C-Store: A Column-oriented DBMS

Mike Stonebraker1, Daniel J. Abadi1, Adam Batkin1, Xuedong Chen2, Mitch Cherniack3, Miguel Ferrante4, Edmond Lau5, Anil Venkat Lall6, Sam Madden7, Elizabeth O’Neil8, Pat O’Neil9, Alex Rojin1, Nga Tran1, Stan Zdonik1

MIT CSAIL
Cambridge, MA

Brandeis University
Wayland, MA

UMass Boston
Boston, MA

Brown University
Providence, RI

Abstract

This paper presents the design of a multi-optimized relational DBMS that灾构与 existing column-oriented systems, which are well-optimized. Among the many differences in the design are: (1) the use of column-oriented storage, rather than the current row- or tuple-based systems, (2) column-oriented implementations of transactions which includes high availability and snapshot isolation for multi-thread transactions, and (3) the use of bitmap indices to complement B-tree structures. We present preliminary performance data on a subset of TPC-C and show that the system we are building, C-Store, is substantially faster than popular commercial products. Hence, the architecture looks very promising.

1. Introduction

Most major DBMS vendors implement record-oriented storage engines, where the attributes of a record (or tuple) are placed contiguously in storage. With this row store architecture, a single disk write suffices to update 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 achieving higher rates of data read are column-oriented systems. Data warehouses represent one class of data-oriented system, in which periodically a bulk load of new data is performed, followed by a relatively long period of ad-hoc queries. Other real-time applications include consumer 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 column (or attribute) are stored contiguously, should be more efficient. This efficiency has been demonstrated in the warehouse benchmarking by products like Sybase IQ (PRESTO, STRATA), Adesso (ADAMOND), and KDD (KDDO). 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 would need read the values of columns requested per processing a given query and can avoid bringing into memory elements or attributes. In warehouse environments where typical queries involve aggregates performed over large numbers of data items, a column store has a substantial performance advantage. However, there are several other major distinctions that can be drawn between an architecture that is column-oriented and one that is row-optimized.

Current relational DBMSs were designed to put attributes in rows and were not built to support column-oriented storage in their native data format. It was thought that it was too expensive to drill data values out of row-oriented systems and that this was a problem for data warehouses. However, CPUs are getting faster at a much greater rate than disk bandwidth is increasing. Hence, it makes sense to store CPU cycles, which are abundant, for disk bandwidth, which is not. This trend will perpetuate especially profitable in a multi-thread 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 abbreviations require 16 bits and a variable length character string for the name of the state requires many more. Second, one should avoid 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 compactness advantages of a
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
A relation (table) has rows and columns
Row Store vs. Column Store

Row Store

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>100</td>
<td>fall</td>
</tr>
<tr>
<td>F</td>
<td>95</td>
<td>fall</td>
</tr>
<tr>
<td>F</td>
<td>98</td>
<td>spring</td>
</tr>
<tr>
<td>M</td>
<td>79</td>
<td>spring</td>
</tr>
</tbody>
</table>

**Row store**: fields in a row are contiguously stored on disk
- Write optimized
Row Store vs. Column Store

<table>
<thead>
<tr>
<th>Row Store</th>
<th>Column Store</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>M</strong></td>
<td>100</td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>95</td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>98</td>
</tr>
<tr>
<td><strong>M</strong></td>
<td>79</td>
</tr>
<tr>
<td><strong>M</strong></td>
<td>100</td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>95</td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>98</td>
</tr>
<tr>
<td><strong>M</strong></td>
<td>79</td>
</tr>
</tbody>
</table>

**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

Advantages of column store
– Only needed attributes are loaded into memory
Row Store vs. Column Store

### Row Store

<table>
<thead>
<tr>
<th>M</th>
<th>100</th>
<th>fall</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>95</td>
<td>fall</td>
<td>...</td>
</tr>
<tr>
<td>F</td>
<td>98</td>
<td>spring</td>
<td>...</td>
</tr>
<tr>
<td>M</td>
<td>79</td>
<td>spring</td>
<td>...</td>
</tr>
</tbody>
</table>

### Column Store

<table>
<thead>
<tr>
<th>M</th>
<th>100</th>
<th>fall</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>95</td>
<td>fall</td>
<td>...</td>
</tr>
<tr>
<td>F</td>
<td>98</td>
<td>spring</td>
<td>...</td>
</tr>
<tr>
<td>M</td>
<td>79</td>
<td>spring</td>
<td>...</td>
</tr>
</tbody>
</table>

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>100</td>
<td>fall</td>
</tr>
<tr>
<td>F</td>
<td>95</td>
<td>fall</td>
</tr>
<tr>
<td>F</td>
<td>98</td>
<td>spring</td>
</tr>
<tr>
<td>M</td>
<td>79</td>
<td>spring</td>
</tr>
</tbody>
</table>

### Column Store

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>100</td>
<td>fall</td>
</tr>
<tr>
<td>F</td>
<td>95</td>
<td>fall</td>
</tr>
<tr>
<td>F</td>
<td>98</td>
<td>spring</td>
</tr>
<tr>
<td>M</td>
<td>79</td>
<td>spring</td>
</tr>
</tbody>
</table>

### 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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>100</td>
<td>fall</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>95</td>
<td>fall</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>98</td>
<td>spring</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>79</td>
<td>spring</td>
<td></td>
</tr>
</tbody>
</table>

#### Column Store

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>100</td>
<td>fall</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>95</td>
<td>fall</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>98</td>
<td>spring</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>79</td>
<td>spring</td>
<td></td>
</tr>
</tbody>
</table>

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
**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:

- 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(\text{name}, \text{age} | \text{age})
- EMP2(\text{dept}, \text{age}, \text{DEPT.floor} | \text{DEPT.floor})
- EMP3(\text{name}, \text{salary} | \text{salary})
- DEPT1(\text{dname}, \text{floor} | \text{floor})

Same attribute can belong to multiple projections, and be sorted in different orders
**Segment**: Each projection is horizontally partitioned into segments
- Called *row groups* in parquet format

\[ \text{EMP1(name, age | age)} \]

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Segment 1

Segment 2
**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

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>SK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>

Segment 1

Segment 2
Join Indices store the mapping between projections that are anchored at the same table (one-to-one mapping).
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
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

Mask

(SID, SK)

select

select

0

1

0

1

1

0

0

1

0

0
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

<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>
# Evaluation

## No materialized view in baselines

<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>

## With materialized view in baselines

<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>11,900 GB</td>
<td>4,090 GB</td>
</tr>
</tbody>
</table>

### Query Results

<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>
Q/A – C-Store

How can the idea adapt for distributed databases?
Does industry welcome column store?
Any way to optimize for write performance?
Impact of not doing prepare phase in 2PC?
Why storage keys calculated on the fly instead of being stored?
Multiple projections amplify space usage.
Before Next Lecture

Submit review for