UNIVERSITY OF WISCONSIN–MADISON

CS 764: Topics in Database Management Systems

Lecture 5: Column Store

Xiangyao Yu

9/18/2025

Today’s Paper: C-Store

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[‡]

[*]MIT CSAIL Cambridge, MA
[+]Brandeis University Waltham, MA
[†]UMass Boston Boston, MA
[‡]Brown University Providence, RI

Abstract

This paper presents the design of a read-optimized relational DBMS that contrasts sharply with most current systems, which are write-optimized. Among the many differences in its design are: storage of data by column rather than by row, careful coding and packing of objects into storage including main memory during query processing, storing an overlapping collection of column-oriented projections, rather than the current fare of tables and indexes, a non-traditional implementation of transactions which includes high availability and snapshot isolation for read-only transactions, and the extensive use of bitmap indexes to complement B-tree structures.

We present preliminary performance data on a subset of TPC-H and show that the system we are building, C-Store, is substantially faster than popular commercial products. Hence, the architecture looks very encouraging.

1. Introduction

Most major DBMS vendors implement record-oriented storage systems, where the attributes of a record (or tuple) are placed contiguously in storage. With this *row store* architecture, a single disk write suffices to push all of the fields of a single record out to disk. Hence, high performance writes are achieved, and we call a DBMS with a row store architecture a *write-optimized* system. These are especially effective on OLTP-style applications.

In contrast, systems oriented toward ad-hoc querying of large amounts of data should be *read-optimized*. Data warehouses represent one class of read-optimized system, in which periodically a bulk load of new data is performed, followed by a relatively long period of ad-hoc queries. Other read-mostly applications include customer relationship management (CRM) systems, electronic library card catalogs, and other ad-hoc inquiry systems. In such environments, a *column store* architecture, in which the values for each single column (or attribute) are stored contiguously, should be more efficient. This efficiency has been demonstrated in the warehouse marketplace by products like Sybase IQ [FREN95, SYBA04], Addamark [ADDA04], and KDB [KDB04]. In this paper, we discuss the design of a column store called C-Store that includes a number of novel features relative to existing systems.

With a column store architecture, a DBMS need only read the values of columns required for processing a given query, and can avoid bringing into memory irrelevant attributes. In warehouse environments where typical queries involve aggregates performed over large numbers of data items, a column store has a sizeable performance advantage. However, there are several other major distinctions that can be drawn between an architecture that is read-optimized and one that is write-optimized.

Current relational DBMSs were designed to pad attributes to byte or word boundaries and to store values in their native data format. It was thought that it was too expensive to shift data values onto byte or word boundaries in main memory for processing. However, CPUs are getting faster at a much greater rate than disk bandwidth is increasing. Hence, it makes sense to trade CPU cycles, which are abundant, for disk bandwidth, which is not. This tradeoff appears especially profitable in a read-mostly environment.

There are two ways a column store can use CPU cycles to save disk bandwidth. First, it can code data elements into a more compact form. For example, if one is storing an attribute that is a customer’s state of residence, then US states can be coded into six bits, whereas the two-character abbreviation requires 16 bits and a variable length character string for the name of the state requires many more. Second, one should *densepack* values in storage. For example, in a column store it is straightforward to pack N values, each K bits long, into N * K bits. The coding and compressibility advantages of a

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

VLDB 2005

Agenda

Row store vs. column store

C-store

- Architecture
- Data model
- Data encoding
- Query execution
- Transaction updates
- Evaluation

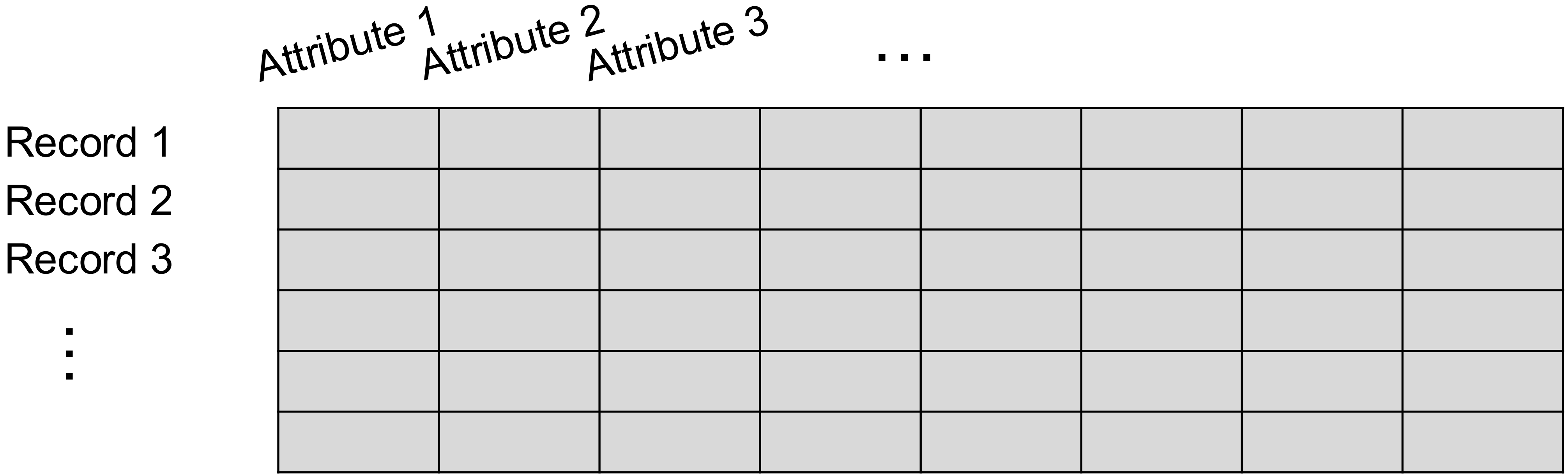
Agenda

Row store vs. column store

C-store

- Architecture
- Data model
- Data encoding
- Query execution
- Transaction updates
- Evaluation

Relational Database



A relation (table) has rows and columns

Row Store vs. Column Store

Row Store

M	100	fall	…

F	95	fall	…

F	98	spring	…

M	79	spring	…

Row store: fields in a row are contiguously stored on disk

– Write optimized

Row Store vs. Column Store

Row Store

M	100	fall	…
F	95	fall	…
F	98	spring	…
M	79	spring	…

Column Store

M	100	fall	…
F	95	fall	…
F	98	spring	…
M	79	spring	…

Row store: fields in a row are contiguously stored on disk

– Write optimized

Column store: fields in a column are contiguously stored on disk

– Read optimized

Row Store vs. Column Store

Row Store

M	100	fall	…
F	95	fall	…
F	98	spring	…
M	79	spring	…

Column Store

M	100	fall	…
F	95	fall	…
F	98	spring	…
M	79	spring	…

Advantages of column store

- Only needed attributes are loaded into memory

Row Store vs. Column Store

Row Store

M	100	fall	…
F	95	fall	…
F	98	spring	…
M	79	spring	…

Column Store

M	100	fall	…
F	95	fall	…
F	98	spring	…
M	79	spring	…

Advantages of column store

- Only needed attributes are loaded into memory
- Store data in more compact layout (avoid word and page alignment)

Row Store vs. Column Store

Row Store

M	100	fall	…
F	95	fall	…
F	98	spring	…
M	79	spring	…

Column Store

M	100	fall	…
F	95	fall	…
F	98	spring	…
M	79	spring	…

Advantages of column store

– Only needed attributes are loaded into memory

– Store data in more compact layout (avoid word and page alignment)

– Easier to compress data

Row Store vs. Column Store

Row Store

M	100	fall	…
F	95	fall	…
F	98	spring	…
M	79	spring	…

update one row

Column Store

M	100	fall	…
F	95	fall	…
F	98	spring	…
M	79	spring	…

update all columns

Advantages of column store

– Only needed attributes are loaded into memory

– Store data in more compact layout (avoid word and page alignment)

– Easier to compress data

Disadvantages of column store

– Updates are less efficient

Agenda

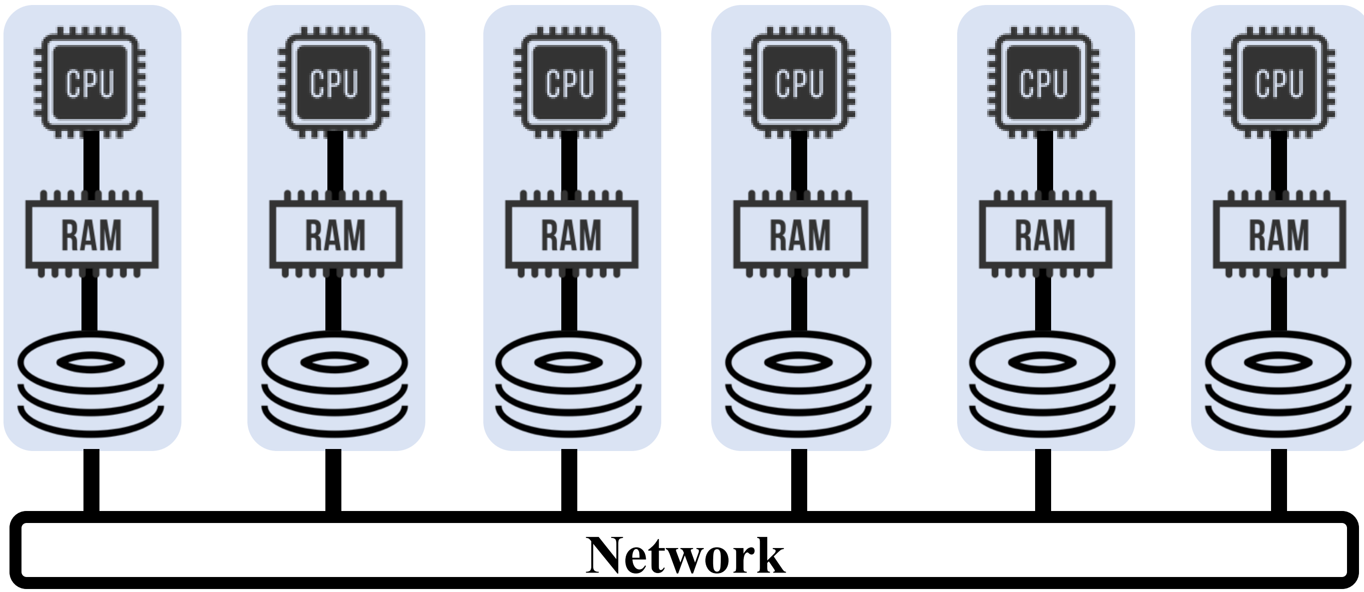
Row store vs. column store

C-store

- **Architecture**
- Data model
- Data encoding
- Query execution
- Transaction updates
- Evaluation

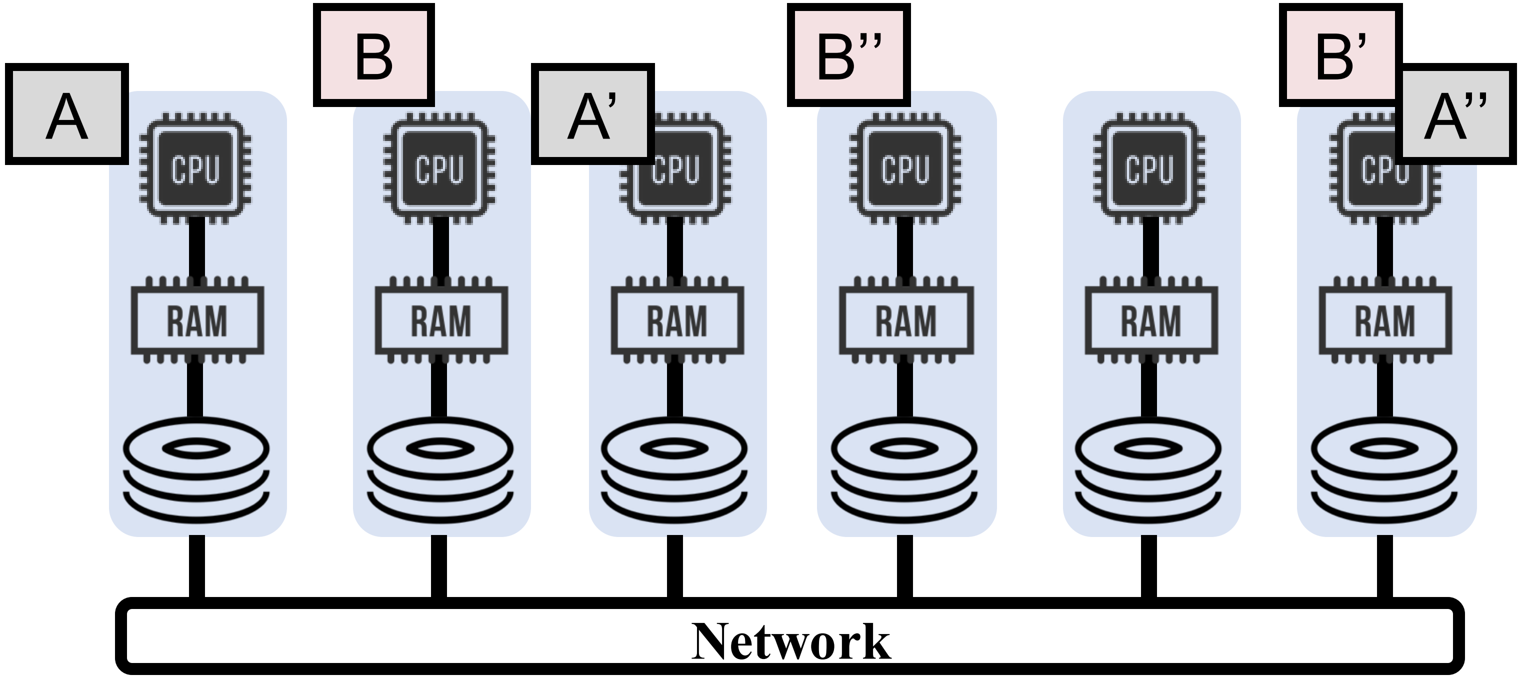
C-Store Architecture — Shared Nothing

- Data is partitioned across servers in a cluster



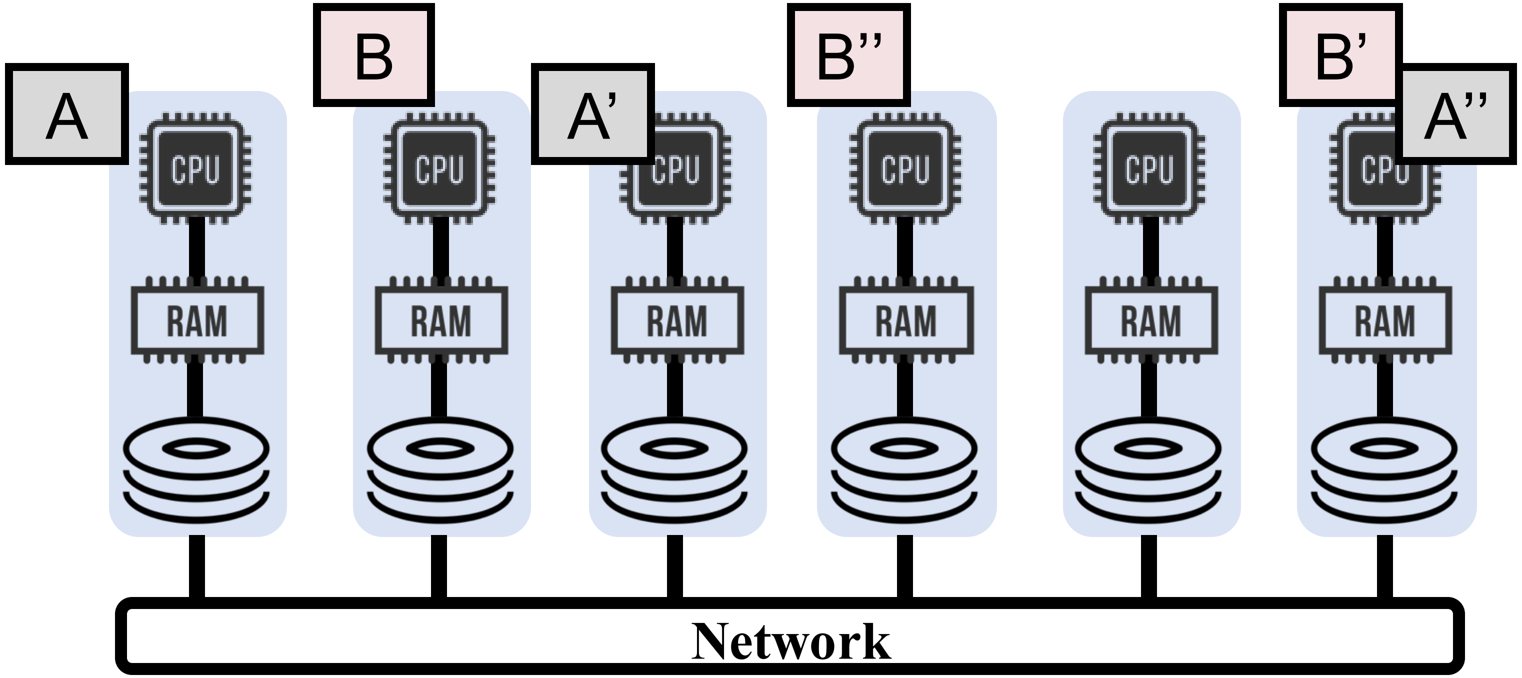
C-Store Architecture — Shared Nothing

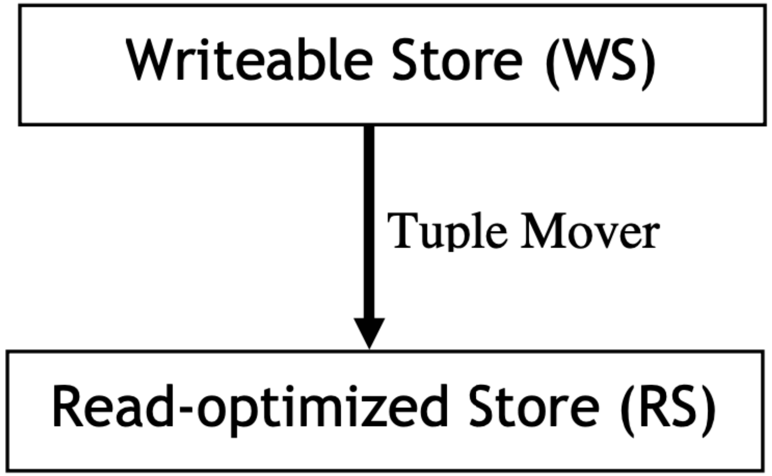
- Data is partitioned across servers in a cluster
- Each piece of data is stored in multiple replicas for high availability
  - If one replica fails, can read from other replicas



C-Store Architecture — Shared Nothing

- Data is partitioned across servers in a cluster
- Each piece of data is stored in multiple replicas for high availability
  - If one replica fails, can read from other replicas
- Separate reads and writes to different stores





Agenda

Row store vs. column store

C-store

- Architecture
- **Data model**
- Data encoding
- Query execution
- Transaction updates
- Evaluation

C-Store Data Model

Projection: A group of columns sorted on the same attributes

Example:

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

C-Store Data Model

Projection: A group of columns sorted on the same attributes

Example:

Sort key

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

C-Store Data Model

Projection: A group of columns sorted on the same attributes

Example:

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

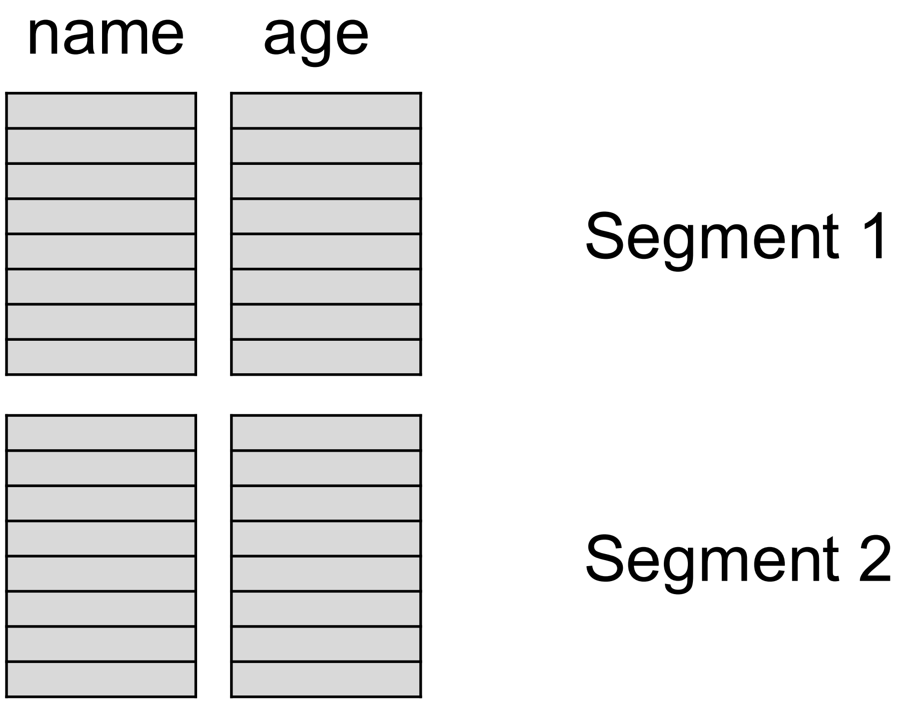
Same attribute can belong to multiple projections, and be sorted in different orders

C-Store Data Model

Segment: Each projection is horizontally partitioned into segments

- Called row groups in parquet format

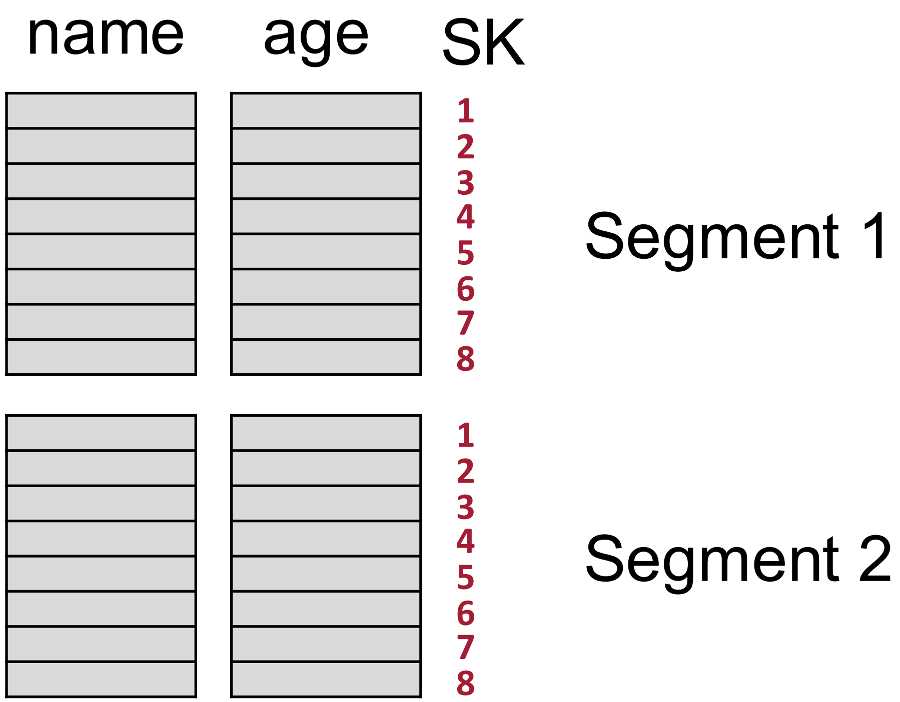
```
EMP1(name, age| age)
```



C-Store Data Model

Storage Key: Each segment associates every data value of every column with a storage key, SK

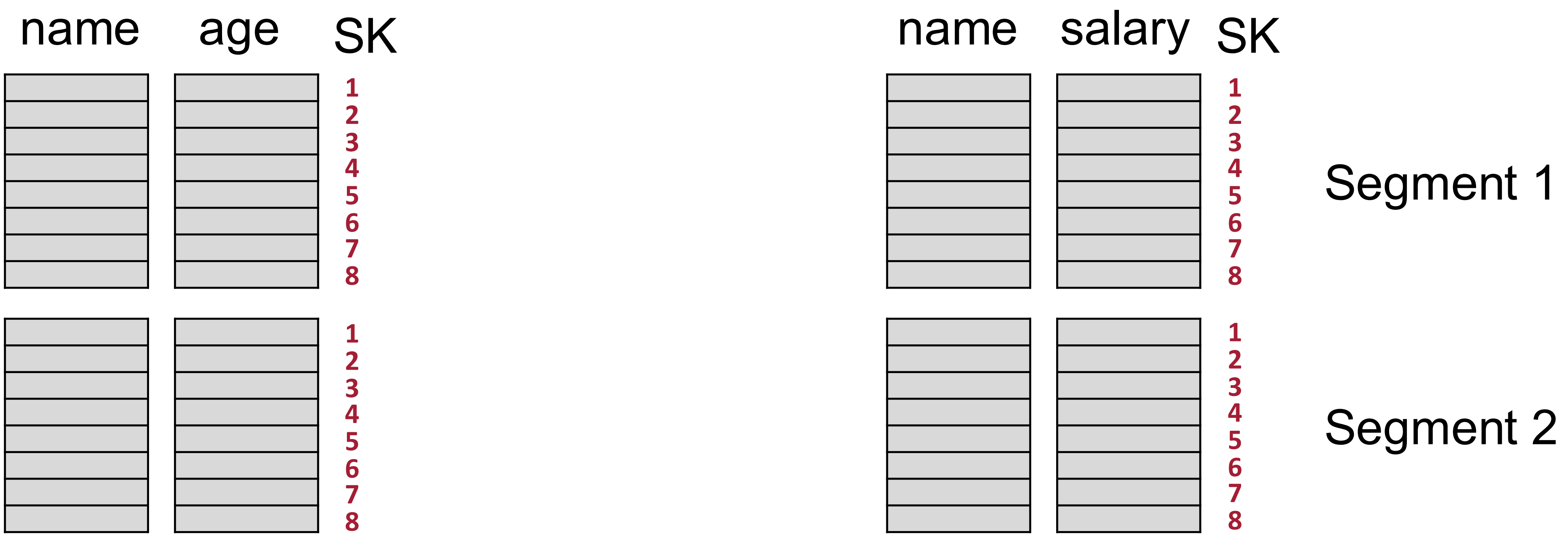
– For records in RS, SK is the physical position in the column



C-Store Data Model

Join Indices store the mapping between projections that are anchored at the same table (one-to-one mapping)

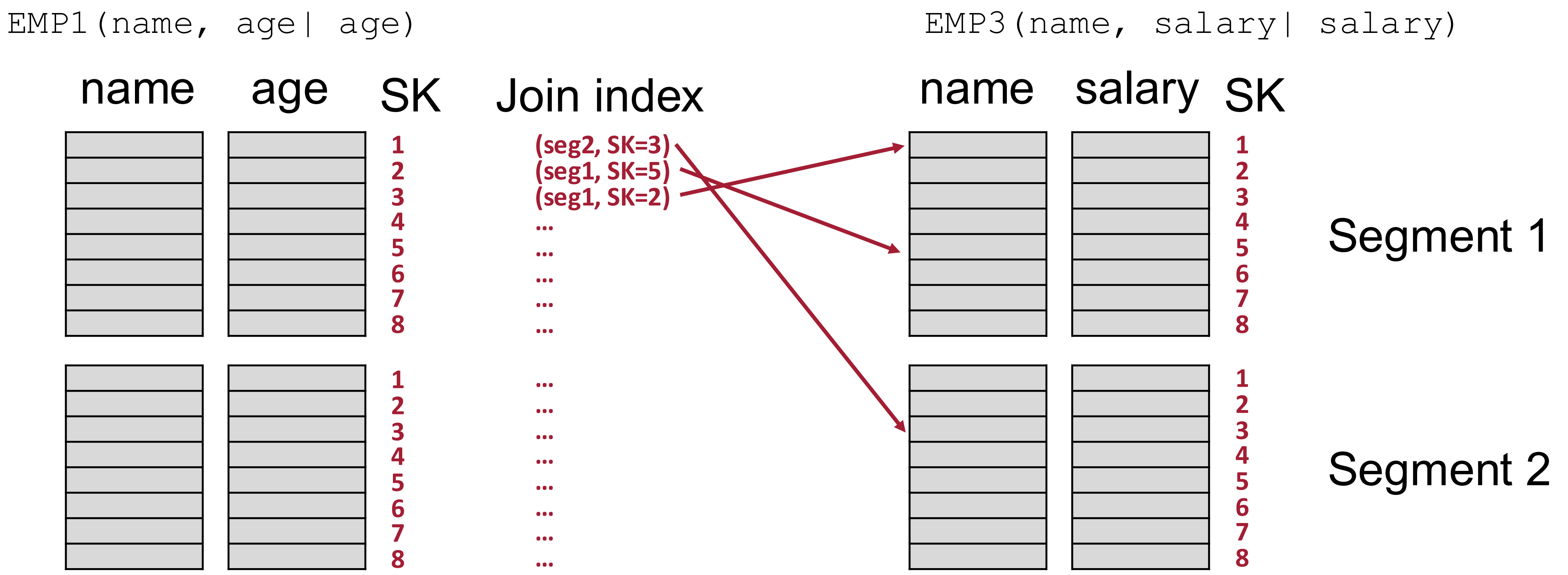
`EMP1(name, age| age)` `EMP3(name, salary| salary)`



C-Store Data Model

Join Indices store the mapping between projections that are anchored at the same table (one-to-one mapping)

- (segment_ID, SK) to locate the matching record



Agenda

Row store vs. column store

C-store

- – Architecture
- – Data model
- – **Data encoding**
- – Query execution
- – Transaction updates
- – Evaluation

Data Encoding

Type 1: Self-order, few distinct values

- (value, first-appear-position, number-of-appearance)
- Similar to run length encoding (RLE)

Data Encoding

Type 1: Self-order, few distinct values

- (value, first-appear-position, number-of-appearance)
- Similar to run length encoding (RLE)

Type 2: Foreign-order, few distinct values

- Bitmap encoding (value, bitmap)

Data Encoding

Type 1: Self-order, few distinct values

- (value, first-appear-position, number-of-appearance)
- Similar to run length encoding (RLE)

Type 2: Foreign-order, few distinct values

- Bitmap encoding (value, bitmap)

Discussion Question:
Is there an encoding scheme that can achieve higher compression ratio than bitmap encoding? (Hint: consider 4 unique values)

Data Encoding

Type 1: Self-order, few distinct values

- (value, first-appear-position, number-of-appearance)
- Similar to run length encoding (RLE)

Type 2: Foreign-order, few distinct values

- Bitmap encoding (value, bitmap)

Type 3: Self-order, many distinct values

- Delta encoding

Data Encoding

Type 1: Self-order, few distinct values

- (value, first-appear-position, number-of-appearance)
- Similar to run length encoding (RLE)

Type 2: Foreign-order, few distinct values

- Bitmap encoding (value, bitmap)

Type 3: Self-order, many distinct values

- Delta encoding

Type 4: Foreign-order, many distinct value

- No encoding

Other Data Encoding Schemes

Bitpacking

Frame of reference (FOR)

Dictionary encoding

Agenda

Row store vs. column store

C-store

- Architecture
- Data model
- Data encoding
- **Query execution**
- Transaction updates
- Evaluation

Query Execution

- Decompress
- Select
- Mask
- Project
- Sort
- Aggregation
- Concat
- Permute
- Join
- Bitstring operators

Query Execution

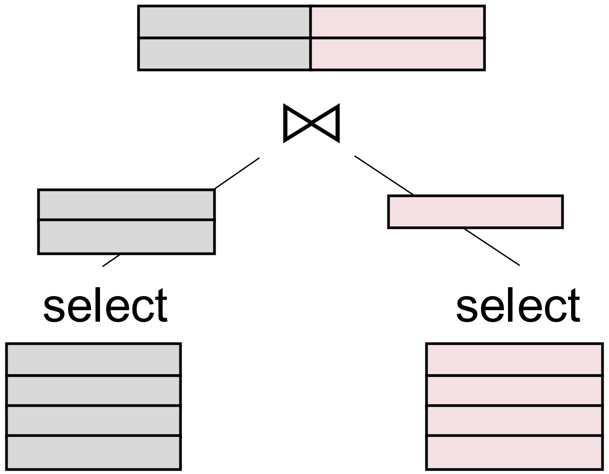
- Decompress
- Select
- Mask
- Project
- Sort
- Aggregation
- Concat
- Permute
- Join
- Bitstring operators

Impact on query optimizers

- Choose the best projections to run queries
- Cost model includes the compression type

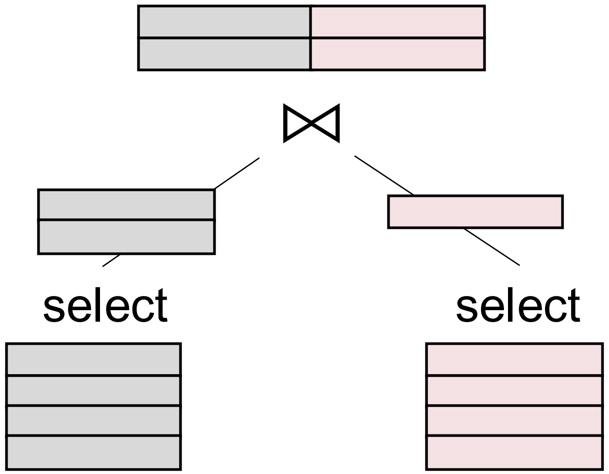
Query Execution Example

Join in row store



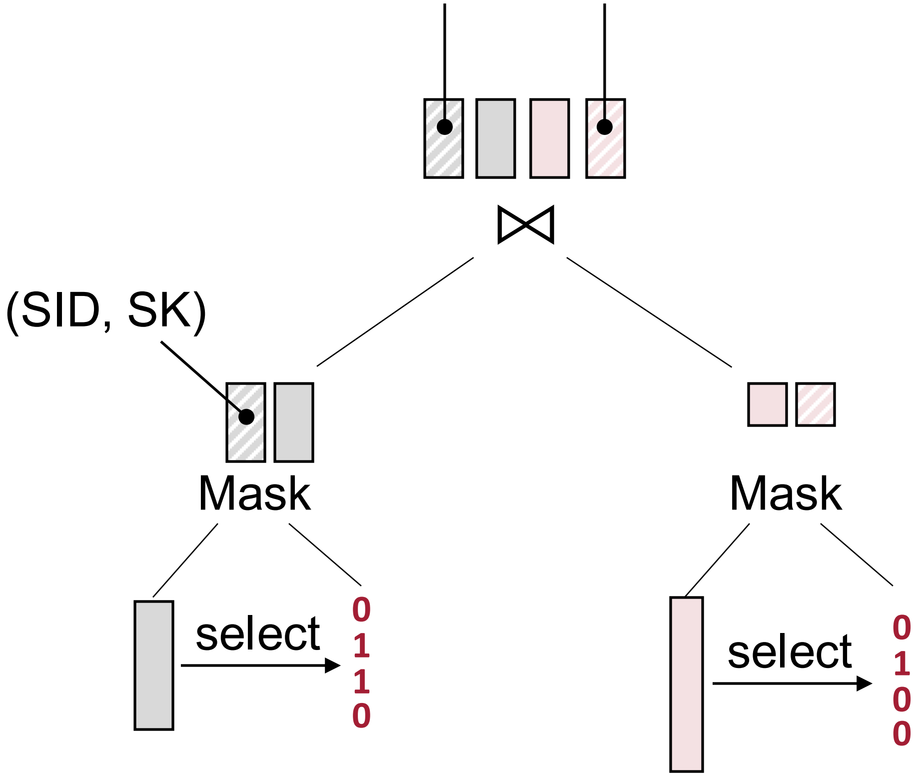
Query Execution Example

Join in row store



Join in column store

Use (SID, SK) pairs to join with other columns
(**late materialization**)



Agenda

Row store vs. column store

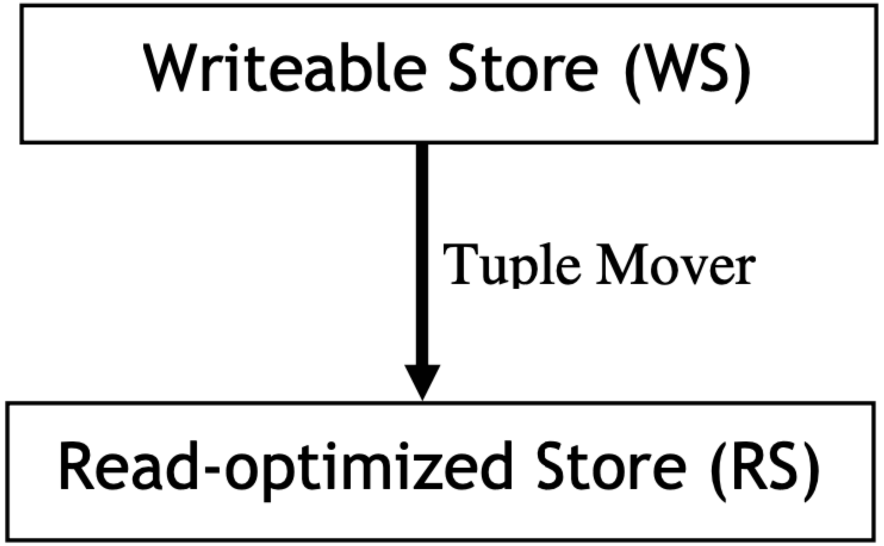
C-store

- Architecture
- Data model
- Data encoding
- Query execution
- **Transaction updates**
- Evaluation

Transaction Updates

Write Store (WS)

- 1:1 mapping between RS and WS
- Storage keys are explicitly stored
- No compression
- Snapshot isolation



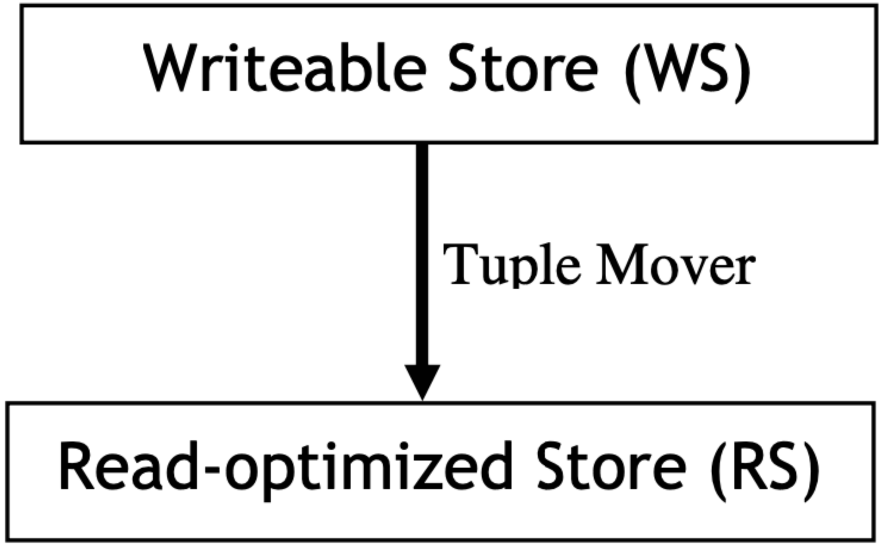
Transaction Updates

Write Store (WS)

- 1:1 mapping between RS and WS
- Storage keys are explicitly stored
- No compression
- Snapshot isolation

Tuple mover

- Periodically merge WS and RS into a new RS'



Agenda

Row store vs. column store

C-store

- Architecture
- Data model
- Data encoding
- Query execution
- Transaction updates
- **Evaluation**

Evaluation

No materialized view in baselines

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

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

Evaluation

No materialized view in baselines

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

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

With materialized view in baselines

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

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

Questions

- Why join index, why not having all columns in one projection?
- Real-world implementation? [Vertica]
- More details on tuple mover
- Columnar benefits shrink for SELECT *?
- How does C-store work in cloud-native databases?
- How are projections chosen in practice?

Before Next Lecture

Submit review for

Laurens Kuiper, et al., Robust External Hash Aggregation in the Solid State Age. ICDE 2024