Query processing on raw files

Vítor Uwe Reus
Outline

1. Introduction
2. Adaptive Indexing
3. Hybrid MapReduce
4. NoDB
5. Summary
Outline

1. Introduction
2. Adaptive Indexing
3. Hybrid MapReduce
4. NoDB
5. Summary
Raw Files

Information Storing

Sometimes human-readable, open format

Not physically optimized for querying

 Might be useful in some cases
Big Data

Traditional DBMS may not be a good option

Internet-scale business

Scientific data
The fourth paradigm

For scientific discovery

Experimental

Theoretical

Computational (simulations)

Data driven
Interoperability
Interoperability

Information interoperability

Application interoperability
Human-sourced Information
How to query raw files?
State of the art

Raw file as storage

A-priori loading
Raw file parsing

AWK

Oracle external table

MySQL CSV engine

MapReduce

Read entire data all times

No indexing features
A-priori loading

Load into a DBMS and then query
  Benefit from indexes
Time
Labor intensive
  Loading scripts, schemas
Data duplication
  Big data
Versioning
Load time vs Query time

DBMS

Ideal

RAW
Hybrid querying techniques
Outline

1. Introduction
2. Adaptive Indexing
3. Hybrid MapReduce
4. NoDB
5. Summary
Adaptive indexing

Automatic tuning based on workload

Keep an auxiliary structure

Can benefit raw file parsing

Database Cracking

Adaptive Merging
Database cracking

Physical reorganization of columns

Implemented on MonetDB

A column store, but can be generalized (raw)
**Database Cracking**

<table>
<thead>
<tr>
<th>Column A</th>
<th>Cracker column of A after query Q1</th>
<th>Cracker column of A after query Q2</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>

Q1:
select * from R
where R.A > 10
and R.A < 14

Q2:
select * from R
where R.A > 7
and R.A >= 16

**Cracking a column**

Piece 1: $A \leq 10$
Piece 2: $7 < A \leq 10$
Piece 3: $10 < A < 14$
Piece 4: $14 \leq A \leq 16$
Piece 5: $16 < A$
Database Cracking

Column A $\rightarrow$ Copy to cracker column $A_{CRK}$

AVL tree indexing

Refinement
Tuple reconstruction

Fast if columns are in same order

Cracking compromises original positions

Cracker columns: Value selection

Original columns: Tuple reconstruction
Adaptive merging

Incremental index creation as in cracking

Partitioned B-trees

Focus on merging instead of partitioning
Merging vs cracking

Typical result of merging compared to cracking

*In this case, all queries focus on the same $10^6$ keys in the center of the domain*
## Merging vs cracking

<table>
<thead>
<tr>
<th></th>
<th>Cracking</th>
<th>Merging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Converge</td>
<td>Stable</td>
<td>Faster</td>
</tr>
<tr>
<td>Storage</td>
<td>AVL</td>
<td>B-Tree</td>
</tr>
<tr>
<td>Data is</td>
<td>Partitioned</td>
<td>...and Sorted</td>
</tr>
<tr>
<td>as in..</td>
<td>Quick Sort</td>
<td>Merge Sort</td>
</tr>
</tbody>
</table>
Outline

1. Introduction
2. Adaptive Indexing
3. Hybrid MapReduce
4. NoDB
5. Summary
Hybrid MapReduce

<table>
<thead>
<tr>
<th></th>
<th>Scalability*</th>
<th>High Performance**</th>
</tr>
</thead>
<tbody>
<tr>
<td>MapReduce</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>Parallel Databases</td>
<td>✗</td>
<td>✔️</td>
</tr>
<tr>
<td><strong>What is needed</strong></td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

* 1000s of nodes
** Queries on structured data
HadoopDB

MapReduce using a DBMS instead of HDFS
SMS Planner

SQL → MapReduce → SQL

HIVE

Hadoop

MySQL
Hive query processor

1. Convert HiveQL query to AST
2. Get schema from catalog
3. Create a Query Plan
4. Optimize
5. Converted plan to one or more MR Jobs
**SMS Planner**

1. Convert HiveQL query to AST
   - Update Catalog with DB information
2. Get schema from catalog
3. Create a Query Plan
4. Optimize
   - Reconstruc some SQL to push it to the DB
5. Converted plan to one or more MR Jobs
SELECT YEAR(saleDate), SUM(revenue) FROM sales GROUP BY YEAR(saleDate);
HadoopDB Performance

**Group By**

- 2,500,000 unique groups
- over 20gb of data

**Join**

- 134,000 joined records
- over 20gb of data
HadoopDB loading times

Load Grep (0.5GB/node)

Load UserVisits (20GB/node)
HadoopDB

- Good performance
- Scalable
- Fault tolerant
- Heterogeneous node compatible
- Make any DBMS a distributed system

- Data Loader: All *a-priori* loading problems
Invisible loading

Load DBMS with data from Hadoop at run-time

Invisibility objective

Minmal human effort

Minimal increase in response time

Use a DBMS as a cache for the raw data
Invisible loading

Use code for tuple parsing and extraction to invisibly load the parsed tuples into a DBMS
Invisible loading

On next data access, it can be read from DBMS
Invisible loading: Parser

Parser extends inputFormat

getAttribute(int index)

Code for tuple parsing and extraction

Map takes a Parser as input
Invisible loading

- Incremental data reorganization
- Almost no overhead on MR Jobs
- Optimizes future access speeds
- Data duplication (No GC)
NoDB

New DBMS paradigm

Do not require data loading

Maintains feature set of modern DBMS

Replaces physical storage with raw files
PostgresRaw

NoDB Implementation

Replaces TableScan Operator

CSV Files

Optimizations
PostgresRaw Optimizations

Selective...
   a. Tokenizing
   b. Parsing
   c. Tuple formation

Indexing

Auto Tuning

Caching

Statistics
a. Selective tokenizing

111;222;"third";garbage;...

Supposing we want attributes 1 and 3

We can stop tokenizing at the third

Saves CPU time
b. Selective parsing

111;222;"third";garbage;...

In memory:

111  6F
222  32 32 32
"third"  74 68 69 72 64

Also: delayed parsing
c. Selective tuple formation

111;222;"third";garbage;...

(111, "third")

Final tuple containing only attributes 1 and 3

CPU bound
## Indexing

<table>
<thead>
<tr>
<th>Year</th>
<th>Make</th>
<th>Model</th>
<th>Liters</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td>BMW;</td>
<td>E89;</td>
<td>2.34</td>
</tr>
<tr>
<td>2011</td>
<td>Mercedes</td>
<td>SLS;</td>
<td>2</td>
</tr>
</tbody>
</table>

Looks nice :)
Indexing

NOT :(

Year;Make;Model;Liters
1997;BMW;E89;2,34
2011;Mercedes;SLS;2

Sequentially reading each time is not an option

Solution

Keep an index of the already used attributes
Skip file reading to this positions
Indexing

Positional Map
Dynamically created according to queries

<table>
<thead>
<tr>
<th>Tuple 1</th>
<th>Tuple 2</th>
<th>Tuple 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute 1</td>
<td>Attribute 3</td>
<td>Attribute 1</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>23</td>
</tr>
</tbody>
</table>

Example values:
Year;Make;Model;Liters 1997;BMW;E89;2,34 2011;Mercedes;SLS;2
## Updates

First case, no positions change

<table>
<thead>
<tr>
<th>Attribute 1</th>
<th>Attribute 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>23</td>
<td>32</td>
</tr>
<tr>
<td>41</td>
<td>55</td>
</tr>
</tbody>
</table>

- **Tuple 1:** 1989; BBB; CCC; 4, 4492011; Mercedes; SLS; 2
Updates

Second case, positions change.
First option, update index.

<table>
<thead>
<tr>
<th>Tuple 1</th>
<th>Tuple 2</th>
<th>Tuple 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute 1</td>
<td>Attribute 3</td>
<td>Attribute 1</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>23</td>
</tr>
<tr>
<td>Year;Make;Model;Liters</td>
<td>1989;B;C;4,44</td>
<td>2011;Mercedes;SLS;2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32 30 (-2)</td>
<td>41 37 (-4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>55 51 (-4)</td>
</tr>
</tbody>
</table>
Updates

Second case, positions change.
Second option, throw it partially (or fully) away.

<table>
<thead>
<tr>
<th>Tuple 1</th>
<th>Tuple 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute 1</td>
<td>Attribute 3</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

Year;Make;Model;Liters;1989;B;C;4,44;2011;Mercedes;SLS;2

Index will automatically reconstruct itself
Traditional optimizations

Caching

Statistics
NoDB Performance Compared
NoDB

- Great DBMS + Raw hybrid
- Competitive performance with traditional DBs
- Eliminates loading times
- Queries get faster with time
- Updates
Outline

1. Introduction
2. Adaptive Indexing
3. Hybrid MapReduce
4. NoDB
5. Summary
Summary

Mature solutions: high load or query time
  No index → High query time
  Load all data → High delay (load time)

Hybrid solutions
  Bring indexes to in-situ processing
  Adaptive indexing
  HadoopDB
  NoDB
Remember..
Conclusions
References

3. Renata Borovica, Stratos Idreos, and Anastasia Ailamaki. **NoDB**: Efficient Query Execution on Raw Data Files *Categories and Subject Descriptors. pages 241–252.*
7. Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. **Here are my data files. here are my queries. where are my results.** *Proceedings of 5th Biennial Conference on Innovative Data Systems Research, pages 57–68, 2011.*
8. Christopher Olston, Benjamin Reed, Ravi Kumar, and Andrew Tomkins. **Pig Latin**: A Not-So-Foreign Language for Data Processing.
Questions?
Thank you!
MapReduce

Can be classified as distributed raw file parsing
Adaptive merging
Database Cracking

Column A

Cracking

13
4
2
7
19
14
8

4
2
7
8
13
14
19

A <= 7
7 < A <= 14
14 < A