# C-STORE: A COLUMN-ORIENTED DBMS

Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, Stan Zdonik

22/6/22

CS 848 Modern Database Systems

Presenter- Amy Bhatia

UNIVERSITY OF WATERLOO | FACULTY OF MATHEMATICS

## Outline

- Introduction
- Data Model
- RS
- WS
- Storage Management
- Updates and Transactions
- Recovery
- Tuple Mover
- Query Execution
- Performance
- Conclusion

UNIVERSITY OF WATERLOO | FACULTY OF MATHEMATICS

## The proposal

<u>Row-store DBMS</u>

- Attributes of a row are placed contiguously in storage
- *Write-optimized* system
- More suitable for OLTP-style applications

<u>Column-store DBMS</u>

- Values for each attribute (column) are stored contiguously
- *Read-optimized* system
- Data warehouses, customer relationship management systems, ad-hoc inquiry systems

## Why Column-store DBMS?

- Can only bring required attribute values into memory
- Typical queries involve aggregates over important attributes
- Easy to encode together column values of the same data type
- Data can be packed densely together rather than aligned by byte or word boundaries.

## C-Store

- Stores a group of columns in the form of projections
- A distributed architecture where different projections could occupy different nodes
- Supports k-safe failure mode
- Has both a read-optimized column store and an update/insert oriented writeable store

## Architecture

- Support read-optimized workload
- Avoid locking with write based workload
- Two logical sets:
  - WS supports high performance inserts
  - RS supports high performance reads
  - Tuple mover moves updated records from WS to RS

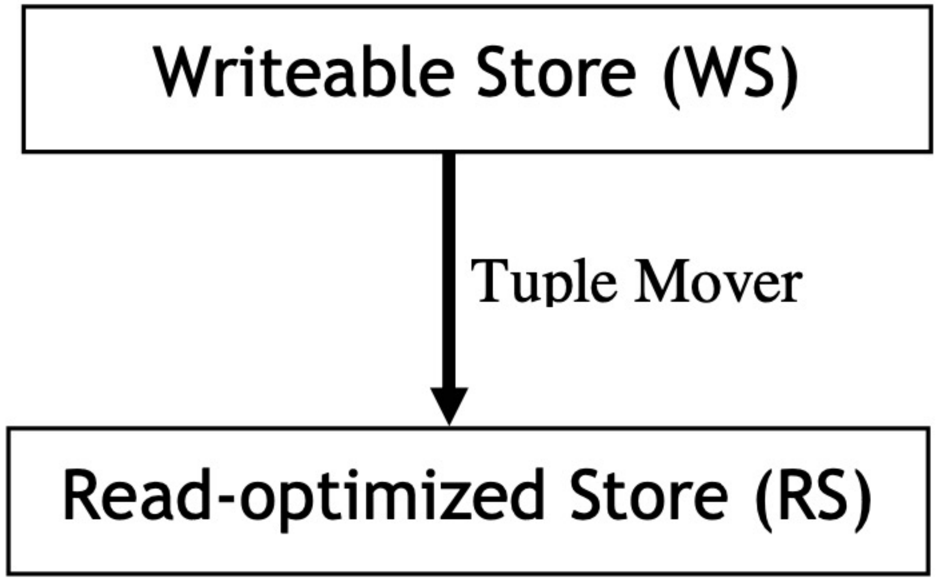


**Figure 1. Architecture of C-Store**

## Architecture

- Queries (in SQL) must access data in both storage systems.
- WS-inserts, RS-deletes
- Updates work as inserts followed by deletes
- Read only queries use snapshot isolation
- Supports large ad-hoc queries, small updates, continuous inserts

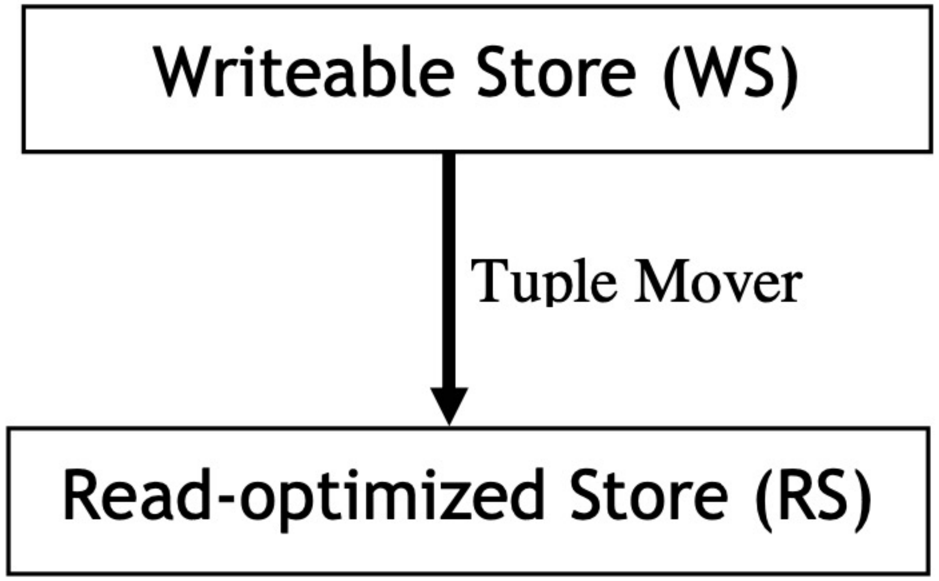
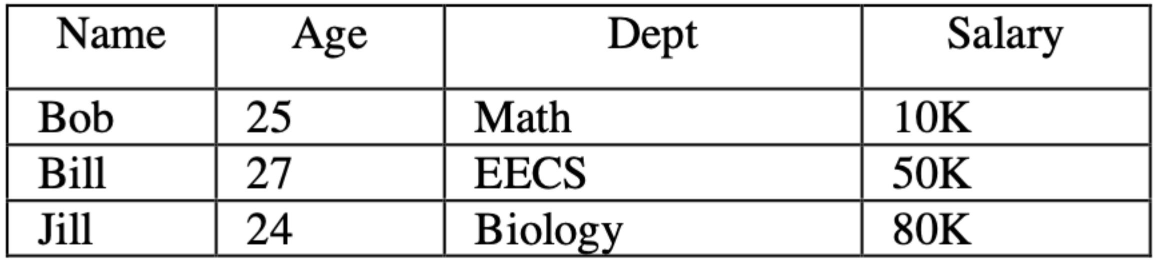


**Figure 1. Architecture of C-Store**

## Projections

- Groups of columns which are stored in physically contiguous manner
- *Anchored* on a logical table **T**
- Can also contain columns from other tables that are related to **T** through foreign keys, etc.



| Name | Age | Dept | Salary |
|---|---|---|---|
| Bob | 25 | Math | 10K |
| Bill | 27 | EECS | 50K |
| Jill | 24 | Biology | 80K |

**Table 1: Sample EMP data**

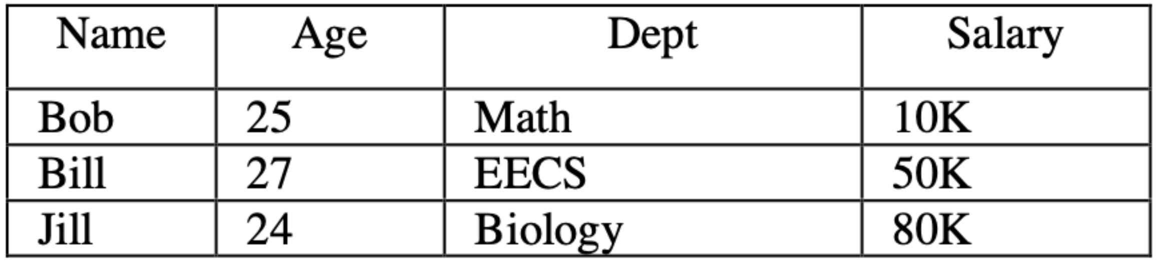
Another table DEPT(dname, floor)

*Set of possible projections:*

EMP1 (name, age)
EMP2 (dept, age, DEPT.floor)
EMP3 (name, salary)
DEPT1 (dname, floor)

## Sort Key

- Tuples in a projection are stored column-wise
- K attributes will be stored in K-data structures
- All are sorted on a *sort key* which is a column or a set of columns in the projection



| Name | Age | Dept | Salary |
|---|---|---|---|
| Bob | 25 | Math | 10K |
| Bill | 27 | EECS | 50K |
| Jill | 24 | Biology | 80K |

**Table 1: Sample EMP data**

Another table DEPT(dname, floor)

*Set of possible projections with sort order:*

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

## Segments and Storage Keys

### Segments

- Every projection is divided into segments
- Value-based partitioning on the sort key
- Each has a segment identifier **Sid**>0

### Storage Keys

- In each segment, different column values belonging to the same record are identified by **SK**
- In RS- Inferred from tuple's physical position
- In WS- Integers largest than the largest integer storage key for any segment in RS

## Join Indices

- Used to reconstruct a table from its projections

Table *T* has projections *T1, T2*.

A tuple in a segment of T1 will have corresponding join index ***(s,k)***

s: SID in T2

k: Storage key of corresponding tuple in T2

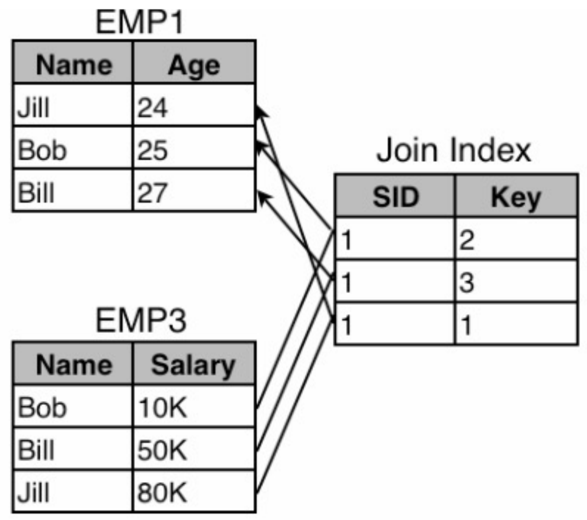


**Figure 2: A join index from EMP3 to EMP1.**

Maintaining join index is **expensive**. Any update to a projection leads to an update of the incoming or outgoing join index. Therefore, each column is stored in several projections.

## RS encoding schemes

*1. Self-order, few distinct values:*

**(v,f,n)**

1,1,1,1,3,3,3,3,3,3,3,5,5,5

(1, 1,4), (3,5,7),(5,12,3)

*2. Foreign-order, few distinct values:*

**(v,b)**

0,0,1,1,2,1,0,2,1

(0, 110000100), (1, 001101001), (2, 000010010)

*3. Self-order, many distinct values:*

1,4,7,7,8,12

(1,3,3,0,1,4)

*4. Foreign-order, few distinct values:*

leave it unencoded

## WS

- Similar physical structure as RS to avoid writing query optimizers differently
- The difference being columns are not encoded since it is assumed that WS is trivial in size as compared to RS
- This also ensures efficient updates
- Sid and SK identify the same tuple in RS and WS
- Every column represented as *(v, sk)* and a B-tree is built for *sk*
- Sort key *s* represented as *(s, sk)*

## Storage Management

- A *storage allocator* will allocate segments to nodes in a grid system
- Co-locate the following
  - same columns in the projection
  - same segments of RS and WS
  - the sender segment like EMP3 and the join index as was shown in Figure 2

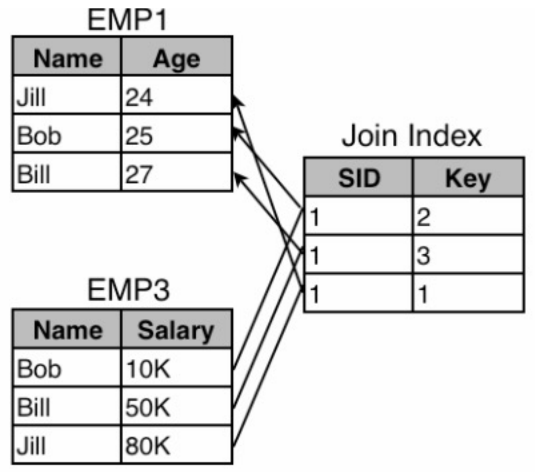


**Figure 2: A join index from EMP3 to EMP1.**

## Updates and Transactions

- Inserts (WS)- updating all the columns in the given projection and also the sorted-key column.
- Node is responsible for creating a new storage key for this entry.
- The unique storage key= node_id + local_counter. This local_counter is unique and greater than the largest key in RS to ensure a unique value across all nodes.
- Locking is minimized using *snapshot isolation*

## Snapshot Isolation

- Every projection segment in WS has Insertion Vector (IV)
- IV has epochs or time-ranges at which records are inserted
- Every projection has Deleted Record Vector (DRV) which contains the deletion status and epoch for each record
- High watermark (HWM) denotes the time at which snapshot isolation ran
- Low watermark (LWM) indicates the latest time at which a read-only query can be run on the snapshot

## Snapshot Isolation

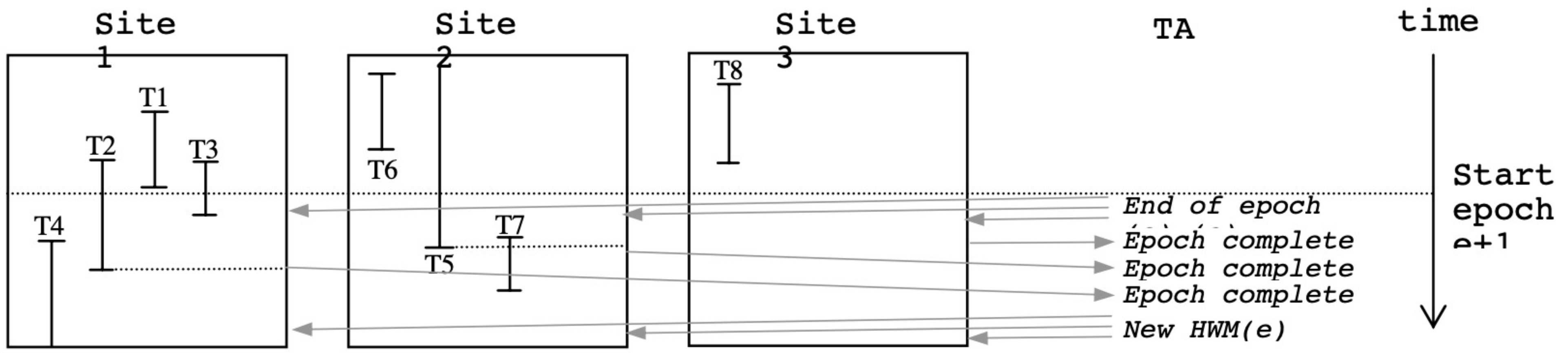


**Figure 3. Illustration showing how the HWM selection algorithm works. Gray arrows indicate messages from the TA to the sites or vice versa. We can begin reading tuples with timestamp *e* when all transactions from epoch *e* have committed. Note that although T4 is still executing when the HWM is incremented, read-only transactions will not see its updates because it is running in epoch *e*+1.**

## Read-write transactions

- Each transaction gets a master.
- Masters will split the work of the transaction and assign it to appropriate nodes
- Once all nodes finish their work, master will send a “commit” message, release logs and delete the undo log

## Recovery

- K-safety: K sites can fail in time t to still keep transactional consistency
- If failed node has no loss of data, it can be rectified by using updates that will be queued for it elsewhere in the network.
- If both RS and WS crash, reconstruct both segments from other projections and joint indexes in the system
- WS being damaged is a more common case.
  - The first step would be to check if it can be retrieved completely from other projections.
  - If the common case becomes otherwise, the authors propose to make the tuple mover log all the information it moves to RS

## Tuple Mover

- Move blocks of tuples from WS to RS segment
- The records deleted before LWM are ignored
- The rest were moved by creating a new segment in RS with the updated data and then deleting the older version of that segment
- Each tuple gets a new storage key and SID and join indices also need to be maintained

## Query Operators and Plan Format

- 10 node types with possible operators: decompress, select, mask, project, sort, aggregation operators, concat, permute, join and bitstring operators.
- Query plan has a tree of the operators with access methods at leaves and iterators serving as the interface between connected nodes.
- A non-leaf node gets data from its children through the interface

## Query Optimization

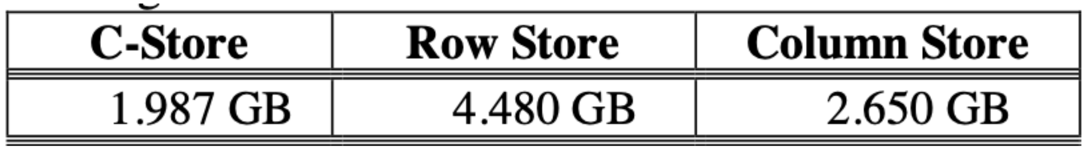
- A *Selinger-style optimiser* that uses cost-based estimation for plan construction
- Query optimization in this setting differs from traditional query optimization:
  - the need to consider compressed representations of data, and
  - the decisions about when to mask a projection using a bitstring.
- Execution cost depends on the compression type of the input
- The main optimizer decision: decide which set of projections to use for a given query

## Performance

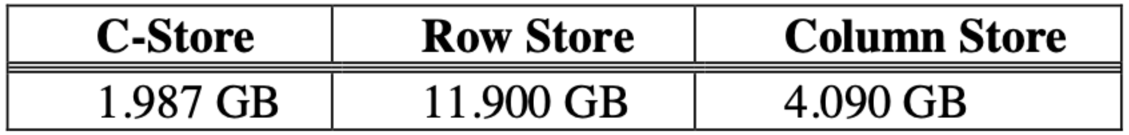
- Have implemented only single site read-only queries
- Used a simplified version of TPC-H benchmark with 7 queries
- Compared the performance of three systems each with a storage budget of 2.7GB- C-Store, a row-oriented DBMS, a-column oriented DBMS

## Performance Test Results

- Disk Usage in standard relational schemas (left) materialized schemas (right)

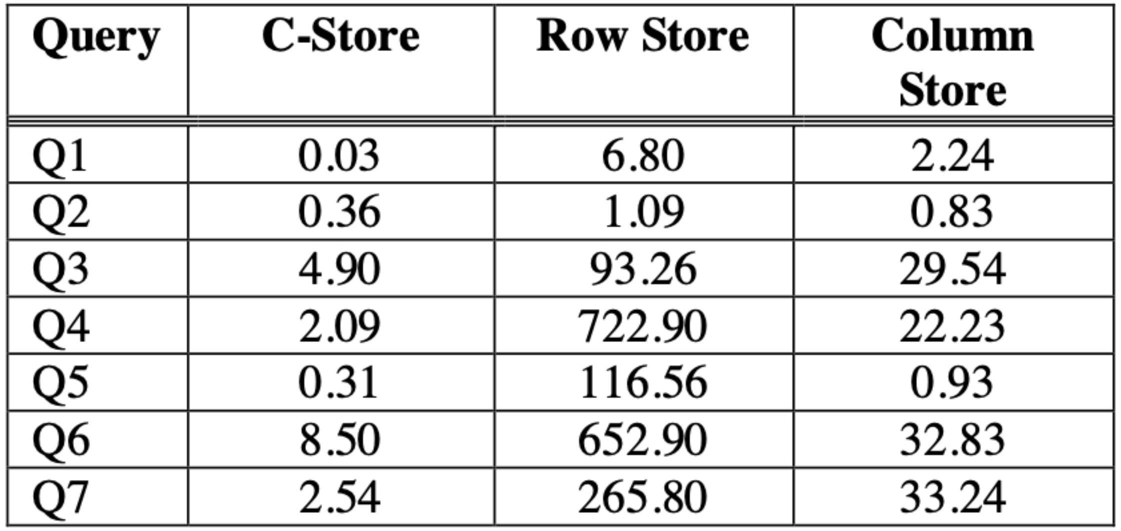


| C-Store | Row Store | Column Store |
|---|---|---|
| 1.987 GB | 4.480 GB | 2.650 GB |

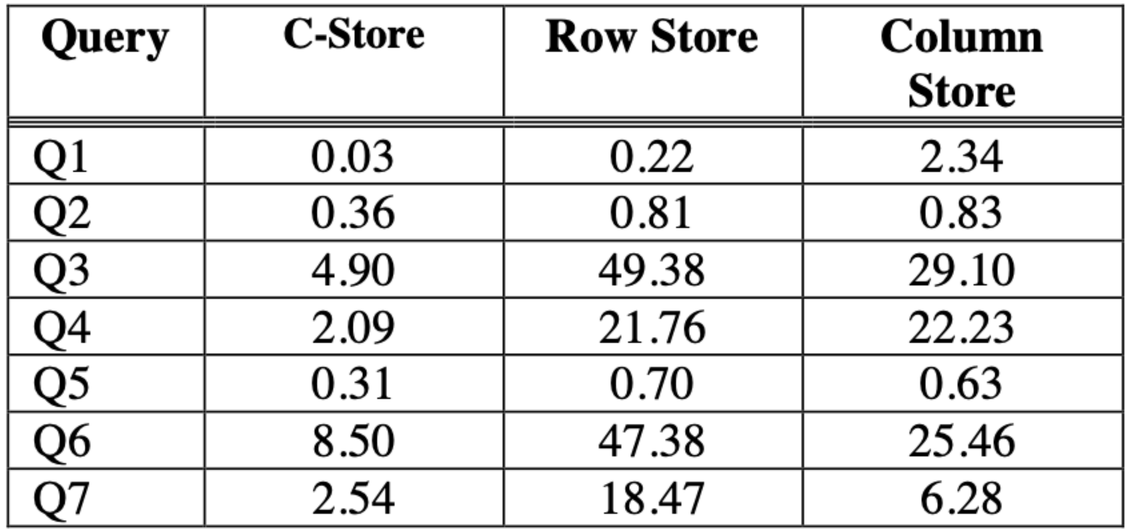


| C-Store | Row Store | Column Store |
|---|---|---|
| 1.987 GB | 11.900 GB | 4.090 GB |

- Query execution time in standard relational schemas (left) materialized schemas (right)



| 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 |



| 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 |

## Performance Test Results Summary

- C-store is much faster than either commercial product
  - *Column representation* – avoids reads of unused attributes
  - *Storing overlapping projections, rather than the whole table* – allows storage of multiple orderings of a column as appropriate.
  - *Better compression of data* – allows more orderings in the same space
  - *Query operators operate on compressed representation* – mitigates the storage barrier problem of current processors.

## Summary

- Proposed a concept of read-mostly database
- A hybrid architecture that permits column store transactions
- An emphasis on compressing data and coding data values to save space while storing representations of data on disc
- A data model that, instead of the usual tables, secondary indexes, and projections, consists of overlapping projections of tables
- Effective isolation of snapshots

## References

*Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-store: a column-oriented DBMS. In Proceedings of the 31st international conference on Very large data bases (VLDB '05). VLDB Endowment, 553–564.*

UNIVERSITY OF
WATERLOO

FACULTY OF MATHEMATICS

## THANK YOU