, our goal is to simultaneously achieve good performance on update workloads and ad-hoc queries. Consequently, C-Store differs dramatically from a data mirror in its design.

Storing data via columns has been implemented in several systems, including Sybase IQ, Addamark, Bubba [COPE88], Monet [BONC04], and KDB. Of these, Monet is probably closest to C-Store in design philosophy. However, these systems typically store data in entry sequence and do not have our hybrid architecture nor do they have our model of overlapping materialized projections.

Similarly, storing tables using an inverted organization is well known. Here, every attribute is stored using some sort of indexing, and record identifiers are used to find corresponding attributes in other columns. C-Store uses this sort of organization in WS but extends the architecture with RS and a tuple mover.

There has been substantial work on using compressed data in databases; Roth and Van Horn [ROTH93] provide an excellent summary of many of the techniques that have been developed. Our coding schemes are similar to some of these techniques, all of which are derived from a long history of work on the topic in the broader field of computer science [WITT87]. Our observation that it is possible to operate directly on compressed data has been made before [GRAE91, WESM00].

Lastly, materialized views, snapshot isolation, transaction management, and high availability have also been extensively studied. The contribution of C-Store is an innovative combination of these techniques that simultaneously provides improved performance, K-safety, efficient retrieval, and high performance transactions.

11. Conclusions

This paper has presented the design of C-Store, a radical departure from the architecture of current DBMSs. Unlike current commercial systems, it is aimed at the "read-mostly" DBMS market. The innovative contributions embodied in C-Store include:

- A column store representation, with an associated query execution engine.
- A hybrid architecture that allows transactions on a column store.
- A focus on economizing the storage representation on disk, by coding data values and dense-packing the data.
- A data model consisting of overlapping projections of tables, unlike the standard fare of tables, secondary indexes, and projections.
- A design optimized for a shared nothing machine environment.
- Distributed transactions without a redo log or two phase commit.
- Efficient snapshot isolation.

Acknowledgements and References

We would like to thank David DeWitt for his helpful feedback and ideas.

This work was supported by the National Science Foundation under NSF Grant numbers IIS-0086057 and IIS-0325525.

[ADDA04] http://www.addamark.com/products/sls.htm

[BERE95] Hal Berenson et al. A Critique of ANSI SQL Isolation Levels. In *Proceedings of SIGMOD*, 1995.

[BONC04] Peter Boncz et. al.. MonetDB/X100: Hyper-pipelining Query Execution. *In Proceedings CIDR 2004*.

[CERI91] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *VLDB*, 1991.

[COPE88] George Copeland et. al. Data Placement in Bubba. *In Proceedings SIGMOD 1988.*

[DEWI90] David Dewitt et. al. The GAMMA Database machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March, 1990.

[DEWI92] David Dewitt and Jim Gray. Parallel Database Systems: The Future of High Performance Database Processing. *Communications of the ACM,* 1992.

[FREN95] Clark D. French. One Size Fits All Database Architectures Do Not Work for DSS. In *Proceedings of SIGMOD*, 1995.

[GRAE91] Goetz Graefe, Leonard D. Shapiro. Data Compression and Database Performance. In *Proceedings of the Symposium on Applied Computing*, 1991.

[GRAE93] G. Graefe. Query Evaluation Techniques for Large Databases. *Computing Surveys*, 25(2), 1993.

[GRAY92] Jim Gray and Andreas Reuter. *Transaction Processing Concepts and Techniques*, Morgan Kaufman, 1992.

[GRAY97] Gray et al. DataCube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1), 1997.

[HONG92] Wei Hong and Michael Stonebraker. Exploiting Inter-operator Parallelism in XPRS. In *SIGMOD*, 1992.

[KDB04] http://www.kx.com/products/database.php

[KOTI99] Yannis Kotidis, Nick Roussopoulos. DynaMat: A Dynamic View Management System for Data Warehouses. In *Proceedings of SIGMOD*, 1999.

[MOHA92] C. Mohan et. al: ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *TODS,* March 1992.

[ONEI96] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil, The Log-Structured Merge-Tree. *Acta Informatica* 33, June 1996.

[ONEI97] P. O'Neil and D. Quass. Improved Query Performance with Variant Indexes, In *Proceedings of SIGMOD*, 1997.

[ORAC04] Oracle Corporation. *Oracle 9i Database for Data Warehousing and Business Intelligence*. White Paper. http://www.oracle.com/solutions/ business_intelligence/Oracle9idw_bwp.

[PAPA04] Stratos Papadomanolakis and Anastassia Ailamaki. AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In *SSDBM 2004.*

[RAMA02] Ravishankar Ramamurthy, David Dewitt. Qi Su: A Case for Fractured Mirrors. In *Proceedings of VLDB*, 2002.

[ROTH93] Mark A. Roth, Scott J. Van Horn: Database Compression. *SIGMOD Record* 22(3). 1993.

[SELI79] Patricia Selinger, Morton Astrahan, Donald Chamberlain, Raymond Lorie, Thomas Price. Access Path Selection in a Relational Database. In *Proceedings of SIGMOD*, 1979.

[SLEE04] http://www.sleepycat.com/docs/

[STAU96] Martin Staudt, Matthias Jarke. Incremental Maintenance of Externally Materialized Views. In *VLDB*, 1996.

[STON86] Michael Stonebraker. The Case for Shared Nothing. In *Database Engineering*, 9(1), 1986.

[SYBA04] http://www.sybase.com/products/databaseservers/sybaseiq

[TAND89] Tandem Database Group: NonStop SQL, A Distributed High Performance, High Availability Implementation of SQL. In Proceedings *of HPTPS*, 1989.

[VSAM04] http://publibb.boulder.ibm.com/Redbooks.nsf/ RedbookAbstracts/sg246105.html?OpenDocument

[WESM00] Till Westmann, Donald Kossmann, Sven Helmer, Guido Moerkotte. The Implementation and Performance of Compressed Databases. *SIGMOD Record* 29(3), 2000.

[WEST00] Paul Westerman. *Data Warehousing: Using the Wal-Mart Model*. Morgan-Kaufmann Publishers , 2000.

[WITT87] I. Witten, R. Neal, and J. Cleary. Arithmetic coding for data compression. *Comm. of the ACM*, 30(6), June 1987.

[ZHAO97] Y. Zhao, P. Deshpande, and J. Naughton. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. In *Proceedings of SIGMOD*, 1997.