Workload?
OLAP
Streaming
Scan-oriented
Archiving
Log-processing
Web-search
Streaming

OLTP

OLAP

Archiving

Log-processing

Web-search

Scan-oriented
Streaming

OLTP

OLAP

Archiving

Log-processing

Web-search

Scan-oriented
• Primary/Secondary Index
• Materialized Views
• Vertical Partitioning
• Horizontal Partitioning
• Primary/Secondary Index
• Materialized Views
• Vertical Partitioning
• Horizontal Partitioning

• Human Intervention
• Slow Response Time
• Inefficient
• Online Indexes
• Dynamic Materialized Views
• Database Cracking
• Adaptive Merging
• Online Indexes
• Dynamic Materialized Views
• Database Cracking
• Adaptive Merging

• Still, does not change core system features
• Several physical designs at schema level
• No true physical data independence
• Physical design not effective
PLAN B?
OLAP
Log-processing
• Tedious
• Expensive
• Complex ETL
• Inefficient
Mixed Workloads
Dynamic Workloads
‘Zoo’ of systems
Indexes

Row

Column

Raw files

Row+Column
• Flexible data storage layer
• Adapt layout to workload
• Logical journal of data operations
• Arbitrary physical representations
• New concept: Storage Views
### OctopusDB

- Flexible data storage layer
- Adapt layout to workload
- Logical journal of data operations
- Arbitrary physical representations
- New concept: 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></td>
<td>omg</td>
<td>90</td>
<td>148.9</td>
</tr>
</tbody>
</table>
• Flexible data storage layer
• Adapt layout to workload
• Logical journal of data operations
• Arbitrary physical representations
• New concept: Storage Views
• Flexible data storage layer
• Adapt layout to workload
• Logical journal of data operations
• Arbitrary physical representations
• New concept: Storage Views
OctopusDB

• Flexible data storage layer
• Adapt layout to workload
• Logical journal of data operations
• Arbitrary physical representations
• New concept: Storage Views
### OctopusDB

- **Flexible data storage layer**
- **Adapt layout to workload**
- **Logical journal of data operations**
- **Arbitrary physical representations**
- **New concept: Storage Views**

<table>
<thead>
<tr>
<th>Log</th>
<th>Column grouped</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lkj</td>
</tr>
<tr>
<td></td>
<td>Row</td>
</tr>
<tr>
<td></td>
<td>Column</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>56</td>
<td>887.9</td>
</tr>
<tr>
<td>89</td>
<td>445.35</td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>234.67</td>
<td></td>
</tr>
</tbody>
</table>
OctopusDB

- Flexible data storage layer
- Adapt layout to workload
- Logical journal of data operations
- Arbitrary physical representations
- New concept: Storage Views
• Flexible data storage layer
• Adapt layout to workload
• Logical journal of data operations
• Arbitrary physical representations
• New concept: Storage Views
Example: Flight Tickets

\[
\pi_{\text{customer.}*}(\sigma_{a_1 = x_1 \ldots a_n = x_n}(\text{tickets.customer}_\text{id} \bowtie \text{customer.id}))
\]
Example: Flight Tickets

Log SV

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

Result

Col SV

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

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

Result

Row SV

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

\[ \pi_{\text{customer}.*}(\sigma_{\text{bag}=\text{customers}}(\text{tickets.customer_id} \land \text{customer.id})) \]

Result
Example: Flight Tickets
Example: Flight Tickets
<table>
<thead>
<tr>
<th>Use-Case (traditional systems)</th>
<th>Storage view definition type</th>
<th>example query</th>
</tr>
</thead>
<tbody>
<tr>
<td>row store</td>
<td>Row SV</td>
<td>any</td>
</tr>
<tr>
<td>column store</td>
<td>Col SV</td>
<td>any</td>
</tr>
<tr>
<td>PAX</td>
<td>PAX SV</td>
<td>any</td>
</tr>
<tr>
<td>fractured mirrors</td>
<td>Row SV</td>
<td>same query for both Row SV and Col SV</td>
</tr>
<tr>
<td>column groups</td>
<td>Row SV</td>
<td>$\pi a_1, \ldots, a_k \quad \pi a_{k+1}, \ldots, a_m$</td>
</tr>
<tr>
<td>index</td>
<td>Index SV</td>
<td>any</td>
</tr>
<tr>
<td>indexed row store</td>
<td>Index SV(Row SV)</td>
<td>any</td>
</tr>
<tr>
<td>indexed column store</td>
<td>Index SV(Col SV)</td>
<td>any</td>
</tr>
<tr>
<td>read-optimized indexed column store + differential write-optimized row store</td>
<td>Index SV(Col SV)</td>
<td>$\sigma_{t&lt;now()-1day}$ Row SV $\sigma_{t\geq now()-1day}$</td>
</tr>
<tr>
<td>partial index</td>
<td>Index SV</td>
<td>$\sigma_{420 \leq a_k \leq 42000}$</td>
</tr>
<tr>
<td>projection index</td>
<td>Col SV</td>
<td>$\pi a_k$</td>
</tr>
<tr>
<td>partial projection index</td>
<td>Index SV(Col SV)</td>
<td>$\pi a_k \left(\sigma_{420 \leq a_k \leq 42000}\right)$</td>
</tr>
<tr>
<td>DSMS</td>
<td>Index SV</td>
<td>$\sigma_{t\geq now()-5min}$</td>
</tr>
<tr>
<td>DSMS + archive</td>
<td>Index SV</td>
<td>$\sigma_{t\geq now()-5min}$ $\sigma_{t&lt;now()-5min}$</td>
</tr>
<tr>
<td>snapshot</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td>replicated row store</td>
<td>Row SV</td>
<td>same query for both Row SV</td>
</tr>
<tr>
<td>Use-Case (new system)</td>
<td>Storage view definition type</td>
<td>example query</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>OLTP + OLAP</td>
<td>Row SV</td>
<td>$\sigma_{t \geq \text{now}() - 1\text{day}}$</td>
</tr>
<tr>
<td></td>
<td>Col SV</td>
<td>$\sigma_{t &lt; \text{now}() - 1\text{day}}$</td>
</tr>
<tr>
<td>DSMS + OLTP</td>
<td>Index SV</td>
<td>$\sigma_{t \geq \text{now}() - 5\text{min}}$</td>
</tr>
<tr>
<td></td>
<td>Row SV</td>
<td>$\sigma_{t &lt; \text{now}() - 5\text{min}}$</td>
</tr>
<tr>
<td>DSMS + archive OLTP</td>
<td>Index SV</td>
<td>$\sigma_{t \geq \text{now}() - 5\text{min}}$</td>
</tr>
<tr>
<td></td>
<td>Row SV</td>
<td>$\sigma_{\text{now}() - 1\text{day} \leq t &lt; \text{now}() - 5\text{min}}$</td>
</tr>
<tr>
<td></td>
<td>Col SV</td>
<td>$\sigma_{t &lt; \text{now}() - 1\text{day}}$</td>
</tr>
<tr>
<td>other hybrid</td>
<td>any combination of the above</td>
<td>any</td>
</tr>
</tbody>
</table>
OctopusDB

- ‘Mimic’ several systems
- No ‘zoo’ overheads
- One-size-fits-all

- Flexible data storage layer
- Adapt layout to workload
- Logical journal of data operations
- Arbitrary physical representations
- New concept: Storage Views
Interesting
Trojan Techniques

- Good Trojans
- Existing system
- Source-code not required
- Inject additional layouts
How does it work?

• Exploit UDFs provided by existing systems
• Inject pieces of code
• Hack layouts into the UDFs
• UDF as mapping between logical and physical view of data
How does it work?

- Exploit UDFs provided by existing systems
- Inject pieces of code
- Hack layouts into the UDFs
- UDF as mapping between logical and physical view of data
How does it work?

- Exploit UDFs provided by existing systems
- Inject pieces of code
- Hack layouts into the UDFs
- UDF as mapping between logical and physical view of data
- Novel use of UDFs
Use Case 1: OLAP in Row-stores
OLTP
OLAP
OLTP

OLAP?

• Can we push the limits of row stores?
Trojan Columns

Relation

<table>
<thead>
<tr>
<th>Customer</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>

In this section, we present Trojan Columns: a novel way of integrating column store technology into existing database products without invading or making heavy changes in the system. Trojan Columns uses lightweight User Defined Functions (UDFs) to store and access the data. Unlike other techniques that either reuses the query optimizer or injects column store functionality into existing closed source database products, Trojan Columns is radically different from data stores: either having a radically different technique or doing deep seated changes.

Typically database systems support three kinds of UDFs based on their return types: (i) scalar value returning UDFs, (ii) row returning UDFs, and (iii) table (of rows) returning UDFs. We describe how to rewrite the user queries in order to use certain UDFs within the database system and exploit them whenever we need to store or access data. The data is actually stored in a compressed column-oriented fashion on disk. But the UDFs translate it into the row layout for the query processor. This means that the physical data in the system is entirely agnostic of the column store functionality injected within, i.e. no new SQL keywords, or data types, or any entries in the system catalog have to be added.

Third, Trojan Columns uses standard database tables to store the blobs. The data is actually stored as a separate BLOB (binary large object) in a physical table. Experiments show that we can achieve better performance with Trojan Columns compared to Column Indexes not only in data storage, but also in data access. Similarly, we found bigger segment sizes, e.g. 25K for Smith and 128K for Stevens (inserting records alternatively)

Our idea is to use User Defined Functions (UDFs) as an access layer for data storage and retrieval. To do so, we create and install certain UDFs within the database system and exploit them whenever we need to store or access data. All the while, users' view remains (almost) unchanged. User queries remain (almost) unchanged. Reuses the query optimizer (at least partially) and therefore no modification is required. Does not invade or make heavy changes in the system; rather, the changes are transparently injected inside the DBMS. However, in practice, Trojan Columns requires changes to database products e.g. IBM DB2, Oracle.

Trojan Columns is plug-and-play, but also allows us to customize the data storage functionality. Additionally, we could simply let the database system apply the default compression method for blobs, e.g. TOAST for dumps. However, in practice, Trojan Columns uses lightweight UDFs to store and access the data. Essentially, we have full control and flexibility to decide how to access the data. For example, for an airline company, instead of making deep changes in the data access layer in order to access the data in a column store, we trick the database into believing that the data is still stored in a blob, instead of a new index type in case of Column Indexes. The database uses its own physical storage mechanism to persist the blob, instead of a new index type in case of Column Indexes. The consequence is that the blobs must reside on disk.

For example, consider the following entries of a Customer relation.

```sql
CREATE TABLE Customer_trojan (segment_ID serial, attribute_ID serial, blob_data bytea);
```

We store each entry in the blob_data column of the above table as a separate BLOB, thus mimicking a column-oriented storage. Experimentally, we found bigger segment sizes, e.g.

<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>
Trojan Columns

Relation

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

Physical Table

<table>
<thead>
<tr>
<th>Customer_trojan</th>
<th>blob_data</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>
Trojan Columns

Relation

| Customer | | | | |
| name | phone | market_segment |
| smith | 2134 | automobile |
| john | 3425 | household |
| kim | 6756 | furniture |
| joe | 9878 | building |
| mark | 4312 | building |
| steve | 2435 | automobile |
| jim | 5766 | household |
| ian | 8789 | household |

Physical Table

<p>| Customer_trojan | | | | |</p>
<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

Relation

<table>
<thead>
<tr>
<th>Customer</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>phone</td>
<td>market_segment</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>

Physical Table

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

Relation

<table>
<thead>
<tr>
<th>Customer</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>phone</td>
<td>market_segment</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>

Physical Table

<table>
<thead>
<tr>
<th>Customer_trojan</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>segment_ID</td>
<td>attribute_ID</td>
<td>blob_data</td>
</tr>
<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

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

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

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

Scan

lineitem

Also note that the query plan in Figure 1 works well because the output of UDF can be processed by the output operator to the UDF. This is because we need to interpret Trojan Columns. Since Trojan Columns internally store data in column-oriented fashion, we need to translate the data back to row representation (for part not the row representation) for processing. For queries having too many operators down the UDF, exploring these in more detail is pushed inside the UDF, while the join is still performed outside. This holds true even for nested queries, e.g. TPC-\text{H}.

For queries having join conditions, we simply push down the scan, selection, and project attributes. Since the UDF return type is still the complete projected attributes. Since Trojan Columns internally store data in column-oriented fashion, we need to translate the data back to row representation (for part not the row representation) for processing. For queries having too many operators down the UDF, exploring these in more detail is pushed inside the UDF, while the join is still performed outside. This holds true even for nested queries, e.g. TPC-\text{H}.

Thus, we see that UDFs can be seamlessly integrated into the database query executor. Figure 1 shows the logical query plan for query 6. Below, let’s see how we process queries using scanUDF and join them based on the join condition. The advantage would be there is no need to push down the UDF. The UDF would then have to access two physical tables to scan the table. Additionally, we may also push down other selection attributes first. Then, before returning the tuple, the UDF inspects the next selection attribute value to see if it satisfies the conditions. When the conditions are not satisfied, the UDF inspects the next selection attribute value. When the conditions are satisfied, the UDF inspects the next selection attribute value. When the conditions are not satisfied, the UDF inspects the next selection attribute value. When the conditions are satisfied, the UDF inspects the next selection attribute value.

Furthermore, in the presence of Trojan Columns, processing queries using scanUDF works well because the output of UDF can be processed by the output operator to the UDF. This is because we need to interpret Trojan Columns. Since Trojan Columns internally store data in column-oriented fashion, we need to translate the data back to row representation (for part not the row representation) for processing. For queries having too many operators down the UDF, exploring these in more detail is pushed inside the UDF, while the join is still performed outside. This holds true even for nested queries, e.g. TPC-\text{H}.

Thus, we see that UDFs can be seamlessly integrated into the database query executor. Figure 1 shows the logical query plan for query 6. Below, let’s see how we process queries using scanUDF and join them based on the join condition. The advantage would be there is no need to push down the UDF. The UDF would then have to access two physical tables to scan the table. Additionally, we may also push down other selection attributes first. Then, before returning the tuple, the UDF inspects the next selection attribute value to see if it satisfies the conditions. When the conditions are not satisfied, the UDF inspects the next selection attribute value. When the conditions are satisfied, the UDF inspects the next selection attribute value. When the conditions are not satisfied, the UDF inspects the next selection attribute value. When the conditions are satisfied, the UDF inspects the next selection attribute value. When the conditions are not satisfied, the UDF inspects the next selection attribute value. When the conditions are satisfied, the UDF inspects the next selection attribute value.

Furthermore, in the presence of Trojan Columns, processing queries using scanUDF works well because the output of UDF can be processed by the output operator to the UDF. This is because we need to interpret Trojan Columns. Since Trojan Columns internally store data in column-oriented fashion, we need to translate the data back to row representation (for part not the row representation) for processing. For queries having too many operators down the UDF, exploring these in more detail is pushed inside the UDF, while the join is still performed outside. This holds true even for nested queries, e.g. TPC-\text{H}.

Thus, we see that UDFs can be seamlessly integrated into the database query executor. Figure 1 shows the logical query plan for query 6. Below, let’s see how we process queries using scanUDF and join them based on the join condition. The advantage would be there is no need to push down the UDF. The UDF would then have to access two physical tables to scan the table. Additionally, we may also push down other selection attributes first. Then, before returning the tuple, the UDF inspects the next selection attribute value to see if it satisfies the conditions. When the conditions are not satisfied, the UDF inspects the next selection attribute value. When the conditions are satisfied, the UDF inspects the next selection attribute value. When the conditions are not satisfied, the UDF inspects the next selection attribute value. When the conditions are satisfied, the UDF inspects the next selection attribute value. When the conditions are not satisfied, the UDF inspects the next selection attribute value. When the conditions are satisfied, the UDF inspects the next selection attribute value.

Furthermore, in the presence of Trojan Columns, processing queries using scanUDF works well because the output of UDF can be processed by the output operator to the UDF. This is because we need to interpret Trojan Columns. Since Trojan Columns internally store data in column-oriented fashion, we need to translate the data back to row representation (for part not the row representation) for processing. For queries having too many operators down the UDF, exploring these in more detail is pushed inside the UDF, while the join is still performed outside. This holds true even for nested queries, e.g. TPC-\text{H}.
Example: TPC-H Query 6

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

\[
\begin{align*}
\gamma_{\text{agg}} (\text{extendedprice} \times \text{discount}) \\
\sigma \text{shipdate BETWEEN '1994-01-01' AND '1995-01-01'} \\
\pi \text{quantity, discount} \\
\pi \text{extendedprice, shipdate} \\
\text{SCAN} \quad \text{lineitem}
\end{align*}
\]
evaluates the selection predicate. If the predicates satisfy then the selection predicate to the UDF, as shown in Figure I.

Exploring these in more detail attributes as parameters to the UDF. The UDF now returns only the down the projection operator to the UDF, i.e. pass the projected at-

is much smaller table and it does not pay off to use a UDF for ing the standard database access method. This is because
database query executor. Let's consider query 6

in column-oriented fashion, we need to translate the data back to

3. QUERYING TROJAN COLUMNS

Figure 2: Example UDF query plan for TPC-H query 14.
C-Store Benchmark

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

Query Time (sec)

- Standard Row
- Trojan Columns

Q1 | Q6 | Q12 | Q14

Q1: 75 75 75 75
Q6: 75 75 75 75
Q12: 25 25 25 25
Q14: 25 25 25 25

* tpch.org/tpch
## Micro-Benchmark

<table>
<thead>
<tr>
<th># referenced attributes (r)</th>
<th>1E-06</th>
<th>1E-05</th>
<th>1E-04</th>
<th>1E-03</th>
<th>1E-02</th>
<th>1E-01</th>
<th>1E+00</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>2.13</td>
<td>2.13</td>
<td>2.11</td>
<td>2.06</td>
<td>1.55</td>
<td>0.47</td>
<td>0.06</td>
</tr>
<tr>
<td>15</td>
<td>4.64</td>
<td>4.62</td>
<td>4.55</td>
<td>4.27</td>
<td>2.57</td>
<td>0.55</td>
<td>0.06</td>
</tr>
<tr>
<td>13</td>
<td>5.00</td>
<td>5.00</td>
<td>4.94</td>
<td>4.61</td>
<td>2.70</td>
<td>0.57</td>
<td>0.06</td>
</tr>
<tr>
<td>11</td>
<td>5.79</td>
<td>5.82</td>
<td>5.75</td>
<td>5.24</td>
<td>2.87</td>
<td>0.56</td>
<td>0.06</td>
</tr>
<tr>
<td>9</td>
<td>6.39</td>
<td>6.38</td>
<td>6.25</td>
<td>5.79</td>
<td>3.11</td>
<td>0.54</td>
<td>0.06</td>
</tr>
<tr>
<td>7</td>
<td>7.00</td>
<td>6.96</td>
<td>6.80</td>
<td>6.23</td>
<td>3.17</td>
<td>0.56</td>
<td>0.06</td>
</tr>
<tr>
<td>5</td>
<td>10.96</td>
<td>10.94</td>
<td>10.55</td>
<td>9.27</td>
<td>3.75</td>
<td>0.57</td>
<td>0.06</td>
</tr>
<tr>
<td>3</td>
<td>12.86</td>
<td>13.57</td>
<td>13.22</td>
<td>11.03</td>
<td>4.16</td>
<td>0.56</td>
<td>0.06</td>
</tr>
<tr>
<td>1</td>
<td>17.43</td>
<td>17.61</td>
<td>16.61</td>
<td>13.57</td>
<td>4.39</td>
<td>0.57</td>
<td>0.06</td>
</tr>
</tbody>
</table>

selectivity (fraction of tuples accessed)
 versus Column Stores

![Query Time Comparison Graph]

- Standard Row
- Trojan Columns
- DBMS-Y

Query Time (sec)

- Q1
- Q6
- Q12
- Q14

- TPC-H
- D. J. Abadi, S. Madden, and M. Ferreira. Integrating Compression into an Existing Database System Environment (Without It Even Noticing). In(SIGMOD, 3(1), 2010.
Trojan Columns
Advantages

- Column + Row storage
- Much better performance
- Closed source system
Use Case 2: Big Data Analytics
Web-log Processing

- Scan Tasks

User visits from different countries
Web-log Processing

- Scan Tasks
- Selection Tasks

User visits with duration greater than 10s
Web-log Processing

- Scan Tasks
- Selection Tasks
- Projection Tasks

URL and duration of each user visit
Web-log Processing

- Scan Tasks
- Selection Tasks
- Projection Tasks
- Join Tasks

Average PageRank visited by each user IP
Web-log Processing

- Scan Tasks
- Selection Tasks
- Projection Tasks
- Join Tasks
Trojan Index

- Each HDFS block sorted
- Each block contains an index
Trojan Index

- Each HDFS block sorted
- Each block contains an index
- Index access in UDF
Trojan Index Creation

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

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 \text{indexBuilder}_{a_i}(ivs))]\)
2.1 The Hadoop Plan

As a physical operator DAG. To our knowledge, this paper and DBMS have the same expressiveness, i.e. any MapReduce task

All experiments are run on Amazon's EC2 Cloud.

We make the following contributions:

We provide a non-invasive, DBMS-

Not to be confused with the distributed file system

A Trojan

to integrate

... for details.

Asymmetry may occur if a mapper subplan (here: M3) consumes less input data and/or

M1 shows a subplan processing three spill files. These spill files are then retrieved from

... stored on the distributed file system

which pre-reduces the data.

The output is materialized on disk

This is faster than pert

... created by the framework

We set the operation defined above

The distributed file system stores the data for

Input

Determine the number of reducer subplans (here: R).

1. Again, we apply

... as a value containing all attributes of the

SplitID

key of record

T T T T T

6

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. 
5. 
6. 
7. 
8. 
9. 
10. 
11. 
12. 
13. 
14. 
15. 
16. 
17. 

return splits;
Trojan Index Access

The Hadoop Plan: Hadoop’s processing pipeline expressed as a physical query execution plan. Let’s analyze The Hadoop Plan in more detail:

**Data Load Phase**
- Fetch
- Store
- Scan
- PPartSplit

**Map Phase**
- Fetch
- Store
- Scan
- PPartSplit

**Shuffle Phase**
- Fetch
- Buffer
- Merge
- Store

**Reduce Phase**
- Fetch
- Buffer
- Merge
- Store

**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);
```
**Trojan Index Access**

**Algorithm 3: Trojan Index itemize.next UDF**

<table>
<thead>
<tr>
<th>Input</th>
<th>KeyType key, ValueType value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>has more records</td>
</tr>
</tbody>
</table>

1. if offset < splitEnd then
2. Record nextRecord = ReadNextRecord(split);
3. offset += nextRecord.size();
4. if nextRecord.key < highKey then
5.   SetKey Value(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
Trojan Index Advantages

- Each HDFS block sorted
- Each block contains an index
- Index access in UDF
- Scan + Index data accesses
- Parallel index lookups
- Non-invasive system changes
- Much better performance
Trojan Join

- Each HDFS block co-partitioning over two relations
- Join relations are co-located
Trojan Join

• Each HDFS block co-partitioning over two relations
• Join relations are co-located
• Co-partitioned join in UDF
This paper has proposed new index and join techniques. The experimental results demonstrate that our implementation of these techniques on top of Hadoop++ results in better performance of fault-tolerant jobs. We also observe that as we increase the split sizes, the performance benefits stem from exploiting schema knowledge on runtime and fault tolerance of Hadoop jobs. This symbolizes a tradeoff: as the index coverage also increases, performance of fault-tolerant jobs decreases with larger splits as it requires more time to recompute lost tasks. This tradeoff is further improved for both selection and join tasks. As we increase the split sizes, the underlying Hadoop framework may improve over Hadoop, but again, there is no need to use a Hadoop++ to Hadoop interface we believe that we do not have issues.

We would like to thank all students of the course "Data Processing on Large Clusters" for the fruitful discussions.

We implemented our Trojan techniques on top of Hadoop++, using Hadoop as a foundation.

Figure 0: Benchmark Results related to Indexing and Join Processing

---

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

- Re- + Co- partitioned join
- Parallel join processing
- Non-invasive system changes
- Much better performance

- Each HDFS block co-partitioning over two relations
- Join relations are co-located
- Co-partitioned join in UDF
Trojan Layouts

- Each HDFS block in row or column
- Each block replica in different layout
- Pick right layout in UDF
Trojan Layouts provide for a good trade-off between redundant attribute access and tuple reconstruction overhead. Table 1 summarizes this observation. Hence, per-replica Trojan Layouts significantly reduce reconstruction cost by co-locating attributes in the same column, amortizing tuple reconstruction cost in Hadoop-Row up to factor of 4 and in Hadoop-PAX up to factor of 3 in the worst case (e.g., Q2). This is because Hadoop-Row performs worse than Hadoop-PAX, having at least the same performance as Hadoop-PAX. Thus, Trojan Layouts do not perform worse than Hadoop-Row — e.g., LineItem and Customer do not have. On the other hand, we observe that for those queries referencing many attributes, all attributes are referenced and Trojan Layouts have an extra tuple reconstruction cost. Further, the results show that Trojan Layouts never perform worse than Hadoop-Row and to perform slightly outperforms Hadoop-PAX in the worst case. This is because Hadoop-PAX increases as the number of referenced attributes increases as well. Trojan Layouts amortize tuple reconstruction cost in Hadoop-Row and Hadoop-PAX up to a factor of 3 in the worst case.
Trojan Layouts Advantages

- Row, PAX, Column-group layouts
- Several layouts at the same time
- Non-invasive system changes
- Much better performance

- Each HDFS block in row or column
- Each block replica in different layout
- Pick right layout in UDF
Open Issues

• Automatically rewriting user queries
• Avoiding UDF call overheads for low selectivity
• Putting all Trojan Techniques together in a single system
• What to store, How to store, Where to store
• Trojan Techniques: one way of approaching OctopusDB
• Trojan Techniques: first step towards OctopusDB
• Storage View optimization: selection, transformation, update propagation
Mixed Workloads

Big Problem

OLAP

Streamlining

OLTP

Log-processing

Web-search

Big Data

Not Sufficient

Complicated
Mixed Workloads

Big Problem

Not Sufficient

Complicated

OLAP

OLTP

Archiving

Log-processing

Web-search

Scan-oriented

Big Data

OLAP

Streaming

Trojan Techniques

• Existing system
• Inject additional layouts
• Source-code not required
• Good use of Trojans

• No heavy changes
• Affect from inside
• Similar to PAX, fP+tree

Use Case 1:
OLAP in Row-stores

Use Case 2:
Big Data Analytics

OctopusDB

• ‘Mimic’ several systems
• No ‘zoo’ overheads
• One-size-fits-all

Flexible data storage layer
• Adapt layout to workload
• Logical journal of data operations
• Arbitrary physical representations
• New concept: Storage Views

• Flexible data storage layer
• Adapt layout to workload
• Logical journal of data operations
• Arbitrary physical representations

Our Vision

First Steps

OLAP in Row-stores

[VLDB PhD Workshop, 2010]

[CIDR, 2011]

Big Data Analytics

[VLDB, 2010]

[SOCC, 2011]

[VLDB, 2010]

[CIDR, 2013]

(Under Submission)