Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing)

Jens Dittrich
Jorge-Arnulfo Quiané-Ruiz
Alekh Jindal
Yagiz Kargin
Vinay Setty
Jörg Schad

1 Information Systems Group, Saarland University
http://infosys.cs.uni-saarland.de

2 International Max Planck Research School for Computer Science
http://www.imprs-cs.de/
The Parallel DBMS vs MapReduce Debate

<table>
<thead>
<tr>
<th></th>
<th>Parallel DBMS</th>
<th>MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>licensing costs</td>
<td>usually high</td>
<td>none</td>
</tr>
<tr>
<td>administration</td>
<td>difficult</td>
<td>easy</td>
</tr>
<tr>
<td>upfront schema</td>
<td>must have</td>
<td>not required</td>
</tr>
<tr>
<td>user</td>
<td>advanced</td>
<td>beginner</td>
</tr>
<tr>
<td>scalability</td>
<td>10-100es of nodes</td>
<td>&gt;10,000 nodes</td>
</tr>
<tr>
<td>failover, large clusters</td>
<td>suboptimal</td>
<td>very good</td>
</tr>
<tr>
<td>performance</td>
<td>very good</td>
<td>suboptimal</td>
</tr>
</tbody>
</table>

- see also [Pavlo etal, SIGMOD 2009] comparison
- benchmark to compare Parallel DBMS with MapReduce
- showed superiority of Parallel DBMS over MapReduce
MapReduce ≠ MapReduce ≠ MapReduce

- but, MapReduce is **three different** things:

1. a **programming paradigm**:
   - it allows users to specify analytical tasks
   - need to provide two functions only: `map()` and `reduce()`

2. a description of a **processing pipeline and system**:
   - that system computes the result to a MapReduce-job
   - MapReduce-job: `map()`, `reduce()`, and some input data
   - scales to very large clusters, > 10,000 nodes

3. several implementations of (2):
   - Google‘s proprietary MapReduce, Hadoop, ...
### Related Work

<table>
<thead>
<tr>
<th>(2) Processing pipeline and system</th>
<th>(1) Programming Paradigm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MapReduce</td>
</tr>
<tr>
<td></td>
<td>Hadoop</td>
</tr>
<tr>
<td></td>
<td>PDBMS</td>
</tr>
<tr>
<td></td>
<td>Hybrid</td>
</tr>
</tbody>
</table>

- **(1) Programming Paradigm**
  - MapReduce
  - SQL
  - Hybrid

- **(2) Processing pipeline and system**
  - MapReduce
  - PDBMS
  - Hybrid
## Related Work

<table>
<thead>
<tr>
<th>(2) Processing pipeline and system</th>
<th>(1) Programming Paradigm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MapReduce</td>
</tr>
<tr>
<td><strong>MapReduce</strong></td>
<td>Hadoop</td>
</tr>
<tr>
<td><strong>PDBMS</strong></td>
<td>Greenplum</td>
</tr>
<tr>
<td><strong>Hybrid</strong></td>
<td></td>
</tr>
</tbody>
</table>
## Related Work

<table>
<thead>
<tr>
<th>(2) Processing pipeline and system</th>
<th>(1) Programming Paradigm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MapReduce</td>
</tr>
<tr>
<td><strong>MapReduce</strong></td>
<td>Hadoop</td>
</tr>
<tr>
<td><strong>PDBMS</strong></td>
<td>proprietary, expensive</td>
</tr>
<tr>
<td><strong>Hybrid</strong></td>
<td></td>
</tr>
</tbody>
</table>
### Related Work

<table>
<thead>
<tr>
<th>(2) Processing pipeline and system</th>
<th>(1) Programming Paradigm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MapReduce</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PDBMS</td>
</tr>
<tr>
<td></td>
<td>Hybrid</td>
</tr>
</tbody>
</table>

**Table:**

<table>
<thead>
<tr>
<th>MapReduce</th>
<th>SQL</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadoop</td>
<td>Hive</td>
<td>admin costs?</td>
</tr>
</tbody>
</table>
| proprietary, expensive | back to initial interface hurdle | }  

**Diagram:**

- **MapReduce:** Hadoop
- **SQL:** Hive
- **Hybrid:** admin costs?
Related Work

Research Challenge:

Can we invent a system that:

1. keeps the MapReduce programming paradigm and the MapReduce execution engine?
2. approaches Parallel DBMSs in performance?
Related Work

<table>
<thead>
<tr>
<th>(2) Processing pipeline and system</th>
<th>(1) Programming Paradigm</th>
<th>MapReduce</th>
<th>SQL</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>MapReduce</td>
<td>Hadoop++</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PDBMS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hybrid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hadoop++ System Vision

(1) MapReduce programming paradigm

MapReduce program analysis
e.g. [Cafarella and Ré, WebDB2010]
[Iu and Zwaenepoel, EuroSys 2010]

Optimization
e.g. cost models [Morton et.al. SIGMOD 2010]

MapReduce program generation
this paper, Hadoop++

(2) MapReduce processing pipeline and system
Features of Hadoop++

(1) we **do not change** the existing Hadoop framework at all

  *advantage:* no need to maintain and test Hadoop code changes
  *advantage:* future improvements of Hadoop orthogonal to Hadoop++

(2) **inject** our technology inside Hadoop, hide it

  *advantage:* clear layering
  *advantage:* no extra operators, no pipeline changes

(3) **do not change** the MapReduce programming paradigm

  *advantage:* nothing changes from the user-side

(4) still trick Hadoop into using **more efficient plans**

  *advantage:* improve runtime performance considerably

**How do we do this?**

Well, let’s first better understand the existing Hadoop processing pipeline....
• partition data into blocks
• replicate data to nodes
• store data

• scan input data blocks
• form splits
• send data to processing nodes
• break data into records
• call map() for each record
• pregroup and preaggregate output
• store output locally

• redistribute data over processing nodes
• merge subsets belonging to same reducer into single file

• perform final grouping
• call reduce() for each group
• store output
Observations on The Hadoop Plan

- again: no real operators, all hard-coded
- large distributed external merge sort
- sort in order to do a sort-based grouping
- full scan access at all times
- not only two functions, i.e. `map` and `reduce`,
- but...
The Hadoop Plan has ten user-defined functions (UDFs):

- block
- split
- itemize
- mem
- map
- sh
- cmp
- grp
- combine
- reduce

The diagram shows a flowchart of the Hadoop execution plan with ten user-defined functions. It illustrates the Data Load, Map, Shuffle, and Reduce phases, highlighting how the data is processed through blocks, splits, and other operations. The figure is an example with 4 mappers and 2 reducers.
Hadoop++ Approach: Trojan Techniques

- **Trojan Index:**
  - at data load time: create index
  - at query time: use index access plan

- **Trojan Join:**
  - at data load time: create co-partitions
  - at query time: compute all join results locally
Trojan Index Creation

Desired layout:

- **Index Creation Algorithm:**
  - read input split
  - add small clustered Trojan index (we use a CSS-tree)
  - add some metadata

- **Implementation:**
  - a MapReduce program

E.g. 8MB of index for 1GB of data
Trojan Index Creation

\[
\text{map}(key \ k, \ \text{value} \ \nu) \mapsto [(\text{getSplitID}() \oplus \text{prj}_a (k \oplus \nu), \ k \oplus \nu)]
\]

form intermediate key with splitID and index key \(a\)

figure shows example with 4 mappers and 2 reducers

+: concatenate schemas
### The Hadoop Plan

Hadoop uses a plan to determine how data is processed. The plan may be run on a distributed file system to contain analytical data processing.

### Trojan Join

 Trojan Joins do neither require a hard-coded SQL nor SQL to do so. Trojan Indexes are created at data load time and thus have independent indexing technique coined Trojan Index. The Hadoop Plan shows the MapReduce plan corresponding to the index creation.

### Index Creation

#### 3.1 Index Creation

Here we discuss the Trojan Index creation and subsequent query processing. Since we are building Trojan Index per split, we need to prepare each split separately.

#### 3.2 Trojan Index Creation

**Phase**

<table>
<thead>
<tr>
<th><strong>Map Phase</strong></th>
<th><strong>Shuffle Phase</strong></th>
<th><strong>Reduce Phase</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch → Store</td>
<td>Fetch → Store</td>
<td>Fetch → Store</td>
</tr>
<tr>
<td>Store</td>
<td>Store</td>
<td>Store</td>
</tr>
<tr>
<td>Scan</td>
<td>Scan</td>
<td></td>
</tr>
</tbody>
</table>

**Data Load Phase**

- **Fetch**
- **Store**
- **Scan**

**Algorithm 1**

```java
foreach File[] files
    while Long InputStream == splitEnd
        if Long InputStream == splitStart
            ReadIndex
        newSplit = getSplitSize
        if splitEnd - splitStart > newSplit
            skip the split,
            set to the beginning of the split
        else
            seek and read
            put stream
            input stream
            file
            seek and read
            put stream
```

**Figure 5**

The figure shows example with 4 mappers and 2 reducers.
Trojan Index Creation

Data Load Phase

Map Phase

Shuffle Phase

\[
\text{sh(key } k, \text{ value } v, \text{ int numPartitions}) \mapsto \text{k.splitID } \% \text{ numPartitions}
\]

figure shows example with 4 mappers and 2 reducers

\[\oplus:\text{concatenate schemas}\]
Trojan Index Creation

Data Load Phase

Map Phase

Reduce Phase

\[ \text{grp(key } k1, \text{key } k2) \rightarrow \text{compare}(k1.s\text{plitID}, k2.s\text{plitID}) \]

build groups on splitID only

figure shows example with 4 mappers and 2 reducers

\[ \oplus: \text{concatenate schemas} \]
Trojan Index Creation

3.1 Index Creation

We discuss the Trojan Index creation and subsequent query processing.

reduce(key \(ik\), vset \(ivs\)) \(\mapsto\) 
\([ivs \oplus \text{indexBuilder}_a(ivs)]]\)

build CSS-tree for each ivs set

\(\oplus\): concatenate schemas

figure shows example with 4 mappers and 2 reducers
Trojan Index Query Processing

- **Query Algorithm:**
  - for each split:
    - read footer to obtain split size
    - read header to obtain \([\text{key}_{\text{min}}, \text{key}_{\text{max}}]\)-range of index
    - if search key overlaps \([\text{key}_{\text{min}}, \text{key}_{\text{max}}]\)-range:
      - read CSS-tree into main memory
      - read only records qualifying for search predicate
      - only pass those records to map()
    - else
      - skip this split

- **Implementation:**
  - a MapReduce program
  - provide `split` and `itemize` UDF
  - everything else unchanged
Trojan Join Co-Partitioning

Desired layout:

- **Co-Partition Creation Algorithm:**
  - read input data
  - create co-partitioned data based on join keys of two relations
  - add some metadata

- **Implementation:**
  - a MapReduce program
Trojan Join Co-Partitioning Details

map(key k, value v) →

\[
\begin{cases}
([\text{pr}j_a (k \oplus v), k \oplus v]) & \text{if input}(k \oplus v) = T, \\
([\text{pr}j_b (k \oplus v), k \oplus v]) & \text{if input}(k \oplus v) = S.
\end{cases}
\]

form intermediate key with join key a from T and b from S

Figure 3: MapReduce Plans

Join T.a=S.b ⊕: concatenate schemas

The Merge phase is a join operator which follows the reduce phase.

Input

Record of Relation T

FileType key

ValueType value

Output

SetKeyValue(index key value of the next record is less than the high key r)

The Merge phase is a join operator which follows the reduce phase.

Figure 3: MapReduce Plans

Join T.a=S.b ⊕: concatenate schemas

The Merge phase is a join operator which follows the reduce phase.

Figure 3: MapReduce Plans

Join T.a=S.b ⊕: concatenate schemas

The Merge phase is a join operator which follows the reduce phase.
Trojan Join Co-Partitioning Details

- Data Load Phase
  - H1: Fetch, Store, Scan
  - H2: Fetch, Store
  - H3: Fetch, Store, Scan
  - H4: Fetch, Store

- Map Phase
  - L: Fetch, Store, Scan
  - T1: Fetch, Replicate, Block
  - PPart: Block, Replicate
  - T6: Replicate, Replicate

- Shuffle Phase
  - Fetch, Buffer, Store
  - Merge

- Reduce Phase
  - Fetch, Buffer, Merge, Store
  - MM: Combine

cmp(key k1, key k2) use default

sort on join key only

join T.a=S.b +: concatenate schemas

Figure shows example with 4 mappers and 2 reducers.
Trojan Join Co-Partitioning Details

sh(key k, value v, int numPartitions)

use default

shuffle on join

join T.a=S.b

⊕: concatenate schemas

Phase

figure shows example with 4 mappers and 2 reducers
The Hadoop Plan consists of three user-defined parameters: M, R, and P. These parameters are used to determine the number of mappers, reducers, and data storage. We achieve this by providing appropriate settings.

The Hadoop Plan is shaped by three user-defined parameters. M, R, and P. These parameters are used to determine the number of mappers, reducers, and data storage. We achieve this by providing appropriate settings.

Index enriches logical input splits by bulkloaded read optimized. We achieve this by providing appropriate settings. Section 1r

Contribution.

The Hadoop Plan is shown in Figure zw. We observe that the Hadoop Plan consists of three main stages: load, map, and reduce. Each stage is performed by a set of tasks that are executed in parallel.

The Hadoop Plan is shown in Figure zw. We observe that the Hadoop Plan consists of three main stages: load, map, and reduce. Each stage is performed by a set of tasks that are executed in parallel.

The Hadoop Plan is shown in Figure zw. We observe that the Hadoop Plan consists of three main stages: load, map, and reduce. Each stage is performed by a set of tasks that are executed in parallel.

The Hadoop Plan is shown in Figure zw. We observe that the Hadoop Plan consists of three main stages: load, map, and reduce. Each stage is performed by a set of tasks that are executed in parallel.

The Hadoop Plan is shown in Figure zw. We observe that the Hadoop Plan consists of three main stages: load, map, and reduce. Each stage is performed by a set of tasks that are executed in parallel.

The Hadoop Plan is shown in Figure zw. We observe that the Hadoop Plan consists of three main stages: load, map, and reduce. Each stage is performed by a set of tasks that are executed in parallel.
reduce(key $ik$, vset $ivs$) $\mapsto [{\{ik\} \times ivs}]$ 

```plaintext
join T.a=S.b  ⊕: concatenate schemas
```
The Hadoop Plan is shaped by three user-defined parameters. In this section we examine how Hadoop computes a MapReduce execution plan. Let's analyze the Hadoop Plan in more detail.

1. **Data Load Phase**
   - Fetch L
   - Store L
   - Scan L
   - PPart L
   - Split L

2. **Map Phase**
   - Fetch T
   - Store T
   - Scan T
   - MPart T
   - Split T

3. **Shuffle Phase**
   - Fetch T
   - Store T
   - Merge T
   - Sort Grp T
   - Split T

4. **Reduce Phase**
   - Fetch T
   - Store T
   - Merge T
   - Sort Grp T
   - Split T

Notice. Write-up of these UDFs in the CR has a small bug. See note on our website:
http://infosys.cs.uni-saarland.de/publications/DQJ+10CRv1correction.pdf
Trojan Join Query Processing

- **Query Algorithm:**
  - read footer of each input split to determine split size
  - read records from each co-group in ascending order
  - build cross product for each co-group

- **Implementation:**
  - a MapReduce program
  - provide split UDF

- **Option 1: map-side join**
  - trick: map function keeps some state
  - perform local join in map()
  - advantage: no Reduce Phase (see paper)
  - drawback: need to keep some state in map() for sort-based grouping
Option 2: state-less map-side join

- Algorithm:
  - change `itemize` to return `[joinkey, entire co-group]`
  - then `map` is being called with the data belonging to an entire co-group
  - inside `map`: break co-group into tuples and compute cross product

- Advantages:
  - no Reduce Phase as in Option 1
  - but also: no need to keep state in `map`

- In fact: we exploited an interesting order plus `itemize` to semantically reduce data in `map`!

![Diagram of Hadoop++](image)
Trojan Index plus Trojan Join

- may combine both techniques
- may use index on join key
- may use index on different key
- may create multiple indexes inside the split
- in any case:
  - both scan access and index access paths possible
Experiments

- used benchmark as proposed in [Pavlo et al, SIGMOD 2009]
- benchmark defines several tasks
- two of them related to indexing and join processing
  - Selection Task
  - Join Task
- used up to 100 EC2 nodes as in HadoopDB-paper [Abouzeid et al, VLDB 2009]
- report average of three executions
- Some twist, see our paper:
  *Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance*
  Jörg Schad, Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz
  *VLDB 2010*
  Research Session-14: Experimental Analysis and Performance (i.e., yesterday)

*therefore:* also executed scaled-down experiments on small local cluster to verify
Selection Task

![Bar chart showing runtime of different systems for 10, 50, and 100 nodes. The systems compared are Hadoop, HadoopDB, HadoopDB Chunks, Hadoop++ (256MB), and Hadoop++ (1GB).]
Join Task

The chart compares the runtime (in seconds) of different systems and node counts.

- **Hadoop**
- **HadoopDB**
- **Hadoop++ (256MB)**
- **Hadoop++ (1GB)**

The x-axis represents the number of nodes (10, 50, 100) and the y-axis represents runtime in seconds.

- At 10 nodes:
  - Hadoop: ~1500 seconds
  - HadoopDB: ~1500 seconds
  - Hadoop++ (256MB): ~200 seconds
  - Hadoop++ (1GB): ~250 seconds

- At 50 nodes:
  - Hadoop: ~2000 seconds
  - HadoopDB: ~2000 seconds
  - Hadoop++ (256MB): ~450 seconds
  - Hadoop++ (1GB): ~500 seconds

- At 100 nodes:
  - Hadoop: ~2500 seconds
  - HadoopDB: ~2500 seconds
  - Hadoop++ (256MB): ~600 seconds
  - Hadoop++ (1GB): ~650 seconds
Failover

- we inherit fault tolerance from Hadoop!
- the Trojan effect!
Lessons Learned for our Community

- indexing, co-partitioning, preprocessing, etc....
- ...are **not exclusive** to database management systems
- all these techniques may be successfully used in **any** data processing system, not only DBMS
- just one thing matters:
- **“Do we know anything about the schema and the anticipated workload in advance?“**
- if **yes**, we may:
  - create appropriate indexes
  - create co-partitions
  - etc.
- this holds for **both**
  - DBMS
  - and MapReduce/Hadoop
Conclusions

- we proposed Hadoop++
- a new approach to large scale data analysis
- keep the MapReduce interface and the MapReduce execution engine
- still: rewrite incoming MapReduce programs to more efficient ones
- inject code through Trojan techniques
- execute plans using existing MapReduce pipeline unchanged
- experiments with SIGMOD 2009 benchmark
- strong improvements in selection and join tasks
- up to a factor of 18 better than Hadoop
Future Work

- other Trojan techniques  
  ongoing

- research challenges when executing MapReduce on the Cloud

  **Flying Yellow Elephant: Predictable and Efficient MapReduce in the Cloud**
  Jörg Schad
  VLDB PhD Workshop 2010 (see VLDB USB stick or online)

- marry Hadoop++ with OctopusDB* one-size-fits-all DBMS

  **The Mimicking Octopus: Towards a one-size-fits-all Database Architecture**
  Alekh Jindal
  VLDB PhD Workshop 2010 (see VLDB USB stick or online)