Obsevations about existing problems

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. Systems oriented toward ad-hoc querying of large amounts of data should be read optimized. Data warehouses represents 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. 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.
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 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.
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 sortorders opens opportunities for optimization.C-Store also allows redundant objects to be stored in different sort orders providing higher retrieval performance in addition to high availability. 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.

Key Contributions AND BASIC IDEA

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.
Features of C-Store are:
1. A column-oriented optimizer and executor, with different primitives than in a row-oriented system.
2. Heavily compressed columns using one of several coding schemes.
3. A hybrid architecture with a WS component optimized for frequent insert and update and an RS component optimized for query performance.
4. High availability and improved performance through K-safety using a sufficient number of overlapping projections.

5. 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.

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).
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.
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. 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.
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. Join Indices. To reconstruct all of the records in a table T from its various projections, C-Store uses join indexes. In practice, we expect to store each column in several projections, thereby allowing us to maintain relatively few join indices. 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.
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. 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. 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.
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 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.
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.
The query optimizer will accept a SQL query and construct a query plan of execution nodes. A C-Store query plan consists of a tree of the operators, with access methods at the leaves and iterators serving as the interface between connected nodes.
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.