Matt Might, The Illustrated Guide to a Ph.D.: http://matt.might.net/articles/phd-school-in-pictures
ONE SIZE DOES NOT FIT ALL
Streaming

OLAP

OLTP

Archiving

Log-processing

Web-search

Scan-oriented
Streaming
Log-processing
Indexes

Column

Row

Row+Column

Raw files
## Storage Views

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>abc</td>
<td>56</td>
<td>887.9</td>
</tr>
<tr>
<td>2</td>
<td>fdg</td>
<td>89</td>
<td>445.35</td>
</tr>
<tr>
<td>3</td>
<td>poe</td>
<td>67</td>
<td>234.67</td>
</tr>
<tr>
<td>4</td>
<td>lkj</td>
<td>12</td>
<td>385.92</td>
</tr>
<tr>
<td>5</td>
<td>yui</td>
<td>17</td>
<td>612.13</td>
</tr>
<tr>
<td>6</td>
<td>omg</td>
<td>90</td>
<td>148.9</td>
</tr>
</tbody>
</table>
# Storage Views

<table>
<thead>
<tr>
<th>Log</th>
<th>56</th>
<th>887.9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>89</td>
<td>445.35</td>
</tr>
<tr>
<td></td>
<td>67</td>
<td>234.67</td>
</tr>
<tr>
<td>4</td>
<td>lkj</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>yui</td>
<td>17</td>
</tr>
<tr>
<td>6</td>
<td>omg</td>
<td>90</td>
</tr>
</tbody>
</table>
# Storage Views

<table>
<thead>
<tr>
<th>Log</th>
<th>Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td>887.9</td>
</tr>
<tr>
<td>89</td>
<td>445.35</td>
</tr>
<tr>
<td>67</td>
<td>234.67</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>yui</td>
<td>omg</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>90</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>612.13</td>
<td></td>
<td>148.9</td>
</tr>
</tbody>
</table>
## Storage Views

<table>
<thead>
<tr>
<th>Column</th>
<th>Log</th>
<th>Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>yui</td>
<td>17</td>
<td>612.13</td>
</tr>
<tr>
<td>omg</td>
<td>90</td>
<td>148.9</td>
</tr>
</tbody>
</table>

|                  | 56     | 887.9 |
|                  | 89     | 445.35|
|                  | 67     | 234.67|
Storage Views

Log

56 | 887.9
89 | 445.35
67 | 234.67

Column grouped

Row

Column
Storage Views

Log

Index

Row

Column

Column grouped
Storage Views

- Log
- PAX
- Index
- Row
- Column
- Column grouped
Example: Flight Tickets

\[ \Pi_{\text{customer.}*} \left( \sigma_{a_1=x_1..a_n=x_n} \left( \left( \right) \right) \right) \]
Example: Flight Tickets

\[
\pi_{\text{customer.}^*}(\sigma_{a_1=x_1..a_n=x_n}(\text{tickets.customer_id} \bowtie_{\text{customer.id}}))
\]

Log SV

\[
\gamma_{\text{recent}}(\Gamma_{\text{bag, key}}(\sigma_{\text{bag}=\text{tickets}}))
\]

Col SV

\[
\pi_{\text{customer.}^*}(\sigma_{a_1=x_1..a_n=x_n}(\text{tickets.customer_id} \bowtie_{\text{customer.id}}))
\]

Result

Log SV

\[
\gamma_{\text{recent}}(\Gamma_{\text{bag, key}}(\sigma_{\text{bag}=\text{customers}}))
\]

Row SV

\[
\pi_{\text{customer.}^*}(\sigma_{a_1=x_1..a_n=x_n}(\text{tickets.customer_id} \bowtie_{\text{customer.id}}))
\]

Result
Example: Flight Tickets

Log SV \( \pi_{\text{customer.} \cdot (\sigma_{a_1=x_1..a_n=x_n} (\text{tickets.customer_id} \land \text{customer.id}))} \) Result

Log SV \( \gamma_{\text{recent}} (\Gamma_{\text{bag,key}(\sigma_{\text{bag=tickets}})}) \) Col SV \( \pi_{\text{tickets} \cdot (\sigma_{a_1=x_1..a_n=x_n} (\text{tickets.customer_id} \land \text{customer.id}))} \) Result

Col SV \( \gamma_{\text{recent}} (\Gamma_{\text{bag,key}(\sigma_{\text{bag=customers}})}) \) Row SV \( \pi_{\text{customers}} \cdot (\sigma_{a_1=x_1..a_n=x_n} (\text{tickets.customer_id} \land \text{customer.id})) \) Result

Row SV \( \gamma_{\text{recent}} (\Gamma_{\text{bag,key}(\sigma_{\text{bag=tickets}})}) \) Col SV \( \gamma_{\text{recent}} (\Gamma_{\text{bag,key}(\sigma_{\text{bag=customers}})}) \) Row SV

Log SV \( \gamma_{\text{recent}} (\Gamma_{\text{bag,key}(\sigma_{\text{bag=tickets}})}) \) Col SV \( \gamma_{\text{recent}} (\Gamma_{\text{bag,key}(\sigma_{\text{bag=customers}})}) \) Row SV

Log SV

Log Store

Col SV

Index SV

Cold

Index SV

Frequent Fliers
(Adaptive Partial Index)

Col SV

Index SV

Frequent Fliers
(Adaptive Partial Index)

Log SV

Log Store

Col SV

Index SV

Frequent Fliers
(Adaptive Partial Index)
Example: Flight Tickets

- **Primary Log Store**
  - **Row SV**: \( \pi \) \( \sigma_{a_1=x_1..a_n=x_n} (\text{tickets.customer_id} \bowtie \text{customer.id}) \)
  - **Col SV**: \( \pi \) \( \sigma_{a_1=x_1..a_n=x_n} (\text{tickets.customer_id} \bowtie \text{customer.id}) \)

**Result**

- **Primary Log Store**
  - **Index SV**: \( \pi \) \( \sigma_{a_1=x_1..a_n=x_n} (\text{tickets.customer_id} \bowtie \text{customer.id}) \)
  - **Col SV**: \( \pi \) \( \sigma_{a_1=x_1..a_n=x_n} (\text{tickets.customer_id} \bowtie \text{customer.id}) \)

**Result**
WTF!
Where's The Food!
Rodent Store
What to store?

Data Files

- copy 1
- copy 2
- copy 3
How to store?

Data Files

? 

+ a b
Where to store?

Data Files

?
Example Use-cases

• WWHow! File System
• WWHow! RAID
• WWHow! Relational DBMS
• WWHow! Cloud
Store my conferences talks (PDFs 2x and PPTs 1x) using RSA compression on University server

STORE ‘/Users/Bob/Conferences/Talks/.*.*’
WHAT *(pdf | ppt), *.pdf
WHERE vise4
HOW encryption(rsa) FOR *;
I want my conference talks to be highly available

STORE ‘/Users/Bob/Conferences/Talks/*.*’
WHAT *(pdf | ppt), *.pdf
HOW encryption(rsa) FOR *
**PREFERENCE** Availability=‘high’;
I want my conference talks to be highly available

STORE ‘/Users/Bob/Conferences/Talks/*..*’
WHAT *(pdf | ppt), *.pdf
HOW encryption(rsa) FOR *
**PREFERENCE** Availability=‘high’;

job for the WWhow! data storage optimizer
OctopusDB

- Cool Vision
- Tough to realize
C-Store
Trojan Columns

Application

Query Processor

Relations

UDF Storage Layer

Physical Representation

File 1  File 2  File 3  ....  File n
Trojan Columns

Our main contributions are as follows:

- Our idea is to use User Defined Functions (UDFs) as an access technique in Hadoop++
- User queries remain (almost) unchanged.
- Trojan Columns maps logical relations to physical tables as grouped layouts by inserting appropriate UDFs.
- Injects column store functionality into existing closed source systems.
- Does not invade or make heavy changes in the system; rather, just fixes the storage layer (row, column, or even column-grouped layouts) by inserting appropriate UDFs.
- User's logical row view to the physical column view on disk. The system. Trojan Columns uses table returning UDFs.
- Injects column store functionality into a given database system.
- Does not affect the changes from inside without plugging in the system catalog have to be added. The consequence is that within, i.e. no new SQL keywords, or data types, or any entries in the database product, is it possible to introduce column store technology in it to itself. Again, based on user applications, we have full control to decide how to access the data. For example, for an airline company and higher compression ratio (e.g. Huffman) for archive databases.
- Server Column Indexes: affect the changes from inside without.
- 第二, Trojan Columns is quite different from SQL Server Column Indexes not only in data storage, but also in data access. Similar to them, Trojan Columns can tune the blob storage to user applications. For example, we might decide how to push down one or more operators in the query tree to the database product independent. We present experimental results from DBMS-X over TPC-H datasets. We evaluate Trojan Columns over there different benchmarks to investigate the pros and cons of Trojan TPC-H queries as proposed in the C-Store paper.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Customer</th>
<th>Physical Table</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>name</td>
<td></td>
</tr>
<tr>
<td></td>
<td>phone</td>
<td></td>
</tr>
<tr>
<td></td>
<td>market_segment</td>
<td></td>
</tr>
<tr>
<td>smith</td>
<td>2134</td>
<td>automobile</td>
</tr>
<tr>
<td>john</td>
<td>3425</td>
<td>household</td>
</tr>
<tr>
<td>kim</td>
<td>6756</td>
<td>furniture</td>
</tr>
<tr>
<td>joe</td>
<td>9878</td>
<td>building</td>
</tr>
<tr>
<td>mark</td>
<td>4312</td>
<td>building</td>
</tr>
<tr>
<td>steve</td>
<td>2435</td>
<td>automobile</td>
</tr>
<tr>
<td>jim</td>
<td>5766</td>
<td>household</td>
</tr>
<tr>
<td>ian</td>
<td>8789</td>
<td>household</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical Table</th>
<th>Customer_trojan</th>
</tr>
</thead>
<tbody>
<tr>
<td>segment_ID</td>
<td>attribute_ID</td>
</tr>
<tr>
<td>1</td>
<td>name</td>
</tr>
<tr>
<td>1</td>
<td>phone</td>
</tr>
<tr>
<td>1</td>
<td>market_segment</td>
</tr>
<tr>
<td>2</td>
<td>name</td>
</tr>
<tr>
<td>2</td>
<td>phone</td>
</tr>
<tr>
<td>2</td>
<td>market_segment</td>
</tr>
</tbody>
</table>
Our main contributions are as follows:

- Injects column store functionality into existing closed source column store databases.
- Provides a mechanism to decouple logical representation of relations from their physical implementation, a.k.a. column store technology.
- Trojan Columns by varying several query parameters. (Section 2)

### Trojan Columns

**Relation**

<table>
<thead>
<tr>
<th>Customer</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>phone</td>
</tr>
<tr>
<td>smith</td>
<td>2134</td>
</tr>
<tr>
<td>john</td>
<td>3425</td>
</tr>
<tr>
<td>kim</td>
<td>6756</td>
</tr>
<tr>
<td>joe</td>
<td>9878</td>
</tr>
<tr>
<td>mark</td>
<td>4312</td>
</tr>
<tr>
<td>steve</td>
<td>2435</td>
</tr>
<tr>
<td>jim</td>
<td>5766</td>
</tr>
<tr>
<td>ian</td>
<td>8789</td>
</tr>
</tbody>
</table>

### Customer_trojan

<table>
<thead>
<tr>
<th>segment_ID</th>
<th>attribute_ID</th>
<th>blob_data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>name</td>
<td>smith, john, kim, joe</td>
</tr>
<tr>
<td>1</td>
<td>phone</td>
<td>2134, 3425, 6756, 9878</td>
</tr>
<tr>
<td>1</td>
<td>market_segment</td>
<td>automobile, household, furniture, building</td>
</tr>
<tr>
<td>2</td>
<td>name</td>
<td>mark, steve, jim, ian</td>
</tr>
<tr>
<td>2</td>
<td>phone</td>
<td>4312, 2435, 5766, 8789</td>
</tr>
<tr>
<td>2</td>
<td>market_segment</td>
<td>building, automobile, household, household</td>
</tr>
</tbody>
</table>
Trojan Columns

Our main contributions are as follows:

1. Trojan Columns maps logical relations to physical tables as follows. First, Trojan Columns uses UDFs to store the blob data instead of a new index type in case of Column Indexes. The blob, instead of a new index type in case of Column Indexes. The accessor to translate it into the row layout for the query processor. This means we trick the database into believing that the data is still stored in a compressed column-oriented fashion on disk. But the UDFs must not invade or make heavy changes in the system; rather, they change the source code of the system. In the following, we describe how to rewrite the user queries in order to use table returning UDFs, and (iii) table (of rows) returning UDFs.

2. Trojan Columns is quite different from SQL Server Column Indexes not only in data storage, but also in data access. Similarly, we found bigger segment sizes, e.g., typically, we found bigger segment sizes, e.g., 9878, instead of a new index type in case of Column Indexes. The blob, instead of a new index type in case of Column Indexes. The accessor to translate it into the row layout for the query processor. This means we trick the database into believing that the data is still stored in a compressed column-oriented fashion on disk. But the UDFs must not invade or make heavy changes in the system; rather, they change the source code of the system. In the following, we describe how to rewrite the user queries in order to use table returning UDFs, and (iii) table (of rows) returning UDFs.

3. Trojan Columns is a BLOB, thus mimicking a column-oriented storage. Experimentally, we found bigger segment sizes, e.g., 9878, instead of a new index type in case of Column Indexes. The blob, instead of a new index type in case of Column Indexes. The accessor to translate it into the row layout for the query processor. This means we trick the database into believing that the data is still stored in a compressed column-oriented fashion on disk. But the UDFs must not invade or make heavy changes in the system; rather, they change the source code of the system. In the following, we describe how to rewrite the user queries in order to use table returning UDFs, and (iii) table (of rows) returning UDFs. In order to demonstrate the benefits of column-oriented access, we use the TPC-H benchmark. We present experimental results from DBMS-X over TPC-H micro-benchmarks to investigate the pros and cons of Trojan Columns. We compare its performance against that of a commercial row-oriented database system, IBM DB2. The performance results indicate that Trojan Columns is able to achieve higher throughput and lower query execution time than DB2. In particular, we observe a significant reduction in query execution time.

Customer

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
<th>market_segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>smith</td>
<td>2134</td>
<td>automobile</td>
</tr>
<tr>
<td>john</td>
<td>3425</td>
<td>household</td>
</tr>
<tr>
<td>kim</td>
<td>6756</td>
<td>furniture</td>
</tr>
<tr>
<td>joe</td>
<td>9878</td>
<td>building</td>
</tr>
<tr>
<td>mark</td>
<td>4312</td>
<td>building</td>
</tr>
<tr>
<td>steve</td>
<td>2435</td>
<td>automobile</td>
</tr>
<tr>
<td>jim</td>
<td>5766</td>
<td>household</td>
</tr>
<tr>
<td>ian</td>
<td>8789</td>
<td>household</td>
</tr>
</tbody>
</table>

Customer_trojan

<table>
<thead>
<tr>
<th>segment_ID</th>
<th>attribute_ID</th>
<th>blob_data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>name</td>
<td>smith, john, kim, joe</td>
</tr>
<tr>
<td>1</td>
<td>phone</td>
<td>2134, 3425, 6756, 9878</td>
</tr>
<tr>
<td>1</td>
<td>market_segment</td>
<td>automobile, household, furniture, building</td>
</tr>
<tr>
<td>2</td>
<td>name</td>
<td>mark, steve, jim, ian</td>
</tr>
<tr>
<td>2</td>
<td>phone</td>
<td>4312, 2435, 5766, 8789</td>
</tr>
<tr>
<td>2</td>
<td>market_segment</td>
<td>building, automobile, household, household</td>
</tr>
</tbody>
</table>
Example: TPC-H Query 6

Result

\[ \gamma_{\text{agg}} \ (\text{extendedprice} \times \text{discount}) \]

\[ \sigma \ \text{shipdate BETWEEN '1994-01-01' AND '1995-01-01'} \]
\[ \text{AND discount BETWEEN 0.05 AND 0.07} \]
\[ \text{AND quantity < 24} \]

\[ \pi \ \text{quantity, discount} \]
\[ \text{extendedprice, shipdate} \]

Scan

\[ \text{lineitem} \]
Example: TPC-H Query 6

\[ \begin{align*}
\text{Result} & \quad \gamma_{\text{agg}} (\text{extendedprice} \times \text{discount}) \\
\sigma & \quad \text{shipdate BETWEEN '1994-01-01' AND '1995-01-01'} \\
\pi & \quad \text{quantity, discount} \\
\text{Scan} & \quad \text{lineitem}
\end{align*} \]

\[ \begin{align*}
\text{Result} & \quad \gamma_{\text{agg}} (\text{extendedprice} \times \text{discount}) \\
\sigma & \quad \text{shipdate BETWEEN '1994-01-01' AND '1995-01-01'} \\
\pi & \quad \text{quantity, discount} \\
\text{Scan} & \quad \text{lineitem}
\end{align*} \]
Example: TPC-H Query 6

**Result**

\[ \gamma_{agg} \] 
\[ \sigma \text{ shipdate BETWEEN '1994-01-01' AND '1995-01-01'} \]
\[ \pi \text{ quantity, discount extendedprice, shipdate} \]
\[ \text{scan lineitem} \]

**Result**

\[ \gamma_{agg} \] 
\[ \sigma \text{ quantity, discount extendedprice, shipdate} \]
\[ \pi \text{ lineitem} \]

**Result**

\[ \gamma_{agg} \] 
\[ \sigma \text{ shipdate BETWEEN '1994-01-01' AND '1995-01-01'} \]
\[ \pi \text{ quantity, discount extendedprice, shipdate} \]
\[ \text{scan lineitem} \]

\[ \text{scanUDF} \]

\[ \text{selectUDF} \]
**Benchmark Results**

**Figure 3: Comparing TPC-H Query Runtimes of Trojan Columns with Standard Row in DBMS-X.**

- **Stage 1**: Selectivity 0, Stage 1
- **Stage 2**: Selectivity 1, Stage 2
- **Stage 3**: Selectivity 5, Stage 3
- **Stage 4**: Selectivity 15, Stage 4

<table>
<thead>
<tr>
<th>Query</th>
<th>Standard Row</th>
<th>Trojan Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>232.7671</td>
<td>1890.123</td>
</tr>
<tr>
<td>Q2</td>
<td>111.9957</td>
<td>30.788114</td>
</tr>
<tr>
<td>Q3</td>
<td>24.38536</td>
<td>37.331905</td>
</tr>
<tr>
<td>Q4</td>
<td>58.82072</td>
<td>336.6745</td>
</tr>
<tr>
<td>Q5</td>
<td>98.44627</td>
<td>345.8437</td>
</tr>
<tr>
<td>Q6</td>
<td>152.6566</td>
<td>232.7671</td>
</tr>
<tr>
<td>Q7</td>
<td>52.6585</td>
<td>30.788114</td>
</tr>
</tbody>
</table>

*Mike Stonebraker et. al. C-Store: A Column Oriented DBMS. VLDB 2005*
**Benchmark Results**

![Benchmark Results Diagram](image)

- **Q1**: Standard Row vs. Trojan Columns
  - **Q2**: Standard Row vs. Trojan Columns
  - **Q3**: Standard Row vs. Trojan Columns
  - **Q4**: Standard Row vs. Trojan Columns
  - **Q5**: Standard Row vs. Trojan Columns
  - **Q6**: Standard Row vs. Trojan Columns
  - **Q7**: Standard Row vs. Trojan Columns

---

* Mike Stonebraker et. al. C-Store: A Column Oriented DBMS. VLDB 2005
1990s
2000s
HYRISE

2010s
7 Vertical Partitioning Algorithms

- Brute Force
- Navathe’s Algorithm
- HillClimb
- AutoPart
- HYRISE
- O₂P
- Trojan
Four Comparison Metrics

- How Fast?
- How Good?
- How fragile?
- Where does it make sense?
How do the optimization times change with the workload size?

In this section we address the following questions:

1. How much longer to transform the layout than it takes to compute the layout.
2. Much longer to transform the layout than it takes to compute the layout.
3. As discussed in the previous section, we apply the same setting for partitioning algorithms. We can see that the fastest algorithm (O(1), which is around 10,000 times faster than BruteForce). Even the slowest algorithm (O(n^2)) takes much longer to transform the layout than it takes to compute the layout.

The total I/O costs of the entire workload will be the sum of the I/O costs of each query in the workload.

The total I/O costs of the entire workload will be the sum of the I/O costs of each query in the workload.

Finally, for a query Q referencing a partition P, the scan cost is given as:

\[ \text{scan cost} = \text{rows per block} \times \text{blocks per partition} \times \text{disk seek time} \]

The total number of blocks on disk for partition P is given as:

\[ \text{blocks per partition} = \frac{\text{total number of rows}}{\text{partition size}} \]

Assume that we have to perform a seek every time the I/O buffer gets full determines the seek cost of reading partition P.

If the table has N rows, then the total number of blocks on disk for partition P is given as:

\[ \text{blocks per partition} = \frac{N}{\text{partition size}} \]

The seek cost of reading partition P is given as:

\[ \text{seek cost} = \text{disk seek time} \times \text{number of times the I/O buffer needs to be filled} \]

For example, if it takes fifteen minutes to create the layouts (i.e., a large table) then it might be possible to recompute them for each and every hardware/software setting. However, vertical partitioning algorithms can be computationally expensive, therefore it is not possible in data centers these days. However, vertical partitioning algorithms are over different parameters in the cost model and describe them below.

How fast?
## Distance from Column Layouts

### Percentage Difference from Column Layouts [%]

<table>
<thead>
<tr>
<th></th>
<th>AutoPart</th>
<th>HillClimb</th>
<th>HYRISE</th>
<th>Navathe</th>
<th>O2P</th>
<th>Trojan</th>
<th>BruteForce</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPC-H</td>
<td>-21.47</td>
<td>-25</td>
<td>1.58</td>
<td>3.71</td>
<td>5.29</td>
<td>1.64</td>
<td>5.29</td>
</tr>
<tr>
<td>SSB</td>
<td>-27.74</td>
<td>-20</td>
<td>3.71</td>
<td>0.05</td>
<td>3.71</td>
<td>1.64</td>
<td>0.05</td>
</tr>
</tbody>
</table>

### Bar Chart

- **X-axis:** Datasets (AutoPart, HillClimb, HYRISE, Navathe, O2P, Trojan, BruteForce)
- **Y-axis:** Percentage Difference from Column Layouts [%]
- **Legend:**
  - Blue: TPC-H
  - Green: SSB
Effect of Buffer Size

The graph shows the normalized estimated costs (%) of different vertical partitioning algorithms and Column under varying buffer sizes. The x-axis represents the buffer size in MB, with a logarithmic scale, while the y-axis represents the normalized estimated costs.

- **HillClimb**: Blue line
- **Navathe**: Green line
- **Materialized views**: Orange line
- **Column**: Red line

As the buffer size increases, the performance of vertical partitioning algorithms relative to Column changes. For smaller buffer sizes, Column performs better, but as the buffer size grows, HillClimb and Navathe show improved performance compared to Column. Materialized views also show performance improvements with increasing buffer size.

The graph highlights the importance of buffer size in determining the optimal vertical partitioning algorithm. It suggests that for larger buffer sizes, vertical partitioning algorithms can offer significant performance gains compared to the traditional column layout.
Comparison’s Paper: Hadoop Vs PDBMS
Comparison’s Paper: Hadoop Vs PDBMS
### Analytical Query Performance

#### Selection Task

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Vertica</th>
<th>DBMS-X</th>
<th>Hadoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.3</td>
<td>0.8</td>
<td>1.8</td>
</tr>
<tr>
<td>10</td>
<td>1.8</td>
<td>4.7</td>
<td>12.4</td>
</tr>
</tbody>
</table>

#### Join Task

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Vertica</th>
<th>DBMS-X</th>
<th>Hadoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21.5</td>
<td>31.3</td>
<td>36.1</td>
</tr>
<tr>
<td>10</td>
<td>32.8</td>
<td>29.2</td>
<td>29.4</td>
</tr>
<tr>
<td>25</td>
<td>85.0</td>
<td>31.9</td>
<td></td>
</tr>
</tbody>
</table>

---

![Bar Graph](image-url)

*Figure 6: Analytical Query Performance*
We observe, that The Hadoop Plan consists of two reducers (\( R_1 \)). The default number of replicas used by Hadoop is 3, i.e. we obtain subsets of the data called a block. The latter operator uses a UDF that defines how a split is divided into items. Then subplan M1 unions the input blocks and breaks them horizontally into disjoint subsets. These subsets may both be retrieved from subplan H1. Subplan M1 unions the input blocks into records (\( \text{RecRead} \)). The partitioning function itemize (\( \text{PPart} \)) and the number of reducers (\( \text{R} \)) determines the number of reducer subplans (\( \text{R} = 4 \)). The default number of replicas used by Hadoop is 3, i.e. we obtain subsets of the data called a block on different nodes.

The Hadoop Plan: Hadoop's processing pipeline expressed as a physical query execution plan (\( \text{M} \)). As a consequence, we are then able to run a MapReduce job, we first load the data into the distributed file system. This is done by partition the data at data load time. Similarly to Trojan Indexes, Trojan Index enriches logical input splits by bulkloaded read-optimized User Defined Functions (\( \text{sh} \)). The former operator (\( \text{MMap} \)) changes the internal layout of a block. Trojan Joins do neither require a DBMS nor SQL to do so. Trojan Join allows us to combine arbitrarily partition the data at data load time. Similarly to Trojan Indexes, Trojan Index is created at data load time and thus provide a non-invasive, DBMS-independent Trojan Index and Trojan Join may be combined to create arbitrarily independent indexing technique coined Trojan Index (\( \text{LPart} \)).

Let's analyze The Hadoop Plan in more detail: Figure 1. We provide a non-invasive, DBMS-independent execution plan (\( \text{M} \)). As a consequence, we are then able to run a MapReduce job, we first load the data into the distributed file system. This is done by partition the data at data load time. Similarly to Trojan Indexes, Trojan Index is created at data load time and thus provide a non-invasive, DBMS-independent Trojan Index (\( \text{LPart} \)).

Hadoop framework. Experimental comparison.

As in [3] we used the benchmark from SIGMOD 2009 [16]. All experiments are run on Amazon's EC2 Cloud.

We provide a non-invasive, DBMS-independent Trojan Index and Trojan Join may be combined to create arbitrarily independent indexing technique coined Trojan Index (\( \text{LPart} \)). All Trojan-techniques on top of Hadoop framework.

<p>|</p>
<table>
<thead>
<tr>
<th>Data Load Phase</th>
<th>Map Phase</th>
<th>Shuffle Phase</th>
<th>Reduce Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>M1</td>
<td>T1</td>
<td>R1</td>
</tr>
<tr>
<td>Data Node 1</td>
<td>M2</td>
<td>T2</td>
<td>R2</td>
</tr>
<tr>
<td>H2</td>
<td>M3</td>
<td>T3</td>
<td></td>
</tr>
<tr>
<td>Data Node 3</td>
<td>M4</td>
<td>T4</td>
<td></td>
</tr>
</tbody>
</table>

The Hadoop Plan: Hadoop's processing pipeline expressed as a physical operator DAG. To our knowledge, this paper is the first to show how to express MapReduce applications as a physical operator DAG. The Hadoop Plan consists of two reducers (\( R_1 \)). The default number of replicas used by Hadoop is 3, i.e. we obtain subsets of the data called a block. The latter operator uses a UDF that defines how a split is divided into items. Then subplan M1 unions the input blocks into records (\( \text{RecRead} \)). The partitioning function itemize (\( \text{PPart} \)) and the number of reducers (\( \text{R} \)) determines the number of reducer subplans (\( \text{R} = 4 \)). The default number of replicas used by Hadoop is 3, i.e. we obtain subsets of the data called a block on different nodes.

The Hadoop Plan: Hadoop's processing pipeline expressed as a physical operator DAG. To our knowledge, this paper is the first to show how to express MapReduce applications as a physical operator DAG. The Hadoop Plan consists of two reducers (\( R_1 \)). The default number of replicas used by Hadoop is 3, i.e. we obtain subsets of the data called a block. The latter operator uses a UDF that defines how a split is divided into items. Then subplan M1 unions the input blocks into records (\( \text{RecRead} \)). The partitioning function itemize (\( \text{PPart} \)) and the number of reducers (\( \text{R} \)) determines the number of reducer subplans (\( \text{R} = 4 \)). The default number of replicas used by Hadoop is 3, i.e. we obtain subsets of the data called a block on different nodes.

The Hadoop Plan: Hadoop's processing pipeline expressed as a physical operator DAG. To our knowledge, this paper is the first to show how to express MapReduce applications as a physical operator DAG. The Hadoop Plan consists of two reducers (\( R_1 \)). The default number of replicas used by Hadoop is 3, i.e. we obtain subsets of the data called a block. The latter operator uses a UDF that defines how a split is divided into items. Then subplan M1 unions the input blocks into records (\( \text{RecRead} \)). The partitioning function itemize (\( \text{PPart} \)) and the number of reducers (\( \text{R} \)) determines the number of reducer subplans (\( \text{R} = 4 \)). The default number of replicas used by Hadoop is 3, i.e. we obtain subsets of the data called a block on different nodes.
The Hadoop Plan is shaped by three user-defined parameters: $P$, $R$, and $M$. The Hadoop Plan consists of nodes, respectively [10]. An example for a plan with four map tasks. We have analyzed Yahoo!'s Hadoop version 0.19. Note that data nodes with subplans H1–H4. Replicas are stored on different nodes.

In this section we examine how Hadoop computes a MapReduce execution plan. We make Hadoop's hard-coded query process explicit and represent it as a DB-style physical query execution plan. As a consequence, we are then able to reason on that plan. (Section 2)

In this paper, we provide a non-invasive, DBMS-independent Trojan Index and Trojan Join may be combined to create arbitrarily large Index enriches logical input splits by bulkloaded read-optimized UDFs. However, Hadoop

Let's analyze The Hadoop Plan in more detail: The Hadoop Plan: Hadoop's processing pipeline expressed as a physical operator DAG. To our knowledge, this paper is the first to do so in that detail and we term it pressed as a physical operator DAG. An example for a plan with four map-reduce operations is shown in Figure 1. We observe, that the Hadoop Plan consists of $P$ data nodes, $M$ map tasks, and $R$ reduce tasks. (Section 2.1)

Our results confirm that Hadoop has exponential I/O asymmetry may occur if a mapper subplan (here: M3) consumes less input data and/or stores back on disk. (Section 3.2.6 for details).

In this paper, we provide a non-invasive, DBMS-independent Trojan Join. We provide a non-invasive, DBMS-independent Trojan Index. A Trojan Index may be run on a DBMS and vice-versa. Appendices C to E contain experimental comparison, we implemented all Trojan-techniques on top of Hadoop framework. No operator-model needs to be changed or replaced the Hadoop framework. (Section 3)

The Hadoop Plan: Hadoop's processing pipeline expressed as a physical query execution plan. No operator-model of Hadoop framework.

Let's analyze The Hadoop Plan in more detail: The Hadoop Plan: Hadoop's processing pipeline expressed as a physical query execution plan.
to the inital of a subplan \( L \) (2.1 The Hadoop Plan)

We provide a non-invasive, DBMS-independent approach to neuro-structured data management. The work in [10] introduced a physical operator DAG. To our knowledge, this paper is the first to analyze Yahoo!'s Hadoop version 0.19. Note that the additional details and results of our experimental study may be run on a DBMS and vice-versa. Appendices C to E contain more information.

In addition, Appendix A illustrates processing strategies for analytical data processing. Appendix B shows that MapReduce is well suited for index and join-based tasks. (Section 5)

Our results confirm that Hadoop outperforms HadoopDB against Hadoop as well as HadoopDB as proposed at VLDB 2007. In this section we examine how Hadoop computes a MapReduce query execution plan.

Experimental comparison.

As a consequence, we are then able to reason on that plan. (Section 2)

Let's analyze The Hadoop Plan in more detail:

The Hadoop Plan: Hadoop's processing pipeline expressed as a physical operator DAG. To our knowledge, this paper is the first to analyze Yahoo!'s Hadoop version 0.19. Note that the additional details and results of our experimental study may be run on a DBMS and vice-versa. Appendices C to E contain more information.

In addition, Appendix A illustrates processing strategies for analytical data processing. Appendix B shows that MapReduce is well suited for index and join-based tasks. (Section 5)

Our results confirm that Hadoop outperforms HadoopDB against Hadoop as well as HadoopDB as proposed at VLDB 2007. In this section we examine how Hadoop computes a MapReduce query execution plan.

Experimental comparison.

As a consequence, we are then able to reason on that plan. (Section 2)

Let's analyze The Hadoop Plan in more detail:

The Hadoop Plan: Hadoop's processing pipeline expressed as a physical operator DAG. To our knowledge, this paper is the first to analyze Yahoo!'s Hadoop version 0.19. Note that the additional details and results of our experimental study may be run on a DBMS and vice-versa. Appendices C to E contain more information.

In addition, Appendix A illustrates processing strategies for analytical data processing. Appendix B shows that MapReduce is well suited for index and join-based tasks. (Section 5)

Our results confirm that Hadoop outperforms HadoopDB against Hadoop as well as HadoopDB as proposed at VLDB 2007. In this section we examine how Hadoop computes a MapReduce query execution plan.

Experimental comparison.

As a consequence, we are then able to reason on that plan. (Section 2)

Let's analyze The Hadoop Plan in more detail:

The Hadoop Plan: Hadoop's processing pipeline expressed as a physical operator DAG. To our knowledge, this paper is the first to analyze Yahoo!'s Hadoop version 0.19. Note that the additional details and results of our experimental study may be run on a DBMS and vice-versa. Appendices C to E contain more information.

In addition, Appendix A illustrates processing strategies for analytical data processing. Appendix B shows that MapReduce is well suited for index and join-based tasks. (Section 5)

Our results confirm that Hadoop outperforms HadoopDB against Hadoop as well as HadoopDB as proposed at VLDB 2007. In this section we examine how Hadoop computes a MapReduce query execution plan.

Experimental comparison.

As a consequence, we are then able to reason on that plan. (Section 2)

Let's analyze The Hadoop Plan in more detail:

The Hadoop Plan: Hadoop's processing pipeline expressed as a physical operator DAG. To our knowledge, this paper is the first to analyze Yahoo!'s Hadoop version 0.19. Note that the additional details and results of our experimental study may be run on a DBMS and vice-versa. Appendices C to E contain more information.

In addition, Appendix A illustrates processing strategies for analytical data processing. Appendix B shows that MapReduce is well suited for index and join-based tasks. (Section 5)

Our results confirm that Hadoop outperforms HadoopDB against Hadoop as well as HadoopDB as proposed at VLDB 2007. In this section we examine how Hadoop computes a MapReduce query execution plan.

Experimental comparison.

As a consequence, we are then able to reason on that plan. (Section 2)

Let's analyze The Hadoop Plan in more detail:

The Hadoop Plan: Hadoop's processing pipeline expressed as a physical operator DAG. To our knowledge, this paper is the first to analyze Yahoo!'s Hadoop version 0.19. Note that the additional details and results of our experimental study may be run on a DBMS and vice-versa. Appendices C to E contain more information.

In addition, Appendix A illustrates processing strategies for analytical data processing. Appendix B shows that MapReduce is well suited for index and join-based tasks. (Section 5)

Our results confirm that Hadoop outperforms HadoopDB against Hadoop as well as HadoopDB as proposed at VLDB 2007. In this section we examine how Hadoop computes a MapReduce query execution plan.

Experimental comparison.

As a consequence, we are then able to reason on that plan. (Section 2)

Let's analyze The Hadoop Plan in more detail:

The Hadoop Plan: Hadoop's processing pipeline expressed as a physical operator DAG. To our knowledge, this paper is the first to analyze Yahoo!'s Hadoop version 0.19. Note that the additional details and results of our experimental study may be run on a DBMS and vice-versa. Appendices C to E contain more information.

In addition, Appendix A illustrates processing strategies for analytical data processing. Appendix B shows that MapReduce is well suited for index and join-based tasks. (Section 5)

Our results confirm that Hadoop outperforms HadoopDB against Hadoop as well as HadoopDB as proposed at VLDB 2007. In this section we examine how Hadoop computes a MapReduce query execution plan.

Experimental comparison.

As a consequence, we are then able to reason on that plan. (Section 2)
Trojan Index Creation

map(key k, value v) $\mapsto$ [(getSplitID() $\oplus$ prj$_{a_i}$ (k $\oplus$ v), k $\oplus$ v)]

cmp(key k1, key k2) $\mapsto$ compare(k1.a, k2.a)

grp(key k1, key k2) $\mapsto$ compare(k1.splitID, k2.splitID)

sh(key k, value v, int numPartitions) $\mapsto$ k.splitID % numPartitions

reduce(key ik, vset ivs) $\mapsto$ [(ivs $\oplus$ indexBuilder$_{a_i}$ (ivs))]
Our results confirm that Hadoop’s query execution strategy may be ex-
2. HADOOP AS A PHYSICAL QUERY EX-
may be run on a DBMS and vice-versa. Appendices C to E contain
Index enriches logical input splits by bulkloaded read-optimized
able to reason on that plan. (Section 2)

Algorithm 1: Trojan Index/Trojan Join split UDF

Input : JobConf job, Int numSplits
Output: logical data splits
1 FileSplit [] splits;
2 File [] files = GetFiles(job);
3 foreach file in files do
4 Path path = file.getPath();
5 InputStream in = GetInputStream(path);
6 Long offset = file.getLength();
7 while offset > 0 do
8 in.seek(offset—FOOTER_SIZE);
9 Footer footer = ReadFooter(in);
10 Long splitSize = footer.getSplitSize();
11 offset -= (splitSize + FOOTER_SIZE);
12 BlockLocations blocks = GetBlockLocations(path, offset);
13 FileSplit newSplit = CreateSplit(path, offset, splitSize, blocks);
14 splits.add(newSplit);
15 end
16 end
17 return splits;
Trojan Index Access

Algorithm 2: Trojan Index itemize.initialize UDF

Input: FileSplit split, JobConf job

1. Global FileSplit split = split;
2. Key lowKey = job.getLowKey();
3. Global Key highKey = job.getHighKey();
4. Int splitStart = split.getStart();
5. Global Int splitEnd = split.getEnd();
6. Header h = ReadHeader(split);
7. Overlap type = h.getOverlapType(lowKey, highKey);
8. Global Int offset;
9. if type == LEFT_CONTAINED or type == FULL_CONTAINED or type == POINT_CONTAINED then
   10. Index i = ReadIndex(split);
   11. offset = splitStart + i.lookup(lowKey);
12. else if type == RIGHT_CONTAINED or type == SPAN then
   13. offset = splitStart;
14. else
   15. // NOT_CONTAINED, skip the split;
   16. offset = splitEnd;
17. end
18. Seek(offset);
Algorithm 3: Trojan Index itemize.next UDF

Input: KeyType key, ValueType value
Output: has more records

1. if offset < splitEnd then
   2. Record nextRecord = ReadNextRecord(split);
   3. offset += nextRecord.size();
   4. if nextRecord.key < highKey then
      5. SetKeyValue(key, value, nextRecord);
      6. return true;
   7. end
   8. end
9. return false;
Selection Analytical Task *

* Pavlo et. al. A Comparison of Approaches to large-Scale Data Analysis. SIGMOD 2009
Join Analytical Task *

* Pavlo et. al. A Comparison of Approaches to large-Scale Data Analysis. SIGMOD 2009
Trojan Index

Trojan Join
Traditional Layouts

Row
(default)

Column*

PAX**

* A. Floratou et al. Column-Oriented Storage Techniques for MapReduce. PVLDB, April, 2011
** Y. He et al. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. ICDE, 2011
## Traditional Layouts

<table>
<thead>
<tr>
<th>Non-required Reads</th>
<th>Column</th>
<th>PAX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Costs</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Block Placement</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tuple Reconstruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Trojan Data Layouts

Replica 1

Replica 2

Replica 3
# Trojan Data Layouts

<table>
<thead>
<tr>
<th></th>
<th>Row</th>
<th>Column</th>
<th>PAX</th>
<th>Trojan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-required Reads</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Costs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Block Placement</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tuple Reconstruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Layout Quality

<table>
<thead>
<tr>
<th></th>
<th>#Non-required Attributes Read</th>
<th>#Joins in Tuple Reconstruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HADOOP-ROW</strong></td>
<td>525</td>
<td>0</td>
</tr>
<tr>
<td><strong>HADOOP-PAX</strong></td>
<td>0</td>
<td>139</td>
</tr>
<tr>
<td>Trojan Layout</td>
<td>14</td>
<td>20</td>
</tr>
</tbody>
</table>
Projection Analytical Task

![Bar chart showing improvement factors for TPC-H queries. The chart compares the performance of Trojan Layouts over Hadoop-Row and Hadoop-PAX.](chart.png)
Hadoop Aggressive Indexing Library
Individual Jobs: Weblog, RecordReader

![Bar chart showing runtime comparison between Hadoop, Hadoop ++, and HAIL for different MapReduce jobs. The chart displays the following runtimes (in ms) for each job:

- Bob-Q1: Hadoop = 3358, Hadoop ++ = 2776, HAIL = 2864
- Bob-Q2: Hadoop = 2156, Hadoop ++ = 2112, HAIL = 2442
- Bob-Q3: Hadoop = 2470, Hadoop ++ = 2470, HAIL = 2470
- Bob-Q4: Hadoop = 2917, Hadoop ++ = 333, HAIL = 683
- Bob-Q5: Hadoop = 2112, Hadoop ++ = 2156, HAIL = 2776

The chart indicates that Hadoop ++ generally performs better than Hadoop and HAIL for most jobs.
Hadoop Stack

- HBase
- Hive
- Pig
- MapReduce
- Cartilage Query Engine
- HDFS
- Cartilage Upload Pipeline
- Data File 1
- Data File 2
- ...
- Data File n
Hadoop Stack

Queried Data

Cartilage Query Engine

HDFS

Cartilage Upload Pipeline

Input Data
Upload Plans

Data

Parser

Logical Partitioner

Stage 1

Stage 2

Stage 3

Stage 4

Stage 5

Uploader

Serializer

Physical Partitioner

HDFS

Block 1

Block 2

Block 3
IDs (corresponding to node IDs) to each replica of each physical
and assigns a replica ID to it.

in the second block replicates each physical partition three times
before passing them to the second block.

which
physical partitioner assigns the same physical partition IDs to tuples
ID to each tuple.

shows the HDFS upload plan in Cartilage.

as
to preprocess and upload their data. We call these upload data
actually uploaded to the HDFS.

the same block, i.e. a datum is passed on to the next stage as soon as
previous stages. Cartilage has a streaming data
form a datum based on the labels already assigned to it by the pre-
every transformation. Successive UDF stages can selectively trans-
applies a chain of UDFs to transform the data, and assigns a label for

However, the data
inside) and we specify the node locations of the physical
replica with replication factor of one (since here we do the replication from
outside)
blocks into the HDFS. Note that we upload each physical

We saw in the above example that the users can reproduce
the combination of UDF and ID). Cartilage applies a UDF only to
process certain pieces of the data conditionally using
addition to the physical partitioner to create logical partitions based
for each of the UDFs. For example, we can add a
in the upload plan, and (iii) provide their own custom functionality
using Cartilage is that the users now have the
flexibility to: (i) re-

Figure
Stage 5
Stage 1
Stage 2
Stage 3
Stage 4
Stage 1
Serializer 1
Serializer 2
Replicator 2
Physical Partitioner 1
Replicator 1
Logical Partitioner
Data
Stage 3
Locator 1
Stage 4
Locator 2
Stage 5
Uploader
Block 1
Block 2
Block 3
Block 4
Block 5
HDFS
replica 1a
replica 1b
replica 1
replica 2
replica 2
replica 1

Stage 1
Serializer 1
Serializer 2
Replicator 2
Physical Partitioner 1
Replicator 1
Logical Partitioner
Data
Stage 3
Locator 1
Stage 4
Locator 2
Stage 5
Uploader
Block 1
Block 2
Block 3
Block 4
Block 5
HDFS
replica 1a
replica 1b
replica 1
replica 2
replica 1

replica 1
replica 2
Summary
ONE SIZE DOES NOT FIT ALL

CIDR 2011

HYRISE

VLDB 2010

SOCC 2011

CIDR 2013

VLDB 2012

VLDB 2013

SIGMOD (demo)
Acknowledgements

- Jens Dittrich
- Jorge Quiane
- Felix Martin Schuhknecht
- Endre Palatinus
- Karen Khachatryan
- Stefan Richter
- Alexander Bunte

- Sam Madden
- Stefan Richter
- Stefan Schuh
- Joerg Schad
- Yagiz Kargin
- Vinay Setty
- Vladimir Pavlov