Lecture 9

Column Stores

10/5/2022

Lab 2 Due, Lab 3 Out

PEOPLE LIKE THE NEW LAYOUT

BETTER CHANGE THE LAYOUT
Plan for Next Few Lectures

Admission Control
Connection Management

Query System

- Parser
- Rewriter
- Planner
- Executor

Lec 9 – Column Stores (This Lecture)

Lec 10 - Optimizer (Next Lecture)

Lec 8 – Join Algos (Last Lecture)

Storage System

- Access Methods
- Buffer Manager
- Lock Manager
- Log Manager
Join Algorithms

• Nested loops (NL)
• Blocked nested loops
• Index nested loops (INL)
• When tables fit in memory
  – Hash (only 1 needs to fit)
  – Sort merge (both must fit)
• When tables don’t fit into memory
  – Blocked hash join
  – External sort merge
  – Simple hash
  – Grace hash
# External Joins Recap

Notation: P partitions / passes over data; assuming hash is O(1)

<table>
<thead>
<tr>
<th>Sort-Merge</th>
<th>Simple Hash</th>
<th>Grace Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O: 3 (</td>
<td>R</td>
<td>+</td>
</tr>
<tr>
<td>CPU: O(P x {S}/P log {S}/P)</td>
<td>CPU: O({R} + {S})</td>
<td>CPU: O({R} + {S})</td>
</tr>
</tbody>
</table>

Grace hash is generally a safe bet, unless memory is close to size of tables, in which case simple can be preferable

Extra cost of sorting makes sort merge unattractive unless there is a way to access tables in sorted order (e.g., a clustered index), or a need to output data in sorted order (e.g., for a subsequent ORDER BY)

Many fancier versions exist, e.g., Radix Joins to exploit multiple cores while joining
Typical Database Setup

**Transactional database**
- Lots of writes/updates
- Reads of individual records

**Analytics / Reporting Database**
- "Warehouse"
- Lots of reads of many records
- Bulk updates
- Typical query touches a few columns

"Extract, Transform, Load"
Example Warehouse: TPC-H

All use through lineitem_orders – i.e., products purchased by day, or by customer ...
How Long Does a Scan Take?

- Time proportional to amount of data read
- Example:

<table>
<thead>
<tr>
<th>Exchange</th>
<th>date</th>
<th>symbol</th>
<th>price</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>NYSE</td>
<td>1/17/2007</td>
<td>GM</td>
<td>30.77</td>
<td>1,000</td>
</tr>
<tr>
<td>NYSE</td>
<td>1/17/2007</td>
<td>GM</td>
<td>30.77</td>
<td>10,000</td>
</tr>
<tr>
<td>NYSE</td>
<td>1/17/2007</td>
<td>GM</td>
<td>30.78</td>
<td>12,500</td>
</tr>
<tr>
<td>NQDS</td>
<td>1/17/2007</td>
<td>AAPL</td>
<td>93.24</td>
<td>9,000</td>
</tr>
</tbody>
</table>

Even though we only need price, date, and symbol, if data is on disk, must scan over all columns.

Magnetic Disk

Head

Record about to be read

Memory and SSD also transfer a block at a time, so same issue arises.

SELECT avg(price) FROM tickstore WHERE symbol = 'GM' AND date = '1/17/2007'
Column Representation Reduces Scan Time

- Idea: Store each column in a separate file

<table>
<thead>
<tr>
<th></th>
<th>Column Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GM</strong></td>
<td>30.77</td>
</tr>
<tr>
<td><strong>GM</strong></td>
<td>30.77</td>
</tr>
<tr>
<td><strong>GM</strong></td>
<td>30.78</td>
</tr>
<tr>
<td><strong>AAPL</strong></td>
<td>93.24</td>
</tr>
<tr>
<td><strong>1,000</strong></td>
<td></td>
</tr>
<tr>
<td><strong>10,000</strong></td>
<td></td>
</tr>
<tr>
<td><strong>12,500</strong></td>
<td></td>
</tr>
<tr>
<td><strong>9,000</strong></td>
<td></td>
</tr>
<tr>
<td><strong>NYSE</strong></td>
<td></td>
</tr>
<tr>
<td><strong>NYSE</strong></td>
<td></td>
</tr>
<tr>
<td><strong>NYSE</strong></td>
<td></td>
</tr>
<tr>
<td><strong>NQDS</strong></td>
<td></td>
</tr>
<tr>
<td><strong>1/17/2007</strong></td>
<td></td>
</tr>
<tr>
<td><strong>1/17/2007</strong></td>
<td></td>
</tr>
<tr>
<td><strong>1/17/2007</strong></td>
<td></td>
</tr>
<tr>
<td><strong>1/17/2007</strong></td>
<td></td>
</tr>
</tbody>
</table>

Assuming each column is same size, reduces bytes read from disk by factor of 3/5

In reality, databases are often 100’s of columns
### Linearizing a Table – Row store

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Memory/Disk (Linear Array)

- R1 C1
- R1 C2
- R1 C3
- R1 C4
- R1 C5
- R1 C6
- R2 C1
- R2 C2
- R2 C3
- R2 C4
- R2 C5
- R2 C6
- R3 C1
- R3 C2
- R3 C3
- R3 C4
- R3 C5
- R3 C6
- R4 C1
- R4 C2
- R4 C3
- R4 C4
- R4 C5
- R4 C6
Linearizing a Table – Column Store

<table>
<thead>
<tr>
<th></th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory/Disk (Linear Array)
- R1 C1
- R2 C1
- R3 C1
- R4 C1
- R5 C1
- R6 C1
- R1 C2
- R2 C2
- R3 C2
- R4 C2
- R5 C2
- R6 C2
- R1 C3
- R2 C3
- R3 C3
- R4 C3
- R5 C3
- R6 C3
- R1 C4
- R2 C4
- R3 C4
- R4 C4
- R5 C4
- R6 C4
# Tables Can Get Huge

- Data warehouse at Cambridge Mobile Telematics

<table>
<thead>
<tr>
<th>Table</th>
<th>#columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>251</td>
</tr>
<tr>
<td>t2</td>
<td>248</td>
</tr>
<tr>
<td>t3</td>
<td>134</td>
</tr>
<tr>
<td>t4</td>
<td>107</td>
</tr>
<tr>
<td>t5</td>
<td>87</td>
</tr>
<tr>
<td>t6</td>
<td>83</td>
</tr>
<tr>
<td>t7</td>
<td>71</td>
</tr>
<tr>
<td>t8</td>
<td>54</td>
</tr>
<tr>
<td>t9</td>
<td>52</td>
</tr>
<tr>
<td>t10</td>
<td>45</td>
</tr>
</tbody>
</table>

Average query access 4-5 fields

Top 2-3 tables involved in nearly every query

Using a row-store would impose ~200/4 = 50x performance overhead
When Are Columns Right?

- **Warehousing (OLAP)**
  - Read-mostly; batch update
  - Queries: Scan and aggregate a few columns
- **Vs. Transaction Processing (OLTP)**
  - Write-intensive, mostly single record ops.
- **Column-stores: OLAP optimized**
  - In practice >10x performance on comparable HW, for many real world analytic applications
  - True even if w/ Flash or main memory!

*Different architectures for different workloads*
C-Store: Rethinking Database Design from the Ground Up

Inserts

Write optimized storage

Tuple Mover

Shared nothing horizontal partitioning

Column-oriented query executor

Separate Files Column-based Compression

“C-Store: A Column-oriented DBMS” -- VLDB 05
**Query Processing Example**

- **Traditional Row Store**

  ```sql
  SELECT avg(price)
  FROM tickstore
  WHERE symbol = 'GM'
  AND date = '1/17/2007'
  ```
Query Processing Example

- Basic Column Store
- “Early Materialization”

```
SELECT avg(price)
FROM tickstore
WHERE symbol = 'GM'
AND date = '1/17/2007'
```

![Diagram showing query processing example with basic column store and early materialization.]

Fields from same tuple at same index (position) in each column file
Query Processing Example

- C-Store
- “Late Materialization”

See Abadi et al ICDE 07
Why Compress?

- **Database** size is 2x-5x larger than the volume of data loaded into it
- Database performance is proportional to the amount of data flowing through the system

Abadi et al, SIGMOD 06
Column-Oriented Compression

- Query engine processes compressed data
- Transfers load from disk to CPU
- Multiple compression types
  - Run-Length Encoding (RLE), LZ, Delta Value, Block Dictionary Bitmaps, Null Suppression
- System chooses which to apply
- Typically see 50% - 90% compression
- NULLs take virtually no space

<table>
<thead>
<tr>
<th>RLE</th>
<th>Delta</th>
<th>LZ</th>
<th>RLE</th>
<th>RLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3xGM</td>
<td>30.77</td>
<td>1,000</td>
<td>3xNYSE</td>
<td>4 x 1/17/2007</td>
</tr>
<tr>
<td>1XAPPL</td>
<td>30.77</td>
<td>10,000</td>
<td>1XNQDS</td>
<td>1/17/2007</td>
</tr>
<tr>
<td>GM</td>
<td>30.78</td>
<td>12,500</td>
<td>NYSE</td>
<td>1/17/2007</td>
</tr>
<tr>
<td>AAPL</td>
<td>93.24</td>
<td>9,000</td>
<td>NQDS</td>
<td>1/17/2007</td>
</tr>
</tbody>
</table>

Columns contain similar data, which makes compression easy.
Run Length Encoding

- Replace repeated values with a count and a value
- For single values, use a run length of 1
  - Naively, can increase storage space
  - Can use a shorter bit sequence for 1s, at the cost of more expensive decompression
- E.g., 1110002 $\rightarrow$ 3x1, 3x0, 1x2
- Works well for mostly-sorted, few-valued columns
Dictionary Encoding

• Many variants; simplest is to replace string values with integers and maintain a dictionary
• I.e., AAPL, AAPL, IBM, MSFT →
  1,1,2,3 + 1:AAPL, 2:IBM, 3:MSFT
• Works well for few-valued string columns
  • Choice of dictionary not obvious
  • Words? Records?
Lempel Ziv Encoding

• LZ ("Lempel Ziv") Compression
• General purpose lossless data compression
• Builds data dictionary dynamically as it runs
  • Add new bit strings to the dictionary as they are encountered
• Treat entire column as a document
• **AAPLAAPLIBMMAAPL**

  Dictionary: A:1, B:2, …, F:6, …, I:9, …, L:12, M:13, …, P:16

  Output:
LZ Example

- **AAPLAAPLIBMAAPL**


  Output:  1
LZ Example

- **AAPLAAPLIBMAAPL**

  Output: 1
LZ Example

- AAPLAAPLIBMAAPL

Dictionary:  A:1, B:2, ... , F:6, ... , I:9, ..., L:12, M:13, ..., P:16, ..., AA:27, AP:28

Output:  1 1
LZ Example

• AAPLAAPLIBMAAPL


Output: 1 1 16
LZ Example

- AAPLAAPLIBMAAPL


Output:   1 1 16 12
LZ Example

- AAPL AAPLMAAPL

Dictionary:    A:1, B:2, ... ,F:6, ... ,I:9, ..., L:12, M:13, ..., P:16, ..., AA:27, AP:28, PL: 29, LA: 30

Output:        1 1 16 12
LZ Example

• AAPLÅAAPLIBMAAPL

Dictionary: A:1, B:2, ... , F:6, ... , I:9, ..., L:12, M:13, ..., P:16, ..., AA:27, AP:28, PL: 29, LA: 30

Output: 1 1 16 12 27
LZ Example

• AAPL\red{AA}APL\red{IBMAAPL}

Dictionary:  A:1, B:2, …, F:6, …, I:9, …, L:12, M:13, …, P:16, …, A\red{A}:27, A\red{P}:28, P\red{L}:29, L\red{A}:30, A\red{A}P:31

Output:  1 1 16 12 27
LZ Example

• AAPLAAPL BMAAPL


Output: 1 1 16 12 27
LZ Example

- **AAPL**AA**APL** BMA**APL**


Output:  1 1 16 12 27 29
LZ Example

- AAPLAAPLÍBMAAAPL


Output:  1 1 16 12 27 29 9
LZ Example

- AAPLAAPLIBMAAPL


Output:  1 1 16 12 27 29 9 2
LZ Example

• AAPLAAPLIBMAAPL


Output:  1 1 16 12 27 29 9 2 13
LZ Example

- AAPLAAPLIBMMAAPL


Output:  1 1 16 12 27 29 9 2 13
LZ Example

• AAPLAAPL LIB MBAAPL


Output: 1 1 16 12 27 29 9 2 13 31
LZ Example

• AAPLAAPLIBMAAAPL


Output: 1 1 16 12 27 29 9 2 13 31
LZ Example

- AAPLAAPLIBMAAAPL


Output: 1 1 16 12 27 29 9 2 13 31 12
LZ Example

- **AAPLAAPLIBMAAPL**


Output: 1 1 16 12 27 29 9 2 13 31 12

Reduced from 15 to 11 symbols

But future AAPL patterns will be emitted as 1 byte instead of 4

Dictionary can be further encoded, e.g., using entropy encoding to make most common patterns use least bits (“Huffman encoding”)
Bit Packing

- Encode values with fewest possible bits
- Each value becomes bit-length (e.g., 0-8 or 0-32), followed by value in that many bits
- E.g.,: 1 2 37 7
  - Need 1, 2, 6, and 3 bits respectively
  - Each number becomes 3 bit header and encoded value
    - 1: 0x001, 0x1
    - 2: 0x010, 0x10
    - 37: 0x110, 0x10010
    - 7: 0x011, 0x111
- 3 x 4 + 12 = 24 bits to encode, vs e.g., 8x4 = 32
Delta Encoding

- Consecutive values encoding as difference to previous values
- $1.1, 1.2, 1.3 \rightarrow 1.1, +.1, +1$
  - After encoding as deltas, bit-pack
  - Works if deltas can be represented in fewer bits than whole values
- Works well for e.g., floats with small variations
Bitmap Encoding

• Encode few valued columns as bitmaps
• M M M F F → 11100, 00011
  • If fewer distinct values than bitwidth of field, saves space
  • Bitmaps can be further compressed, e.g., using RLE
• Bitmaps are very good for certain kinds of operations, e.g., filtering
Sorted Data

- Delta and RLE work great on sorted data
- Trick: Secondary sorting

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
</tr>
</tbody>
</table>

Sort on X, then Y

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>

Y is not sorted, but if many duplicates of X, will be “mostly” sorted
Operating on Compressed Data

Only possible with late materialization!
Direct Operation Optimizations

- Compressed data used directly for position lookup
  - RLE, Dictionary, Bitmap

- Direct Aggregation and GROUP BY on compressed blocks
  - RLE, Dictionary

- Join runs of compressed blocks
  - RLE, Dictionary

- Min/max directly extracted from sorted data
Query: SELECT colY, SUM(colX)
FROM lineItem
GROUP BY colY

TPC-H Scale 10 (60M records)
Sorted on colY, then colX
colY uncompressed, cardinality varies
Compression + Sorting is a Huge Win

- How can we get more sorted data?
- **Store duplicate copies of data**
  - Use different physical orderings

- Improves ad-hoc query performance
  - Due to ability to directly operate on sorted, compressed data

- Supports fail-over / redundancy
Study Break: Compression

• For each of the following columns, what compression method would you recommend?

(Choose from RLE, LZ, Dictionary, Bitmap, Delta, Bit-packing)

An unsorted column of integers in the range 0-100
A mostly sorted column of arbitrary strings
A mostly sorted column of integers in the range 0-10
A sorted column of floats
An unsorted column of strings w/ 3 values
### Study Break: Compression

For each of the following columns, what compression method would you recommend?

(Choose from RLE, LZ, Dictionary, Bitmap, Delta, Bit-packing)

<table>
<thead>
<tr>
<th>Column Description</th>
<th>Recommended Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>An unsorted column of integers in the range 0-100</td>
<td>Delta/Bit-packing (LZ/dictionary also OK)</td>
</tr>
<tr>
<td>A mostly sorted column of arbitrary strings</td>
<td>LZ</td>
</tr>
<tr>
<td>A mostly sorted column of integers in the range 0-10</td>
<td>RLE</td>
</tr>
<tr>
<td>A sorted column of floats</td>
<td>Delta</td>
</tr>
<tr>
<td>An unsorted column of strings w/ 3 values</td>
<td>Bitmap</td>
</tr>
</tbody>
</table>
Write Performance

Trickle load: Very Fast Inserts

> Write-optimized Column Store (WOS)
  Memory: mirrored projections in insertion order (uncompressed)

Query read from both WOS and ROS

Tuple Mover
Asynchronous Data Movement

Batched
Amortizes seeks
Amortizes recompression
Enables continuous load

> Read-optimized Column Store (ROS)

Disk: data is sorted and compressed

(A B C | A)
When to Rewrite ROS Objects?

- Store multiple ROS objects, instead of just one
  - Each of which must be scanned to answer a query
- Tuple mover writes new objects
  - Avoids rewriting whole ROS on merge
- Periodically merge ROS objects to limit number of distinct objects that must be scanned ("Log structured merge tree")

---

**Read-optimized Column Store (ROS)**
- Disk: data is sorted and compressed

**Write-optimized Column Store (WOS)**
- Memory: mirrored projections in insertion order (uncompressed)
Problem: Lots of Partitions

• Performance will degrade as you get many partitions
• Idea: merge some partitions together, but how?

• Log structured merge tree: arrange so partitions merge a logarithmic number of times
Problem: Lots of Partitions

• Performance will degrade as you get many partitions
• Idea: merge some partitions together, but how?

• Log structured merge tree: arrange so partitions merge a logarithmic number of times
Problem: Lots of Partitions

• Performance will degrade as you get many partitions
• Idea: merge some partitions together, but how?

• Log structured merge tree: arrange so partitions merge a logarithmic number of times
Problem: Lots of Partitions

• Performance will degrade as you get many partitions
• Idea: merge some partitions together, but how?

• Log structured merge tree: arrange so partitions merge a logarithmic number of times
Problem: Lots of Partitions

• Performance will degrade as you get many partitions
• Idea: merge some partitions together, but how?

• Log structured merge tree: arrange so partitions merge a logarithmic number of times

P1 has merged 2 times, but won’t merge again until after 8 more partitions arrive
Log Structure Merge Tree

Exponentially Larger & Less Frequent Merges

1-2

1-4

3-4

P1

P2

P3

P4

5-6

7-8

P5

P6

P7

P8
C-Store Performance

• How much do these optimizations matter?

• Wanted to compare against best you could do with a commercial system
Emulating a Column Store

• Two approaches:

  1. **Vertical partitioning**: for \( n \) column table, store \( n \) two-column tables, with \( i \)th table containing a tuple-id, and attribute \( i \)
     • Sort on tuple-id
     • Merge joins for query results

  2. **Index-only plans**
     • Create a secondary index on each column
     • Never follow pointers to base table
Two Emulation Approaches

Option A: Vertical Partitioning

<table>
<thead>
<tr>
<th>Last Name</th>
<th>First Name</th>
<th>E-mail</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Option B: Index Every Column

Last Name Index

First Name Index

Daniel Abadi -- Yale University
**Bottom Line**

- **SSBM (Star Schema Benchmark -- O’Neil et al ICDE 08)**
  - Data warehousing benchmark based on TPC-H
  - Scale 100 (60 M row table), 17 columns
  - Average across 12 queries
  - Row store is a commercial DB, tuned by professional DBA vs C-Store

---

### Performance Comparison

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-Store, Compression</td>
<td>4</td>
</tr>
<tr>
<td>C-Store, No Compression</td>
<td>15</td>
</tr>
<tr>
<td>C-Store, Early Materialize</td>
<td>41</td>
</tr>
<tr>
<td>Rows</td>
<td>26</td>
</tr>
<tr>
<td>Rows, Vert. Part.</td>
<td>80</td>
</tr>
<tr>
<td>Rows, All Indexes</td>
<td>221</td>
</tr>
</tbody>
</table>

**Notes:**
- **SSBM** is a commercial system that does not benefit from vertical partitioning.
Problems with Vertical Partitioning

① Tuple headers
  - Total table is 4 GB
  - Each column table is ~1.0 GB
  - Factor of 4 overhead from tuple headers and tuple-ids

② Merge joins
  - Answering queries requires joins
  - Row-store doesn’t know that column-tables are sorted
    - Sort hurts performance
  - Would need to fix these, plus add direct operation on compressed data, to approach C-Store performance
Recommendations for Row-Store Designers

- Might be possible to get C-Store like performance
  1. Need to store tuple headers elsewhere (not require that they be read from disk w/ tuples)
  2. Need to provide efficient merge join implementation that understands sorted columns
  3. Need to support direct operation on compressed data
    - Requires “late materialization” design
Column-Oriented Data In Modern Systems

• C-Store commercialized as Vertica
• Although it wasn’t the first column-oriented DB, it led to a proliferation of commercial column-oriented systems
• Now the de-facto way that analytic database systems are built, including Snowflake, Redshift, and others.

• One popular open source option: Parquet
Efficient Data Loading: Parquet

- Parquet is column-oriented file format that is MUCH more efficient than CSV for storing tabular data
- Vs CSV, Parquet is stored in binary representation
  - Uses less space
  - Doesn’t require conversion from strings to internal types
  - Doesn’t require parsing or error detection
  - Column-oriented, making access to subsets of columns much faster
### Parquet Format

- Data is partitioned sets of rows, called “row groups”
- Within each row group, data from different columns is stored separately

<table>
<thead>
<tr>
<th>Row Group 1</th>
<th>Data Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Col 1 Block 1</td>
<td>Col 2 Block 1</td>
</tr>
<tr>
<td>Col 1 Block 2</td>
<td>Col 2 Block 2</td>
</tr>
<tr>
<td>Col 1 Block 3</td>
<td>Col 2 Block 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Row Group 2</th>
<th>Data Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Col 1 Block 4</td>
<td>Col 2 Block 4</td>
</tr>
<tr>
<td>Col 1 Block 5</td>
<td>Col 2 Block 5</td>
</tr>
<tr>
<td>Col 1 Block 6</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Row Group N</th>
<th>Data Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Col 1 Block i</td>
<td>Col 2 Block j</td>
</tr>
<tr>
<td>Col 1 Block i+1</td>
<td>Col 2 Block j+1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

- **Header**: Offset of start of each row / column group, and ranges of records in each row group

- **Using header**, can efficiently read any subset of columns or rows without scanning whole file (unlike CSV)

- **Within a row group**, data for each column is stored together
Predicate Pushdown w/ Parquet & Pandas

```python
pd.read_parquet(‘file.pq’, columns=[‘Col 1’, ‘Col 2’])
```

- Only reads col1 and col2 from disk
- For a wide dataset saves a ton of I/O
Performance Measurement

- Compare reading CSV to parquet to just columns we need

```python
import time
import pandas as pd

t = time.perf_counter()
df = pd.read_csv("FARS2019NationalCSV/Person.CSV", encoding = "ISO-8859-1")
print(f"csv elapsed = {time.perf_counter() - t:.3} seconds")

t = time.perf_counter()
df = pd.read_parquet("2019.pq")
print(f"parquet elapsed = {time.perf_counter() - t:.3} seconds")

t = time.perf_counter()
df = pd.read_parquet("2019.pq", columns = ['STATE','ST_CASE','DRINKING','PER_TYP'])
print(f"parquet subset elapsed = {time.perf_counter() - t:.3} seconds")
```

csv elapsed = 1.18 seconds
parquet elapsed = 0.338 seconds
parquet subset elapsed = 0.025 seconds

47x speedup
When to Use Parquet?

• Will always be more efficient than CSV
• Converting from Parquet to CSV takes time, so only makes sense to do so if working repeatedly with a file
• Parquet requires a library to access/read it, whereas many tools can work with CSV
• Because CSV is text, it can have mixed types in columns, or other inconsistencies
  • May be useful sometimes, but also very annoying!
  • Parquet does not support mixed types in a column
Summary

• Column oriented databases are a different way to “linearize” data to disk than the row-oriented representation we have studied

• A good fit for “warehousing” workloads that mostly read many records of a few tables

• C-Store system implements many additional ideas:
  • “Late materialization” execution
  • Column-specific compression and direct execution on compressed data
  • Read/write optimized stores

• Ideas have found their way into many modern systems and libraries, e.g., Parquet