Summary of C-Store: A Column-oriented DBMS

By S. Xiao Fernández Marín

1 Summary

This paper raises the design of a read-optimized relational DBMS. It implements a column store (C-store) based system, instead of in a row. The latter is the traditional implementation of transactions, where the data is stored as all columns of a row together without having to store it sorted. It optimizes writing and is efficient on online transactions. The former stores all values of a column together (sorted). It is better optimized for reading, as the values of each column are stored in order, and efficient for query processing.

C-store does not have the problem of providing updates and optimizing data structures for reading as it divides the read store (RS- read and is very restricting when inserting) and write store (WS). WS and RS are connected by a tuple mover, that does batch updates from WS to RS. C-store also stores an overlapping collection of column-oriented projections: "group of columns stored by the same key". The thing that improves performance but also leaves redundant elements of a table, that can help for a query to be solved easily.

A projection contains one or more attributes from a table that is anchored but also some other attributes from other tables if there is a sequence of N:1 relationship. It is important to take into consideration that this projection is not the same used in others contexts, as it can have rows from other tables and it is just stored the projection result. There is also a to answer SQL queries in C-store there must be a covering set of projections for every table and be able to reconstruct rows from tables. It is done by using *storage keys* and *join indices.*

The storage keys (SK) are used to assign every data value of every column for each segment that is explicit in WS, represented as integers. The join indices are a mapping between SKs if there is more than one projection of just one table, and always the join index table be anchored to one of the tables.

The columns in RS are compressed using encodings, there are different types of encodings and the type is chosen depending on *ordering* or the proportion of *distinct values* that it has.

- Type 1: With format $(value, first, number)$. For example, $("word", 4, 9)$ would have the word "word" from position 4 to 13 (as 4 + 9 = 13). It is stored as a B-tree index. It is self-ordering (ordered by values of that column) and few distinct values.
- Type 2: With foreign-order (ordered by values of other column) and few distinct values. The format is $(value, bitmap)$. As an example, if we have a column like "word, word, plan, bottle, word", the encoding could be like $(word, 11001)$, $(plan, 00100)$ or $(bottle, 00010)$.
- Type 3: The idea of this scheme is to represent each column as a delta value from the previous one. For instance, a column of values $(1, 3, 6, 7, 21, 34)$ would be represented as $(1, 2, 3, 14, 13)$. It is self-oriented and has many distinct values.
- Type 4: Using a foreign order and many distinct values. Leaving the values unencoded as there are a large number of values.

The columns are stored using a storage allocator, as a collection of columns. An insert is seen as a collection of new objects and has a unique counter to generate a unique storage key, locally maintained by each node. For isolating the read-only transactions, we use snapshot isolation: allowing accessing the database as of some time in the recent past.

The execution of the queries are from 10 types that can produce results of 3 different types: **projection**, **single column** or **bitstring**. There are also query operators as predicates, join indexes, attribute names and expressions. The operators are: Decompress, select, mask, project, sort, aggregation operators, concat, premute, join and bristling operators.