

# Efficiently Indexing AND Querying Big Data in Hadoop MapReduce

Jens Dittrich



This talk consists of three parts.

MapReduce  
Intro

Hadoop++

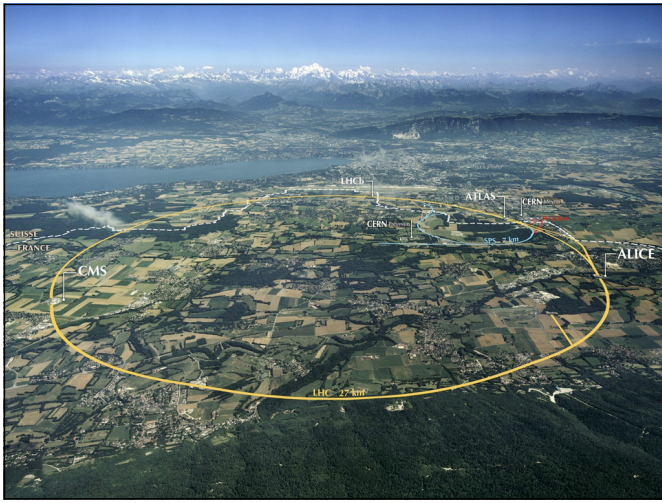
HAIL

First Part

MapReduce  
Intro

Big data is the new “very large”.

**Big** Data



Big data is everywhere: CERN...

<http://cdsweb.cern.ch/record/1295244>



...moving objects indexing...

<http://www.istockphoto.com/stock-video-4518244-la-traffic-a-time-lapse.php>



...astronomy...

<http://www.flickr.com/photos/14924974@N02/2992963984/>



basically whenever you point a satellite dish up in the air, you collect tons of data

but also in...

[http://it.wikipedia.org/wiki/File:KSC\\_radio\\_telescope.jpg](http://it.wikipedia.org/wiki/File:KSC_radio_telescope.jpg)

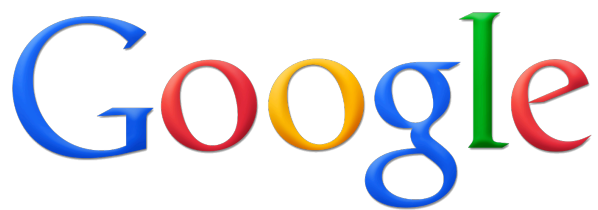


...genomics...

<http://www.istockphoto.com/stock-illustration-16136234-dna-strands.php>

The Facebook logo, consisting of the word "facebook" in white lowercase letters on a dark blue rectangular background.

...social networks...

The Google logo, featuring the word "Google" in its characteristic multi-colored font (blue, red, yellow, blue, green, red) with a 3D effect.

...and search engines.

[Dean et al, OSDI'04]

They proposed a system to effectively analyze big data.

# MapReduce

That system was coined “MapReduce“. The system is Google-proprietary.



Hadoop is the open source variant. It has a large community of developers and start-up companies.

## Big Data Tutorial [VLDB 2012b]

We presented a tutorial on Big Data Processing in Hadoop MapReduce at VLDB 2012.

It contains many details on data layouts, indexing, query processing and so forth.

The tutorial slides are available online:

<http://infosys.uni-saarland.de/publications/BigDataTutorialSlides.pdf>

## Semantics:

Let's briefly revisit the MapReduce interface:



just two functions: map() and reduce()

`map(key, value) -> set of (ikey, ivalue)`

`reduce(ikey, set of ivalue) -> (fkey, fvalue)`

Let's look at a concrete use-case:

## Google-Use Case:

This is vital for Google's search service you use everyday.

## Web-Index Creation

In this use-case the map function...

`map(key, value)`  
->  
set of (ikey, ivalue)

```
map(docID, document)
->
set of (term, docID)
```

...takes a docID and a document (the contents of the document)  
and returns a set of (term,docID)-pairs.

For instance...

```
map(44,
  "This is text on a website!"
)
->
{
  ("This", 44),
  ("is", 44),
  ("text", 44),
  ("on", 44),
  ("a", 44),
  ("website", 44)
}
```

...map() will be called for document 44 with its contents  
"This is text on a website!".

The map()-function breaks this into pairs, one pair for  
each term occurring on website 44.

```
map(42,
  "This is just another website!"
)
->
{
  ("This", 42),
  ("is", 42),
  ("just", 42),
  ("another", 42),
  ("website", 42)
}
```

the same happens for document 42

```
map(43,
  "One more boring website!"
)
->
{
  ("One", 43),
  ("more", 43),
  ("boring", 43),
  ("website", 43)
}
```

and so forth

What about reduce()?

```
reduce(ikey, set of ivalue)
->
(fkey, fvalue)
```

```
reduce(term, set of docID)
->
(term, (posting list of docID, count))
```

```
reduce("This",
      {42,
       43}
)
->
("This", ([42, 43], 2))
```

```
reduce("is",
      {42,
       43}
)
->
("is", ([42, 43], 2))
```

For Web-index creation reduce() receives a term and the set of docIDs containing that term. reduce() then returns a pair of the input term and an ordered posting list of docIDs plus a count, i.e. the number of web pages having that term.

Note: there are many variants how to do web-indexing with MapReduce. The actual semantics used by Google may differ; the core idea however is the same.

For instance: documents 42 and 43 contain "This". reduce() simply returns an ordered posting list plus the count.

Documents 42 and 43 contain "is". reduce() simply returns an ordered posting list plus count for this as well.

```
reduce(`boring`,  
      {43}  
)  
->  
(`boring`, ([43], 1))  
  
etc.
```

and so forth

## Other Applications:

### Search

```
rec.a==42,  
rec.contains(`bla`),  
rec.contains(0011001)
```

### Machine Learning

k-means,  
mahout library

### Web-Analysis

sum of all accesses to page  
Y from user X

etc.

Many things can be mapped to the map()/reduce()-interface, but not all.

Think about twice before blindly using MapReduce. It is useful for many things, but not all.

Many important extensions have been done in the past to support more application classes, i.e. iterative problems.

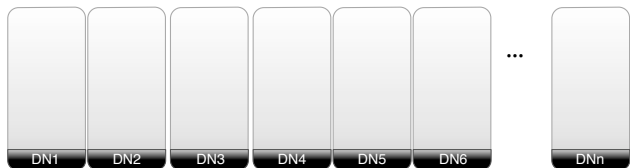
map() and reduce() with

# Big Data ?

Bob

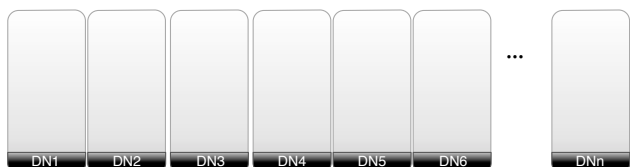
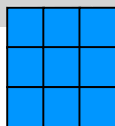


HDFS



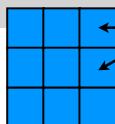
HDFS

horizontal partitions

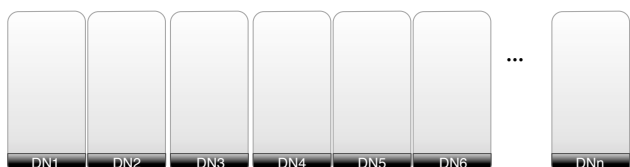


HDFS

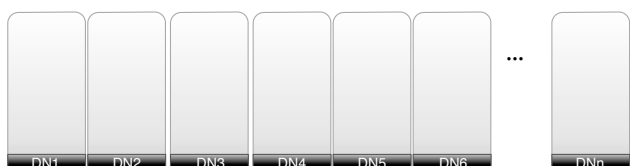
horizontal partitions



HDFS blocks  
64MB (default)



HDFS



Let's assume a user Bob who wants to analyze a large file.

Notice that this is a simplified explanation.

For details on how Hadoop works see our paper: Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing), VLDB 2010 <http://infosys.cs.uni-saarland.de/publications/DQJ+10CRv2.pdf>

[http://www.istockphoto.com/file/\\_closeup.php?id=591134](http://www.istockphoto.com/file/_closeup.php?id=591134)

Bob first needs to upload his file to Hadoop's Distributed File System (HDFS). HDFS partitions his data into large horizontal partitions.

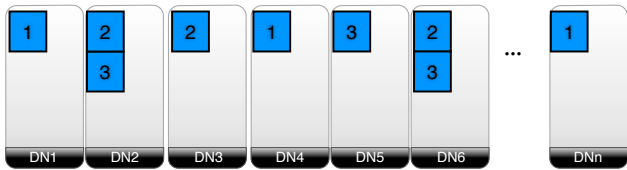
Those horizontal partitions are termed HDFS blocks. They are relatively large: at least 64MB up to 1GB. Do not confuse these large HDFS blocks with the typically small database pages (which are only a few KB in size).

Each HDFS block receives a unique ID.



HDFS

4	5	6
7	8	9

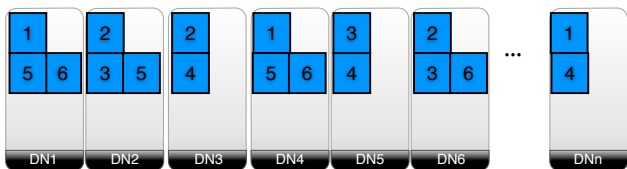


The HDFS blocks get distributed and replicated over the cluster. Each HDFS block gets replicated to at least three different data nodes (DN1, ...DNn in this example).



HDFS

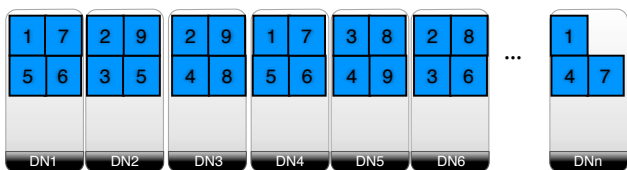
7	8	9
---	---	---



HDFS does this for every HDFS block of the input file.



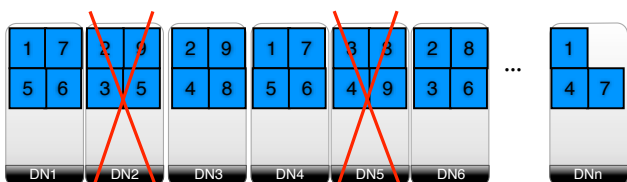
HDFS



Eventually all HDFS blocks have been sent to the datanodes.

## Failover

HDFS



We gain nice failover properties: even if two datanodes go offline, we still have one copy of the block.

Assume that we want to retrieve HDFS block 3. We lost the copies on DN2 and DN5. However, we can still retrieve a copy of block 3 from DN6.

Notice that once datanodes go offline HDFS (should) copy blocks to other nodes to get back to having three copies of each block again.

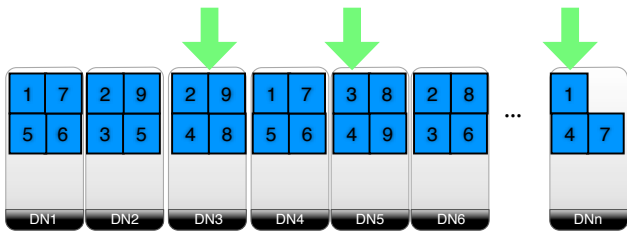


## Load Balancing



I would like to have block 4!

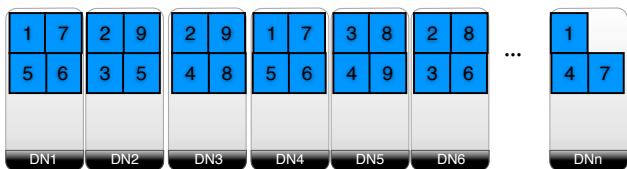
HDFS



Another advantage of having three copies for each block is load balancing. Whenever a user or an application asks for a particular block, we have three options for retrieving that block. The decision which datanode to use may be made based on network locality, network congestion, and the current load of the datanodes.



HDFS



So now we have our data stored in HDFS.

What about MapReduce?

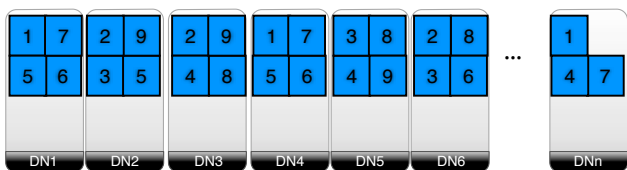
And by “MapReduce” I mean “Hadoop MapReduce” in the following.



map(docID, document) -> set of (term, docID)

MapReduce

HDFS



MapReduce is another software layer on top of HDFS. MapReduce consists of three phases.

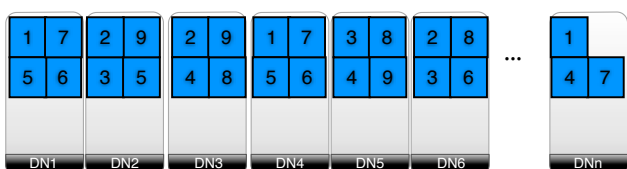
## Map Phase



MapReduce

map(docID, document) -> set of (term, docID)

HDFS



In the first phase (the Map Phase) only the map-function is considered.

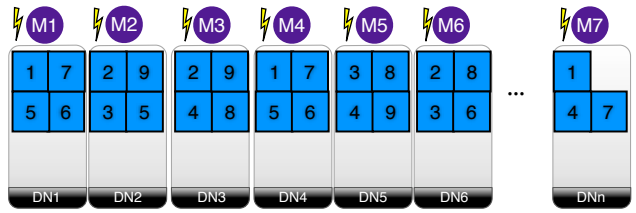
## Map Phase



MapReduce

map(docID, document) -> set of (term, docID)

HDFS



MapReduce assigns a thread (AKA Mapper) at every datanode having data to be processed for this job.

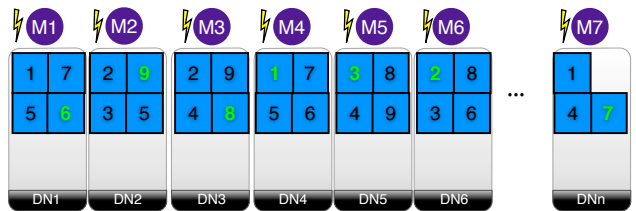
## Map Phase



MapReduce

map(docID, document) -> set of (term, docID)

HDFS



Each mapper reads one of the HDFS blocks...

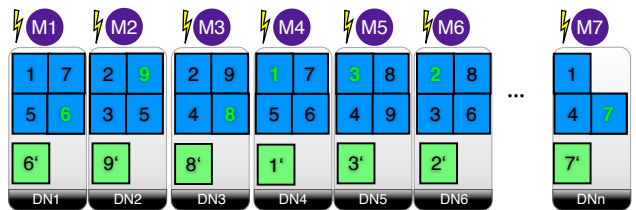
## Map Phase



MapReduce

map(docID, document) -> set of (term, docID)

HDFS



..and breaks that HDFS blocks into records. This can be customized with the RecordReader. For each record map() is called. The output to that file (also called intermediate results) is collected on the local disks of the datanodes. For instance, for block 6 the output is collected in file 6'.

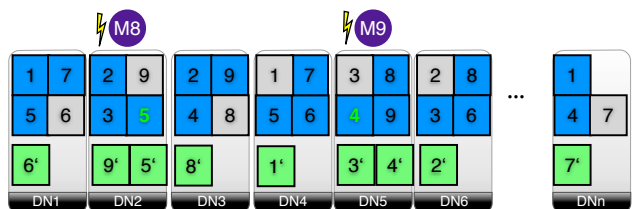
## Map Phase



MapReduce

map(docID, document) -> set of (term, docID)

HDFS

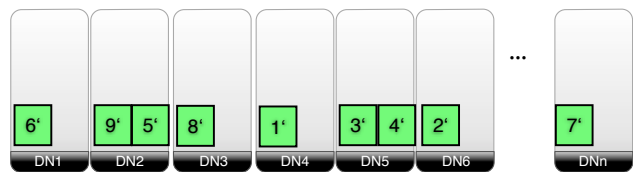


This is done for every block of the input file. Obviously we do not have to do this with every copy of an HDFS block. Processing one copy is enough. With this the Map Phase is finished.

## Shuffle Phase

MapReduce group by term

HDFS

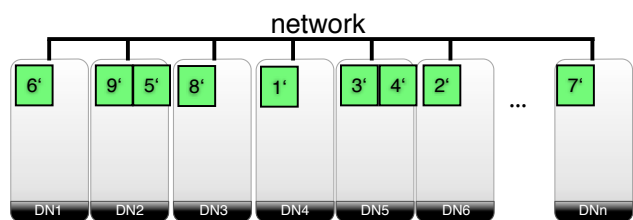


Now, the Shuffle Phase starts.

## Shuffle Phase

MapReduce group by term

HDFS

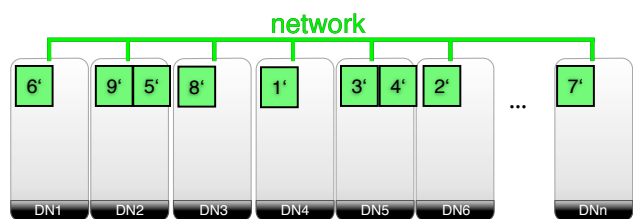


In the Shuffle Phase all intermediate results are redistributed over the different datanodes. In this example we want to redistribute the intermediate results based on the term, i.e. we want to group all intermediate results by term.

## Shuffle Phase

MapReduce group by term

HDFS

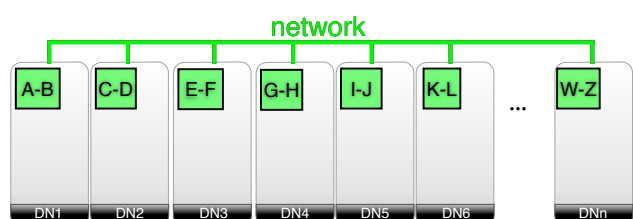


This means, after shuffling, we obtain a range partitioning on terms....

## Shuffle Phase

MapReduce group by term

HDFS



For instance, DN1 contains all intermediate results having terms starting with A or B. In turn, DN2 has only terms starting with C or D and so forth. Once all data has been redistributed, the Shuffle Phase is finished.

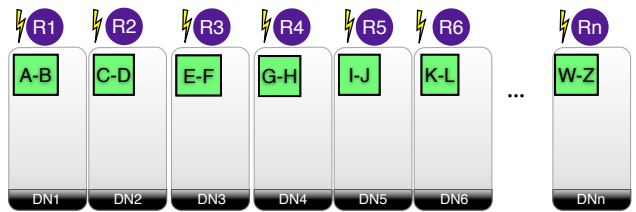
## Reduce Phase



reduce(term, set of docID) -> set of  
(term, (posting list of docID, count))

MapReduce

HDFS



In the final Reduce Phase, MapReduce assigns threads to the different datanodes having intermediate results. These threads are termed reducers. The reducers read the intermediate results and for each distinct key they call `reduce()`. Notice that `reduce()` may only be called once for each distinct key on the entire cluster. Otherwise the semantics of the `map()/reduce()`-paradigm would be broken.

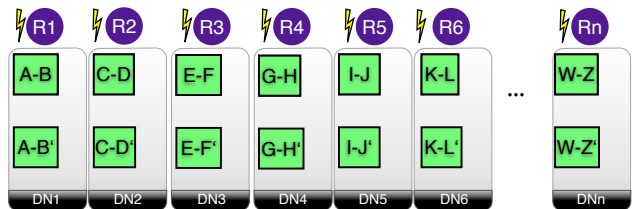
## Reduce Phase



MapReduce

reduce(term, set of docID) -> set of  
(term, (posting list of docID, count))

HDFS



The output of the `reduce()`-calls is stored on disk again. In this example this is visualized to store the output on local disk. However, Hadoop stores the output on HDFS by default, i.e. the output of the Reduce Phase gets replicated by HDFS again.

When to replicate which data for fault tolerance in MapReduce is an interesting discussion. See our paper RAFT for more details:  
<http://infosys.uni-saarland.de/publications/QPSD11.pdf>

## Hadoop MapReduce Advantages

failover

scalability

schema-later

ease of use

# Hadoop MapReduce Disadvantages

And this and how to fix it is what the following material is about.

## Performance

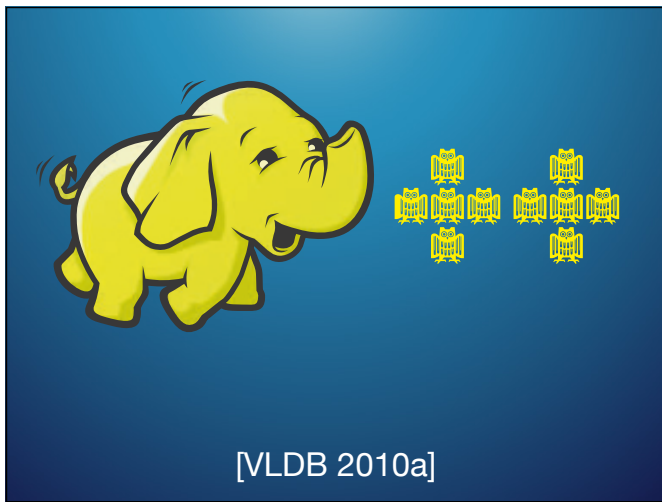
MapReduce  
Intro

Hadoop++

HAIL

Second Part

Hadoop++



Jens Dittrich, Jorge-Arnulfo Quiane-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, Jörg Schäd  
Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing)  
VLDB 2010/PVLDB, Singapore  
<http://infosys.cs.uni-saarland.de/publications/DQJ+10CRv2.pdf>  
slides:  
<http://infosys.cs.uni-saarland.de/publications/DQJ+10talk.pdf>

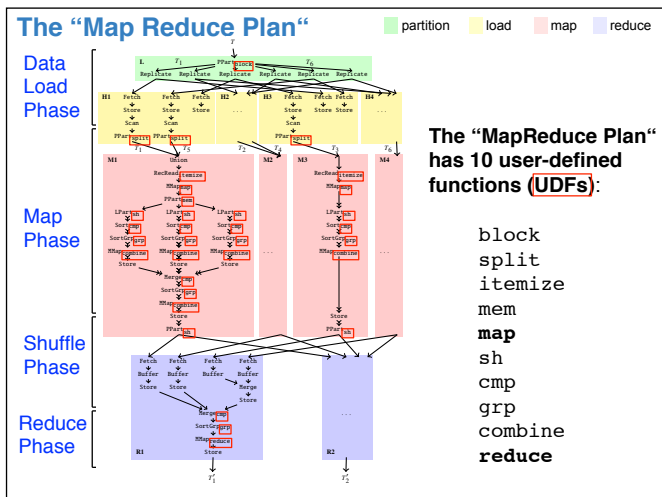
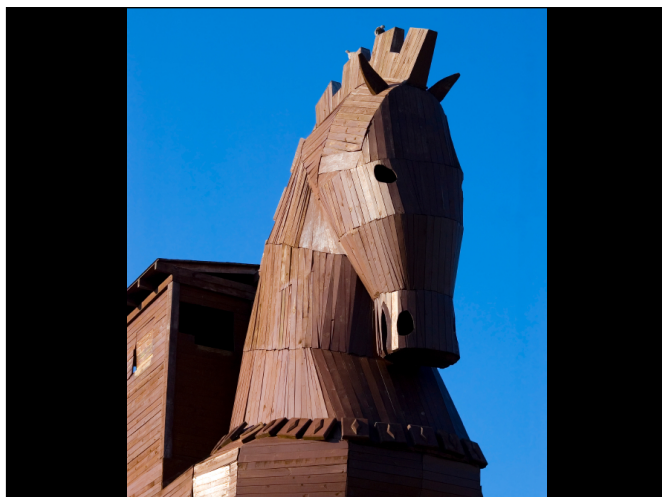


figure shows example with 4 mappers and 2 reducers  
Hadoop MapReduce uses a hard-coded pipeline. This pipeline cannot be changed. This is in sharp contrast to database systems which may use different pipelines for different queries.

However, Hadoop Map Reduce uses 10 user-defined functions (UDFs).  
Theses UDFs can be used to inject arbitrary code into Hadoop...



...including code that was not intended to be injected into Hadoop.

Our idea is somewhat similar to a trojan horse or a trojan, i.e. a virus that is injected into a computer system to harm or destory the system. However, we inject trojans to improve or heal the system.

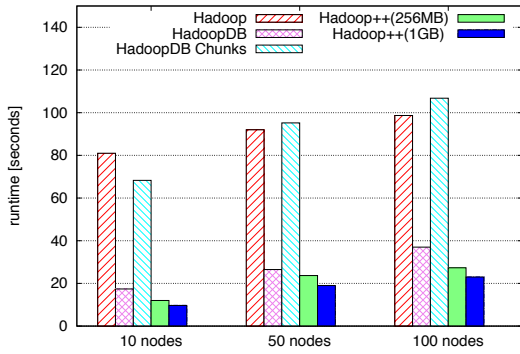
Therefore we are calling them...

<http://www.istockphoto.com/stock-photo-1824642-trojan-horse.php?st=0bab152>

Good Trojans!



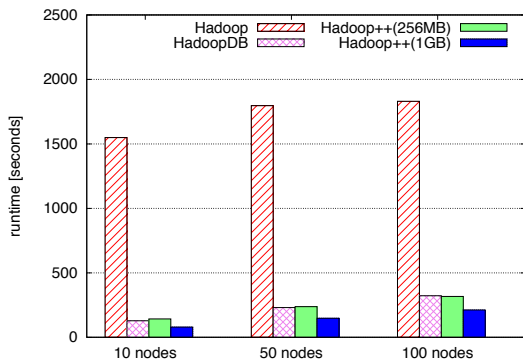
## Selection Task



results taken from our VLDB 2010-paper

Hadoop++ is in the same ballpark or even faster than HadoopDB (now spun-off as Hadapt)

## Join Task



Hadoop++ is up to a factor 18 faster than Hadoop

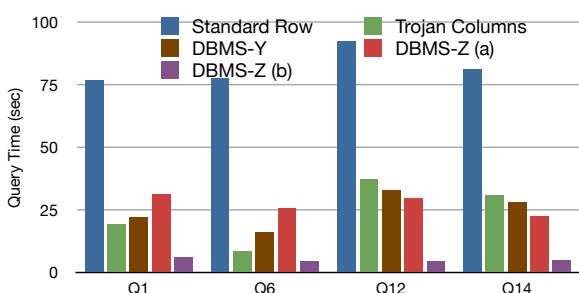
...even though we do not modify the underlying HDFS and Hadoop MapReduce source code at all!  
Our improvements are all done through UDFs only.

But wait: UDFs are also available in traditional database systems. What happens if we exploit those UDFs to inject “better technology” into an existing database system?

Say we inject column store technology into a commercial, closed-source row store. How would that look like?

<sup>1</sup>Good Trojans in a DBMS?...

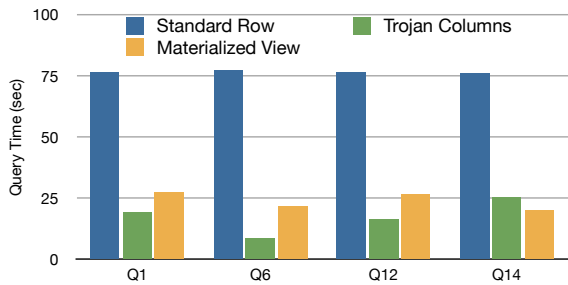
## Good Trojans Versus Closed Source Column Store DBMS



[CIDR 2013a]

It looks like this. For details see our paper: Alekh Jindal, Felix Martin Schuhknecht, Jens Dittrich, Karen Khachatryan, Alexander Bunte. How Achaeans Would Construct Columns in Troy. CIDR 2013, Asilomar, USA. <http://infosys.uni-saarland.de/publications/How%20Achaeans%20Would%20Construct%20Columns%20in%20Troy.pdf> You can get much faster than a row store. We are not as fast as a from scratch implementation of a column store. This has to do with the different QP technology used. However for some customers the performance of a native column store might not even be required - especially for medium-sized datasets. ...

## Good Trojans in a Closed Source Row Store DBMS



[CIDR 2013a]

... Why buy a car with 1000hp if 200hp are just enough?

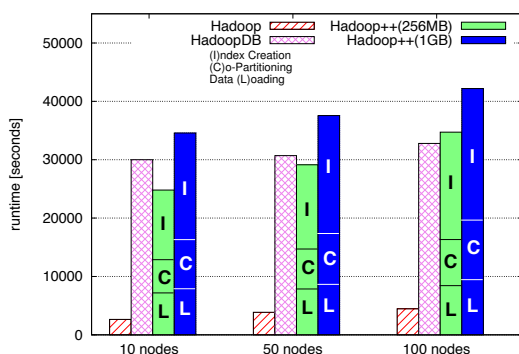
An interesting result from our work is also that we can beat materialized views for some queries.

With this let's close this footnote and go back to Hadoop MapReduce performance. UDFs in Hadoop allow us to boost query performance without changing the underlying system. However, what are the...

<sup>1</sup>...Good Trojans in a DBMS!

## Problems:

### Upload-Times



from Hadoop++ paper: The problem is that in order to have fast queries we first have to “massage” the data before, i.e. create indexes, co-partition and so forth. This takes time. Hadoop does not have to spend this time and therefore uploading the data to HDFS is fast. In contrast, for Hadoop++ (but also HadoopDB) we also have to do a lot of extra work. This extra work is very costly. So costly that only after running many queries these investments are amortized. In other words, if we only want to run a few queries exploiting our indexes and co-partitioning, we shouldn't use Hadoop++ in the first place but rather run the queries directly on Hadoop! How could we fix this problem?

=> ~~back to scanning?~~

=> ~~index selection  
algorithms?~~

=> ~~coarse-granular  
indexes~~

We could drop the idea of using indexes: just scan everything.

Well we are not gonna follow this approach.

We could invest into better index selection algorithms. If we pick the wrong index, index creation is unlikely to be amortized. Therefore making the right choice is important. Therefore...

Well we are not gonna follow this approach.

Or: as it is expensive to create all these indexes, we better investigate coarse-granular indexes, i.e. indexes that are cheaper to construct and yet give some benefit at query time.

Well we are not gonna follow this approach.

We do something different. Which brings me to...

MapReduce  
Intro

Hadoop++

HAIL

... the third part of my talk.

The approach I would like to present is coined HAIL.

HAIL

HAIL means Hadoop Aggressive Indexing Library.

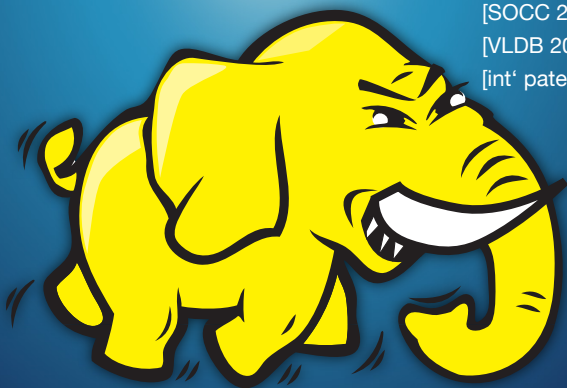
For details see our paper:

Jens Dittrich, Jorge-Arnulfo Quijane-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, Jörg Schadt. Only Aggressive Elephants are Fast Elephants. VLDB 2012/ PVLDB, Istanbul, Turkey. <http://infosys.uni-saarland.de/publications/HAIL.pdf>

A predecessor of this work focussing on data layouts in HDFS is our paper:

Alekh Jindal, Jorge-Arnulfo Quijane-Ruiz, Jens Dittrich. Trojan Data Layouts: Right Shoes for a Running Elephant. ACM SOCC 2011, Cascais, Portugal. <http://infosys.uni-saarland.de/publications/JQD11.pdf>

[SOCC 2011]  
[VLDB 2012a]  
[int' patent]

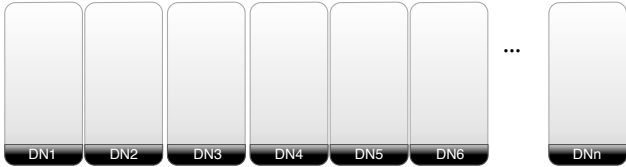


Hadoop Aggressive Indexing Library

Bob



HAIL

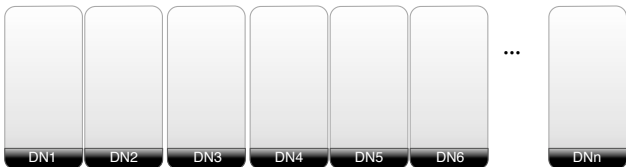


So back to Bob again. Recall that Bob wants to analyze a large file with Hadoop MapReduce. So he first has to upload his file to HDFS. In our approach we replace HDFS with HAIL. HAIL is an extension of HDFS.

As before Bob's file gets partitioned into HDFS blocks...

HAIL

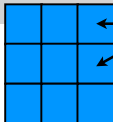
horizontal partitions



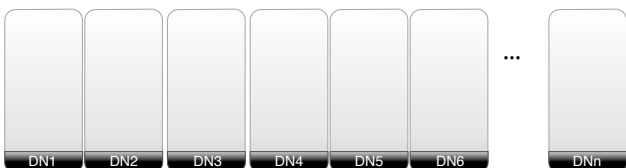
those blocks are relatively large, at least 64MB

HAIL

horizontal partitions

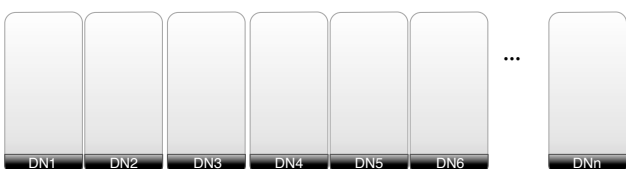


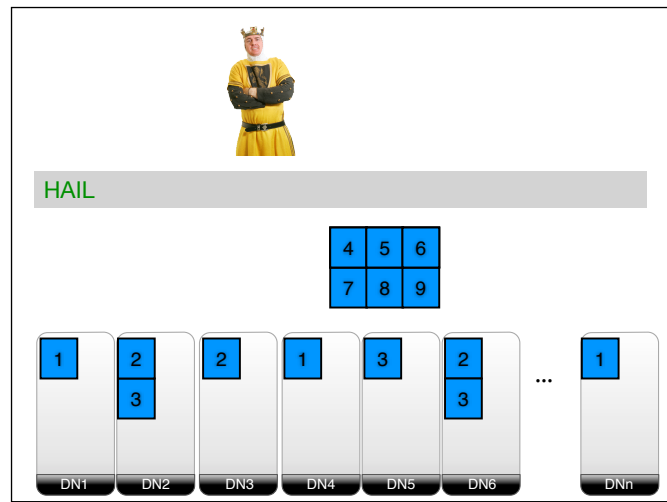
HDFS blocks  
64MB (default)



then those blocks get partitioned to the different datanodes (just as above)

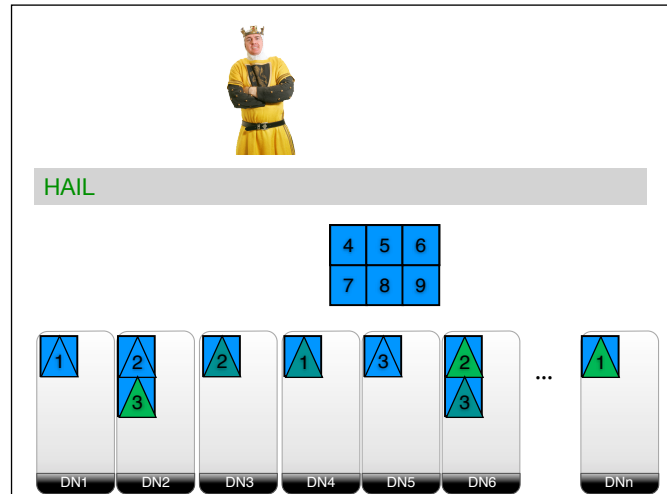
HAIL





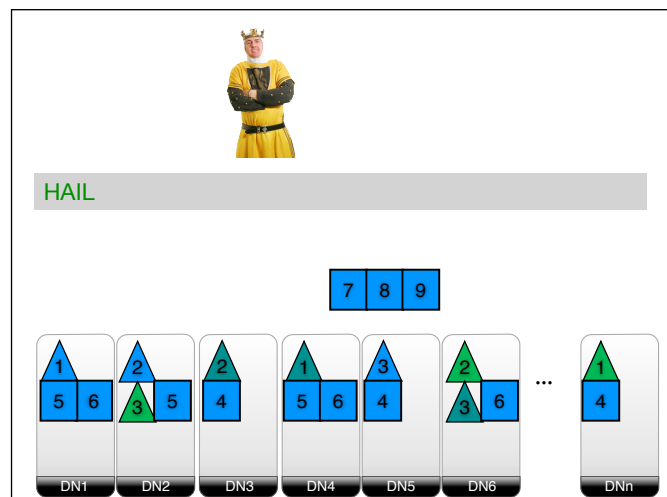
the HDFS blocks also get replicated (just as above)

but then, before writing the data to the local disks on the different datanodes, we do something in addition:

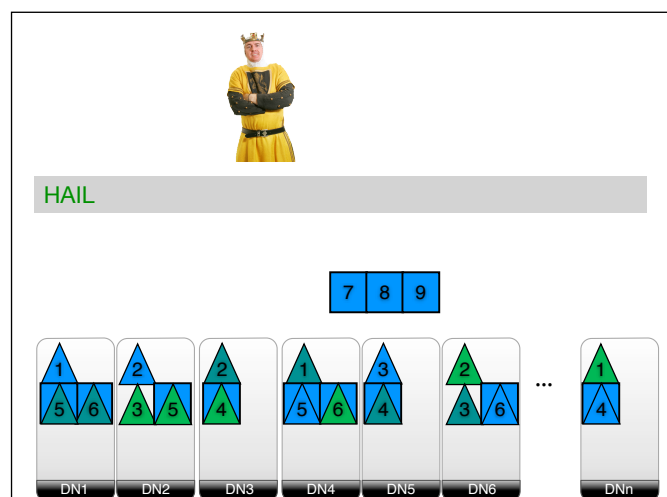


we sort the data on each HDFS block in main memory. Each replica is sorted using a different sort criteria. This means after sorting each HDFS block is available in three different sort orders - roughly corresponding to three different clustered indexes.

Notice that we do not redistribute data across HDFS blocks! Data that was on one particular block in standard HDFS will sit on the same HDFS block in HAIL. In other words: the different copies of the block contain the same data - yet in different sort orders.



Again, we do this for each and every copy of a block.

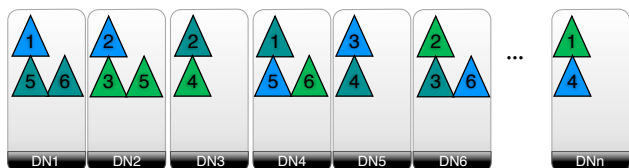


Notice that this is done without introducing additional I/O. We fully piggy-back on the existing HDFS upload pipeline.

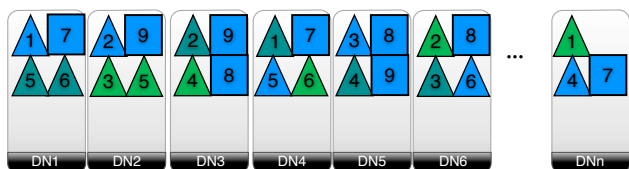


HAIL

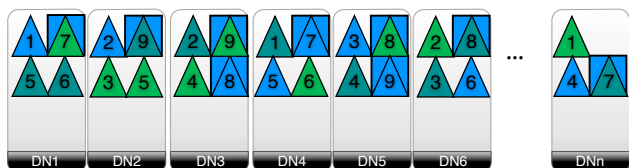
7 8 9



HAIL



HAIL

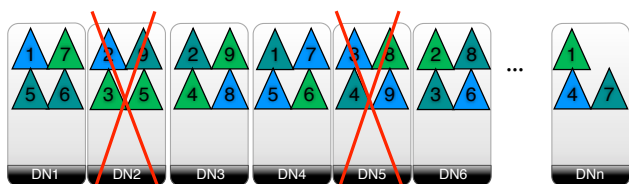


Eventually uploading (and indexing) is finished.

What does this mean for HDFS failover?

## Failover

HAIL



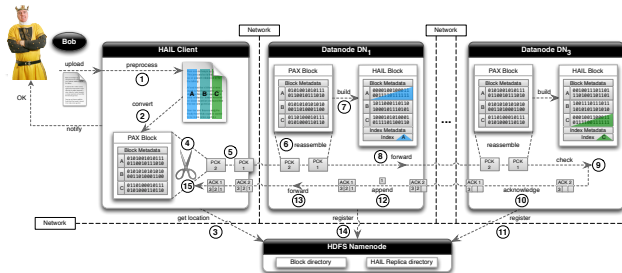
Well actually, nothing changes. All data sits on the same HDFS blocks as before. For instance, if we lose DN2 and DN6, we can still retrieve block 3 from DN6. That block might not be sorted along the desired sort criteria, but it contains all the data. And we can use the remaining block to recreate additional copies in other sort orders.



## little Details

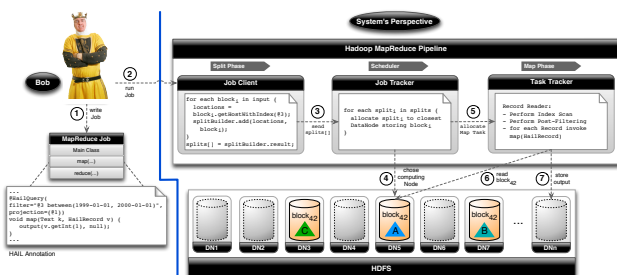
Well it is all somewhat more complex than explained in the previous example.

### HAIL Upload Pipeline



We play some other tricks. For instance, while reading the input file, we immediately parse the data into a binary PAX (column like) layout. The PAX data is then sent to the different datanodes. We also had to make sure to not break the involved data consistency-checks used by HDFS (packet acknowledge). In addition, we extended the namenode to record additional information on the sort orders and layouts used for the different copies of an HDFS block. The latter is needed at query time. None of these changes affects principle properties of HDFS. We just extend and piggyback.

### HAIL Query Pipeline



At query time HAIL needs to know how to filter input records and which attributes to project to in intermediate result tuples. This can be solved in many ways. In our current implementation we allow users to annotate their map-function with the filter and projection conditions. However this could also be done fully transparently by using static code analysis as shown in: Michael J. Cafarella, Christopher Ré: Manimal: Relational Optimization for Data-Intensive Programs. WebDB 2010. That code analysis could be used directly with HAIL. Another option, if the map()/reduce()-functions are not produced by a user, is to adjust the application. ...

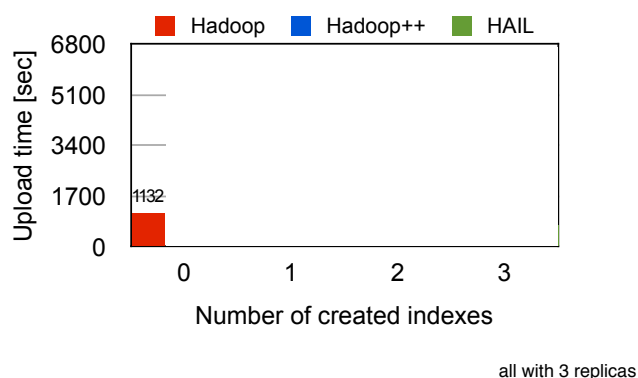
## Experiments

...Possible “applications” might be Pig, Hive, Impala or any other system producing map()/reduce() programs as its output. In addition, any other application not relying on MapReduce but just relying on HDFS might use our system, i.e. applications diurectly working with HDFS files.

# Upload Times

What happens if we upload a file to HDFS?

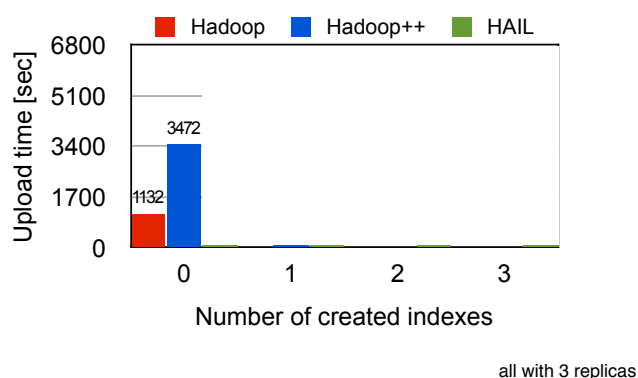
## Upload Time



Let's start with the case that no index is created by any of the systems.

Hadoop takes about 1132 seconds

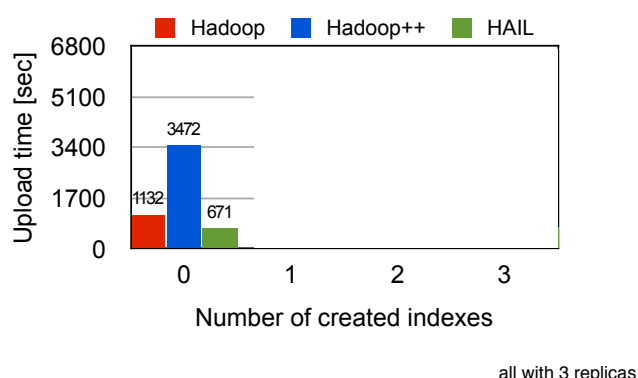
## Upload Time



Hadoop++ is considerably slower. Even though we switch off index creation here, Hadoop++ runs an extra job to convert the input data to binary. This takes a while.

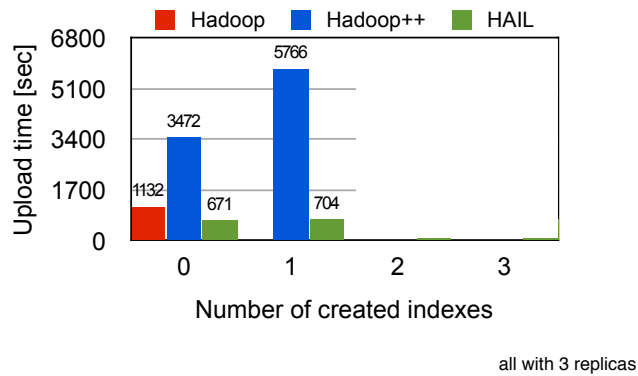
What about HAIL?

## Upload Time



HAIL is faster than Hadoop HDFS. How can this be? We are doing more work than Hadoop HDFS, right? For instance, we convert the input file to binary PAX during upload directly. This can only be slower than Hadoop HDFS but NOT faster. Well, when converting to binary layout it turns out that the binary representation of this dataset is smaller than the textual representation. Therefore we have to write less data and SAVE I/O. Therefore HAIL is faster for this dataset. This does not necessarily hold for all datasets. Notice that for this and the following experiments we are not using compression yet. We expect compression to be even more beneficial for HAIL.

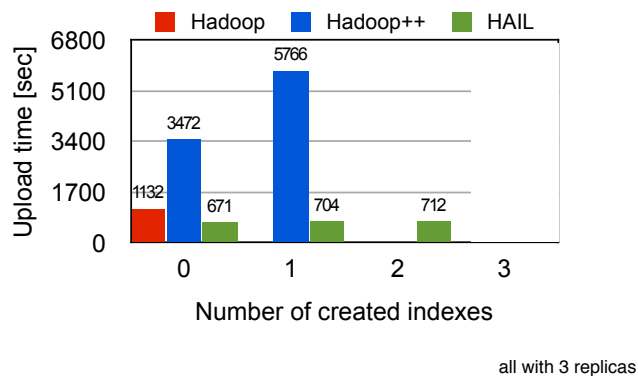
## Upload Time



So what happens if we start creating indexes? We should then feel the additional index creation effort in HAIL.

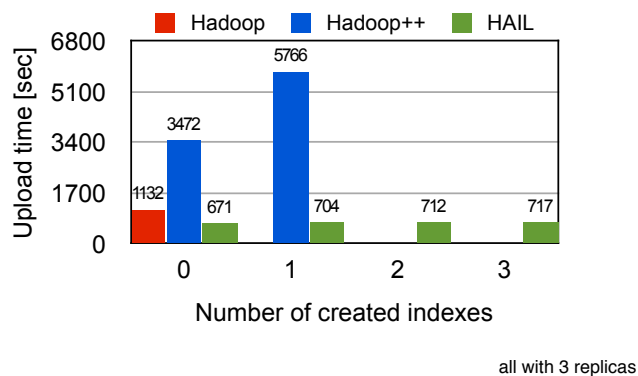
For Hadoop++ we observe long runtimes. For HAIL, in contrast to what we expected, we observe only a small increase in the upload time.

## Upload Time



The same observation holds when creating two clustered indexes with HAIL...

## Upload Time

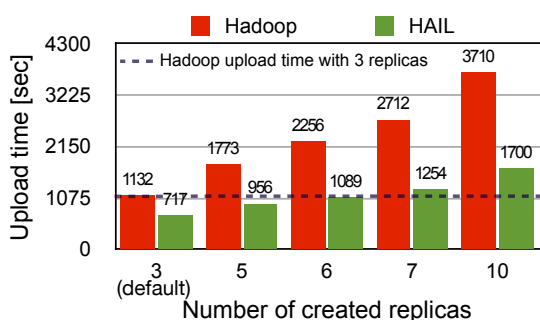


...or three.

This is because, standard file upload in HDFS is I/O-bound. The CPUs are mostly idle. HAIL simply exploits the unused CPU ticks that would be idling otherwise. Therefore the additional effort for indexing is hardly noticeable.

Disk space is cheap. For some situations It could be affordable to store more than three copies of an HDFS block. What would be the impact on the upload times? The next experiment shows the results...

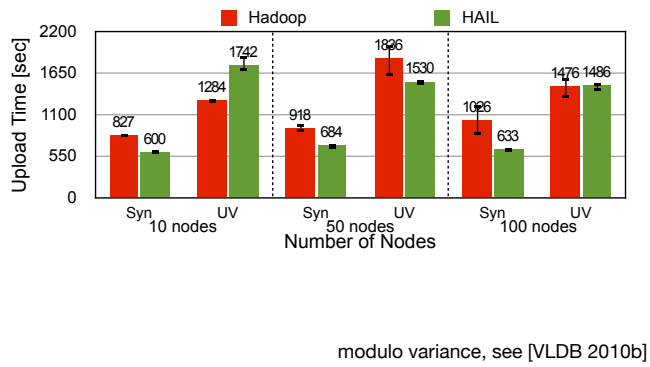
## Replica Scalability



Here we create up to 10 replicas - corresponding to 10 different clustered indexes.

We observe that in the same time HDFS uploads the data without creating any index, HAIL uploads the data, converts to binary PAX, and creates six different clustered indexes.

## Scale-Out



We also evaluated upload times on the cloud using EC2 nodes. Notice that experiments on the cloud are somewhat problematic due to the high runtime variance in those environments.

For details see our paper:

Jörg Schäd, Jens Dittrich, Jorge-Arnulfo Quijano-Ruiz  
Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance  
VLDB 2010/PVLDB, Singapore.

<http://infosys.cs.uni-saarland.de/publications/SDQ10.pdf>

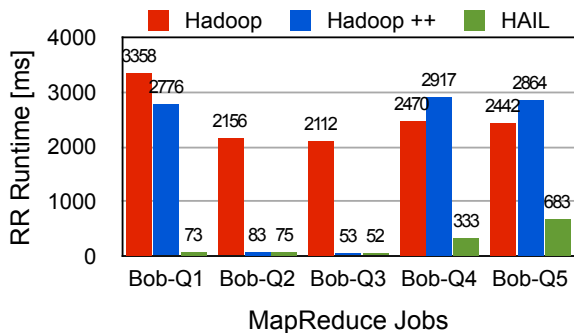
slides: <http://infosys.cs.uni-saarland.de/publications/SDQ10talk.pdf>

slides: <http://infosys.cs.uni-saarland.de/publications/SDQ10talk.pdf>

## Query Times

What about query times?

### Individual Jobs: Weblog, RecordReader



Here we display the RecordReader times. They correspond roughly to the data access time, i.e. the time for further processing, which is equal in all systems - is factored out.

We observe that HAIL improves query runtimes dramatically.

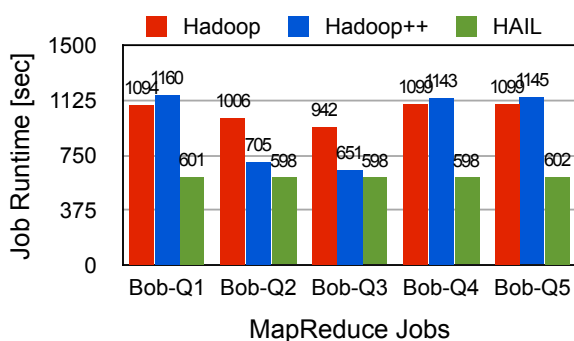
Hadoop resorts to full scan in all cases.

Hadoop++ can benefit from its index if the query happens to hit the right filter condition.

In contrast, HAIL supports many more filter conditions.

What does this mean for the overall job runtimes?

### Individual Jobs: Weblog, Job



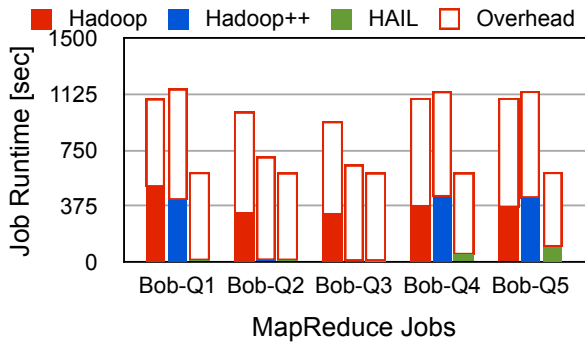
Well, those results are not so great.

The benefits of HAIL over the other approaches are marginal?

How come?

It has to do with...

## Scheduling Overhead



...the Hadoop scheduling overhead. Hadoop was designed having long running tasks in mind. Accessing indexes in HAIL, however, is in the order of milliseconds. These milliseconds of index access are overshadowed by scheduling latencies.

You can try this out with a simple experiment. Write a MapReduce job that does not read any input, does not do anything, and does not produce any output. This takes about 7 seconds - for doing nothing.

How could we fix this problem?

By...

...introducing "HAIL scheduling".

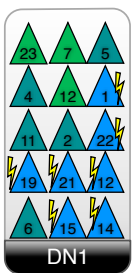
But let's first look back at standard Hadoop scheduling:

## HAIL Scheduling

### Hadoop Scheduling

MapReduce map(row) -> set of (ikey, value)

HAIL



sort order

⇒ 7 map tasks (aka waves)

⇒ 7 times scheduling overhead

In Hadoop, each HDFS block will be processed by a different map task. This leads to waves of map tasks each having a certain overhead.

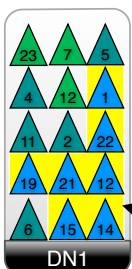
However the assignment of HDFS blocks to map tasks is not fixed.

Therefore, ...

### HAIL Scheduling

MapReduce map(row) -> set of (ikey, value)

HAIL



sort order

⇒ 1 map task (aka wave)

⇒ 1 times scheduling overhead

HAIL Split

...in HAIL Scheduling we assign all HDFS blocks that need to be processed on a datanode to a single map task. This is achieved by defining appropriate "splits". see our paper for details.

The overall effect is that we only have to pay the scheduling overhead once rather than 7 times (in this example).

Notice that in case of failover we can simply reschedule index access tasks - they are fast anyways.

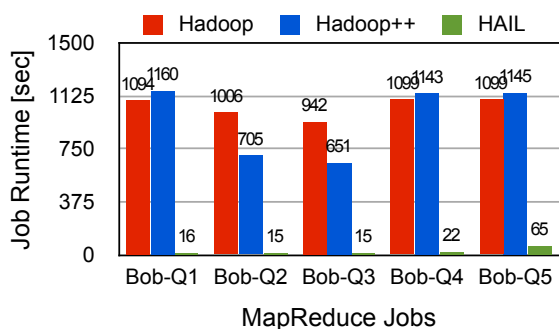
Additionally, we could combine this with the recovery techniques from RAFT, ICDE 2011.

# Query Times

with HAIL Scheduling

What are the end-to-end query runtimes with HAIL Scheduling?

## Individual Jobs: Weblog

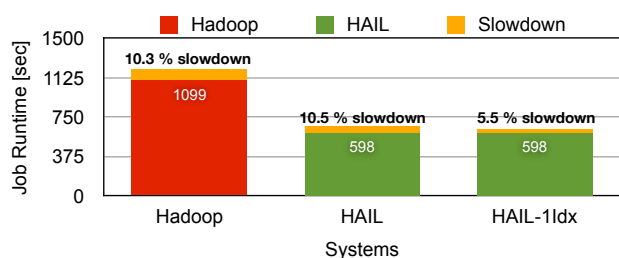


Now the good RecordReader times seen above translate to (very) good query times.

## Failover

What about failover?

## Failover



10 nodes, one killed

We use two configurations for HAIL. First, we configure HAIL to create indexes on three different attributes, one for each replica. Second, we use a variant of HAIL, coined HAIL-1Idx, where we create an index on the same attribute for all three replicas. We do so to measure the performance impact of HAIL falling back to full scan for some blocks after the node failure. This happens for any map task reading its input from the killed node. Notice that, in the case of HAIL-1Idx, all map tasks will still perform an index scan as all blocks have the same index. Overall result: HAIL inherits Hadoop MapReduce's failover properties.



Summary(Summary(...))

of this talk

MapReduce  
Intro

Hadoop++

HAIL

What I tried to explain to you in my talk is:

BigData  
=>



Hadoop++

HAIL

Hadoop MapReduce is THE engine for big data analytics.

BigData  
=>



HAIL

By using good trojans you can improve system performance dramatically AND after the fact --- even for closed-source systems.



HAIL allows you to have fast index creation AND fast query processing at the same time.

project page:  
<http://infosys.uni-saarland.de/hadoop++.php>

Copyright of alls Slides Jens Dittrich 2012



annotated slides are available on that page