C-STORE: A COLUMN-ORIENTED DBMS

MIT CSAIL, Brandeis University, UMass Boston And Brown University
Proceedings Of The 31st VLDB Conference, Trondheim, Norway, 2005
Presented By: Udit Panchal
Timeline of C-Store

- **2005**
  - C-Store paper published at 31st VLDB
  - Commercialization of C-Store
  - Vertica founded by Mike and Andrew Palmar

- **2011**
  - Acquired by HP
Professor Dan Abadi awarded 2015 VLDB 10-Year Best Paper Award
Contributors

• MIT
  • Sam Madden
  • Mike Stonebraker

• Yale
  • Danel Abadi

• Brown University
  • Stan Zdonik

• Brandeis University
  • Mitch Cherniack
  • Nga Tran

• UMass Boston
  • Pat O’Neil
  • Xuedong Chen
Agenda

1. History of WHY column store?
2. C-Store
3. Data Model
4. RS
5. WS
6. Storage Management
7. Updates and Transactions
8. Query execution and optimizer
9. Performance Comparison
10. Conclusion
11. Reflection
1. History of WHY column store?

- Row Store Architecture:
  - Contiguously storage of attributes of a record (tuple).
  - One write: To push all the fields in one short.
  - High write performance → Write-Optimized systems.
  - E.g. OLTP-style applications.

- Systems with ad-hoc querying nature, requires read-optimized systems.
  - E.g. Data Warehouses, CRM Systems, Electronic library card catalogs and ad-hoc inquiry systems.

- Need of Read-optimized systems, in which
  - Values of single column are stored contiguously.
  - Aggregation over the column values become time-efficient.
1. History of WHY column store?

- Why write-optimized isn’t a fit for read-only quires?
  - Brings a lot of unwanted data.

[1]: http://people.eecs.berkeley.edu/~kubitron/cs262/lectures/lec17-C-Store-Cracking.pdf
1. History of WHY column store?

- Why write-optimized DBMSs isn’t a fit for read-only queries?
  - Doesn’t bring the needy ones.
  - Padding brings less data and thus, eats up the CPU cycles.
    - Current RDBMS designed to pad attributes and stores in their native data format.
    - Leads to more usage of disk space.
  - Complete storage of tabular data along with auxiliary B-tree indexes.
    - To make the underlying record in order.

- Manufactured Characteristics to counteract the needs for read-only queries.
  - Various ways to use CPU cycles to bring more data, save disk bandwidth.
    - Code the data elements and store in compact form.
    - *Densepack* values in storage.
2. C-Store

- Introduction
  - Read-optimized relational DBMS.
- Novelties in its architecture are:
  - Only materialized views of tables are stored.
  - Active data compression
  - Hybrid architecture of:
    - WS component
    - RS component
  - Column-oriented query optimizer and operator
  - High availability and improved performance through replication-based concurrency control and recovery
  - Use of snapshot isolation to avoid 2PC and locking.

[1]: http://people.eecs.berkeley.edu/~kubitron/cs262/lectures/lec17-C-Store-Cracking.pdf
2. C-Store: Need of Hybrid Architecture

- Tension between providing on-line updates and read-only optimized data structures.
  - Warehouses need to perform on-line updates to correct errors.
  - CRM to have real-time data visibility for updates.
- In KDB and Addamark,
  - Columns of data maintains insertion order.
  - Suffice insertion, but not read
  - Non-entry sequence makes insertion difficult and expensive.
- Approached the dilemma from fresh perspective.
  - Writeable Store (WS): Support high performance inserts and updates
  - Read-Optimized Store (RS): Optimized for read and support.
  - Batch movement of records from WS to RS, preformed by tuple mover

Supports transactions of many large ad-hoc queries (RS), smaller update transaction (WS) and continues inserts (Tuple mover).

[1]: http://people.eecs.berkeley.edu/~kubitron/cs262/lectures/lec17-C-Store-Cracking.pdf
3. Data Model

- Supports the standard relational logical data model,
  - Where a database consist of named tables, each with collections of named columns.
  - Columns in C-Store can form a primary key or be a foreign key.
- Unlike other relational DBMS,
  - C-store doesn’t store the data physically using the same logical model
  - C-store stores projections of the logical data.
- Projections: “Group of columns sorted on some attributes of a given logical table”.
  - Same column may have multiple projections, sorted on a different attributes.

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Dept</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>25</td>
<td>Math</td>
<td>10K</td>
</tr>
<tr>
<td>Bill</td>
<td>27</td>
<td>EECS</td>
<td>50K</td>
</tr>
<tr>
<td>Jill</td>
<td>24</td>
<td>Biology</td>
<td>80K</td>
</tr>
</tbody>
</table>

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

- Supports the standard relational logical data model,
  - Where a database consist of named tables, each with collections of named columns.
  - Columns in C-Store can form a primary key or be a foreign key.
- Unlike other relational DBMS,
  - C-store doesn’t stores the data physically using the same logical model
  - C-store implements projections.
- Projections: “Group of columns sorted on the same attributes for a given logical table”.
  - Same column may have multiple projections, sorted on a different attributes.
  - Aggressive compression techniques would control the explosion in memory space.
  - Leads to high retrieval performance and high availability, making DBMS K-safe.

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Dept</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>25</td>
<td>Math</td>
<td>10K</td>
</tr>
<tr>
<td>Bill</td>
<td>27</td>
<td>EECS</td>
<td>50K</td>
</tr>
<tr>
<td>Jill</td>
<td>24</td>
<td>Biology</td>
<td>80K</td>
</tr>
</tbody>
</table>

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

- Projection with sort orders
  - Each column in projection is sorted based on the sort key.
  - The sort key can be a column or group of columns for a given projection.
  - Projection EMP1 would have all the column values sorted based on age.
  - Is EMP1 same as EMP4(name, age| name)?

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Dept</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>25</td>
<td>Math</td>
<td>10K</td>
</tr>
<tr>
<td>Bill</td>
<td>27</td>
<td>EECS</td>
<td>50K</td>
</tr>
<tr>
<td>Jill</td>
<td>24</td>
<td>Biology</td>
<td>80K</td>
</tr>
</tbody>
</table>

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

- Horizontal Partitioning of Projections:
  - Projection horizontally partitioned to 1 or more segments, which are given segment identifier.
  - Partitioned based on values, i.e. each segment of a given projection is associated with a key range of the sort key for the projection.
  - Must have **covering sets of projections** for every logical tables in database.
3. Data Model

- Reconstructing the complete tuple
  - Join segments from different projections
  - Using storage key and join indexes.
- Storage keys:
  - A segment may consist of one or more columns.
  - Each value of every column in segment has its associated storage key.
  - Values from different columns in the same segment, having same SK, belongs to same logical row.

So, SK is simply the logical id of tuple in logical table.
3. Data Model

- Reconstructing the complete tuple
  - Join segments from different projections
  - Using storage key and join indexes.
- Join Indices (Mapping Table):
  - To reconstruct all of the records, from various projections.
  - For T1 and T2 projections covering logical table T, the join indices would have

\[(s: \text{SID in T2, k: Storage key in segment s of T1}).\]
3. Data Model

- C-Store physical design problem:
  - To determine the collections of projections
  - Number of segments
  - Sort keys
  - Join indices
  - And to make K-safety as well as the best overall performance.
4. RS

- It’s a read-optimized column store.
  - Any segments of any projection broken into columns.
  - Each column is stored in order of the sort key for the given projection.
- Encoding schemes: Chosen by the ordering of the values in columns (Self v/s foreign)
  - Type-1: Self-order, few distinct values:
    - Columns represented as $(v, f, n)$, where,
      - $V$ is the value stored in column,
      - $F$ is the position in the column, where $f$ first appears and
      - $N$ is the number of $V$'s in the column.
    - One triple for every distinct value in the column.
    - We have a B-tree indexes over the column’s value fields to support search queries.
  - For example,
    - If a group of 4’s appears in position 12-18, then its captured by the entry of $(4, 12, 7)$. 
4. RS

- It’s a read-optimized column store.
  - Any segments of any projection broken into columns.
  - Each column is stored in order of the sort key for the given projection.
- Encoding schemes: Chosen by the ordering of the values in columns (Self v/s foreign)
  - Type-2: Foreign-order, few distinct values:
    - Columns represented as \((v, b)\), where,
      - \(V\) is the value stored in column,
      - \(B\) is the bitmap indicating the positions in which the value \(V\) is stored
    - We have a offset indexes to find the values in the columns encoded by this type.
    - For example,

```
<table>
<thead>
<tr>
<th>Column C1</th>
<th>V=0</th>
<th>V=1</th>
<th>V=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```
4. RS

- It’s a read-optimized column store.
  - Any segments of any projection broken into columns.
  - Each column is stored in order of the sort key for the given projection.
- Encoding schemes: Chosen by the ordering of the values in columns (Self v/s foreign)
  - Type-3: Self-order, many distinct values:
    - Columns represented as delta from the previous value in the column
  - For example,
    - If column has values like 1, 4, 7, 7, 8, 12, then its Type-3 encoded values would be, 1, 3, 3, 0, 1, 4
4. RS

- It’s a read-optimized column store.
  - Any segments of any projection broken into columns.
  - Each column is stored in order of the sort key for the given projection.
- Encoding schemes: Chosen by the ordering of the values in columns (Self v/s foreign)
  - Type-3: Foreign-order, many distinct values:
    - No existing compression techniques for this situation.
5. **WS**

- It’s a write-optimized column store.
  - Implements the identical physical DBMS design as RS.
  - Thus, projections and join indexes are similar for RS and WS.
  - But, storage representation is radically different, to facilitate the efficient update transactions.
    - SK, for each record is explicitly stored in the WS segment.
    - There is 1:1 mapping between the RS and WS segment.
      - I.e. for a given join index, we can get the same record in RS and WS segment.
    - Data in WS in stored in un-compressed manner, thus B-tree maintains logical sort-key order.
5. WS

- Column Representation in WS
  - Represented as a collection of (v, sk) pairs, where,
    - V is the value in the column
    - sk is its corresponding storage key
  - Represented as a conventional B-tree on the sk’s value

- Sort Keys of Each Projections in WS
  - Represented as a collection of (s, sk) pairs
    - s: a sort key value
    - sk: the storage key describing where s first appears.
  - Also, this structure is represented as a conventional B-tree on the sort key fields.
5. **WS**

- Searches in WS: Two Steps
  - 1\textsuperscript{st} Step:
    - Use B-tree over the (s, sk) pairs
      - Search condition: search criteria of the query
      - Search result: corresponding storage keys
  - 2\textsuperscript{nd} Step:
    - Use the B-tree over the each column represented by (v, sk) pairs
      - Search condition: storage keys found in 1\textsuperscript{st} step
      - Search result: data values in the column.
6. Storage Management

- Issue about allocation of segments to nodes in a grid system
  - Performs automatically using a *storage allocator*.
  - Constraints on the allocations:
    - Join index should be co-located with their sender’s segment
    - Each WS segment will be co-located with the RS segment.
7. Updates and Transactions

- In C-Store, updates are represented as collection of new objects in WS
  - Each object is inserted in all projections for a given column, along with sort key structure.
  - All the inserts would have same storage keys, as inserts belong to same record in logical table.
- Issues of synchronizing the storage key, among multiple sites in grid environment
  - To avoid, each site maintains a locally unique counter, to which they append their local site id, thus making it unique.
  - WS is on the top of BERKELEY DB.
    - Leverages B-tree structure, to support their own data structures.
      - Thus, every insert would result in physical inserts on multiple disk pages.
      - This can lead to poor performance for write.
  - Thought make use of large in-memory buffer pool, to have “hot” data structures resident.
7. Updates and Transactions

- Snapshot Isolation
  - C-store expects large number of ad-hoc natured queries and OLTP transactions.
  - If C-Store uses conventional locking, substantial lock contention would likely be observed.
  - Thus, C-Store isolated the read-only transactions using *Snapshot Isolations (SI)*.
  - It allows read-only transactions to access the database as of sometime in the recent past before which we can give guarantee of no uncommitted transactions.
- High Water Mark (HWM): The most recent time, at which the SI can run.
- Low Water Mark (LWM): The most earliest time at which a read-only transaction can run.
7. Updates and Transactions

- Providing Snapshot Isolation
  - Key Problem: “Which records to make visible from WS and RS to read-only transaction at effective time ET”.
    - Make use of epochs as the unit of timestamps.
  - Insertion Vector (IV): Contains epoch for each record in each column of every projection, to describe its time of insert.
  - Deleted Record Vector (DRV): Contains epoch for each record in each column of every projection, to describe its time of delete.
  - The runtime system consults IV and DRV to make the visibility calculation for each query on a record-by-record basis.
7. Updates and Transactions

- Maintaining HVM
  - Designated site called “timestamp authority”.
    - Responsibility of allocating timestamps to other sites.
    - Idea is to divide the time into number of epochs.
      - Epoch number is the number of epochs that have elapsed since the beginning of time.
7. Updates and Transactions

- Maintaining HVM
7. Updates and Transactions

- Locking-based Concurrency Control
  - NO-FORCE STEAL POLICY
    - NO-FORCE: Database system doesn’t forces all the updates of a transaction to disk before committing
    - STEAL: Buffer manager can steal memory pages from the database.
  - Policy differs in
    - By just logging UNDO records
    - Doesn’t use strict two-phase commit, avoiding the PREPARE phase.
  - Locking deadlocks are resolved via timeouts
7. Updates and Transactions

- Distributed Commit Processing

Transaction received by Master

- Assigning Unit of work
- Determining ultimate commit state, by waiting for all the workers to complete outstanding work and issue commit or abort message.

On receiving commit message, they release the locks
Remove undo logs
7. Updates and Transactions

- Three types of recovery
  - No data loss
    - Simply run the queries that are queued in the network for the same site.
  - Catastrophic failure (WS and RS both are destroyed)
    - Construct both the segments from other projections and join indexes
  - WS is damaged and RS in intact
    - Copy the RS segment to WS segment, with the help of tuple mover’s log to fill up IV and DVR data structure.
7. Updates and Transactions

- Efficient recovery of the WS (Recovery from same site)
  - Queries other projections in the same site, to inspect if we have columns that covers the same key range

```
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
```
7. Updates and Transactions

- Efficient recovery of the WS
  - Queries other projections in the same site, to inspect if we have columns that covers the same key range
  - To recover the WS from remote sites
    - Force tuple mover to log, before the tuples are moved from WS to RS
      - For each tuple, its storage key and epoch number.
  - Truncate the log to timestamp of the oldest tuple still in the WS on any site.
    - Any recovering site can use a remote WS segment S and the log to solve the query.
7. Updates and Transactions

- Tuple Mover
  - Job: Move blocks of tuples in a WS segment to corresponding RS Segment.
    - To support high-speed tuple mover, used LSM-tree concept, which supports merge-out process (MOP)
      - Helps in moving ordered WS data objects to large RS blocks efficiently.
  - Updating join indexes
  - MOP will find all the records in the WS segment, with insertion time at or before LWM, and divide them into two groups
    - Once deleted at or before LWM
    - Once not deleted or deleted after LVM
  - MOP will create RS'.
    - It reads blocks from the columns of the RS segments
      - Deletes the RS items with a value in the DRV less than or equal to LWM.
      - Merges in new column values from WS.
      - The merged data is then written out to the new RS'
8. C-Store Query Execution and Optimization

- Query Execution
  - The execution plan that optimizer outputs, consist of various executing nodes, which accepts operands or produces results of type projections, column or bitstrings
    - Decompress
    - Select
    - Mask
    - Project
    - Sort
    - Aggregation Operators
    - Concat
    - Permute
    - Join
    - Bitstring operator.
8. C-Store Query Execution and Optimization

- Query Optimization
  - It differs from the traditional query optimization because,
    - Need to consider compressed representation
    - Decision about when to mask a projection using a bitstring.
    - Which available projection best fits in the given query’s search criteria.
  - C-Store operators have capability to operate on compressed and uncompressed data,
    - But the cost of execution is dependent on type of compression.
  - Thus cost model should be sensitive to representation of input and output columns
9. C-Store Performance Comparison

- Benchmarking System: 3.0 Ghz Pentium
  2 GB RAM and 750 GB of Disk

- Standard data for TPC-H scale for 1.8 GB of data.

- Tested against three systems, each having 2.7 GBs of storage budget.
  - Among three systems,
    - C-Store
    - Commercial Row Store (Turned off locking and logging)
    - Commercial Column Store (Turned off locking and logging)
9. C-Store Performance Comparison

- Benchmarking System: 3.0 Ghz Pentium
  2 GB RAM and 750 GB of Disk

- Standard data for TPC-H scale for 1.8 GB of data.

- Tested against three systems, each having 2.7 GBs of storage budget.
  - Among three systems,
    - C-Store
    - Commercial Row Store (Turned off locking and logging)
    - Commercial Column Store (Turned off locking and logging)

<table>
<thead>
<tr>
<th>C-Store</th>
<th>Row Store</th>
<th>Column Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.987 GB</td>
<td>4.480 GB</td>
<td>2.650 GB</td>
</tr>
</tbody>
</table>
9. C-Store Performance Comparison

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
      c_custkey = c_nationkey AND
      l_returnflag = 'R'
GROUP BY c_nationkey
9. C-Store Performance Comparison

Reasons for faster performance
- Column representation
- Storing overlapping projections
- Better compression of data
- Query Operators

<table>
<thead>
<tr>
<th>Query</th>
<th>C-Store</th>
<th>Row Store</th>
<th>Column Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>0.03</td>
<td>6.80</td>
<td>2.24</td>
</tr>
<tr>
<td>Q2</td>
<td>0.36</td>
<td>1.09</td>
<td>0.83</td>
</tr>
<tr>
<td>Q3</td>
<td>4.90</td>
<td>93.26</td>
<td>29.54</td>
</tr>
<tr>
<td>Q4</td>
<td>2.09</td>
<td>722.90</td>
<td>22.23</td>
</tr>
<tr>
<td>Q5</td>
<td>0.31</td>
<td>116.56</td>
<td>0.93</td>
</tr>
<tr>
<td>Q6</td>
<td>8.50</td>
<td>652.90</td>
<td>32.83</td>
</tr>
<tr>
<td>Q7</td>
<td>2.54</td>
<td>265.80</td>
<td>33.24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query</th>
<th>C-Store</th>
<th>Row Store</th>
<th>Column Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>0.03</td>
<td>0.22</td>
<td>2.34</td>
</tr>
<tr>
<td>Q2</td>
<td>0.36</td>
<td>0.81</td>
<td>0.83</td>
</tr>
<tr>
<td>Q3</td>
<td>4.90</td>
<td>49.38</td>
<td>29.10</td>
</tr>
<tr>
<td>Q4</td>
<td>2.09</td>
<td>21.76</td>
<td>22.23</td>
</tr>
<tr>
<td>Q5</td>
<td>0.31</td>
<td>0.70</td>
<td>0.63</td>
</tr>
<tr>
<td>Q6</td>
<td>8.50</td>
<td>47.38</td>
<td>25.46</td>
</tr>
<tr>
<td>Q7</td>
<td>2.54</td>
<td>18.47</td>
<td>6.28</td>
</tr>
</tbody>
</table>
10. Conclusion

- C-Store, a radical departure from the current DBMSs.

- Innovative features of C-Store are
  - Hybrid Architecture
  - Focus on economizing the storage representation on disk.
    - Coding data values and dense-packing the data.
  - Overlapping projections of logical table.
  - Shared nothing machine environment.
  - Efficient snapshot isolation.
  - Distributed transaction without a redo log or two phase commits.
10. Reflection

- Visibility of records calculated record by record and per query.
- No variation in the query set used for performance comparison.
- Results seem half-promising to me as not tested in full environment test bud.
- Would WS as row-store and RS as column-store give more optimistic results?
- Weak Coherence in the distribution of the content.