C-store: A Column-Oriented DBMS

Authors: Stonebraker et. al.

Michael Ibrahim, Sam Lefforge, Amogh Mantri, Yiling Wu, Lixiaotian Zong - Group 4

Introduction and Background: Comparisons

- Traditional DBMS (Row Store)
  - Write-optimized
  - Continuous storage of rows/tuples
  - Efficient for Online Transaction Processing applications
- **C-Store** DBMS (Column Store)
  - Read-optimized
  - Continuous storage of columns
  - Made for querying of large datasets
  - Better for high-read, low-write applications

Introduction and Background: Design Principles

- Takes advantage of balance between CPU and Disk
  - Modern CPU speed growth has outpaced disk bandwidth growth
  - More CPU cycles are better than more disk usage
- Only reads columns required for query processing
  - Irrelevant attributes are ignored
- Storage Optimization
  - Dense packing of values
  - Storage of data in compact format

Introduction and Background: Innovation

- Has a hybrid architecture
  - Writeable Store
  - Readable Store
  - Connected by Tuple Mover
- Stores Overlapping projections
  - Multiple sort orders of same column
  - Uses bitmap indexes and materialized views
- Deployed to grid computing environment (distributed)
  - K-Safety achieved

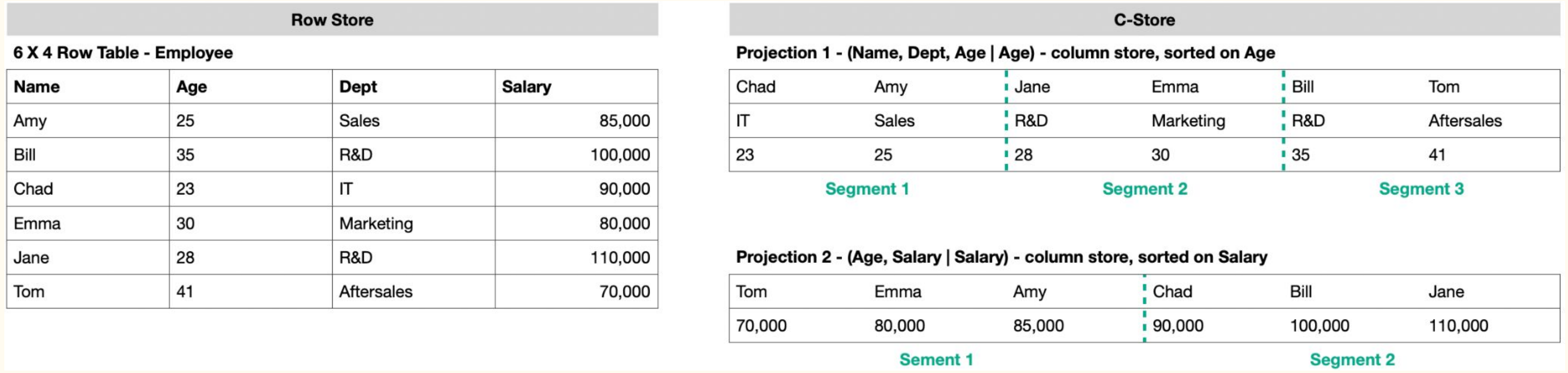
Yiling Wu

Data Model

How is data stored in C-Store DBMS?

- **Projection:** a projection is a data structure including columns sorted on same attribute -> fewer column & compression -> less space

- **Segment:** for a specific projection, a segment is a partition of the projection based on the value of the sorted attribute (image how B Tree works on the indexed column in Row Table) -> faster lookup



Row Store

6 X 4 Row Table - Employee

Name	Age	Dept	Salary
Amy	25	Sales	85,000
Bill	35	R&D	100,000
Chad	23	IT	90,000
Emma	30	Marketing	80,000
Jane	28	R&D	110,000
Tom	41	Aftersales	70,000

C-Store

Projection 1 - (Name, Dept, Age | Age) - column store, sorted on Age

Chad	Amy	Jane	Emma	Bill	Tom
IT	Sales	R&D	Marketing	R&D	Aftersales
23	25	28	30	35	41

Segment 1 Segment 2 Segment 3

Projection 2 - (Age, Salary | Salary) - column store, sorted on Salary

Tom	Emma	Amy	Chad	Bill	Jane
70,000	80,000	85,000	90,000	100,000	110,000

Sement 1 Segment 2

Figure 2.1 - Example of transforming a 6 X 4 employee table to 2 projections

Data Model

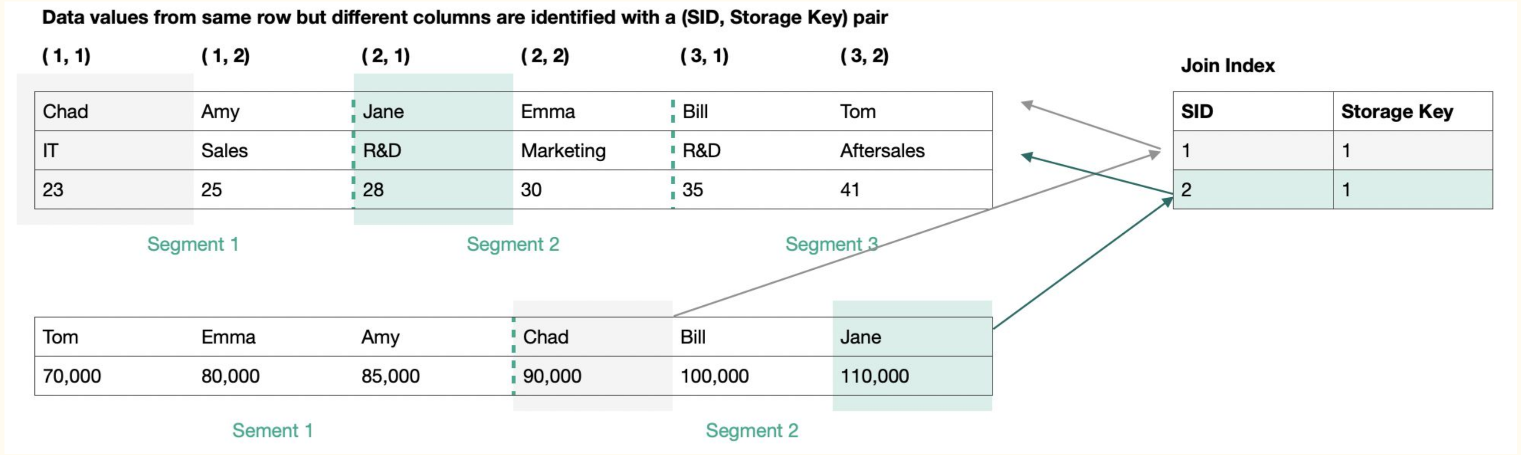
- One table can have multiple projections.
- One projection can include multiple columns, even if columns from another table as long as the join is a n:1 relationship (i.e. not causing extra rows to the projection).
- Every column should appear in at least one projection.

How to reconstruct the original table for random queries?

Data Model

How to reconstruct the original table?

- **Storage Key:** an integer indicating the relative position of data values in a segment
- **Join Index:** a mapping relationship (in the format of two columns, SID & Storage Key) indicating how entries in one projection map with those in another projection to construct a whole row



Data values from same row but different columns are identified with a (SID, Storage Key) pair

(1, 1)	(1, 2)	(2, 1)	(2, 2)	(3, 1)	(3, 2)
Chad	Amy	Jane	Emma	Bill	Tom
IT	Sales	R&D	Marketing	R&D	Aftersales
23	25	28	30	35	41

Join Index

SID	Storage Key
1	1
2	1

Tom	Emma	Amy	Chad	Bill	Jane
70,000	80,000	85,000	90,000	100,000	110,000

Figure 2.2 - Example showing reconstruction of original employee table

Data Model

- Every column is expected to be stored in multiple projection

  -> Expensive to store and maintain Join Indexes

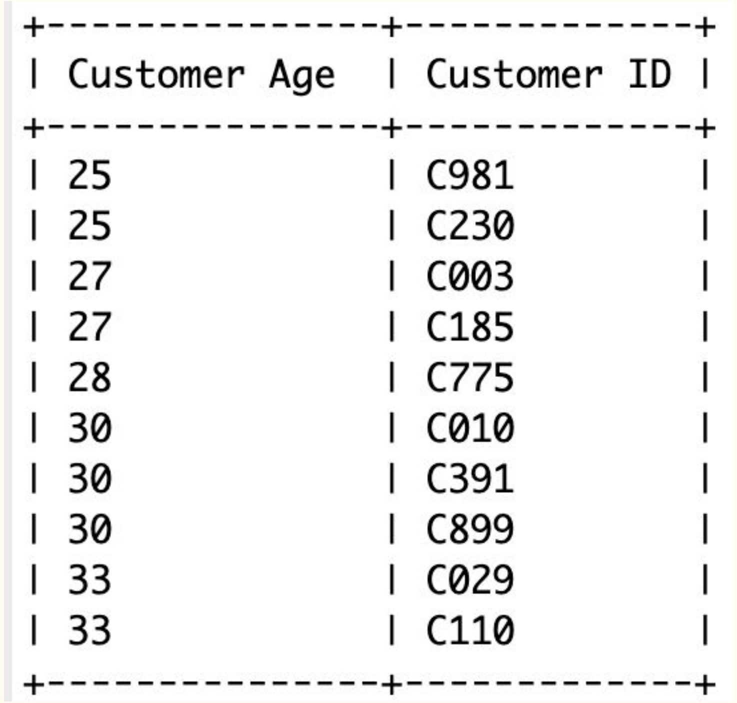
  -> Failure Tolerance (**K-safe**: able to reconstruct the original table even with failure in K nodes.)

The design of a C-Store physical DBMS involves the design of:

Projections / Segments / Storage Key / Join Index

Read-Optimized Stores: Encoding Schemes

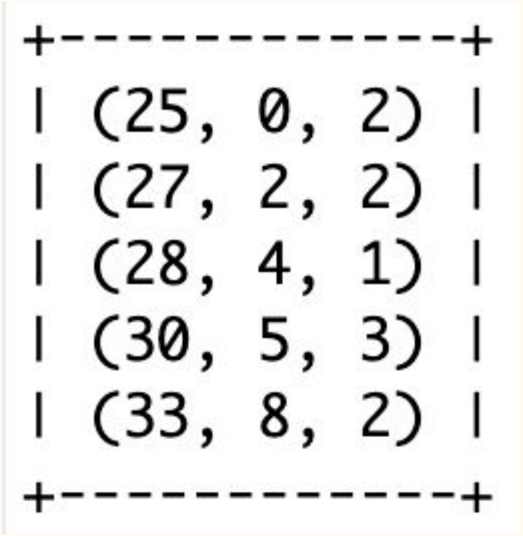
- Data in the Read-Store is encoded
- Tailoring to properties of the data
  - Self-order vs foreign-order
  - High proportion of distinct values vs. low proportion of distinct values



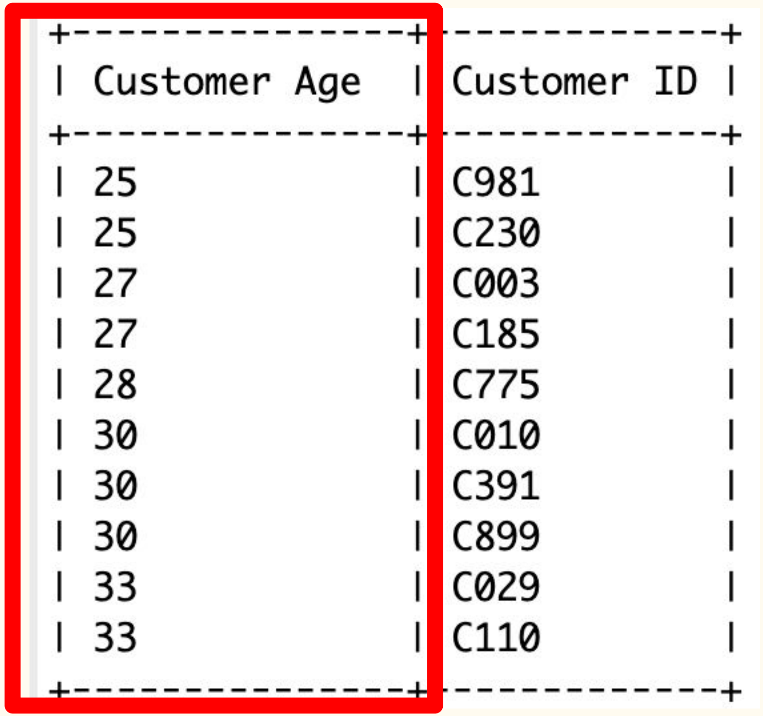
```
+---------------+-------------+
| Customer Age  | Customer ID |
+---------------+-------------+
| 25            | C981        |
| 25            | C230        |
| 27            | C003        |
| 27            | C185        |
| 28            | C775        |
| 30            | C010        |
| 30            | C391        |
| 30            | C899        |
| 33            | C029        |
| 33            | C110        |
+---------------+-------------+
```

Type 1: Self-order, few distinct values

- (v, f, n) triples
  - v = value
  - f = index
  - n = quantity



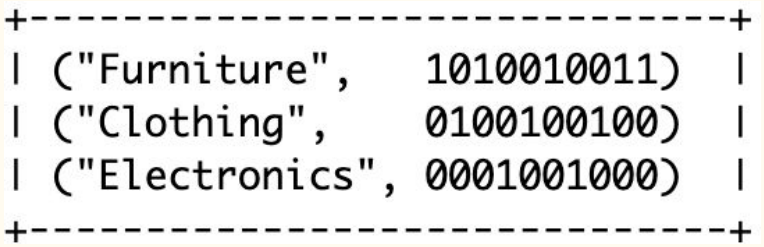
```
+-------------+
| (25, 0, 2) |
| (27, 2, 2) |
| (28, 4, 1) |
| (30, 5, 3) |
| (33, 8, 2) |
+-------------+
```



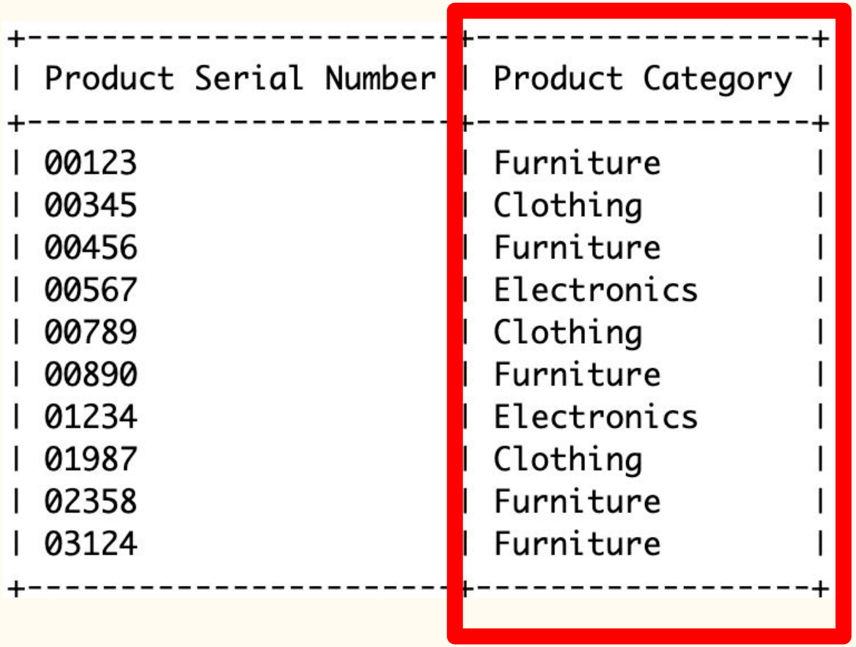
```
+----------------+-------------+
| Customer Age   | Customer ID |
+----------------+-------------+
| 25             | C981        |
| 25             | C230        |
| 27             | C003        |
| 27             | C185        |
| 28             | C775        |
| 30             | C010        |
| 30             | C391        |
| 30             | C899        |
| 33             | C029        |
| 33             | C110        |
+----------------+-------------+
```

Type 2: Foreign-order, few distinct values

- (v, b) tuples
  - v = value
  - b = bitmap



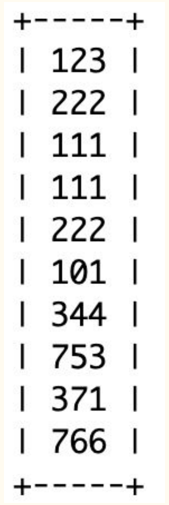
```
+-----------------------------------+
| ("Furniture",    1010010011)  |
| ("Clothing",     0100100100)  |
| ("Electronics", 0001001000)  |
+-----------------------------------+
```



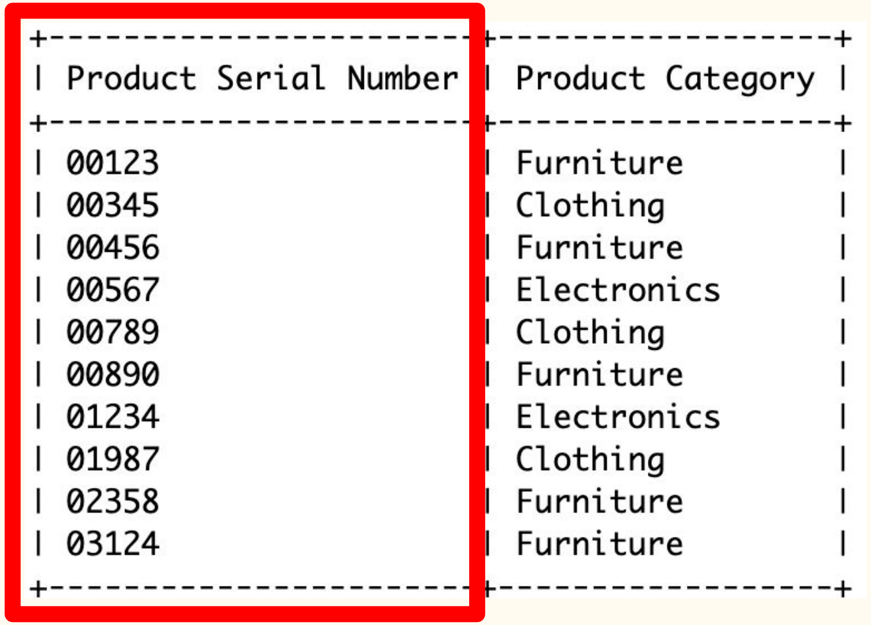
```
+-------------------------+------------------+
| Product Serial Number | Product Category |
+-------------------------+------------------+
| 00123                 | Furniture        |
| 00345                 | Clothing         |
| 00456                 | Furniture        |
| 00567                 | Electronics      |
| 00789                 | Clothing         |
| 00890                 | Furniture        |
| 01234                 | Electronics      |
| 01987                 | Clothing         |
| 02358                 | Furniture        |
| 03124                 | Furniture        |
+-------------------------+------------------+
```

Type 3: Self-order, many distinct values

- Represent every value as the delta of the previous value



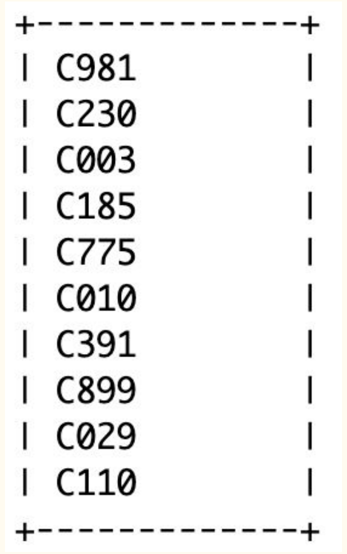
```
+-----+
| 123 |
| 222 |
| 111 |
| 111 |
| 222 |
| 101 |
| 344 |
| 753 |
| 371 |
| 766 |
+-----+
```



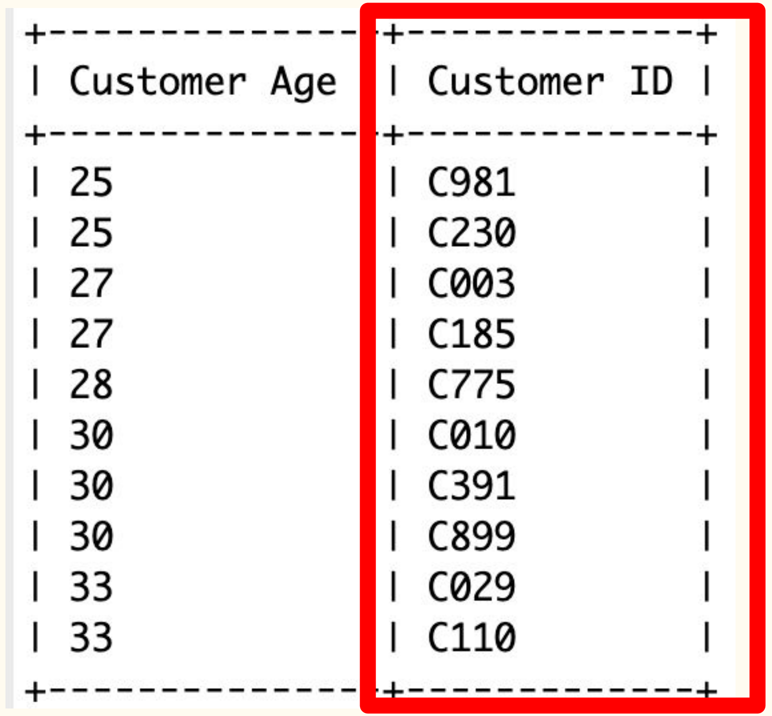
```
+-----------------------+------------------+
| Product Serial Number | Product Category |
+-----------------------+------------------+
| 00123                 | Furniture        |
| 00345                 | Clothing         |
| 00456                 | Furniture        |
| 00567                 | Electronics      |
| 00789                 | Clothing         |
| 00890                 | Furniture        |
| 01234                 | Electronics      |
| 01987                 | Clothing         |
| 02358                 | Furniture        |
| 03124                 | Furniture        |
+-----------------------+------------------+
```

Type 4: Foreign-order, many distinct values

- Leave unencoded



C981
C230
C003
C185
C775
C010
C391
C899
C029
C110

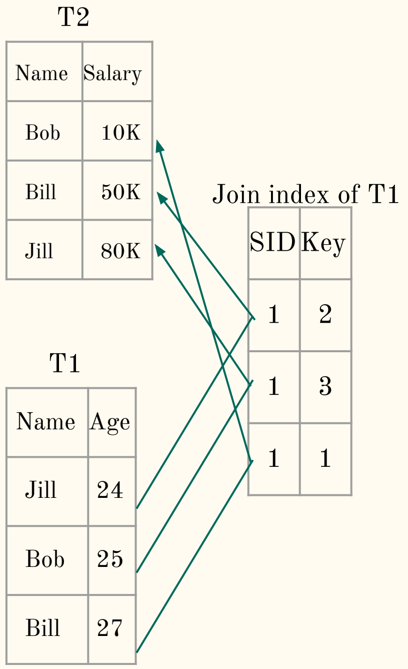


Customer Age	Customer ID
25	C981
25	C230
27	C003
27	C185
28	C775
30	C010
30	C391
30	C899
33	C029
33	C110

Lixiaotian Zong

Read-Optimized Stores: Join Indexes

- Linking Projections: If there are multiple projections (say, T1 and T2) that cover the same table, a join index enables linking these projections by providing a mapping.
- Structure of Join Indexes: The join index contains entries that map tuples between segments in different projections, with each entry containing a segment ID and a storage key pointing to the corresponding tuple in the other projection.
- The path to reconstruct table: A path is a collection of join indexes originating with a sort order specified by some projection(Ti). Path passes through zero or more intermediate join indices and ends with a projection sorted in Ti order.
- Efficiency and Maintenance: Although join indexes facilitate efficient querying, they are costly to store and update, especially when projections are modified.
- Fault Tolerance (K-Safety): C-Store allows for K-safe configurations, meaning the database can withstand the loss of up to K nodes and still reconstruct tables through join indexes and projections, which ensures resilience to node failures.



Write-Optimized Stores: General Structure

- Staging area for temporarily storing dynamic data
  - Efficiently handle inserts and updates without sacrificing compression or performance in RS
  - Batch inserts of stable data from WS to RS via Tuple Mover
- Column store
  - Not as efficient as row store for OLTP
  - Avoids the need for a different optimizer (since RS is already a column store)
  - Same projections and join indexes as RS
  - Same horizontal partitioning as RS (there is 1:1 mapping between RS and WS segments)
- Unique storage key is explicitly stored for each new tuple
  - An integer larger than the number of records the largest segment in RS
  - Included in each projection that contains data from the tuple
- Trivial size compared to RS → no compression needed!

Write-Optimized Stores: Data Storage

- Every column in a projection is represented as a (value, storage key) pair
  - Indexed via a B-tree (1) on the storage key
- Each projection is represented as a (sort key, storage key) pair
  - Indexed via a B-tree (2) on the sort key
- Performing Searches via sort key:
  - Obtain appropriate storage key via B-tree (2)
  - Obtain value from storage key via B-tree (1)
- Join indices
  - Recall: 1:1 mapping between WS and RS segments
  - A join index from a "sender" contains only (segment ID, storage key) of record in "receiver"
  - Join index shares segmentation structure and location of "sender" projection
  - No need for distinction between RS and WS

Storage Management

- Storage management issue
  - Allocation of segments to nodes in a grid
- C-Store will handle storage management automatically
  - Storage Allocator
- Storage Allocator
  - Performs initial allocation
  - Handles reallocation when load is unbalanced
  - Addressed in further detail in future work by authors
- Large columns (MBs) are stored in individual files
  - Lack of overhead in modern file systems allows avoiding raw device storage

Updates & Transactions: Providing Snapshot Isolation

- Read-only transactions utilize snapshot isolation
  - Access database at some time in recent past
  - Most recent time that can be accessed is High Water Mark (HWM)
  - Oldest time that can be accessed is Low Water Mark (LWM)
- Maintaining the HWM:
  - Time is divided into epochs (can be multiple seconds long)
  - Initial HWM is epoch 0, first current epoch is 1
  - A single site is assigned time authority (TA)
  - Periodically, TA sends end of epoch e message to all sites
  - Sites reply with epoch complete once they finish all current transaction
  - HWM is moved to epoch e

Updates & Transactions: Locking-Based Concurrency Control

1. Distributed COMMIT Processing

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.

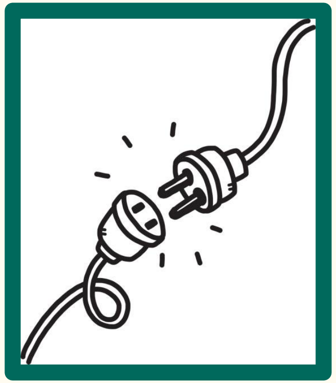
When a COMMIT command is received for a transaction, the master waits until all assigned sites (workers) have completed their designated actions. Only then does it send a commit or abort message to each site.

If a site crashes after receiving a commit directive, the site will reconstruct its state based on updates from other sites in the system, relying on distributed projections to ensure consistency and data integrity.

Updates & Transactions: Recovery

Recovery scenarios:

- No data loss
  - e.g. temporary power outage
  - Replay pending updates
- Catastrophic failure
  - e.g. fire or explosion
  - RS and WS destroyed
  - Reconstruct with data from remote sites
  - Replay pending updates
- Partial damage
  - e.g. system crash during write to WS
  - WS damaged, RS intact
  - Discussed in detail



Efficiently Recovering the WS

- Identify data to be recovered
- Search for available projection segments of lost data
- Ensure segments are stable
- Query and extract necessary data from these segments
- Check if any data has been moved to RS
    - If none has, then the WS has successfully been restored
    - If some has, use tuple mover logs to recover such data
- Once WS is restored, replay updates to align with the current state

Tuple Mover: What Is It?

- Background process that moves tuples from a Writeable Store(WS) segment to a Readable Store(RS) segment
- Bridge between the WS and RS components of C-Store
- Finds and moves **worthy** segment pairs
  - Many records in WS with insertion times $\leq$ LWM
- Performs merge-out process (**MOP**) to transfer data
- Uses LWM and HWM to make decisions on what data must be moved

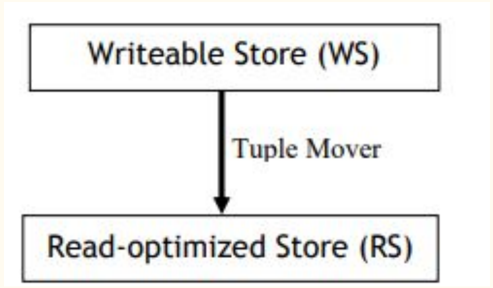


Figure 1. Architecture of C-Store

Amogh Mantri

Tuple Mover: MOP

- First creates new RS segment: RS′ for merged data
- Identifies WS records with insertion times $\leq$ LWM
- Checks:
  - Records deleted before/at LWM $\rightarrow$ discard
  - Records not deleted or deleted after LWM $\rightarrow$ transfer
- Updates join indexes for new storage keys
- Maintains Deleted Record Vector
- Reads and merges blocks from RS columns
- Finally cuts over from RS to RS′

C-Store Query Execution

- ## Query Operators and Plan Format

10 node types : Decompress, Select, Mask, Project, Sort, Aggregation, Concat, Permute, Join, Bitstring Operators (BAnd, BOr, BNot)

Each accepts operands or produces results of type projection (Proj), column(Col), or bitstring (Bits). In addition, C-Store query operators accept predicates (Pred), join indexes (JI), attribute names(Att), and expressions (Exp) as argument.

- ## Query Optimization

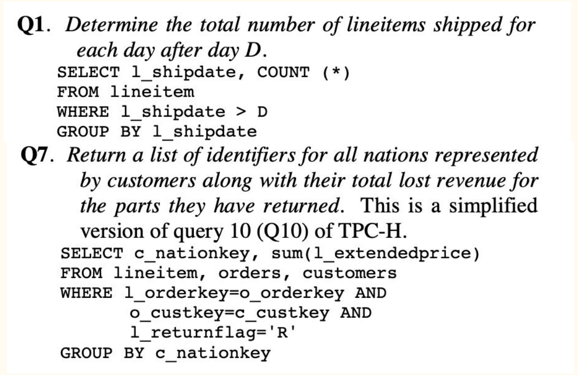
C-Store operators have the capability to operate on both compressed and uncompressed input. An operator's execution cost (both in terms of I/O and memory buffer requirements) is dependent on the compression type of the input. Thus, the cost model is sensitive to the representations of input and output columns.

The major optimizer decision is which set of projections to use for a given query. The optimizer must decide where in the plan to mask a projection according to a bitstring.

Yiling Wu

Performance Comparison

- Tested on **read-only queries** (a subset from the TPC-H benchmark) on a single-site storage setting.
- Queries performed on **C-Store / Row Store / Column Store** designs. Projections/Schemas designed in a way that is believed to achieve the best possible performance.
- **Benchmarking system**: 3.0 Ghz Pentium, RedHat Linux, 2GB Memory, 750 GB disk



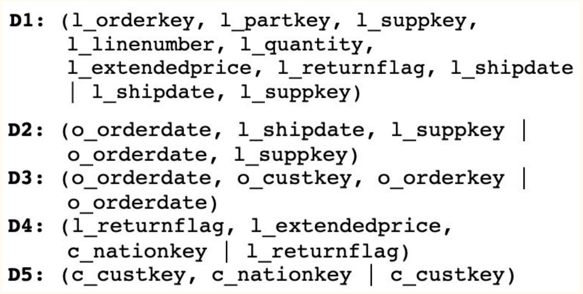
Q1. *Determine the total number of lineitems shipped for each day after day D.*

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

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
```

Figure 9.1 - Sample Queries



```
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)
```

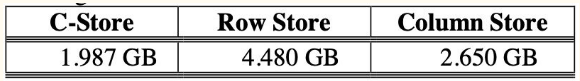
Figure 9.2 - Projection Design

Performance Comparison

- **Space Usage Constrained (2.7GB)**

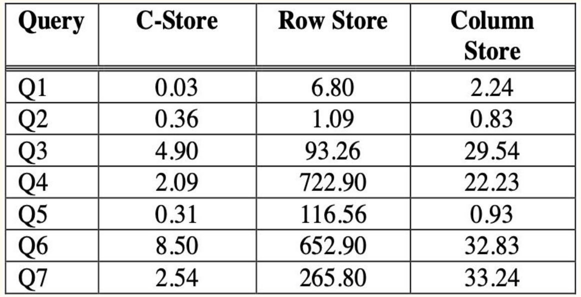
Compared to Row Store: C-Store is **164 times faster**, row-store UNABLE to operate within constraint

Compared to Column Store: C-Store is **21 times faster**



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

Figure 9.3 - Space Usage with constraint



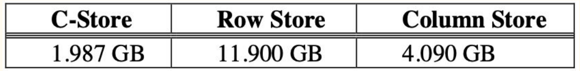
Query	C-Store	Row Store	Column Store
Q1	0.03	6.80	2.24
Q2	0.36	1.09	0.83
Q3	4.90	93.26	29.54
Q4	2.09	722.90	22.23
Q5	0.31	116.56	0.93
Q6	8.50	652.90	32.83
Q7	2.54	265.80	33.24

Figure 9.4 - Time (in seconds) with space constraint

- **Lifting Space Usage Constraint** with materialized views

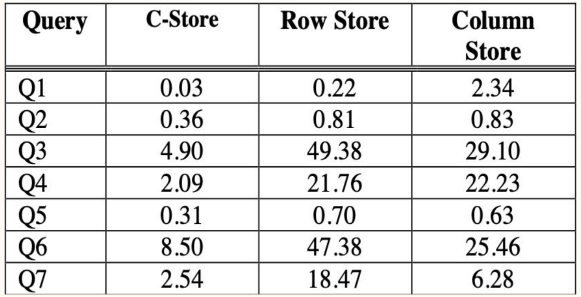
Compared to Row Store: C-Store is **6.4 times faster,** but row-store takes 6 times the space

Compared to Column Store: C-Store is **16.5 times faster**, but column-store requires 1.83 times the space



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

Figure 9.5 - Space Usage without constraint



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

Figure 9.6 - Time (in seconds) without space constraint

Related Work

- Maintaining data cubes:
  - Slicing and dicing datasets: (Gray et. al, 1997)
  - Building and maintaining aggregates of large datasets: (Kotidis and Roussopoulos, 1999), (Zhao et al., 1997)
  - Precomputation for regular queries: (Staudt and Jarke, 1996)
- Two different DBMS in one system:
  - Data mirrors for improved query performance: (Ramamurthy et al., 2002)
- Column stores:
  - Entry sequence-based: (Copeland et al., 1988), (Boncz et al., 2004)
- Data compression:
  - Review of techniques: (Roth and Van Horn, 1993)
  - Direct operation on compressed data: (Graefe, 1993), (Westmann et al., 2000)

Conclusions

The paper introduced the design of "C-Store", which focuses on "**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 **encoding** 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**.

References

- (Gray et. al, 1997): Gray et al. DataCube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. Data Mining and Knowledge Discovery, 1(1), 1997.
- (Kotidis and Roussopoulos, 1999): Yannis Kotidis, Nick Roussopoulos. DynaMat: A Dynamic View Management System for Data Warehouses. In Proceedings of SIGMOD, 1999.
- (Zhao et al., 1997): Y. Zhao, P. Deshpande, and J. Naughton. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. In Proceedings of SIGMOD, 1997. 564
- (Staudt and Jarke, 1996): Martin Staudt, Matthias Jarke. Incremental Maintenance of Externally Materialized Views. In VLDB, 1996.
- (Ramamurthy et al., 2002): Ravishankar Ramamurthy, David Dewitt. Qi Su: A Case for Fractured Mirrors. In Proceedings of VLDB, 2002.
- (Copeland et al., 1988): George Copeland et. al. Data Placement in Bubba. In Proceedings SIGMOD 1988.
- (Boncz et al., 2004): Peter Boncz et. al.. MonetDB/X100: Hyper-pipelining Query Execution. In Proceedings CIDR 2004.
- (Roth and Van Horn, 1993): Mark A. Roth, Scott J. Van Horn: Database Compression. SIGMOD Record 22(3). 1993.
- (Graefe, 1993): G. Graefe. Query Evaluation Techniques for Large Databases. Computing Surveys, 25(2), 1993.
- (Westmann et al., 2000): Paul Westerman. Data Warehousing: Using the Wal-Mart Model. Morgan-Kaufmann Publishers , 2000.

Thank you!
Any Questions?

Amogh Mantri,
Sam Lefforge

Study Questions

- How does C-Store's column-oriented storage design improve read performance and storage efficiency when compared to traditional row-oriented DBMS architectures?
- How does C-Store's hybrid concurrency control using snapshot isolation and two-phase locking benefit read-mostly workloads?