6.5830
Lecture 9

Column Stores
10/4/2023

PS2 Due 10/10

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 8 – Optimizer (Last Time)

Lec 7 – Join Algos

Storage System

Access Methods

Buffer Manager

Lock Manager

Log Manager
Optimization Recap

• Selinger Optimizer is the foundation of modern cost-based optimizers
  – Simple statistics
  – Several heuristics, e.g., left-deep
  – Dynamic programming algo for join ordering

• Easy to extend, e.g., with:
  – More sophisticated statistics
  – Fewer heuristics
**Optimization Steps**

SELECT * FROM `emp`, `dept`, `kids`  
WHERE `sal` > 10k  
AND `emp.dno` = `dept.dno`  
AND `emp.eid` = `kids.eid`  

100 tuples/page  
10 pages RAM  
10 KB/page

**Selectivity**  
\[
\frac{1000}{100 \times 1000} = 0.01
\]

Kids is foreign key;  
Each kid joins with 3  
emps

Join algo?  
\[\bigtriangledown_{\text{dno}=\text{dno}}\]

Join Ordering? Why not kids / emp first?

Index vs scan?

**Steps:**

For each plan alternative:

1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Select best plan
Today: Column Stores
A different way to build a database system
**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 ...

“star schema”
How Long Does a Scan Take?

- Time proportional to amount of data read
- Example

```
GM 30.77 1,000 NYSE 1/17/2007
GM 30.78 12,500 NYSE 1/17/2007
AAPL 93.24 9,000 NQDS 1/17/2007
```

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

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

Memory and SSD also transfer a block at a time, so same issue arises.
Column Representation Reduces Scan Time

- Idea: Store each column in a separate file

<table>
<thead>
<tr>
<th></th>
<th>GM</th>
<th>30.77</th>
<th>1,000</th>
<th>NYSE</th>
<th>1/17/2007</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column</td>
<td>GM</td>
<td>30.77</td>
<td>10,000</td>
<td>NYSE</td>
<td>1/17/2007</td>
</tr>
<tr>
<td></td>
<td>GM</td>
<td>30.78</td>
<td>12,500</td>
<td>NYSE</td>
<td>1/17/2007</td>
</tr>
<tr>
<td></td>
<td>AAPL</td>
<td>93.24</td>
<td>9,000</td>
<td>NQDS</td>
<td>1/17/2007</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>R1</td>
<td>C1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td>C2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td>C3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td>C4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td>C5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td>C6</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>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
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 Often Super Wide

• 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 \( \sim \frac{200}{4} = 50 \)x 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

---

Separate Files
Column-based Compression

Column-oriented query executor

<table>
<thead>
<tr>
<th>SYM</th>
<th>PRICE</th>
<th>VOL</th>
<th>EXCH</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM</td>
<td>100</td>
<td>1024</td>
<td>NYSE</td>
<td>1.17.07</td>
</tr>
<tr>
<td>IBM</td>
<td>102</td>
<td>11245</td>
<td>NYSE</td>
<td>1.17.07</td>
</tr>
<tr>
<td>SUN</td>
<td>58</td>
<td>3455</td>
<td>NQDS</td>
<td>1.17.07</td>
</tr>
<tr>
<td>SUN</td>
<td>58</td>
<td>3455</td>
<td>NQDS</td>
<td>1.17.07</td>
</tr>
</tbody>
</table>

"C-Store: A Column-oriented DBMS" -- VLDB 05
Query Processing Example

- Traditional Row Store

\[
\text{SELECT}\ \text{avg\(\text{(price)\)}}
\]
\[
\text{FROM}\ \text{tickstore}
\]
\[
\text{WHERE}\ \text{symbol} = \text{‘GM’}
\]
\[
\text{AND}\ \text{date} = \text{‘1/17/2007’}
\]

Complete tuples

\[
\text{SELECT}\ \text{sym} = \text{‘GM’}
\]

Complete tuples

\[
\text{SELECT}\ \text{date} = \text{‘1/17/07’}
\]

Complete tuples

<table>
<thead>
<tr>
<th>Disk</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GM</td>
<td>30.77</td>
<td>1,000</td>
<td>NYSE</td>
<td>1/17/2007</td>
<td></td>
</tr>
<tr>
<td>GM</td>
<td>30.77</td>
<td>10,000</td>
<td>NYSE</td>
<td>1/17/2007</td>
<td></td>
</tr>
<tr>
<td>GM</td>
<td>30.78</td>
<td>12,500</td>
<td>NYSE</td>
<td>1/17/2007</td>
<td></td>
</tr>
<tr>
<td>AAPL</td>
<td>93.24</td>
<td>9,000</td>
<td>NQDS</td>
<td>1/17/2007</td>
<td></td>
</tr>
</tbody>
</table>
Query Processing Example

• Basic Column Store

• “Early Materialization”

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

SELECT sym = 'GM'
SELECT date = '1/17/07'
AVG price

Complete tuples
Construct Tuples
GM 30.77 1/17/07

Complete tuples
Disk
GM 30.77 1,000 1/17/07
GM 30.77 10,000 1/17/07
GM 30.78 12,500 1/17/07
AAPL 93.24 9,000 1/17/07

Complete tuples

Row-oriented plan

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

- C-Store
  - “Late Materialization”
  - Much less data flowing through memory

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>3XGM</th>
<th>30.77</th>
</tr>
</thead>
<tbody>
<tr>
<td>1XAPPL</td>
<td>30.77</td>
<td>30.77</td>
<td></td>
</tr>
<tr>
<td>GM</td>
<td>30.78</td>
<td>30.77</td>
<td></td>
</tr>
<tr>
<td>AAPL</td>
<td>93.24</td>
<td>30.77</td>
<td></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 → 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 \rightarrow
  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
LZ Example

- AAPLAAPLIBMAAPL

Dictionary: A:1, B:2, ..., F:6, ..., I:9, ..., L:12, M:13, ..., P:16
Output:
LZ Example

- AAPLAAPLIBMAAPL

Dictionary:  A:1, B:2, ..., F:6, ..., I:9, ..., L:12, M:13, ..., P:16
Output:  1
LZ Example

- **AAPLAAPLIBMAAPL**

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

  Output:  1
LZ Example

• AĂPLAAPLIBMAAPL


Output: 1 1
LZ Example

- AAPLAAPLIBMAAPL


Output: 1 1 16
• AAPLAAPLIBMAAPL


Output:        1 1 16 12
LZ Example

- AAPLÀAAPLIBMAAPL


Output:  1 1 16 12
LZ Example

• AAPLAAAPL LIBMAAPL


Output: 1 1 16 12 27
LZ Example

- AAPLÀAAPLIBMAAPL


Output: 1 1 16 12 27
LZ Example

• AAPLAAPL BMAAPL


Output: 1 1 16 12 27
**LZ Example**

- **AAPLAAPL BMAAPL**


  Output: 1 1 16 12 27 29
LZ Example

- AAPLAAPLÌBMAAPL


Output: 1 1 16 12 27 29 9
LZ Example

- AAPLAAPLIBMAAPL


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

• AAPLAAPL


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

• AAPLAAPLIBMAAAPL


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

- AAPLAAPLIBMAAPL


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: \(0x110, 0x100101\)
    • 7: \(0x011, 0x111\)
• \(3 \times 4 + 12 = 24\) bits to encode, vs e.g., \(8 \times 4 = 32\)
Delta Encoding

• Consecutive values encoding as difference to previous values
  
  • 1.1, 1.2, 1.3 → 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!

Compression Aware

Pos.SELECT
sym = ‘GM’

AND

Position Bitmap (3x1,1x0)

Position Bitmap (3x1,1x0)

Pos.SELECT
date=’1/17/07’

Disk

3xGM
1xAPPL

30.77
+0
+.01
+62.47

1,000
10,000
12,500
9,000

NYSE
NYSE
NYSE
NQDS

4x1/17/2007

NYSE

AVG

Prices

Position Lookup

Position Bitmap (4x1)
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
TPC-H Compression Performance

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

<table>
<thead>
<tr>
<th>Y</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

Sorted, Low Cardinality

- Direct Operation
- Eager Decompression

Sorted, High Cardinality

- Direct Operation
- Eager Decompression
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 A. RLE, B. Dictionary, C. Bitmap, D. Delta, E. Bit-packing)

https://clicker.mit.edu/6.5830/

An unsorted column of integers in the range 0-100
Delta/Bit-packing (LZ/dictionary also OK)

A mostly sorted column of arbitrary strings
LZ

A mostly sorted column of integers in the range 0-10
RLE

A sorted column of floats
Delta

An unsorted column of strings w/ 3 values
Bitmap
Write Performance

Trickle load: Very Fast Inserts

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

Tuple Mover
Asynchronous Data Movement

Batched
- Amortizes seeks
- Amortizes recompression
- Enables continuous load

Queries read from both WOS and ROS

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")
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

P1-2  P3-4  P5-6  P7
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
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

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

```
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
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