C-Store: A Column-oriented DBMS

A short summary by Platon Karageorgis

The following summary introduces a relational DBMS called C-Store. It has special traits separating it than most common systems, like its column storing technique and the read optimized design. For a start, the author is explaining the differences of write and read optimized systems mentioning that the first is better for OLTP systems, when the latter is better when there is a large quantity of data. As for the column storing, C-Store stores a list of columns called projections in case they are being sorted on the same predicate. There is also a reference on the trade-off between CPU cycles and storing values, uncovering the idea of spending more processing time, in order to change the type of some values leading to benefit in terms of cost in storing. Meanwhile, the author presents the model that C-Store implements, which is consisted of a writable store and a read-optimized store, being linked by a tuple mover. This system assigns inserts to the writable store along with updates, which are put into effect by deleting the outdated value and inserting the new one, while reads are handled obviously by the read-optimized store. The tuple mover which uses LSM-trees for its function, is being analyzed later in the paper. Before the main analysis, there is also a short presentation of C-Store's specific strong aspects.

The deeper analyses of the C-Store begins with the data model, where everything is about tables and their predicates which are the source of projections. The projections as mentioned earlier are the only method being supported by C-Store, in contrast to row stores that function differently. A projection can be consisted of arguments from different tables, as long as they are connected with a foreign key, but is based on only one logical table with who they have the same number of total rows. This part also includes examples that illustrate more profoundly the way projections work and introduces the contribution of storage keys and join indices. In a nutshell, a storage key is used in order to give an "ID" to the partitioned segments, so that they can be reconstructed later by the join indices.

In the next part, the authors analyze the read-optimized store which is responsible for reading encoded data. The encoding types are being investigated in this part. To begin with, the first type is self-order, few distinct values which has three variables, the value that is being stored, the number of times it shows up and the place that it is about to be saved. The second type is foreign-order, few distinct values that has two variables, the value being stored and a sparse bitmap. The third type is self-order, many distinct values a method that attempts to link every value in a column as a delta from the previous value and the last type is foreign-order, many distinct values which does not use coding and is the most generic method.

Furthermore, in the following parts the writable store and the storage management in general are being examined. The writable store has many common virtues with the read-optimizing store, like the fact that they are both column stores, but they have a significant difference in the storage part, since the writable store must be able to handle the inserts. In writable stores, since the size is much smaller there is no need for data compression and the data are using B- trees for sorting. The tuples in this system are mostly lists of pairs consisted of the value and its storage key. In addition, C-Store uses a storage allocator whose job is to connect partitioned pieces to nodes. In general, it makes sure that values are located closely in memory with other values that are significantly linked.

Moreover, the authors extensively describe the updates and transactions of C-Store as well as the isolation rules that these take place. In C-Store, every node cannot be updated with each other constantly, so they each keep a local counter that contributes in the creation of a global storage key that makes synchronization unnecessary. In addition, the transactions are being isolated by the snapshot isolation protocol, which allows the read-only transactions to look up the database even though the database might not have been checked recently for uncommitted transactions. Also, read-write transactions implement a two-phase locking system for concurrency control, while they face deadlocks by using specific timeouts. Furthermore, there is an analyses on C-Store's possible recovery issues, stating that C-Store maintains K-safety.

In the last parts, the tuple mover's attributes are being examined along with C-Store's query execution plans. In a few words, the tuple mover is responsible for passing blocks of tuple in a writable store partition, to the proper read-optimized partition, updating meanwhile some join indices. Moreover, it is searching for interesting segment sets of two and subsequently activates a merge-out process, which is supposed to delete some records or relocate them to the read-optimized system. Finally, as long as a query arrives to C-Store, the optimizer creates a plan consisted of nodes and these nodes are briefly explained by the authors.

(Note: The performance/experimental part is skipped as usual)

Which questions aren't answered by this text?

I found the text in general really hard to read. I feel that the introduction is too long and vague while the grounds that C-Store is being created on are not clear. Also, the transition between the last parts does not really have flow and it makes it very hard for the reader.

What has changed since this was written?

I think that not much have changed even though this was written 17 years ago because in crucial parts the authors seem to take into account the evolving technology. Moreover, this system was not used much back then because row-stores had more to despite not having as good performance as C-Store. I do not see why it should be used in modern databases.