Big Data for Data Science

SQL on Big Data
THE DEBATE: DATABASE SYSTEMS VS MAPREDUCE
A major step backwards?

- MapReduce is a step backward in database access
  - Schemas are good
  - Separation of the schema from the application is good
  - High-level access languages are good
- MapReduce is poor implementation
  - Brute force and only brute force (no indexes, for example)
- MapReduce is not novel
- MapReduce is missing features
  - Bulk loader, indexing, updates, transactions…
- MapReduce is incompatible with DMBS tools
Known and unknown unknowns

- Databases only help if you know what questions to ask
  - “Known unknowns”
- What’s if you don’t know what you’re looking for?
  - “Unknown unknowns”
ETL: redux

- Often, with noisy datasets, ETL *is* the analysis!
- Note that ETL necessarily involves brute force data scans
- L, then E and T?
Structure of Hadoop warehouses

Don’t normalize!

Source: Wikipedia (Star Schema)
Relational databases vs. MapReduce

- Relational databases:
  - Multipurpose: analysis and transactions; batch and interactive
  - Data integrity via ACID transactions
  - Lots of tools in software ecosystem (for ingesting, reporting, etc.)
  - Supports SQL (and SQL integration, e.g., JDBC)
  - Automatic SQL query optimization

- MapReduce (Hadoop):
  - Designed for large clusters, fault tolerant
  - Data is accessed in “native format”
  - Supports many query languages
  - Programmers retain control over performance
  - Open source

Source: O'Reilly Blog post by Joseph Hellerstein (11/19/2008)
Philosophical differences

• Parallel relational databases
  – Schema on write
  – Failures are relatively infrequent
  – “Possessive” of data
  – Mostly proprietary

• MapReduce
  – Schema on read
  – Failures are relatively common
  – In situ data processing
  – Open source
MapReduce vs. RDBMS: grep

SELECT * FROM Data WHERE field LIKE 'XYZ';

Source: Pavlo et al. (2009) A Comparison of Approaches to Large-Scale Data Analysis. SIGMOD.
MapReduce vs. RDBMS: select

SELECT pageURL, pageRank
FROM Rankings WHERE pageRank > X;

Source: Pavlo et al. (2009) A Comparison of Approaches to Large-Scale Data Analysis. SIGMOD.
MapReduce vs. RDBMS: aggregation

```
SELECT sourceIP, SUM(adRevenue)
FROM UserVisits GROUP BY sourceIP;
```

Source: Pavlo et al. (2009) A Comparison of Approaches to Large-Scale Data Analysis. SIGMOD.
MapReduce vs. RDBMS: join

Source: Pavlo et al. (2009) A Comparison of Approaches to Large-Scale Data Analysis. SIGMOD.
Why?

• Schemas are a good idea
  – Parsing fields out of flat text files is slow
  – Schemas define a contract, decoupling logical from physical
• Schemas allow for building efficient auxiliary structures
  – Value indexes, join indexes, etc.
• Relational algorithms have been optimised for the underlying system
  – The system itself has complete control of performance-critical decisions
  – Storage layout, choice of algorithm, order of execution, etc.
Storage layout: row vs. column stores

Row store

Column store
Storage layout: row vs. column stores

• Row stores
  – Easy to modify a record
  – Might read unnecessary data when processing

• Column stores
  – Only read necessary data when processing
  – Tuple writes require multiple accesses
Advantages of column stores

• Read efficiency
  – If only need to access a few columns, no need to drag around the rest of the values
• Better compression
  – Repeated values appear more frequently in a column than repeated rows appear
• Vectorised processing
  – Leveraging CPU architecture-level support
• Opportunities to operate directly on compressed data
  – For instance, when evaluating a selection; or when projecting a column
Why not in Hadoop?

No reason why not

RCFile

Row Group

HDFS Block

Row Group 1

Row Group 2

Row Group n

Source: He et al. (2011) RCFile: A Fast and Space-Efficient Data Placement Structure in MapReduce-based Warehouse Systems. ICDE.
Some small steps forward

• MapReduce is a step backward in database access:
  – Schemas are good ✔
  – Separation of the schema from the application is good ✔
  – High-level access languages are good ?

• MapReduce is poor implementation
  – Brute force and only brute force (no indexes, for example) ✔

• MapReduce is not novel

• MapReduce is missing features
  – Bulk loader, indexing, updates, transactions… ?

• MapReduce is incompatible with DMBS tools

Source: Blog post by DeWitt and Stonebraker
MODERN SQL-ON-HADOOP SYSTEMS
# Analytical Database Systems

<table>
<thead>
<tr>
<th>Parallel (MPP):</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Teradata</td>
<td>Paraccel</td>
</tr>
<tr>
<td>Pivotal</td>
<td></td>
</tr>
<tr>
<td>Vertica</td>
<td>Redshift</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Oracle (IMM)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2-BLU</td>
<td>Netteza</td>
</tr>
<tr>
<td>SQLserver (columnstore)</td>
<td>InfoBright</td>
</tr>
<tr>
<td>Vectorwise</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>open source:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>LucidDB</td>
</tr>
<tr>
<td>MonetDB</td>
<td></td>
</tr>
</tbody>
</table>
SQL-on-Hadoop Systems

Open Source:
• Hive (HortonWorks)
• Impala (Cloudera)
• Drill (MapR)
• Presto (Facebook)

Commercial:
• HAWQ (Pivotal)
• Vortex (Actian)
• Vertica Hadoop (HP)
• BigQuery (IBM)
• DataBricks
• Splice Machine
• CitusData
• InfiniDB Hadoop
Analytical DB engines for Hadoop

**Storage**
- **columnar storage** + compression
- table partitioning / distribution
- exploiting correlated data

**Query-Processor**
- CPU-efficient query engine (vectorized or JIT codegen)
- many-core ready
- rich SQL (+authorization+..)

**System**
- batch update infrastructure
- scaling with multiple nodes
- MetaStore & file formats
- YARN & elasticity
Columnar Storage

row-store

+ easy to add/modify a record
- might read in unnecessary data

column-store

+ only need to read in relevant data
- tuple writes require multiple accesses

Rightarrow suitable for read-mostly, read-intensive, large data repositories
Analytical DB engines for Hadoop

**storage**
- columnar storage + compression
- table partitioning / distribution
- exploiting correlated data

**query-processor**
- CPU-efficient query engine (vectorized or JIT codegen)
- many-core ready
- rich SQL (+authorization+..)

**system**
- batch update infrastructure
- scaling with multiple nodes
- MetaStore & file formats
- YARN & elasticity
Columnar Compression

• Trades I/O for CPU
  – A winning proposition currently
  – Even trading RAM bandwidth for CPU wins
    • 64 core machines starved for RAM bandwidth

• Additional column-store synergy:
  – Column store: data of the same distribution close together
    • Better compression rates
    • Generic compression (gzip) vs Domain-aware compression
  – Synergy with vectorized processing (see later)
    compress/decompress/execution, SIMD
  – Can use extra space to store multiple copies of data in different sort orders (see later)
Run-length Encoding

<table>
<thead>
<tr>
<th>Quarter ID</th>
<th>Product ID</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Q1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>Q1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Q1</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>Q1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Q1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Q1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Q2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Q2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Q2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Q2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quarter</th>
<th>(value, start_pos, run_length)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Q1, 1, 300)</td>
<td></td>
</tr>
<tr>
<td>(Q2, 301, 350)</td>
<td></td>
</tr>
<tr>
<td>(Q3, 651, 500)</td>
<td></td>
</tr>
<tr>
<td>(Q4, 1151, 600)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quarter ID</th>
<th>(value, start_pos, run_length)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 1, 5)</td>
<td></td>
</tr>
<tr>
<td>(2, 6, 2)</td>
<td></td>
</tr>
<tr>
<td>(1, 301, 3)</td>
<td></td>
</tr>
<tr>
<td>(2, 304, 1)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

...
Bitmap Encoding

- For each unique value, $v$, in column $c$, create bit-vector $b$
  - $b[i] = 1$ if $c[i] = v$
- Good for columns with few unique values
- Each bit-vector can be further compressed if sparse

<table>
<thead>
<tr>
<th>Product ID</th>
<th>ID: 1</th>
<th>ID: 2</th>
<th>ID: 3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

“Integrating Compression and Execution in Column-Oriented Database Systems” Abadi et al., SIGMOD ’06
Dictionary Encoding

- For each unique value create dictionary entry
- Dictionary can be per-block or per-column
- Column-stores have the advantage that dictionary entries may encode multiple values at once

```
Quarter
Q1    Q2
Q4    Q1
Q3    Q1
Q1    Q2
Q4    Q3
Q3...
```

```
Quarter
0
1
3
0
2
0
0
1
3
2
2
```

Dictionary
```
0: Q1
1: Q2
2: Q3
3: Q4
```

“Integrating Compression and Execution in Column-Oriented Database Systems” Abadi et. al, SIGMOD ’06

www.cwi.nl/~boncz/bigdatacourse
Differential Encoding

- Encodes values as $b$ bit offset from previous value
- Special escape code (just like frame of reference encoding) indicates a difference larger than can be stored in $b$ bits
  - After escape code, original (uncompressed) value is written
- Performs well on columns containing increasing/decreasing sequences
  - inverted lists
  - timestamps
  - object IDs
  - sorted / clustered columns

“Improved Word-Aligned Binary Compression for Text Indexing” Ahn, Moffat, TKDE’06

Exceptions (there are better ways to deal with exceptions)

2 bits per value
Heavy-Weight Compression Schemes

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Decompression Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>BZIP</td>
<td>10 MB/s</td>
</tr>
<tr>
<td>ZLIB</td>
<td>80 MB/s</td>
</tr>
<tr>
<td>LZO</td>
<td>300 MB/s</td>
</tr>
</tbody>
</table>

- Modern disks (SSDs) can achieve > 1GB/s
- 1/3 CPU for decompression → 3GB/s needed

→ Lightweight compression schemes are better
→ Even better: operate directly on compressed data
Operating Directly on Compressed Data

Examples

- \( \text{SUM}_i(\text{rle-compressed column}[i]) \rightarrow \text{SUM}_g(\text{count}[g] \times \text{value}[g]) \)
- \((\text{country} == \text{"Asia"}) \rightarrow \text{countryCode} == 6\)

\text{strncpy} \quad \text{SIMD}

 Benefits:

- I/O - CPU tradeoff is no longer a tradeoff (CPU also gets improved)
- Reduces memory–CPU bandwidth requirements
- Opens up possibility of operating on multiple records at once
Analytical DB engines for Hadoop

**storage**
- columnar storage + compression
- table partitioning / distribution
- exploiting correlated data

**query-processor**
- CPU-efficient query engine (vectorized or JIT codegen)
- many-core ready
- rich SQL (+authorization+..)

**system**
- batch update infrastructure
- scaling with multiple nodes
- MetaStore & file formats
- YARN & elasticity

www.cwi.nl/~boncz/bigdatacourse
Table Partitioning and Distribution

- Data is spread based on a Key
  - Functions: Hash, Range, List
- “Distribution”
  - Goal: parallelism
    - Give each compute node a piece of the data
    - Each query has work on every piece (keep everyone busy)
- “Partitioning”
  - Goal: data lifecycle management
    - Data warehouse e.g. keeps last six months
    - Every night: load one new day, drop the oldest partition
  - Goal: improve access pattern
    - When querying for May, drop Q1, Q3, Q4 (“partition pruning”)

Which kind of function would you use for which method?
Data Placement in Hadoop

- Each node writes the partitions it owns
  - Where does the data end up, really?
- HDFS default block placement strategy:
  - Node that initiates writes gets first copy
  - 2nd copy on the same rack
  - 3rd copy on a different rack
- Rows from the same record should on the same node
  - Not entirely trivial in column stores
    - Column partitions should be co-located
    - Simple solution:
      - Put all columns together in one file (RCFILE, ORCFILE, Parquet)
  - Complex solution:
    - Replace the default HDFS block placement strategy by a custom one

www.cwi.nl/~boncz/bigdatatable
Popular File Formats in Hadoop

- Good old CSV
  - Textual, easy to parse (but slow), better compress it!

- Sequence Files
  - Binary data, faster to process

- RCfile
  - Hive first attempt at column-store

- ORCfile
  - Columnar compression, MinMax

- Parquet
  - Proposed by Twitter and Cloudera Impala
  - Like ORCfile, no MinMax
Example: Parquet Format

On-disk, Parquet data is in binary form using its own formally-specified columnar file format.

A row group stores all the column values for a range of rows in a columnar layout.

A column chunk contains all the values for an individual column in the row group.

The footer contains schema details, object model metadata and metadata about the row groups and columns.

Shaded boxes are part of the Parquet project.
Example: Parquet Format

Table Format
Analytical DB engines for Hadoop

**storage**
- columnar storage + compression
- table partitioning / distribution
- exploiting correlated data

**query-processor**
- CPU-efficient query engine (vectorized or JIT codegen)
- many-core ready
- rich SQL (+authorization+..)

**system**
- batch update infrastructure
- scaling with multiple nodes
- MetaStore & file formats
- YARN & elasticity
Exploiting Natural Order

- Data is often naturally ordered
  - very often, on date
- Data is often correlated
  - orderdate/paydate/shipdate
  - marketing campaigns/date
  - ..correlation is everywhere
    ..hard to predict

Zone Maps
- Very sparse index
- Keeps MinMax for every column
- Cheap to maintain
- Just widen bounds on each modification

---

**Q:** key BETWEEN 13 AND 15?

**Q:** acctno BETWEEN 150 AND 200?

---

**Accounts**

<table>
<thead>
<tr>
<th>KEY</th>
<th>acctno</th>
<th>name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>019</td>
<td>Isabella</td>
<td>269.38</td>
</tr>
<tr>
<td>01</td>
<td>038</td>
<td>Jackson</td>
<td>914.11</td>
</tr>
<tr>
<td>02</td>
<td>072</td>
<td>Lucas</td>
<td>346.61</td>
</tr>
<tr>
<td>03</td>
<td>156</td>
<td>Sophia</td>
<td>266.55</td>
</tr>
<tr>
<td>04</td>
<td>153</td>
<td>Mason</td>
<td>850.90</td>
</tr>
<tr>
<td>05</td>
<td>282</td>
<td>Ethan</td>
<td>521.60</td>
</tr>
<tr>
<td>06</td>
<td>389</td>
<td>Emily</td>
<td>647.38</td>
</tr>
<tr>
<td>07</td>
<td>314</td>
<td>Lily</td>
<td>119.40</td>
</tr>
<tr>
<td>08</td>
<td>332</td>
<td>Chloe</td>
<td>526.08</td>
</tr>
<tr>
<td>09</td>
<td>302</td>
<td>Emma</td>
<td>497.19</td>
</tr>
<tr>
<td>10</td>
<td>533</td>
<td>Aiden</td>
<td>22.03</td>
</tr>
<tr>
<td>11</td>
<td>592</td>
<td>Ava</td>
<td>140.67</td>
</tr>
<tr>
<td>12</td>
<td>808</td>
<td>Mia</td>
<td>383.69</td>
</tr>
<tr>
<td>13</td>
<td>896</td>
<td>Jacob</td>
<td>899.41</td>
</tr>
</tbody>
</table>

---

**Accounts,MinMax**

<table>
<thead>
<tr>
<th>zone</th>
<th>KEY</th>
<th>acctno</th>
<th>name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>019</td>
<td>Isabella</td>
<td>269.38</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>038</td>
<td>Jackson</td>
<td>914.11</td>
</tr>
<tr>
<td></td>
<td>02</td>
<td>072</td>
<td>Lucas</td>
<td>346.61</td>
</tr>
<tr>
<td></td>
<td>03</td>
<td>156</td>
<td>Sophia</td>
<td>266.55</td>
</tr>
<tr>
<td></td>
<td>04</td>
<td>153</td>
<td>Mason</td>
<td>850.90</td>
</tr>
<tr>
<td></td>
<td>05</td>
<td>282</td>
<td>Ethan</td>
<td>521.60</td>
</tr>
<tr>
<td></td>
<td>06</td>
<td>389</td>
<td>Emily</td>
<td>647.38</td>
</tr>
<tr>
<td></td>
<td>07</td>
<td>314</td>
<td>Lily</td>
<td>119.40</td>
</tr>
<tr>
<td></td>
<td>08</td>
<td>332</td>
<td>Chloe</td>
<td>526.08</td>
</tr>
<tr>
<td></td>
<td>09</td>
<td>302</td>
<td>Emma</td>
<td>497.19</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>533</td>
<td>Aiden</td>
<td>22.03</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>592</td>
<td>Ava</td>
<td>140.67</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>808</td>
<td>Mia</td>
<td>383.69</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>896</td>
<td>Jacob</td>
<td>899.41</td>
</tr>
</tbody>
</table>

---

**Q:** acctno BETWEEN 150 AND 200?
Analytical DB engines for Hadoop

**storage**
- columnar storage + compression
- table partitioning / distribution
- exploiting correlated data

**query-processor**
- CPU-efficient query engine (*vectorized* or JIT codegen)
- many-core ready
- rich SQL (+authorization+..)

**system**
- batch update infrastructure
- scaling with multiple nodes
- MetaStore & file formats
- YARN & elasticity

[www.cwi.nl/~boncz/bigdatacourse](http://www.cwi.nl/~boncz/bigdatacourse)
DBMS Computational Efficiency?

TPC-H 1GB, query 1

• selects 98% of fact table, computes net prices and aggregates all

• Results:
  - C program: ?
  - MySQL: 26.2s
  - DBMS “X”: 28.1s

“MonetDB/X100: Hyper-Pipelining Query Execution” Boncz, Zukowski, Nes, CIDR’05
DBMS Computational Efficiency?

TPC-H 1GB, query 1

• selects 98% of fact table, computes net prices and aggregates all

• Results:
  – C program: 0.2s
  – MySQL: 26.2s
  – DBMS “X”: 28.1s

“MonetDB/X100: Hyper-Pipelining Query Execution” Boncz, Zukowski, Nes, CIDR’05
How Do Query Engines Work?

```
SELECT id, name 
    (age-30)*50 AS bonus
FROM employee
WHERE age > 30
```
How Do Query Engines Work?

Operators

- **Iterator interface**
  - `open()`
  - `-next()`: tuple
  - `-close()`

```
next()

102  ivan  350

next()

PROJECT

next()

SELECT

next()

SCAN
```
How Do Query Engines Work?

Primitives

Provide computational functionality

All arithmetic allowed in expressions, e.g. Multiplication

$$7 \times 50$$

$$\text{mult}(\text{int},\text{int}) \Rightarrow \text{int}$$

www.cwi.nl/~boncz/bigdatacourse
Observations:

“Vectorized In Cache Processing”

vector = array of ~100

processed in a tight loop

CPU cache Resident

```
next()

102  ivan  350
104  peggy  750

101  alice  22
102  ivan  37
104  peggy  45
105  victor  25

37  45  25
750 750 750

next()

102  ivan  37
104  peggy  45

next()

101  alice  22
102  ivan  37
104  peggy  45
105  victor  25

next()

101  alice  22
102  ivan  37
104  peggy  45
105  victor  25

-30 * 50
>30 ?

350
750
```

"MonetDB/X100: Hyper-Pipelining Query Execution"
Boncz, Zukowski, Nes, CIDR’05
Observations:

next() called much less often \(\rightarrow\) more time spent in primitives less in overhead

primitive calls process an array of values in a loop:

- CPU Efficiency depends on “nice” code
  - out-of-order execution
  - few dependencies (control, data)
  - compiler support

- Compilers like simple loops over arrays
  - loop-pipelining
  - automatic SIMD

---

<table>
<thead>
<tr>
<th>next()</th>
<th>SCAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>next()</th>
<th>PROJECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>ivan</td>
</tr>
<tr>
<td>104</td>
<td>peggy</td>
</tr>
<tr>
<td></td>
<td>350</td>
</tr>
<tr>
<td></td>
<td>750</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>next()</th>
<th>SELECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>alice</td>
</tr>
<tr>
<td>102</td>
<td>ivan</td>
</tr>
<tr>
<td>104</td>
<td>peggy</td>
</tr>
<tr>
<td>105</td>
<td>victor</td>
</tr>
<tr>
<td>22</td>
<td>FALSE</td>
</tr>
<tr>
<td>37</td>
<td>TRUE</td>
</tr>
<tr>
<td>45</td>
<td>TRUE</td>
</tr>
<tr>
<td>25</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

- 30 \* 50
- > 30 ?

---

“MonetDB/X100: Hyper-Pipelining Query Execution”
Boncz, Zukowski, Nes, CIDR’05

www.cwi.nl/~boncz/bigdatacourse
Observations:

next() called much less often → more time spent in primitives less in overhead

CPU Efficiency depends on “nice” code
- out-of-order execution
- few dependencies (control, data)
- compiler support

Compilers like simple loops over arrays
- loop-pipelining
- automatic SIMD

```
for(i=0; i<n; i++)
    res[i] = (col[i] > x)
```

```
for(i=0; i<n; i++)
    res[i] = (col[i] - x)
```

```
for(i=0; i<n; i++)
    res[i] = (col[i] * x)
```
Varying the Vector size

MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’05

Less and less iterator.next() and primitive function calls (“interpretation overhead”)
Vectors start to exceed the CPU cache, causing additional memory traffic.
Systems That Use Vectorization

- Actian Vortex (Vectorwise-on-Hadoop)
- Hive, Drill

Vectorization

- Drill operates on more than one record at a time
  - Word-sized manipulations
  - SIMD instructions
    • GCC, LLVM and JVM all do various optimizations automatically
  - Manually code algorithms
- Logical Vectorization
  - Bitmaps allow lightning fast null-checks
  - Avoid branching to speed CPU pipeline
Analytical DB engines for Hadoop

**storage**
- columnar storage + compression
- table partitioning / distribution
- exploiting correlated data

**query-processor**
- CPU-efficient query engine (vectorized or JIT codegen)
- many-core ready
- rich SQL (+authorization+..)

**system**
- batch update infrastructure
- scaling with multiple nodes
- MetaStore & file formats
- YARN & elasticity

www.cwi.nl/~boncz/bigdatacourse
Analytical DB engines for Hadoop

**storage**
- columnar storage + compression
- table partitioning / distribution
- exploiting correlated data

**query-processor**
- CPU-efficient query engine (vectorized or JIT codegen)
- many-core ready
- analytical SQL (windowing)

**system**
- batch update infrastructure
- scaling with multiple nodes
- MetaStore & file formats
- YARN & elasticity
Batch Update Infrastructure (Vertica)

Challenge: hard to update columnar compressed data

Trickle Load

**Write Optimized Store (WOS)**
- Memory based
- Unsorted / Uncompressed
- Segmented
- Low latency / Small quick inserts

**Read Optimized Store (ROS)**
- On disk
- Sorted / Compressed
- Segmented
- Large data loaded direct

(A B C | A)

TUPLE MOVER
Asynchronous Data Transfer
Batch Update Infrastructure (Hive)

Challenge: HDFS read-only + large block size

Merge During Query Processing
Analytical DB engines for Hadoop

**storage**
- columnar storage + compression
- table partitioning / distribution
- exploiting correlated data

**query-processor**
- CPU-efficient query engine (vectorized or JIT codegen)
- many-core ready
- rich SQL (+authorization+..)

**system**
- batch update infrastructure
- **scaling with multiple nodes**
- MetaStore & file formats
- YARN & elasticity

www.cwi.nl/~boncz/bigdatacourse
Analytical DB engines for Hadoop

**storage**
- columnar storage + compression
- table partitioning / distribution
- exploiting correlated data

**query-processor**
- CPU-efficient query engine (vectorized or JIT codegen)
- many-core ready
- rich SQL (+authorization+..)

**system**
- batch update infrastructure
- scaling with multiple nodes
- MetaStore & file formats
- YARN & elasticity

www.cwi.nl/~boncz/bigdatacourse
YARN possibilities and limitations

Containers are used to assign:

• cores
• RAM

Limitations:

• no support for disk I/O, network (thrashing still possible)
• Long-running systems (e.g. DBMS) may want to adjust cores and RAM over time depending on workload ➔ “elasticity”
Conclusion

• SQL-on-Hadoop area is very active
  – many open-source and commercial initiatives

• There are many design dimensions
  – All design dimensions of analytical database systems
    • Column storage, compression, vectorization/JIT, MinMax
      pushdown, partitioning, parallel scaling, update handling, SQL99,
      ODBC/JDBC APIs, authorization
  – Hadoop design dimensions
    • HCatalog support, reading from and getting read from other
      Hadoop tools (/writing to..), file format support, HDFS locality,
      YARN integration
SQL IN THE CLOUD
- BUT NOT ON HADOOP
Amazon Redshift

- Cloud version of ParAccel, a parallel database
  - ParAccel is hard to manage, maintain
  - Redshift invested in simplying management, using web interface
    - No knobs, kind of elastics, User Defined Functions (python)
    - Highly performant, but storage more expensive than S3 (local disks)
Snowflake

- Brand-new, from-scratch system that works in AWS – RedShift competitor
- Stores data on S3 (cheap!) but caches it in local disks for performance
- Highly elastic, supports UDFs using JavaScript, table snapshots (“clone table”)
- Puts JSON documents in automatically recognized table format (queryable)

Snowflake Multi-cluster Shared-data Architecture

- All data in one place
- Dynamically combine storage and compute
- Independently size storage and compute
- No unload / reload to shut off compute
- Every compute cluster can access any data