DATA516/CSED516
Scalable Data Systems and Algorithms

Lecture 8
Column-store DBMSs
Announcements

• Project milestones due on Friday
• The 3 Mini-homeworks due next Friday
• Next Tuesday (this room):
  – Individual project discussions
  – Kexuan: office hour 5-6pm by zoom
  – Dong: office hour 7-8pm this room
  – No section
• Tuesday, Dec. 7: project presentations
Project Milestone

• Hard deadline: Friday night!
• Preliminary draft of your final report
• 2-3 pages.
• Include Title and Author!
• Suggested structure
  – Section 1: what question do you want to ask?
  – Section 2: describe the system(s) and the data
  – Section 3: briefly report what you have tried
  – Section 4: what do you need to do until 12/7?
Class of Tuesday, Nov. 30\textsuperscript{th}

Work on project and HW4!

• Projects:
  – Checkout the google spreadsheet
  – I will meet with you individually, for 5’
  – Prepare a short updated on your project and plans for the next step(s)

• Project/hw4 technical questions:
  – Kexuan: over zoom 5-6pm
  – Dong: here 7-8pm
Tuesday, Dec. 7th

Project presentations: 5pm – ??pm

• You have 5 minutes (4 + 1 for questions)

• Prepare 4 slides in a google presentation. Suggestions:
  – Slide 1: Title slide: project title, your name,
  – Slide 2: Question: What question did you investigate?
  – Slide 3: Method: How did you go about answering it?
  – Slide 4: Results: What did you find?

• VOTE! Everyone votes for every presentation; 3 awards.

• I will ask you to place your google slides on a shared drive; details TBD

More details coming up soon
Today’s Lecture

• Columnar Storage

• Recap of DATA 516
References

• Ailamaki et al. *Weaving Relations for Cache Performance*, VLDB’2001

• The Design and Implementation of Modern Column-Oriented Database Systems Daniel Abadi, et al., Foundations and Trends in Databases

• Also:
  – C-Store: A Column-oriented DBMS. Stonebraker et al. VLDB’05
  – The Vertica Analytic Database: CStore 7 Years Later. Lamb et. al. VLDB’12
Column-Oriented Storage

• C-store ideas and research since 1970’s
• Circa 2000: PAX (will discuss…)
• 2004: C-store research prototype at MIT
  – Started by Mike Stonebraker
  – Lead graduate student Daniel Abadi
  – 2005: Vertica founded by M. Stonebraker & A. Palmer
  – 2011: Vertica acquired by HP
  – 2012: As of VLDB’12 paper, 500 production deployments of Vertica, three over a PB in size
• 2013: All major DB vendors include some column-store implementation
• 2016: PAX adopted by Snowflake
DBMS Architecture

[Adapted from Anatomy of a Db System. J. Hellerstein & M. Stonebraker. Red Book. 4ed.]

Process Manager
- Admission Control
- Connection Mgr

Query Processor
- Parser
- Query Rewrite
- Optimizer
- Executor

Storage Manager
- Access Methods
- Buffer Manager
- Lock Manager
- Log Manager

Shared Utilities
- Memory Mgr
- Disk Space Mgr
- Replication Services
- Admin Utilities
Consider a relation storing tweets:
Tweets(tid, user, time, content)

How should we store it on disk?
Design Exercise

- **Design choice:** **One OS file for each relation**
  - Option 1: DBMS creates one big file with “files” inside
  - Option 2: DBMSs uses disk directly, with “files” inside

- The OS (or DBMS) provides an API of the form
  - Seek to some position (or “skip” over B bytes)
  - Read/Write B bytes
Design Exercise

- Design choice: One OS file for each relation
  - Option 1: DBMS creates one big file with “files” inside
  - Option 2: DBMSs uses disk directly, with “files” inside

- The OS (or DBMS) provides an API of the form
  - Seek to some position (or “skip” over B bytes)
  - Read/Write B bytes
Design Exercise

• Design choice: **One OS file for each relation**
  – Option 1: DBMS creates one big file with “files” inside
  – Option 2: DBMSs uses disk directly, with “files” inside

• The OS (or DBMS) provides an API of the form
  – Seek to some position (or “skip” over B bytes)
  – Read/Write B bytes
Working with Pages

- Reading/writing to/from disk
  - Seeking takes a long time!
  - Reading sequentially is fast
  - Read/write entire blocks

- 1 block = typically 4, 8, or 16 KB

- Buffer manager:
  - Caches a set of blocks in main memory
  - Blocks in MM are called pages
  - 1 page = 1 block
Working with Main Memory

• The Central Processing Unit (CPU) reads/writes data from/to main memory
  – Read/write entire bytes (= 8 bits)
  – Typically: 1 or 2 or 4 or 8 bytes
• CPU much faster than MM
• Solution: CPU cache
  – A very fast, associative memory
  – Cache line = aka cache block
  – Typically: 1 cache line = 64 bytes
Summary so far…

Two bottlenecks:

• The disk I/O bottleneck:
  – Disk is much slower than main memory
  – Read/write one block at a time (8KB-16KB)
  – Buffer pool in main memory: 1page=1block
Summary so far…

Two bottlenecks:

• The disk I/O bottleneck:
  – Disk is much slower than main memory
  – Read/write one block at a time (8KB-16KB)
  – Buffer pool in main memory: 1 page = 1 block

• The main memory bottleneck
  – MM is much slower than CPU
  – Read/write one byte at a time (or 2/4/8)
  – CPU cache: 1 cache line = 64 bytes
Continuing our Design

Key question:
• How should we organize tuples on a page?

Let’s first assume all tuples are of the same size
Tweets(tid int, user char(10),
    time int, content char(140))
Design Exercise 1

• Think how you would store tuples on a page
  – Fixed length tuples
  – Variable length tuples

• Requirements
  – Insert a new tuple
  – Look up a tuple given a RID (= Record ID)
  – Remove a tuple given a RID
  – Modify a tuple
  – Enumerate all tuples
Page Formats

Issues to consider:
• 1 page = 1 disk block = fixed size (e.g. 8KB)
• Records:
  – Fixed length
  – Variable length
• Record id = RID
  – Typically RID = (PageID, SlotNumber)

Why do we need RID’s in a relational DBMS?
Page Formats

Issues to consider:

• 1 page = 1 disk block = fixed size (e.g. 8KB)

• Records:
  – Fixed length
  – Variable length

• Record id = RID
  – Typically RID = (PageID, SlotNumber)

Why do we need RID’s in a relational DBMS?
For indexes, and for transactions
Page Format Approach 1

Fixed-length records: packed representation
Divide page into **slots**. Each slot can hold one tuple
Record ID (RID) for each tuple is **(PageID,SlotNb)**

How do we insert a new record?
Page Format Approach 1

Fixed-length records: packed representation
Divide page into **slots**. Each slot can hold one tuple
Record ID (RID) for each tuple is *(PageID, SlotNb)*

<table>
<thead>
<tr>
<th>Slot&lt;sub&gt;1&lt;/sub&gt;</th>
<th>Slot&lt;sub&gt;2&lt;/sub&gt;</th>
<th>Slot&lt;sub&gt;N&lt;/sub&gt;</th>
<th>Slot&lt;sub&gt;N+1&lt;/sub&gt;</th>
<th>Free Sp.</th>
<th>N</th>
</tr>
</thead>
</table>

How do we insert a new record?
Page Format Approach 1

Fixed-length records: packed representation
Divide page into **slots**. Each slot can hold one tuple
Record ID (RID) for each tuple is \((\text{PageID,SlotNb})\)

How do we insert a new record?

How do we delete a record?
Page Format Approach 1

Fixed-length records: packed representation
Divide page into **slots**. Each slot can hold one tuple
Record ID (RID) for each tuple is **(PageID,SlotNb)**

<table>
<thead>
<tr>
<th>Slot₁</th>
<th>Slot₂</th>
<th>Slotₙ</th>
<th>Slotₙ₊₁</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Free Sp.</td>
</tr>
</tbody>
</table>

How do we insert a new record?

Number of records

How do we delete a record? Cannot remove record (why?)
Page Format Approach 1

Fixed-length records: packed representation
Divide page into slots. Each slot can hold one tuple
Record ID (RID) for each tuple is \((\text{PageID}, \text{SlotNb})\)

<table>
<thead>
<tr>
<th>Slot_1</th>
<th>Slot_2</th>
<th>Slot_N</th>
<th>Slot_{N+1}</th>
</tr>
</thead>
</table>

How do we insert a new record?

How do we delete a record? Cannot remove record (why?)

How do we handle variable-length records?
Page Format Approach 2

Record ID (RID) for each tuple is \((\text{PageID}, \text{SlotNb})\)

Header contains slot directory
+ Need to keep track of nb of slots
+ Also need to keep track of free space (F)

Can handle variable-length records
Can move tuples inside a page without changing RIDs
Record Formats

Fixed-length records => Each field has a fixed length (i.e., it has the same length in all the records)

| Field 1 | Field 2 | ... | ... | Field K |

Information about field lengths and types is in the catalog
Record Formats

Variable length records

Field 1  Field 2  . . .  . . .  Field K

Record header

Remark: NULLS require no space at all (why ?)
Summary so far…

• Page format:
  – Page header
  – Record
  – Record
  – …

• Record format:
  – Record header
  – Field
  – Field
  – …
From Row-Store to Column-Store

Rows stored contiguously on disk (+ tuple headers)
From Row-Store to Column-Store

Rows stored contiguously on disk (+ tuple headers)  

Columns stored contiguously on disk (no headers needed)
Two Options

Column Store:
- 1 column = 1 file
- Requires a complete rewrite of query engine
- Potential for major performance gain for some queries, but need a lot of work to get there (will see this)
Two Options

Column Store:
• 1 column = 1 file
• Requires a complete rewrite of query engine
• Potential for major performance gain for some queries, but need need a lot of work to get there (will see this)

PAX:
• Split the table into blocks (original PAX) or chunks (Snowflake)
• Inside each chunk, store the attribute column-wise
• Obtain most of the performance gain, with very little update to the query engine
An Intermediate Format: PAX

• PAX = Partition Attributes Across

• Addresses memory access bottleneck (not the disk bottleneck)
From Row to Column Storage (Initial Designs - 1985)

**Figure 2.1:** Storage models for storing database records inside disk pages: NSM (row-store) and DSM (a predecessor to column-stores). Figure taken from [5].
Current Scheme: Slotted Pages

Formal name: NSM (N-ary Storage Model)

Records are stored sequentially
- Offsets to start of each record at end of page

Ailamaki VLDB’01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt]
Predicate Evaluation using NSM

select name
from R
where age > 50

NSM pushes non-referenced data to the cache

Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt
**Predicate Evaluation using NSM**

<table>
<thead>
<tr>
<th>PAGE HEADER</th>
<th>RH1</th>
<th>RH2</th>
<th>RH3</th>
<th>RH4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane</td>
<td>30</td>
<td>RH</td>
<td>4322</td>
<td>John</td>
</tr>
<tr>
<td>45</td>
<td>RH3</td>
<td>1563</td>
<td>Jim</td>
<td>20</td>
</tr>
<tr>
<td>7658</td>
<td>Susan</td>
<td>52</td>
<td>2534</td>
<td>Leon</td>
</tr>
</tbody>
</table>

**MAIN MEMORY**

**CACHE**

```
select name
from R
where age > 50
```

NSM pushes non-referenced data to the cache

Ailamaki VLDB'01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt](http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt)
Predicate Evaluation using NSM

select name from R where age > 50

NSM pushes non-referenced data to the cache

Ailamaki VLDB’01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt
select name
from R
where age > 50

NSM pushes non-referenced data to the cache

Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt
Predicate Evaluation using NSM

select name
from R
where age > 50

NSM pushes non-referenced data to the cache

Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt
Need New Data Page Layout

- Eliminates unnecessary memory accesses
- Improves inter-record locality
- Keeps a record’s fields together
- Does not affect I/O performance

and, most importantly, is…

low-implementation-cost, high-impact

Ailamaki VLDB’01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt
Partition Attributes Across (PAX)

Partition data *within* the page for spatial locality

Ailamaki VLDB'01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt](http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt)
Partition data *within* the page for spatial locality.

Ailamaki VLDB’01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt](http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt)
Partition Attributes Across (PAX)

Partition data *within* the page for spatial locality

Ailamaki VLDB'01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt](http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt)
Partition Attributes Across (PAX)

Partition data within the page for spatial locality

Ailamaki VLDB’01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt]
Partition Attributes Across (PAX)

Partition data *within* the page for spatial locality

Ailamaki VLDB’01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt](http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt)
Partition Attributes Across (PAX)

Partition data *within* the page for spatial locality

Ailamaki VLDB’01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt](http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt)
Predicate Evaluation using PAX

Fewer cache misses, low reconstruction cost

Ailamaki VLDB'01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt]
Predicate Evaluation using PAX

Fewer cache misses, low reconstruction cost

Ailamaki VLDB’01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt
A Real NSM Record

NSM: All fields of record stored together + slots

Ailamaki VLDB’01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt
PAX: Detailed Design

PAX: Group fields + amortizes record headers

Ailamaki VLDB’01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt
PAX - Summary

• Improves processor cache locality
• Does not affect I/O behavior
  – Same disk accesses for NSM or PAX storage
  – No need to change the buffer manager

• Today:
  – Most (all?) commercial engines use a PAX layout of the disk
  – Beyond disk: Snowflake partitions tables horizontally into files, then uses column-store inside each file (hence, PAX)
Column-Store

• Store an entire attribute in a different file

• While the idea had been around before PAX, getting all the details right in order to extract the extra performance took a long time
C-Store Illustration

Row-based (4 pages)

<table>
<thead>
<tr>
<th>Page</th>
<th>A</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>4</td>
</tr>
</tbody>
</table>

Column-based (4 pages)

<table>
<thead>
<tr>
<th>Page</th>
<th>A</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>4</td>
</tr>
</tbody>
</table>

C-Store also avoids large tuple headers
Column-Oriented Databases

• Main idea:
  – Physical storage: complete vertical partition; each column stored separately: R.A, R.B, R.A
  – Logical schema: remains the same R(A,B,C)

• Main advantage:
  – Improved transfer rate: disk to memory, memory to CPU, better cache locality
Basic Trade-Off

- **Row stores**
  - Quick to update entire tuple (1 page IO)
  - Quick to access a single tuple

- **Column stores**
  - Avoid reading unnecessary columns
  - Better compression

- **Entire system needs a different design**
  - Not only storage manager
  - To achieve high performance
From Row to Column Storage
(Modern Designs)

**Figure 1.1:** Physical layout of column-oriented vs row-oriented databases.

Basic tradeoffs:
- Reading all attributes of one record, v.s.
- Reading some attributes of many records
Figure 1.2: Performance of C-Store versus a commercial database system on the SSBM benchmark, with different column-oriented optimizations enabled.
Key Architectural Trends (Sec.1)

• Virtual IDs

• Block-oriented and vectorized processing

• Late materialization

• Column-specific compression
Key Architectural Trends (Sec.1)

- Virtual IDs
  - Offsets (arrays) instead of keys
- Block-oriented and vectorized processing
  - Iterator model: one tuple → one block of tuples
- Late materialization
  - Postpone tuple reconstruction in query plan
- Column-specific compression
  - Much better than row-compression (why?)
Vectorized Processing

Review:

• Volcano-style iterator model
  – Next() method
  – Pipelining

• Materialization of all intermediate results

• Discuss in class:

  select avg(A) from R where A < 100
Vectorized Processing

• Vectorized processing:
  – Next() returns a block of tuples (e.g. N=1000) instead of single tuple

• Pros:
  – No more large intermediate results
  – Tight inner loop for selection and/or avg

• Discuss in class:
  
  select avg(A) from R where A < 100
Compression (Sec. 4)

• What is the advantage of compression in databases?

• Discuss main column-at-a-time compression techniques
Compression (Sec. 4)

• What is the advantage of compression in databases?

• Discuss main column-at-a-time compression techniques
  – Bit-vector (see also bit-map indexes)
Compression (Sec. 4)

Row-based (4 pages)

<table>
<thead>
<tr>
<th>Page</th>
<th>A 1</th>
<th>A 2</th>
<th>A 2</th>
<th>A 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Column-based (4 pages)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>4</td>
</tr>
</tbody>
</table>

Compressed (2 pages)

<table>
<thead>
<tr>
<th></th>
<th>4XA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2XB</td>
</tr>
<tr>
<td></td>
<td>2XC</td>
</tr>
<tr>
<td></td>
<td>1X1</td>
</tr>
<tr>
<td></td>
<td>4X2</td>
</tr>
<tr>
<td></td>
<td>5X4</td>
</tr>
</tbody>
</table>

Page
Late Materialization (Sec. 4)

- What is it?
- Discuss $\Pi_B(\sigma_{A='a'} \land D='d')(R(A,B,C,D,...))$
Late Materialization (Sec. 4)

- What is it?
- Discuss $\Pi_B(\sigma_{A='a'} \land D='d')(R(A,B,C,D,...))$
- Early materialization:
  - Retrieve positions with ‘a’ in column A: 2, 4, 5, 9, 25...
  - Retrieve those values in column D: ‘x’, ‘d’, ‘y’, ‘d’, ‘d’, ...
  - Retain only positions with ‘d’: 4, 9, ...
  - Lookup values in column B: $B[4], B[9], ...$
Late Materialization (Sec. 4)

• What is it?
• Discuss $\Pi_B(\sigma_{A='a$ \land D='d'}(R(A,B,C,D,...)))$
• Early materialization:
  – Retrieve positions with ‘a’ in column A: 2, 4, 5, 9, 25…
• Late materialization:
  – Retrieve positions with ‘a’ in column A: 2, 4, 5, 9, 25…
  – Retrieve positions with ‘d’ in column D: 3, 4, 7, 9, 12,…. 
  – Intersect: 4, 9, …
  – Lookup values in column B: B[4], B[9], …
Late Materialization (Sec. 4)

• What is it?
• Discuss $\Pi_B(\sigma_{A='a' \land D='d'}(R(A,B,C,D,...)))$
• Early materialization:
  – Retrieve positions with ‘a’ in column A: 2, 4, 5, 9, 25…
  – Retain only positions with ‘d’: 4, 9, ...
• Late materialization:
  – Retrieve positions with ‘a’ in column A: 2, 4, 5, 9, 25…
  – Retrieve positions with ‘d’ in column D: 3, 4, 7, 9, 12,..
  – Intersect: 4, 9, …
  – Lookup values in column B: $B[4], B[9], …$
Late Materialization (Sec. 4)

- What is it?
- Discuss $\Pi_B(\sigma_{A='a'} \land D='d')(R(A,B,C,D,...))$
- Early materialization:
  - Retrieve positions with ‘a’ in column A: 2, 4, 5, 9, 25…
  - Retain only positions with ‘d’: 4, 9, ...
  - Lookup values in column B: B[4], B[9], …
Late Materialization (Sec. 4)

• What is it?
• Discuss $\Pi_B(\sigma_{A='a'} \land D='d'(R(A,B,C,D,...)))$
• Early materialization:
  – Retrieve positions with ‘a’ in column A: 2, 4, 5, 9, 25…
  – Retain only positions with ‘d’: 4, 9, …
  – Lookup values in column B: B[4], B[9], …
• Late materialization
  – Retrieve positions with ‘a’ in column A: 2, 4, 5, 9, 25…
Late Materialization (Sec. 4)

• What is it?
• Discuss $\Pi_B(\sigma_{A='a' \land D='d'}(R(A,B,C,D,\ldots)))$
• Early materialization:
  – Retrieve positions with ‘a’ in column A:  2, 4, 5, 9, 25…
  – Retain only positions with ‘d’:  4, 9, ...
  – Lookup values in column B:  B[4], B[9], ...
• Late materialization
  – Retrieve positions with ‘a’ in column A:  2, 4, 5, 9, 25…
  – Retrieve positions with ‘d’ in column D:  3, 4, 7, 9, 12,..
Late Materialization (Sec. 4)

• What is it?
• Discuss $\Pi_B(\sigma_{A='a'} \land D='d')(R(A,B,C,D,...))$

• Early materialization:
  – Retrieve positions with ‘a’ in column A: 2, 4, 5, 9, 25…
  – Retain only positions with ‘d’: 4, 9, ...
  – Lookup values in column B: B[4], B[9], ...

• Late materialization
  – Retrieve positions with ‘a’ in column A: 2, 4, 5, 9, 25…
  – Retrieve positions with ‘d’ in column D: 3, 4, 7, 9, 12, ...
  – Intersect: 4, 9, …
Late Materialization (Sec. 4)

- What is it?
- Discuss $\Pi_B(\sigma_{A='a' \land D='d'}(R(A,B,C,D,...)))$
- Early materialization:
  - Retrieve positions with ‘a’ in column A: 2, 4, 5, 9, 25…
  - Retain only positions with ‘d’: 4, 9, …
  - Lookup values in column B: B[4], B[9], …
- Late materialization
  - Retrieve positions with ‘a’ in column A: 2, 4, 5, 9, 25…
  - Retrieve positions with ‘d’ in column D: 3, 4, 7, 9, 12,..
  - Intersect: 4, 9, …
  - Lookup values in column B: B[4], B[9], …
Late Materialization (Sec. 4)

Ex: SELECT R.b from R where R.a=X and R.d=Y

Early materialization

Late materialization

Extract values
Jive Join (Sec. 4)

```
SELECT emp.age, dept.name
FROM emp, dept
WHERE emp.dept_id = dept.id
```
Jive Join (Sec. 4)

```
SELECT emp.age, dept.name
FROM emp, dept
WHERE emp.dept_id = dept.id
```
Jive Join (Sec. 4)

```
SELECT emp.age, dept.name
FROM emp, dept
WHERE emp.dept_id = dept.id
```

Tuple positions

One column will be out of order
Jive Join (Sec. 4)

```
SELECT emp.age, dept.name
FROM emp, dept
WHERE emp.dept_id = dept.id
```
### Jive Join (Sec. 4)

```sql
SELECT emp.age, dept.name
FROM emp, dept
WHERE emp.dept_id = dept.id
```

- **Tuple positions**: One column will be out of order.
- **emp.dept_id**
- **dept.id**
- **Add new indexes**
- **sort**
Jive Join (Sec. 4)

```
SELECT emp.age, dept.name
FROM emp, dept
WHERE emp.dept_id = dept.id
```

- emp.dept_id
- dept.id
- Tuple positions
- One column will be out of order
- dept.name

Add new indexes

Sort

Fetch dept.name
Jive Join (Sec. 4)

SELECT emp.age, dept.name
FROM emp, dept
WHERE emp.dept_id = dept.id

emp.dept_id  dept.id

Tuple positions

One column will be out of order

Add new indexes

Fetch dept.name
Late Materialization

```
select sum(R.a) from R, S
where R.c = S.b
    and 5<R.a<20 and 40<R.b<50
    and 30<S.a<40
```
select sum(R.a) from R, S
where R.c = S.b
and 5 < R.a < 20 and 40 < R.b < 50
and 30 < S.a < 40
Late Materialization

```
select sum(R.a) from R, S
where R.c = S.b
  and 5<R.a<20 and 40<R.b<50
  and 30<S.a<40
```
Late Materialization

\[
\text{select sum}(R.a) \text{ from } R, S \\
\text{where } R.c = S.b \\
\quad \text{and } 5 < R.a < 20 \text{ and } 40 < R.b < 50 \\
\quad \text{and } 30 < S.a < 40
\]
Late Materialization

```
select sum(R.a) from R, S
where R.c = S.b
and 5 < R.a < 20 and 40 < R.b < 50
and 30 < S.a < 40
```
Late Materialization

```
select sum(R.a) from R, S
where R.c = S.b
  and 5<R.a<20 and 40<R.b<50
  and 30<S.a<40
```
Late Materialization

\[
\text{select sum(R.a) from R, S where R.c = S.b and 5<} R.a< 20 \text{ and 40<} R.b< 50 \text{ and 30<} S.a< 40
\]
Late Materialization

select sum(R.a) from R, S
where R.c = S.b
    and 5<R.a<20 and 40<R.b<50
    and 30<S.a<40
Late Materialization

```
select sum(R.a) from R, S
where R.c = S.b
  and 5 <= R.a <= 20
  and 40 <= R.b <= 50
  and 30 <= S.a <= 40
```
More Details

• Sort columns according to some criterion
  – Helps with range queries on that column
  – Helps compressing that column
  – But need to sort all the other columns the same way

• Create additional (redundant) "views", called “projections”, by sorting on different columns
Data organized into *projections*:
Sorted subsets of the attributes
Each table has one super projection
Includes all table attributes

**Original Data**

<table>
<thead>
<tr>
<th>sale_id</th>
<th>cid</th>
<th>cust</th>
<th>date</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>Andrew</td>
<td>01/01/06</td>
<td>$100</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>Chuck</td>
<td>01/05/06</td>
<td>$98</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td>Nga</td>
<td>01/02/06</td>
<td>$90</td>
</tr>
<tr>
<td>4</td>
<td>28</td>
<td>Matt</td>
<td>01/03/06</td>
<td>$101</td>
</tr>
<tr>
<td>5</td>
<td>89</td>
<td>Ben</td>
<td>01/01/06</td>
<td>$103</td>
</tr>
<tr>
<td>1000</td>
<td>89</td>
<td>Ben</td>
<td>01/02/06</td>
<td>$103</td>
</tr>
<tr>
<td>1001</td>
<td>11</td>
<td>Andrew</td>
<td>01/03/06</td>
<td>$95</td>
</tr>
</tbody>
</table>

*Super projection sorted by date*
*Non-super projection containing only (cust, price) attributes, sorted by cust*

From: The Vertica Analytic Database: CStore 7 Years Later. Lamb et. Al. VLDB’12
Parallel Processing

• Segment data horizontally across nodes
• Organize as column store on each node

### Original Data

<table>
<thead>
<tr>
<th>sale_id</th>
<th>cid</th>
<th>cust</th>
<th>date</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>Andrew</td>
<td>01/01/06</td>
<td>$100</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>Chuck</td>
<td>01/05/06</td>
<td>$98</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td>Nga</td>
<td>01/02/06</td>
<td>$90</td>
</tr>
<tr>
<td>4</td>
<td>28</td>
<td>Matt</td>
<td>01/03/06</td>
<td>$101</td>
</tr>
<tr>
<td>5</td>
<td>89</td>
<td>Ben</td>
<td>01/01/06</td>
<td>$103</td>
</tr>
<tr>
<td>1000</td>
<td>89</td>
<td>Ben</td>
<td>01/02/06</td>
<td>$103</td>
</tr>
<tr>
<td>1001</td>
<td>11</td>
<td>Andrew</td>
<td>01/03/06</td>
<td>$95</td>
</tr>
</tbody>
</table>

### Super projection

- Sorted by date & segmented by hash(sale_id)

### Non-super projection

- Containing only (cust, price) attributes, sorted by cust, segmented by hash(cust)
Vertica Data Model Details

Segmented on several nodes

Node 1

Node 2
Vertica Data Partitioning

• Cross-node partitioning called “segmentation”
  – Hash-partitioning
  – Other expression

• Each node assigned multiple local segments
  – To facilitate elasticity
  – Enables moving segments as cluster size changes

• Can also replicate all tuples in projection
Vertica Intra-Node Partitioning

• C-store proposed intra-node data partitioning
  – Similar to other parallel DBMS such as Teradata
• In contrast, Vertica divides each on-disk structure into logical regions at runtime and processing the regions in parallel
• Vertica also supports explicit data partitioning
  – Partitions segments within nodes into smaller pieces
  – CREATE TABLE … PARTITION BY <expr>
  – Benefits:
    • Fast deletion
    • Pruning of partitions during query execution
Segmentation = horizontal partitioning across nodes
→ Each projection has own segmentation
→ More segments than nodes for elasticity

Partition = horizontal within a node
→ Same partition for all projections & nodes

ROS = Read Optimized Store
Each column’s data within its ROS container is stored as a single file
→ Total of 28 files of user data
Updates

• What is the issue?

• How does the paper address this?
Updates

• What is the issue?
  – Updates in a sorted column require reordering of the entire column, and the other columns as well

• How does the paper address this?
Updates

• What is the issue?
  – Updates in a sorted column require reordering of the entire column, and the other columns as well

• How does the paper address this?
  – Update to Write Optimized Store (WOS)
  – Queries on Read Optimized Store (ROS)
C-Store/Vertica Design

Writeable Store (WS)

Tuple Mover

Read-optimized Store (RS)

Uncompressed row-store

compressed column store

Figure 1. Architecture of C-Store

From: C-Store: A Column-oriented DBMS. Stonebraker et. Al. VLDB’05
Read and Write Optimized Stores

• **Write Optimized Store (WOS)**
  – In memory data: buffer delete/insert/update operations
  – Column vs row does not matter

• **Tuples never modified in place**
  – Use “delete vector” to track deleted tuples
  – Eventually removed by tuple mover during ROS merge

• **Tuple mover**
  – Move between WOS and ROS
  – When moving tuples out, creates a new ROS container
  – Merges ROS files together
    • Better compression & faster processing (fewer files to merge)
Read and Write Optimized Stores

• **Read Optimized Store (ROS)**
  – Multiple ROS containers
  – Stored on standard file system
  – Logically contains some number of complete tuples sorted by the *projection’s* sort order, stored as a pair of files per column: position index & data
    • The position index = only metadata per disk block
  – Column files may be independently retrieved
Final Thoughts

Simulating a Column-Store in a Row-Store DBMS:

• **Vertical partitioning**
  – Two-column tables: (key, attribute)

• **Index-only plans**
  – Create a B+ tree index on each attribute
  – Answer queries using indexes only, without reading actual data

• **Materialized views**
  – Each view contains a subset of columns
Recap of DATA 516
Recap of DATA 516

• Relational model, SQL
• Query execution/optimization
• Query optimization
• Spark, MapReduce
• Parallel Query Evaluation
• Graphs, Datalog
• Column Stores
Next, your turn: Projects!