Big data: Apache Hadoop ecosystem

Introduction to Hadoop framework tools

HDFS, HBASE, Spark, Impala, Hive and Pig

TE: 18/12/17

Layout

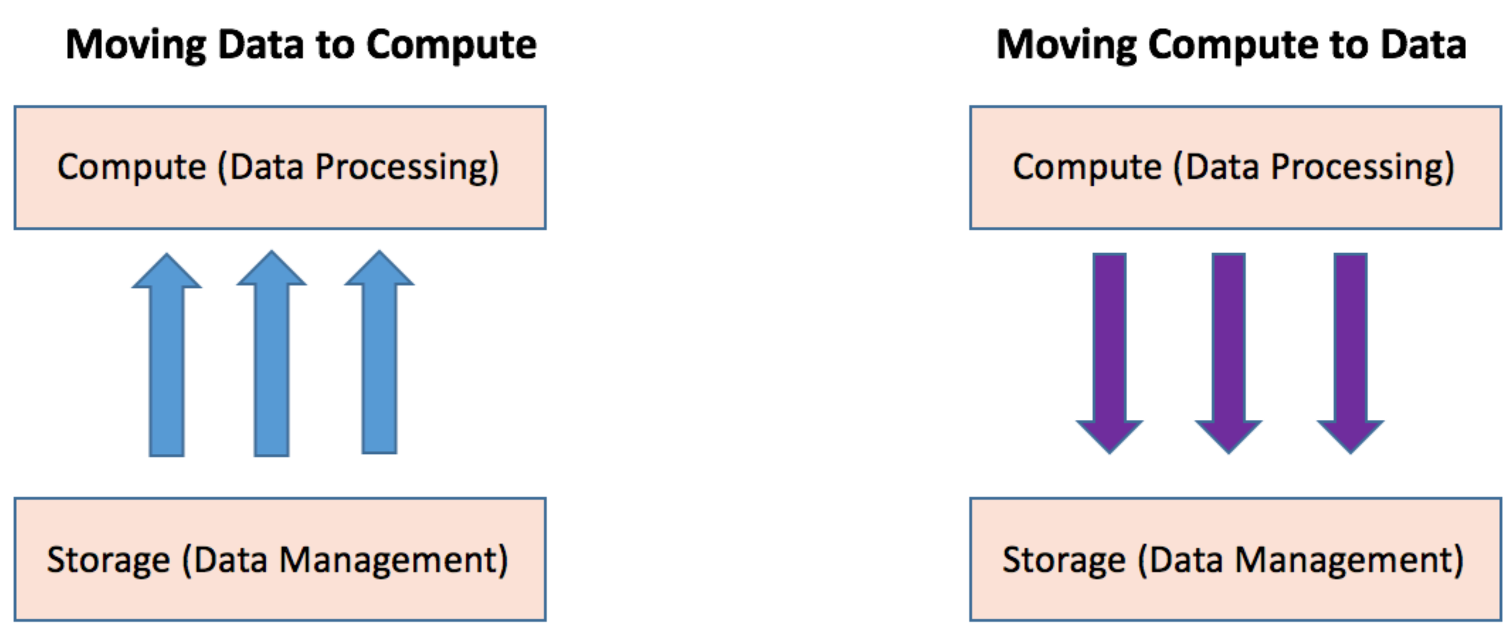
- HDFS: Hadoop distributed storage system
- HBASE: family oriented key-value store
- MapReduce processing alternatives
    - Spark, Impala, Pig, Hive
- Site Reliability Engineering. Big Data projects in Google datacenter context

HDFS

- Distributed file store for low cost hardware
- Tolerance for hardware failure: detection and automatic recovery of nodes
- Simple coherency: append content to end of files, updates are not supported
- Large data sets
- Streaming data access: focus on throughput, not latency

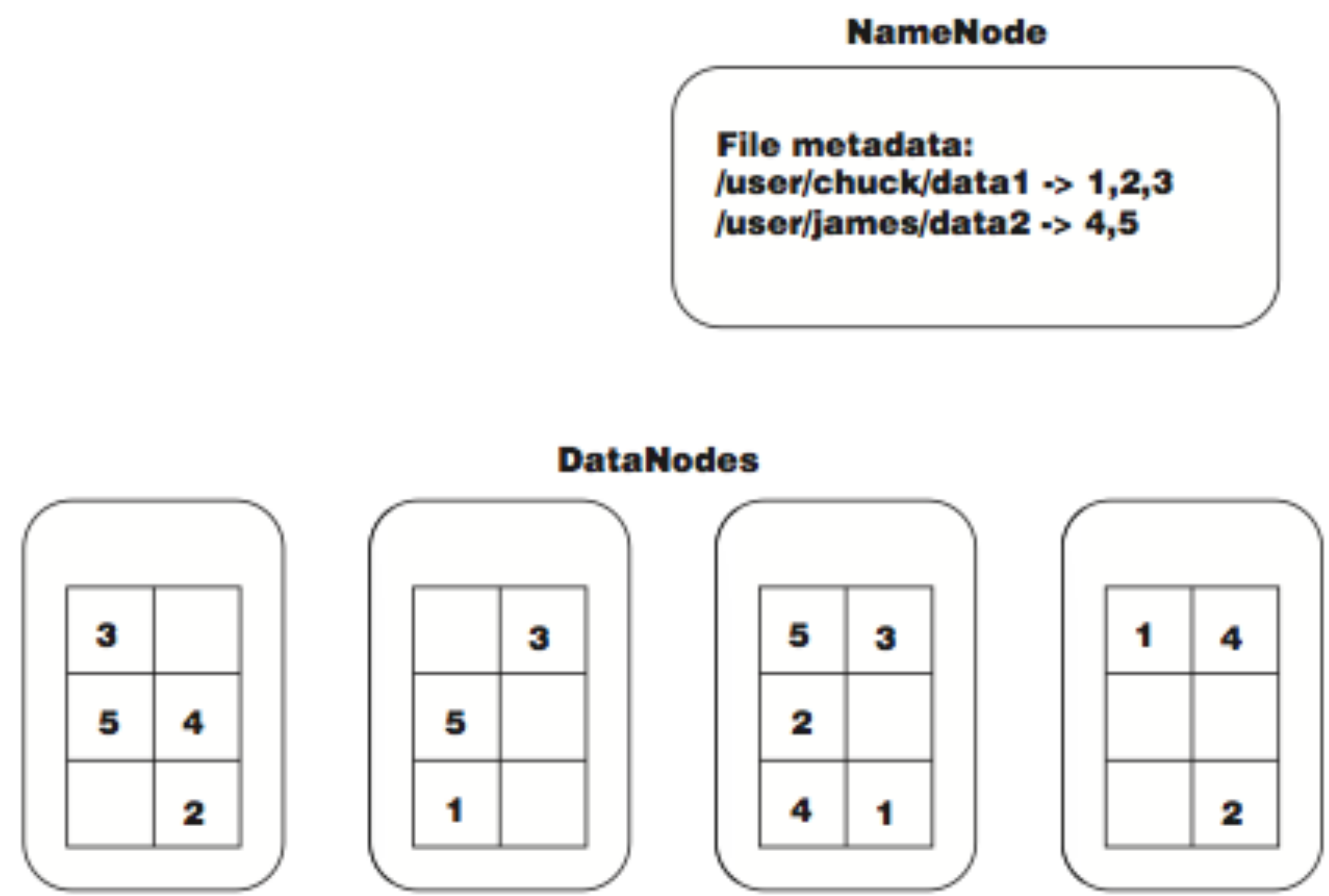
Move computation is cheaper than moving data

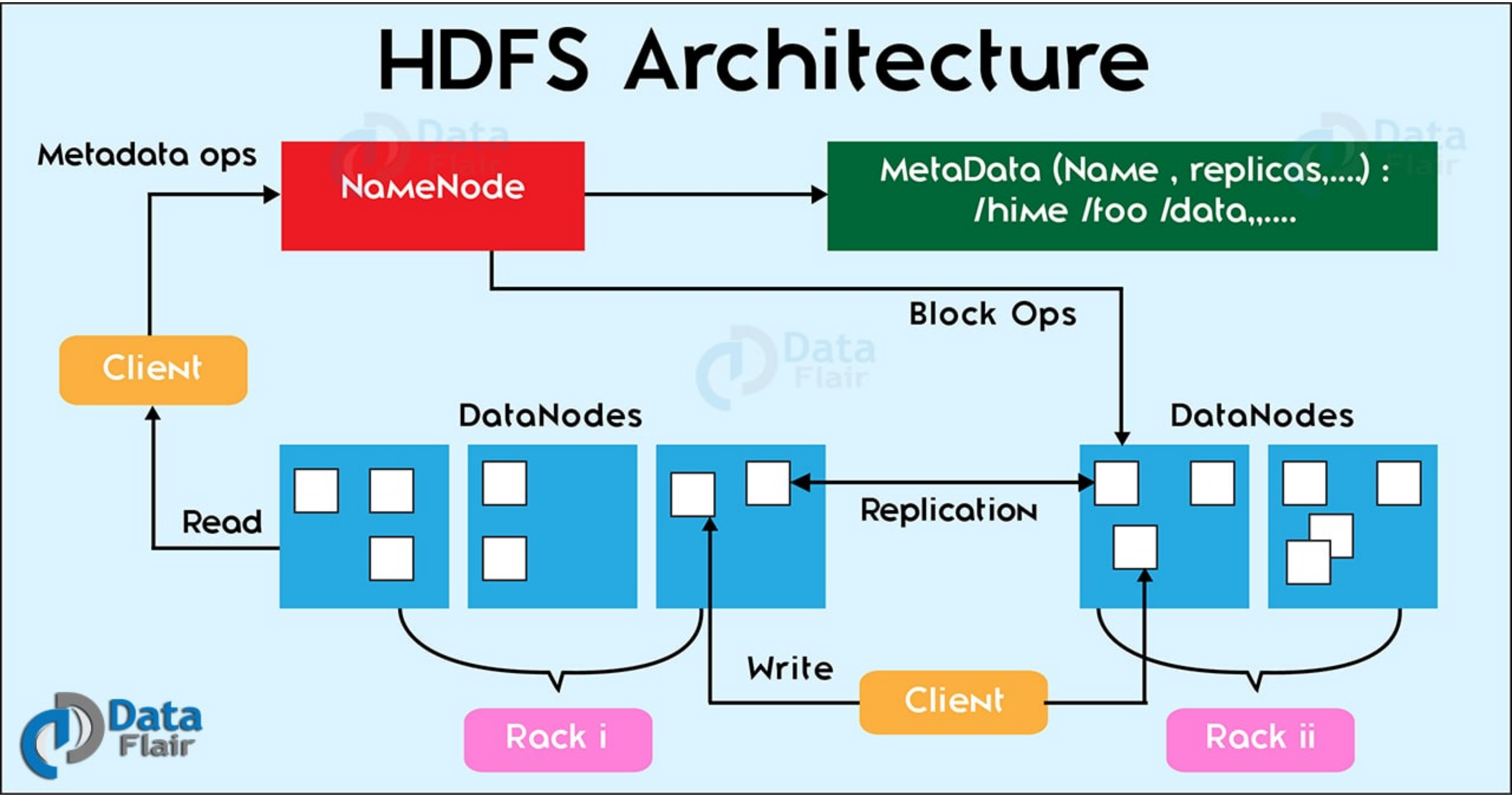
- Do not move data to workers: <u>move workers to data</u>
  - Store data on the local disks of nodes in the cluster
  - Start up workers on available slots with local data
- Disk access (latency) is slow but aggregated throughput is reasonable



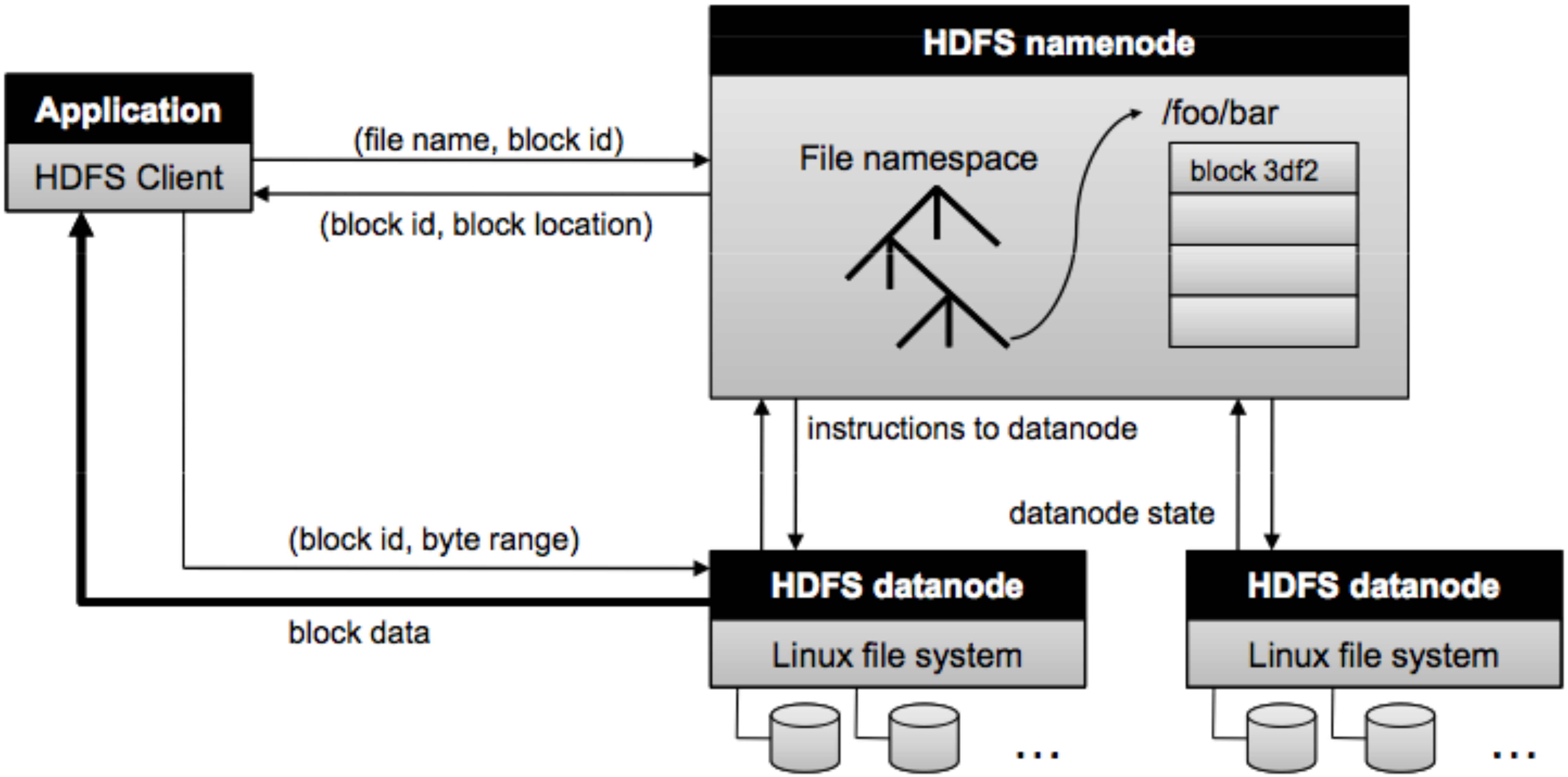
HDFS high level architecture

- Master/worker
- NameNode: file namespace manager
- DataNodes: manage local storage system
- Each file is a list of blocks
- Blocks are stored in a set of DataNodes




HDFS Architecture
Metadata ops
NameNode
MetaData (Name , replicas,....) :
/hime /foo /data,,....
Block Ops
Client
DataNodes
DataNodes
Replication
Read
Write
Client
Rack i
Rack ii
Data Flair

HDFS application view



Data replication

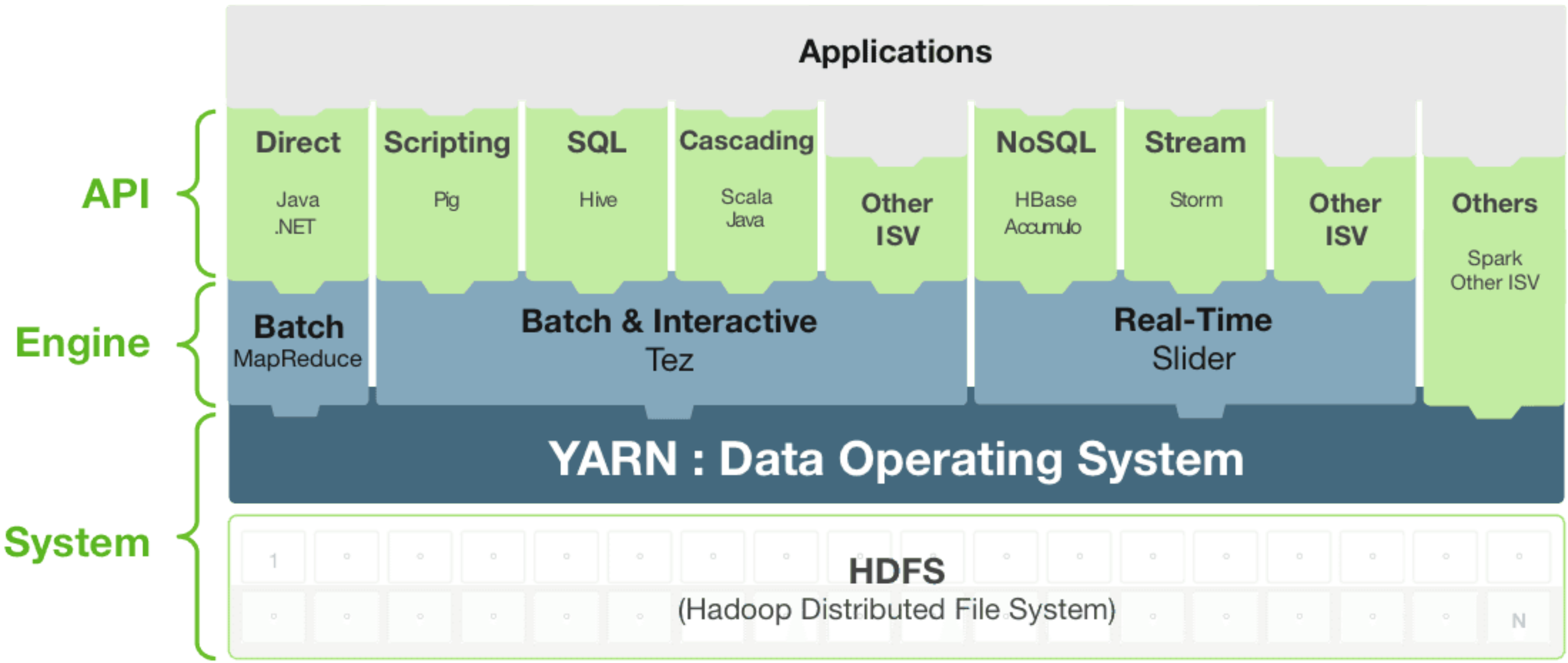
- Each file is a sequence of blocks
- All blocks of a file have the same size except the last
- Blocks of a file are replicated for fault tolerace
- Namenode keeps control of DataNodes' health and applies replication rules
- Algorithm to minimize read latency: get data from replica closest to the user

HDFS operations with files: FS shell

`hadoop dfs -fs` `<command> <directory>`

- Create folder: `hadoop dfs -fs mkdir /user/hadoop/data`
- List files in HDFS： `-ls /user/hadoop/data`
- Copy a file to HDFS: `-put local.txt /user/hadoop/data`
- Get a file from HDFS: `-get /user/hadoop/data/part-00000 d1.txt`
- Cat a file in HDFS: `-cat /user/hadoop/data/part-00000`

Hadoop applications and HDFS



HBASE

Family oriented DB

HBase

- map: key-value based
- persistent
- distributed
- sorted
- multidimensional
- sparse

HBase is a Map

Associative array (PHP), dictionary (Python), Hash (Ruby), or Object (JavaScript)

```
{
  "Smith" : "John",
  "Fernandez" : "Laura",
  "Kent" : "Clark",
  "Parker" : "Peter",
  "Richards" : "Rich"
}
```

HBase is distributed

- HBase and BigTable are built on distributed filesystems so that the underlying file storage can be spread out among an array of independent machines

- HBase sits atop either Hadoop's Distributed File System (HDFS)

- Data is replicated across a number of participating nodes

HBase is sorted

- HBase key/value pairs are kept in strict alphabetical order by key
- That is to say that the row for the key "aaaaa" should be right next to the row with key "aaaab" and very far from the row with key "zzzzz"
- Consider a table whose keys are domain names. It makes the most sense to list them in reverse notation (so "com.mydomain.www" rather than "www.mydomain.com") so that rows about a subdomain will be near the parent domain row.

HBase is multidimensional

```
{
  "1" : {
    "A" : "x",
    "B" : "z"
  },
  "aaaaa" : {
    "A" : "y",
    "B" : "w"
  },
  "aaaab" : {
    "A" : "world",
    "B" : "ocean"
  },
  "xyz" : {
    "A" : "hello",
    "B" : "there"
```

Row

Column family A

Column family B

HBase is multidimensional

- Column families are specified when the table is created
- A column family may have any number of columns, denoted by a column "qualifier" or "label"

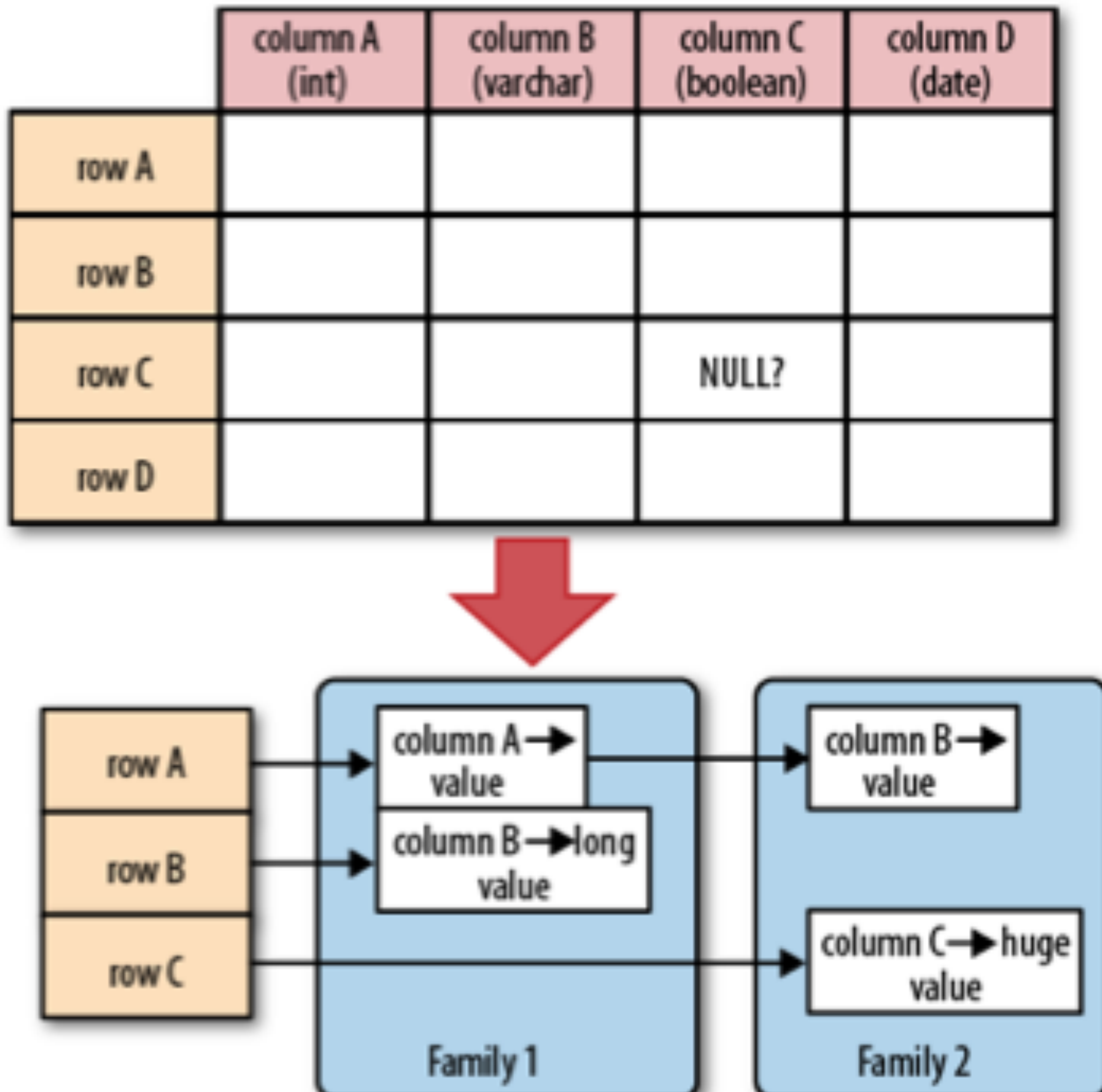
```
{
  "com.cnn.www" : {
    "contents" : {
      "html" : "",
      "txt" : "data"
    },
    "anchor" : {
      "" : "data"
    }
  },
```

Each row of the table has
- family contents, qualifiers “html” and “txt”
- family anchor, qualifier empty: “”

HBase is multidimensional

- When asking HBase for data, you must provide the full column name in the form "<family>:<qualifier>"
- So for example, both rows in the above example have three columns: "content:html", "content:txt" and "anchor:"
- All data is versioned either using an integer timestamp

Sparse: DBMS vs HBASE



Example: rows, families, qualifiers and time

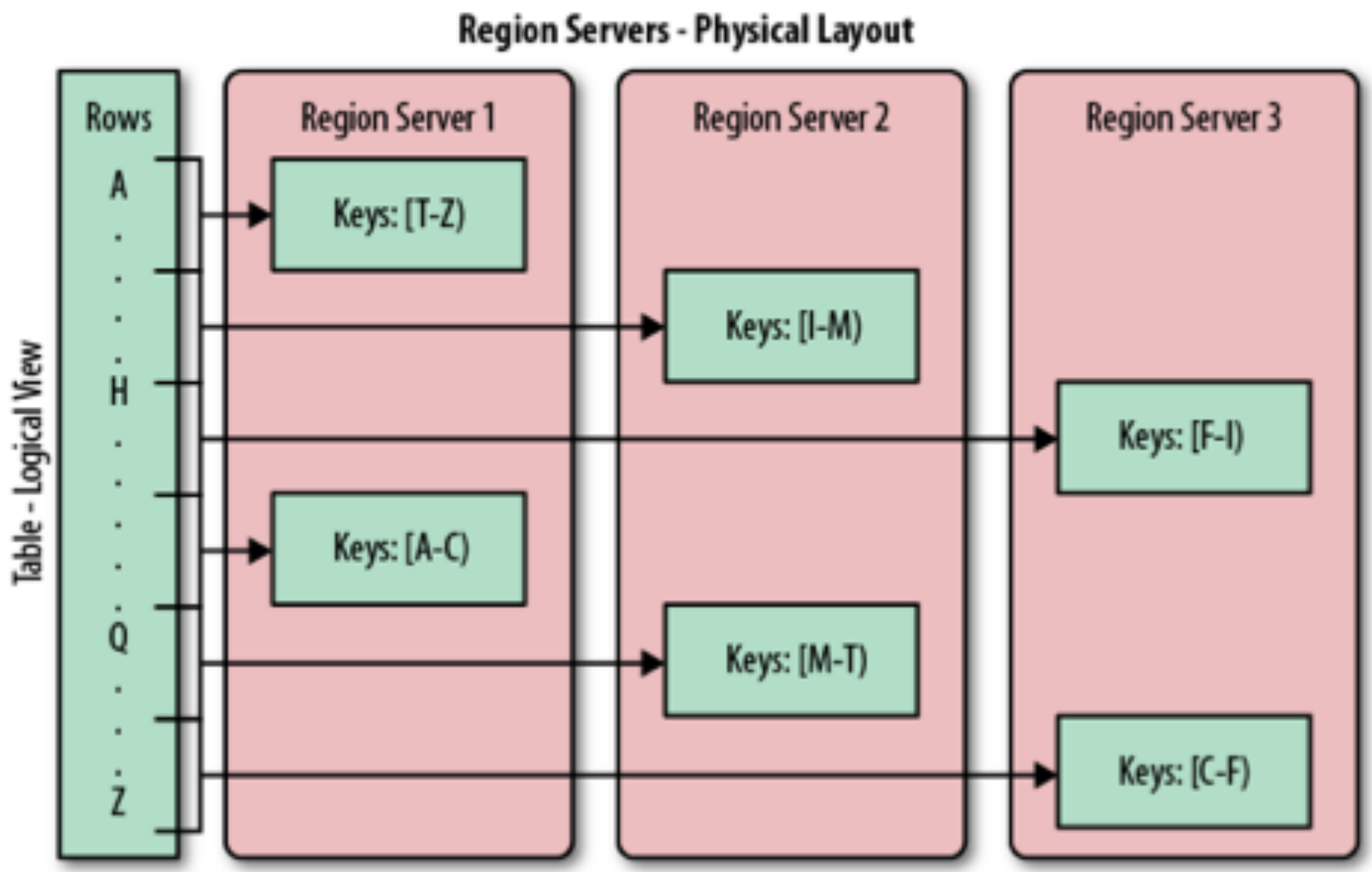
Row Key	Time stamp	Family contents:	Family anchor:
com.cnn.www	t9		anchor:**cnnsi.com** = "CNN"
com.cnn.www	t8		anchor:**my.look.ca** = "CNN.com"
com.cnn.www	t6	contents:**html** = "<html>..."	
com.cnn.www	t5	contents:html = "<html>..."	
com.cnn.www	t3	contents:html = "<html>..."	

qualifiers

Auto-sharding

- Regions: contiguous ranges of rows stored together
- Basic unit of scalability and load balancing
- When regions exceed size limit: they are split in equal halves
- Each region server stores many regions

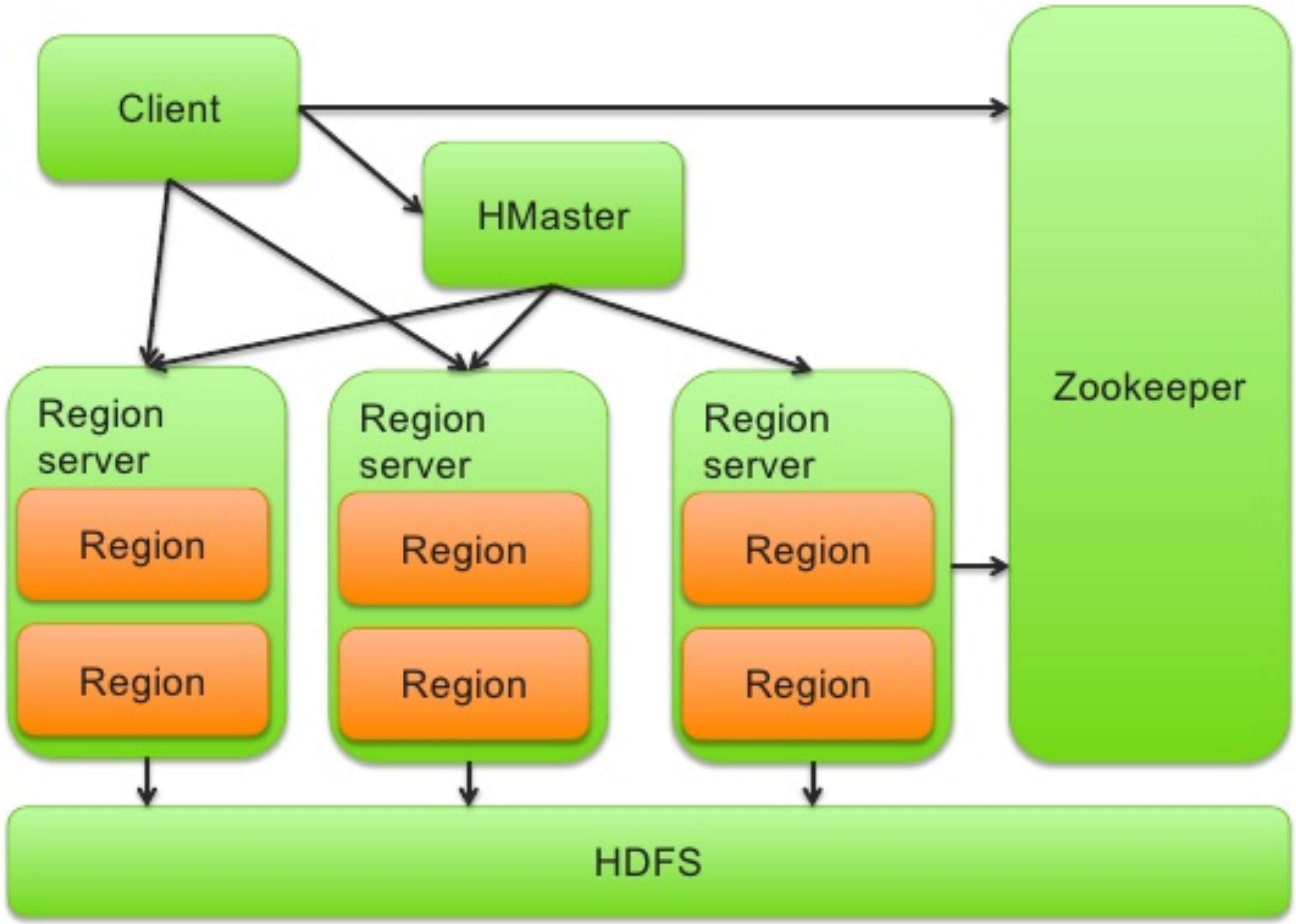
Region servers



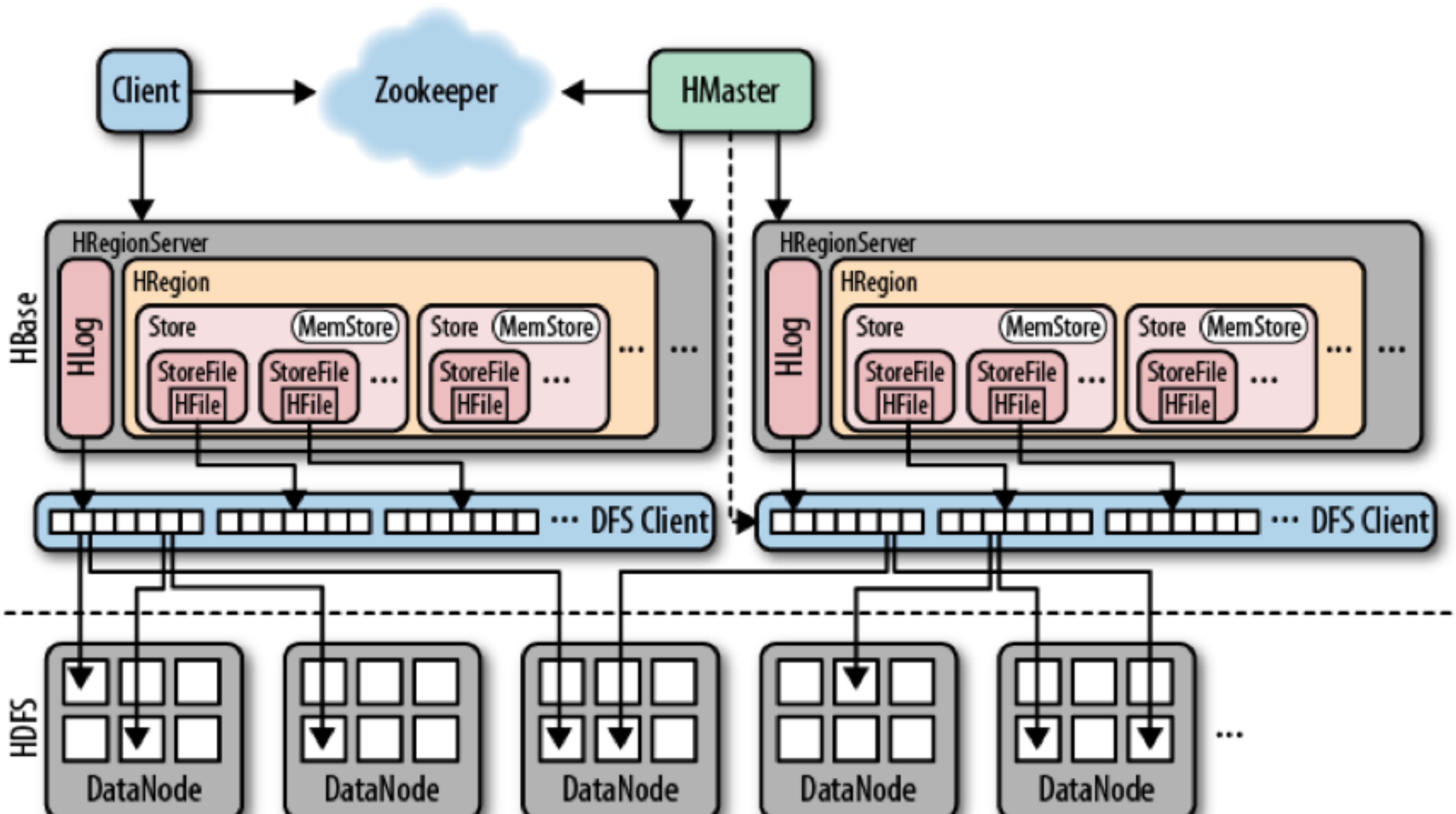
Region counts

- Region count: 10 - 1000 per server
- Each region: 1 - 2 GB
- Autosharding: automatic processes of splitting and serving regions

Apache HBase Architecture



HBASE Architecture



Read process (client)

- new client retrieves server with ROOT region
- gets server name with .META. table region
- get server name with row key
- gets row data (atomic) from server

Read process (server)

- *HRegionServer* opens a region and creates new *HRegion*
- Sets up a *Store* instance for each *HColumnFamily* for the table
- Store has one or more *StoreFiles* to access actual storage file *HFile*

Basic API: Put

```
Configuration conf = HBaseConfiguration.create();
HTable table = new HTable(conf, "testtable");
Put put = new Put(Bytes.toBytes("row1"));
put.add(Bytes.toBytes("colfam1"),
        Bytes.toBytes("qual1"),
        Bytes.toBytes("val1"));
put.add(Bytes.toBytes("colfam1"),
        Bytes.toBytes("qual2"),
        Bytes.toBytes("val2"));
table.put(put);
```

Basic API functions: get

```
Configuration conf = HBaseConfiguration.create();
HTable table = new HTable(conf, "testtable");
Get get = new Get(Bytes.toBytes("row1"));
get.addColumn(Bytes.toBytes("colfam1"),
              Bytes.toBytes("qual1"));
Result result = table.get(get);
byte[] val = result.getValue(Bytes.toBytes("colfam1"),
                             Bytes.toBytes("qual1"));
System.out.println("Value: " + Bytes.toString(val));
```

Scan: get data from HBASE

```
Scan scan = new Scan();
scan.addFamily(Bytes.toBytes("colfamily1"));
ResultScanner scanner = table.getScanner(scan);
for (Result res : scanner) {
  System.out.println(res);
}
scanner.close();
```

Hadoop ecosystem of Data Analysis tools

Other tools for MapReduce programming

- **Impala**
- **Spark**
- Abstractions: **Pig, Hive**

Apache Impala

Impala: low latency query engine

- not based on the MapReduce processing engine.
- *Designed to optimize latency* rather than scale and throughput, its architecture is similar to that of traditional massively parallel data warehouses such as Netezza, Greenplum and Teradata
- Data is read from the disk when the tables are initially scanned, and then remains in memory as it goes through multiple phases of processing.

Impala requirements

- Most of the data set has to fit into memory

- In general Impala requires significantly more memory per node than **MapReduce** based processing. 128GB of RAM are recommended, or large queries may fail when they run out of memory.

- **Impala** query can't recover from the loss of a node like MapReduce. If you lose a node your query will fail.

- **Impala** is recommended for queries that run quickly enough that restarting the entire query in case of a failure is not a major event.

Impala requirements

- **Long running daemons**: Impala daemons always stay on. There is no need for a start up cost and no moving of jar over the network or loading class files. Impala is just always ready.

- **Execution Engine in C++**

- **Use of LLVM**: use of LLVM to compile the query and all the functions used in this query into optimized machine code

Impala broadcast join execution plan

- **Impala** takes the smaller dataset distributes this dataset to all the Impala daemons involved with the query plan, where it will be stored as an in-memory hash table.
- Then each Impala daemon will read the parts of the larger dataset that are local to its node and use the in-memory hash table to find the rows that match between both tables, i.e. perform a hash-join.
- There is no need to read the entire large data set into memory, so Impala uses a 1GB buffer to read the large table and perform the joining part by part.

Impala example

```
select
       *
from
      Huge f JOIN small b on (f.fooBarId = b.barId)
where
      f.fooVal < 500 and
      f.fooVal + b.barVal < 1000;
```

When to use Impala

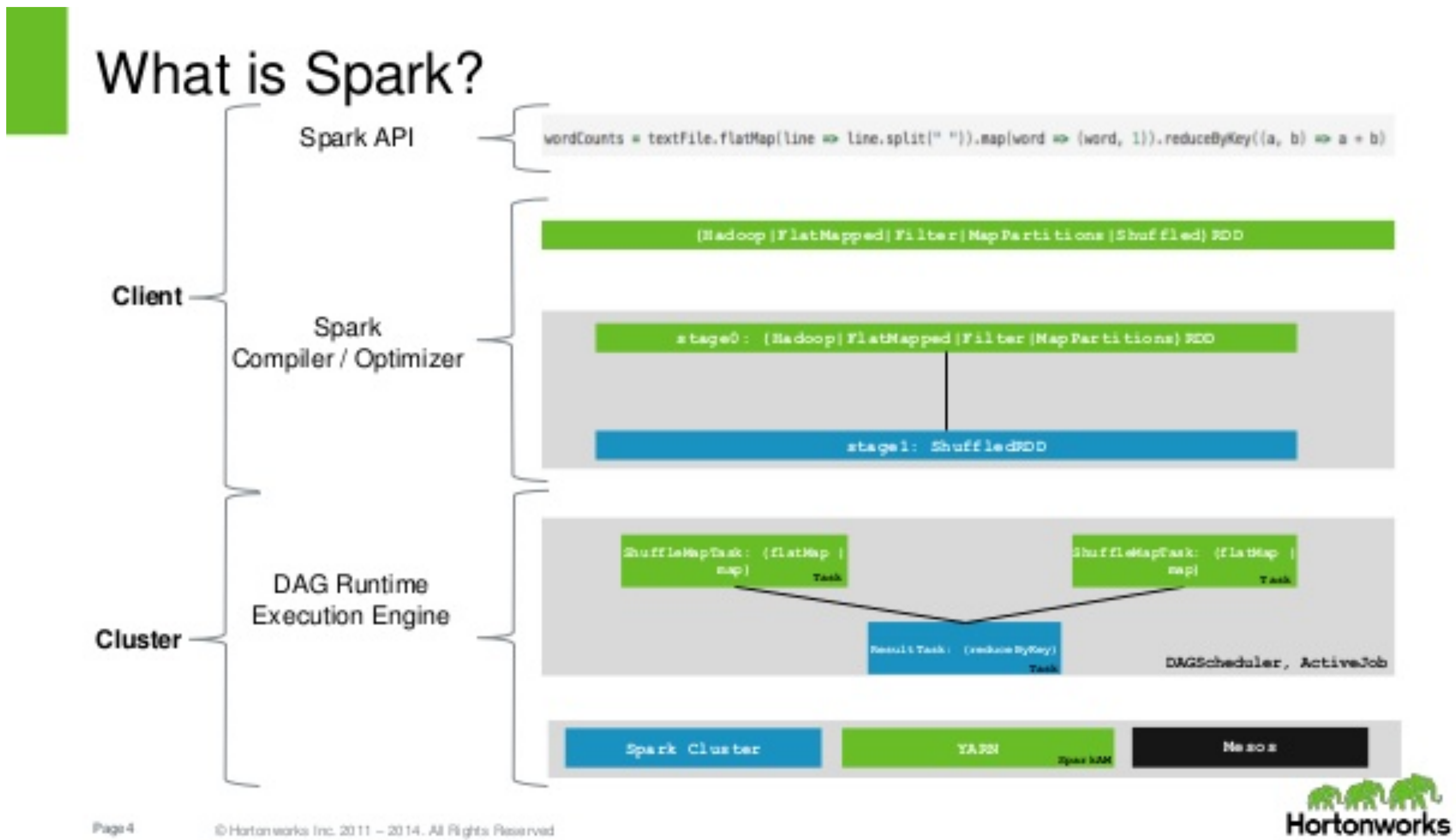
- Given its performance, Impala should be the first choice when using SQL on Hadoop.
- high-concurrency SQL access is a requirement.
- For many users who running SQL queries within Hadoop

Apache Spark

Apache Spark: iterative MapReduce data analysis

- Why **Spark**?
- MapReduce framework is limited to a very rigid data flow model that is unsuitable for many applications.
- reuse a dataset cached in memory for multiple processing tasks
- For example, applications such as iterative machine learning or interactive data analysis

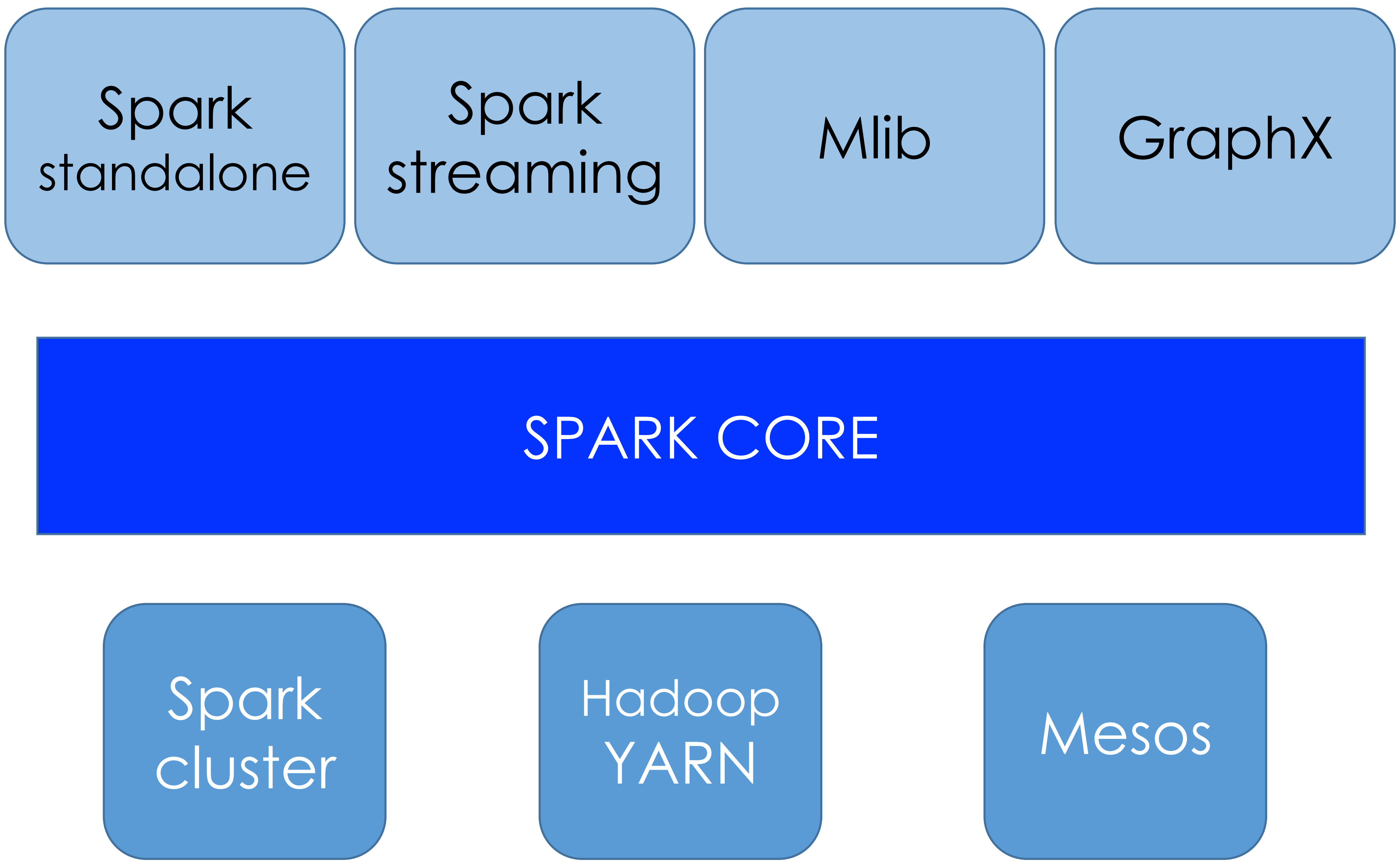
Apache Spark: iterative MapReduce data analysis



Spark overview

- **Support of DAG**: What Spark adds is the fact that the engine itself supports creating those complex chains of steps
- Complex algorithms and data processing pipelines within the same job and allows the framework to optimize the job as a whole
- **Simple**: Spark APIs are significantly cleaner and simpler than those of MapReduce.
- **Versatile:** Spark was built from the ground up to be an extensible, general purpose, parallel processing framework: stream processing engine: **Spark Streaming**, a graph processing engine called **GraphX**, and a machine learning framework called **MlLib**.

Spark architecture



Reduced I/O: Resilient Distributed Datasets

- **Resilient Distributed Datasets (RDD)** to reduce IO and maintaining the processed dataset in memory

- RDDs are collections of serializable elements. The collection may be partitioned, in which case it is stored on multiple nodes

- RDDs are typically created from an Hadoop InputFormat (file on HDFS for example), or by applying transformations on existing RDDs.

- When creating an RDD from InputFormat, the number of partitions is determined by the InputFormat, very similar to the way splits are determined in MapReduce jobs.

Spark data transformations

- **map**: Applies a function on every element of an RDD to produce a new RDD
  - For example: lines.map(s⇒s.length) takes an RDD of Strings (“lines”) and returns an RDD with the length of the strings.

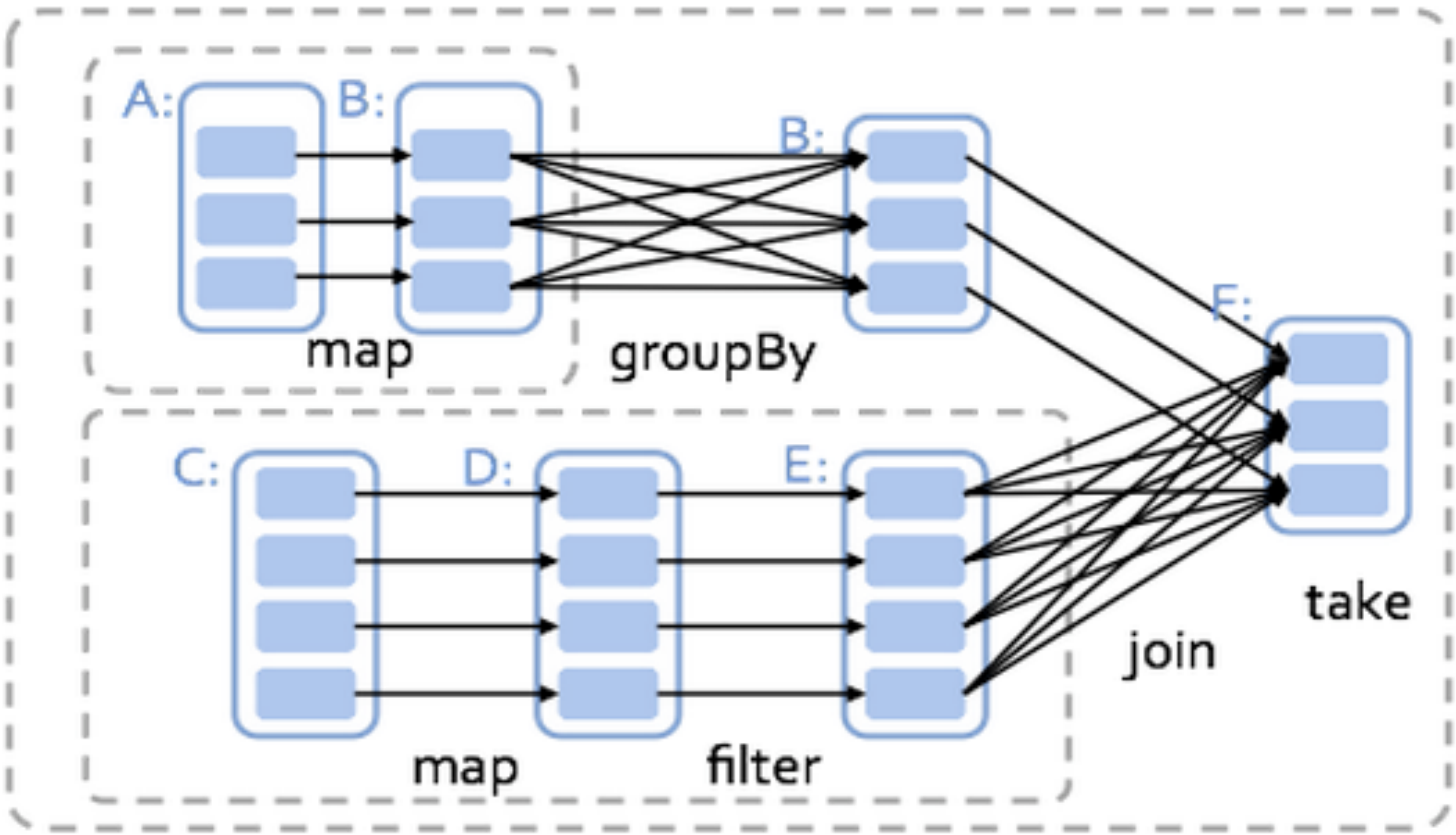
- **filter**: Takes a boolean function as a parameter, executes this function on every element of the RDD, and return a new RDD containing only the elements for which the function returned true.
  - For example lines.filter(s⇒(s.length>50)) returns an RDD containing only the lines with more than 50 characters

- **join**: Joins two key/value RDDs by their keys.
  - For example, lets assume we have two RDDs: lines and more_lines. Each entry in both RDDs containing the line length as the key and the line as the value. lines.join(more_lines) will return for each line length a pair of Strings, one from “lines” RDD and one from “more lines”. Each resulting element looks like: <length,<line,more_line>>

Spark simple example: top 10

```
rdd1.map(lines).filter(”data”)

rdd2.map(lines).groupBy(key)

rdd2.join(rdd1, key).take(10)
```

Spark simple example: top 10



Spark features

- **Storage:** Spark gives the developers a lot of flexibility regarding how RDDs are stored. This includes: In-memory on a single node, in-memory but replicated to multiple nodes, or persisted to disk.
- Developer controls the persistence: RDD can go multiple stages of transformation without storing anything to disk
- **Multi Language** While Spark itself is developed in Scala, Spark APIs are implemented for **Java**, **Scala**, **Python and R**
- **Interactive shell (REPL)** Spark includes a shell (also called REPL). This allows for fast interactive experimentation with the data and easy validation of ideas

Spark two-set join example

- Load “foo” and “bar” data sets into two RDDs

```
var fooTable = sc.textFile("hdfs:///user/root/foo")
var barTable = sc.textFile("hdfs:///user/root/bar”)
```

- Split each row of “foo” into a collection of separate cells

```
var fooSplit = fooTable.map(line => line.split("\\|"))
```

- Filter the splitted “foo” dataset and keep only the elements where the second column is smaller than 500

```
var fooFiltered = fooSplit.filter(cells => cells(1).toInt <= 500)
```

- Convert the results into key/value pairs using the ID column as the key

```
var fooKeyed = fooFiltered.keyBy(cells => cells(2))
```

- Split the columns in “bar” in the same way we splitted “foo” and again convert into key/value pairs with the ID as the key

```
var barSplit = barTable.map(line => line.split("\\|"))
var barKeyed = barSplit.keyBy(cells => cells(0))
```

Spark join example

- Join “bar” and “foo”

```
var joinedValues = fooKeyed.join(barKeyed)
```

- Filter the joined results - The filter function here takes the value of joinedVal RDD, which contains a pair of a “foo” and a “bar” rows. We take the first column from each row and check if their sum is lower than 1000

```
var joinedFiltered =joinedValues.filter(joinedVal =>
           joinedVal._2._1(1).toInt +
           joinedVal._2._2(1).toInt <= 1000)
```

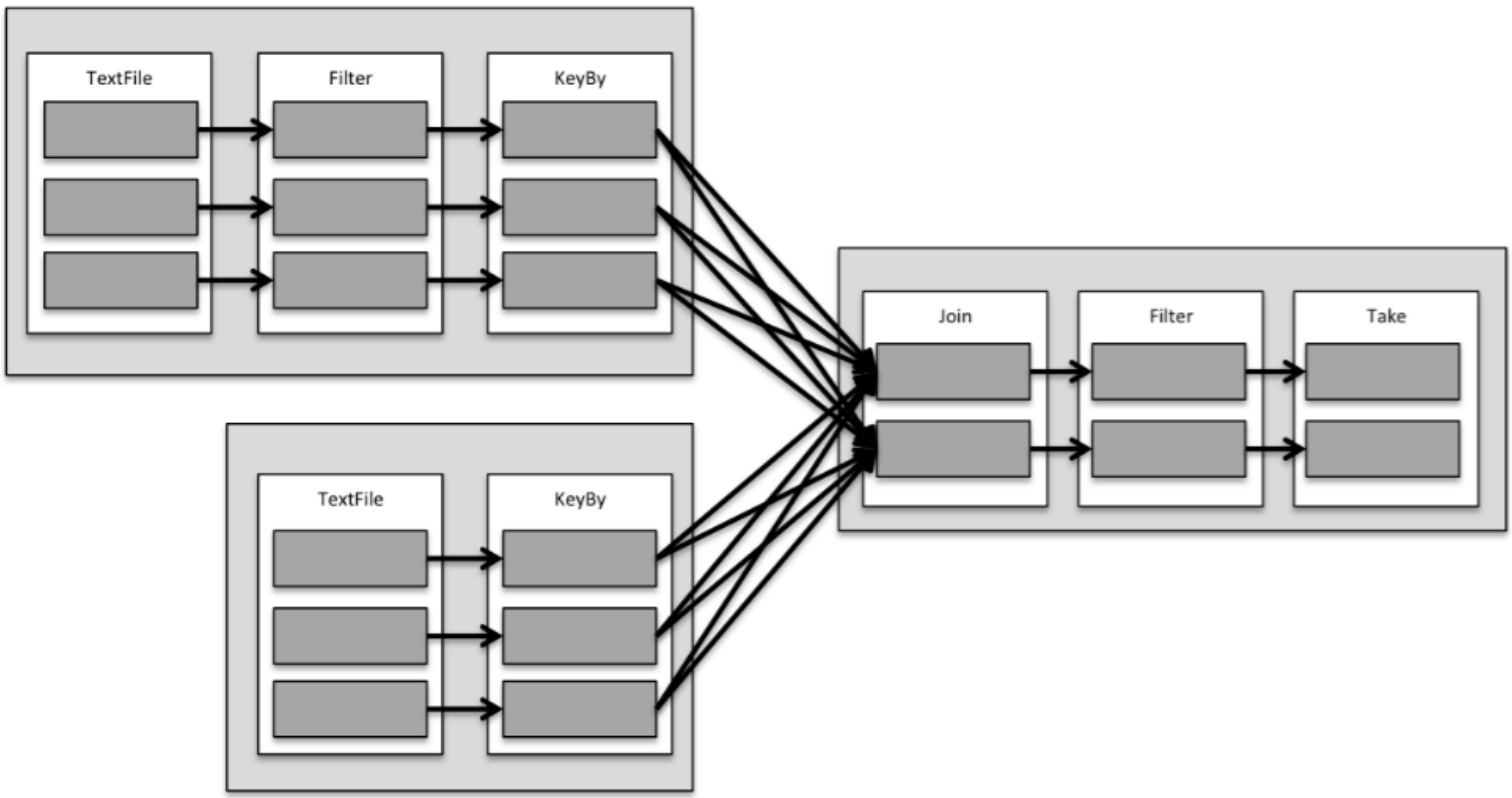
- Show the first 10 results - note that this is the only action in the code, therefore the entire chain of transformations we defined here will only be triggered at this point.

```
joinedFiltered.take(10)
```

Spark example: joining two sets

```
var fooTable = sc.textFile("hdfs:///user/root/foo")

var barTable = sc.textFile("hdfs:///user/root/bar")

var fooSplit = fooTable.map(line => line.split("\\|"))

var fooFiltered = fooSplit.filter(cells => cells(1).toInt
<= 500)

var fooKeyed = fooFiltered.keyBy(cells => cells(2))

var barSplit = barTable.map(line => line.split("\\|"))

var barKeyed = barSplit.keyBy(cells => cells(0))

var joinedValues = fooKeyed.join(barKeyed)

var joinedFiltered =joinedValues.filter(joinedVal =>
           joinedVal._2._1(1).toInt +
           joinedVal._2._2(1).toInt <= 1000)

joinedFiltered.take(10)
```

Example execution plan



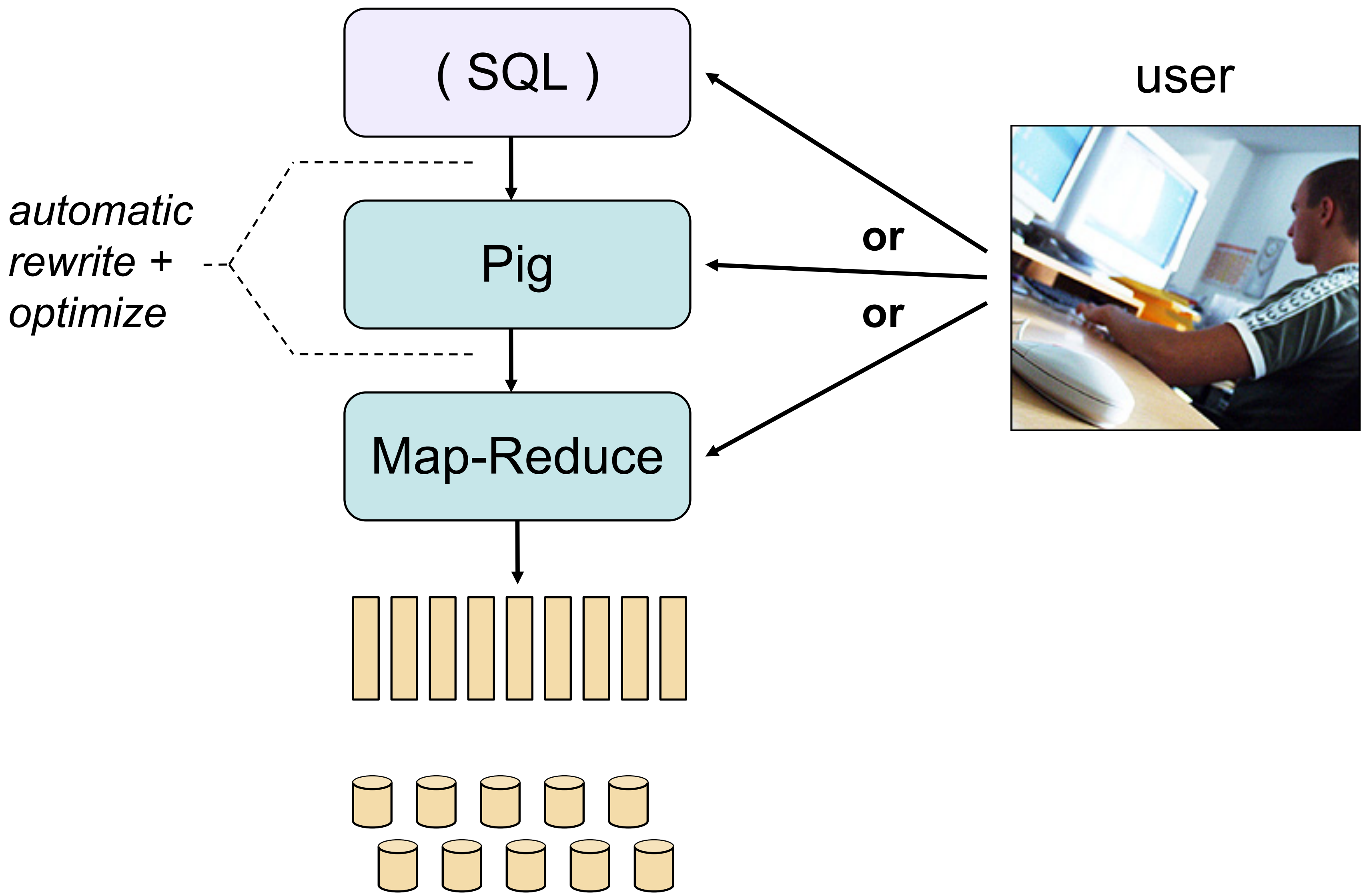
Spark conclusions

- Good implementation for large datasets
- Flexible: selection of high level languages and libraries
- Oriented to DAG processing: data pipelines
- More popular than hadoop map reduce now

MapReduce abstractions

- Since MapReduce was introduced there have been many projects trying to make it easier to use
- **ETL(Extract, Transform and Load)** are optimized to take a given dataset and do a bunch of operations on it to produce a set of outcomes.
  - **Pig** (Yahoo), Crunch, Cascading
- **Query(SQL)** are oriented to use SQL language to ask a question of the data to get an answer.
  - **Hive** (Facebook)

Pig: high level ETL queries

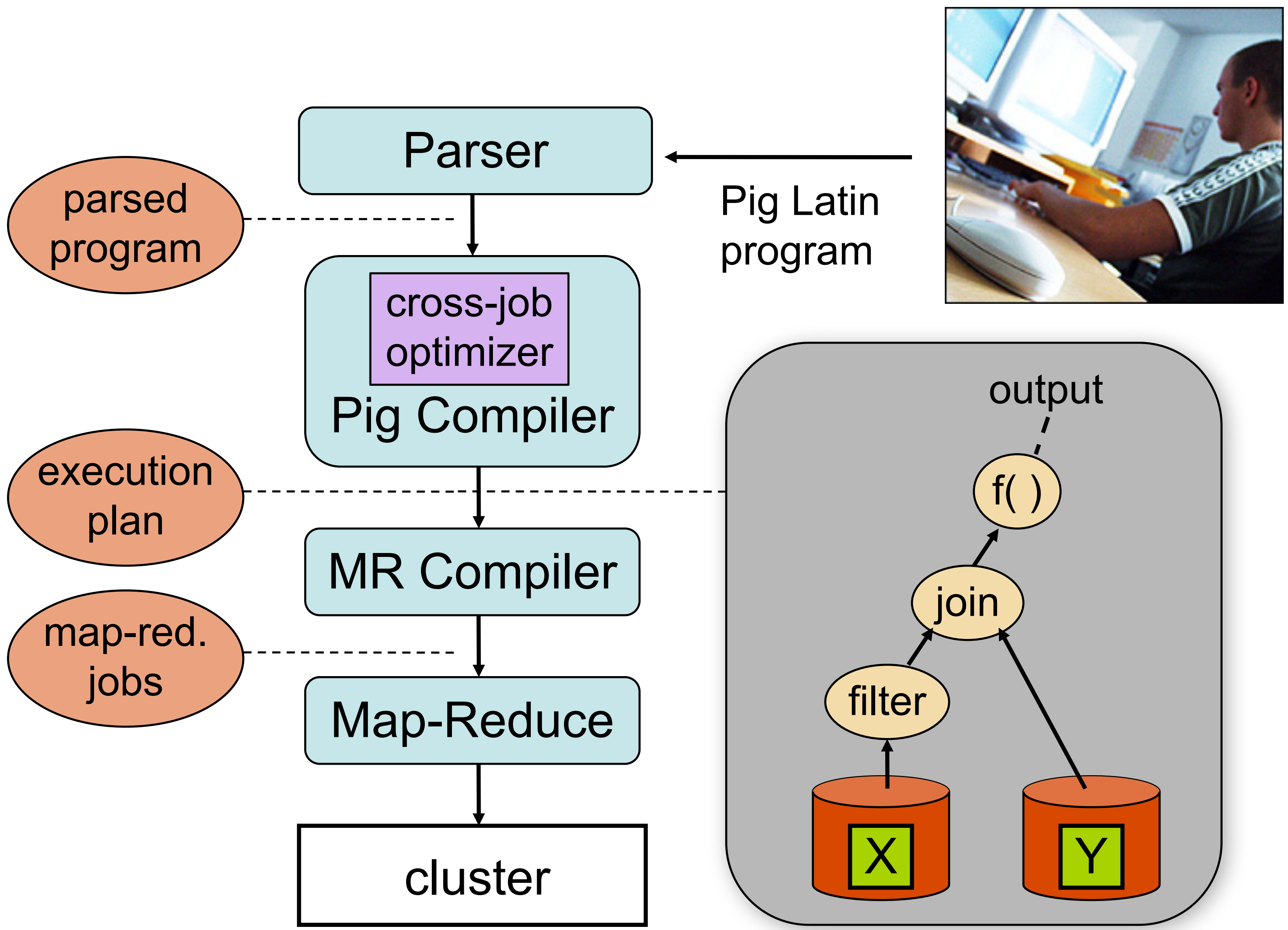


Pig “explicits” MR

- Map-reduce encapsulates three primitives:
    - record processing -> group creation -> group processing
- Pig: three operations:
    - a=FOREACH input GENERATE flatten(Map(*))
    - b=GROUP a BY $0
    - c=FOREACH b GENERATE Reduce(*)
- Operations: filtering, projection, combine tables

Pig vs SQL

- SQL is declarative: what data to get, not how
- Pig: how to get data step by step
  - Longer queries
  - Sequence order is semantic
  - Incremental build using intermediate data: error prone

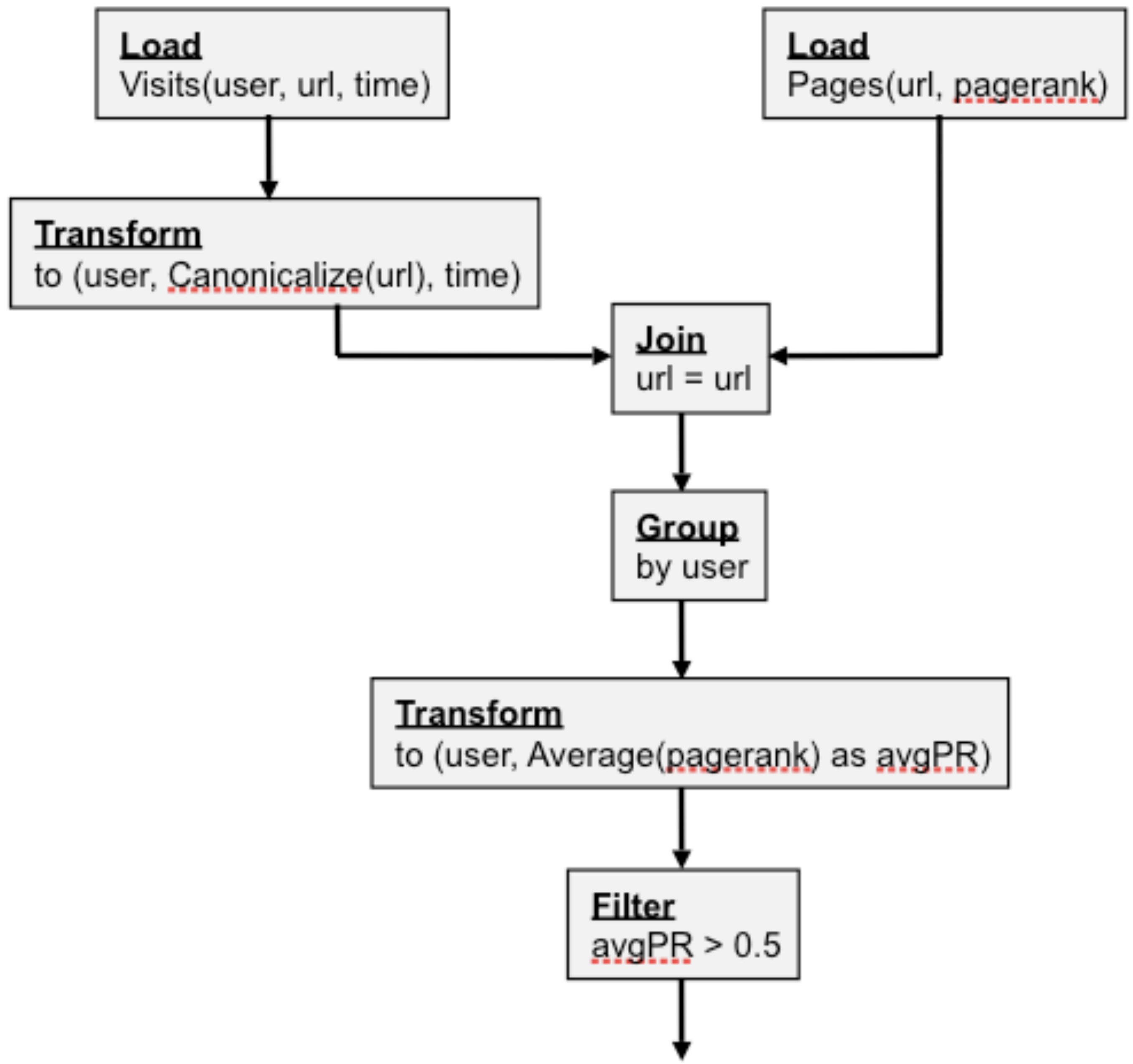

Parser
Pig Latin
program
parsed
program
cross-job
optimizer
Pig Compiler
execution
plan
MR Compiler
map-red.
jobs
Map-Reduce
cluster
output
f( )
join
filter
X
Y

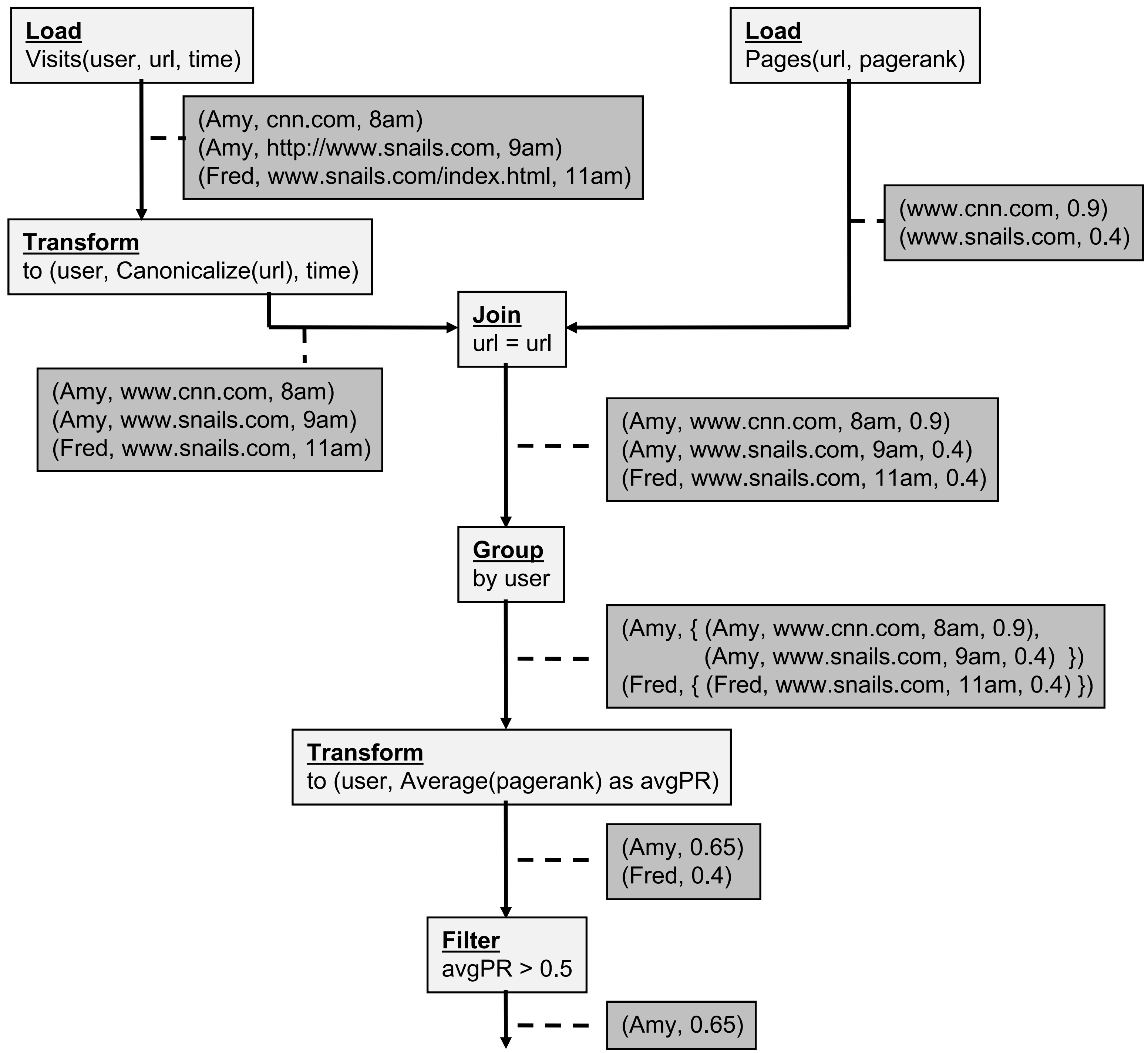
Pig count

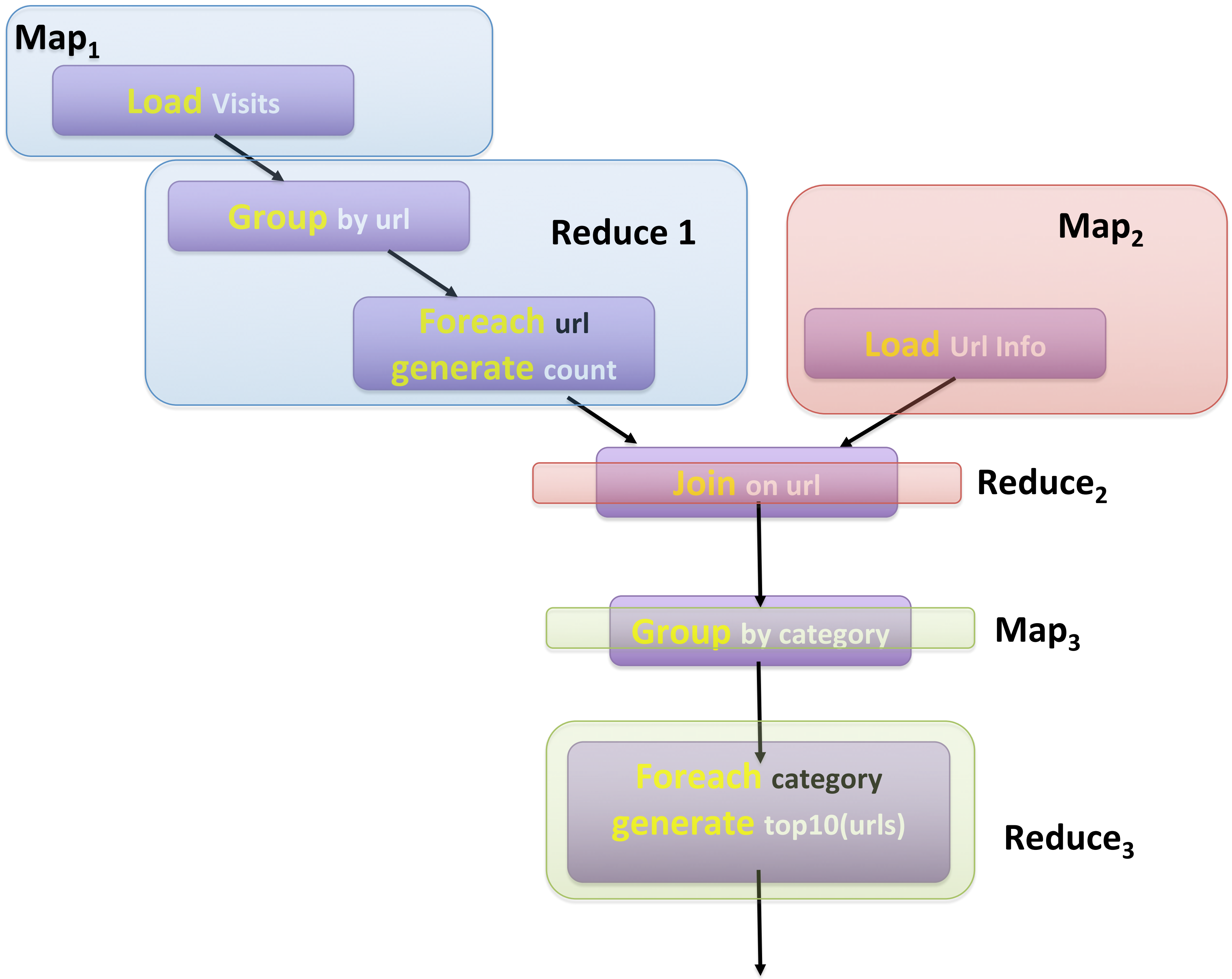
```
input = LOAD ‘input-dir’ USING TextLoader();

words = FOREACH input GENERATE FLATTEN(TOKENIZE(*));

grouped = GROUP words BY $0;

counts = FOREACH grouped GENERATE group, COUNT(words);

STORE counts INTO ‘out-dir’;
```

visual structure

Find users who tend to visit “good” pages.




Load
Visits(user, url, time)
(Amy, cnn.com, 8am)
(Amy, http://www.snails.com, 9am)
(Fred, www.snails.com/index.html, 11am)
Load
Pages(url, pagerank)
(www.cnn.com, 0.9)
(www.snails.com, 0.4)
Transform
to (user, Canonicalize(url), time)
Join
url = url
(Amy, www.cnn.com, 8am)
(Amy, www.snails.com, 9am)
(Fred, www.snails.com, 11am)
(Amy, www.cnn.com, 8am, 0.9)
(Amy, www.snails.com, 9am, 0.4)
(Fred, www.snails.com, 11am, 0.4)
Group
by user
(Amy, { (Amy, www.cnn.com, 8am, 0.9),
(Amy, www.snails.com, 9am, 0.4) })
(Fred, { (Fred, www.snails.com, 11am, 0.4) })
Transform
to (user, Average(pagerank) as avgPR)
(Amy, 0.65)
(Fred, 0.4)
Filter
avgPR > 0.5
(Amy, 0.65)


Map1
Load Visits
Group by url
Reduce 1
Foreach url
generate count
Map2
Load Url Info
Join on url
Reduce2
Group by category
Map3
Foreach category
generate top10(urls)
Reduce3

Hive: data warehousing and analytics on Hadoop

Why create Hive?

- Problem: Data, 2+ TB raw data per day
- Hadoop
  - better scalability/availability than DBs
  - Map-reduce is hard to program to sql users
- HIVE: a SQL language for hadoop

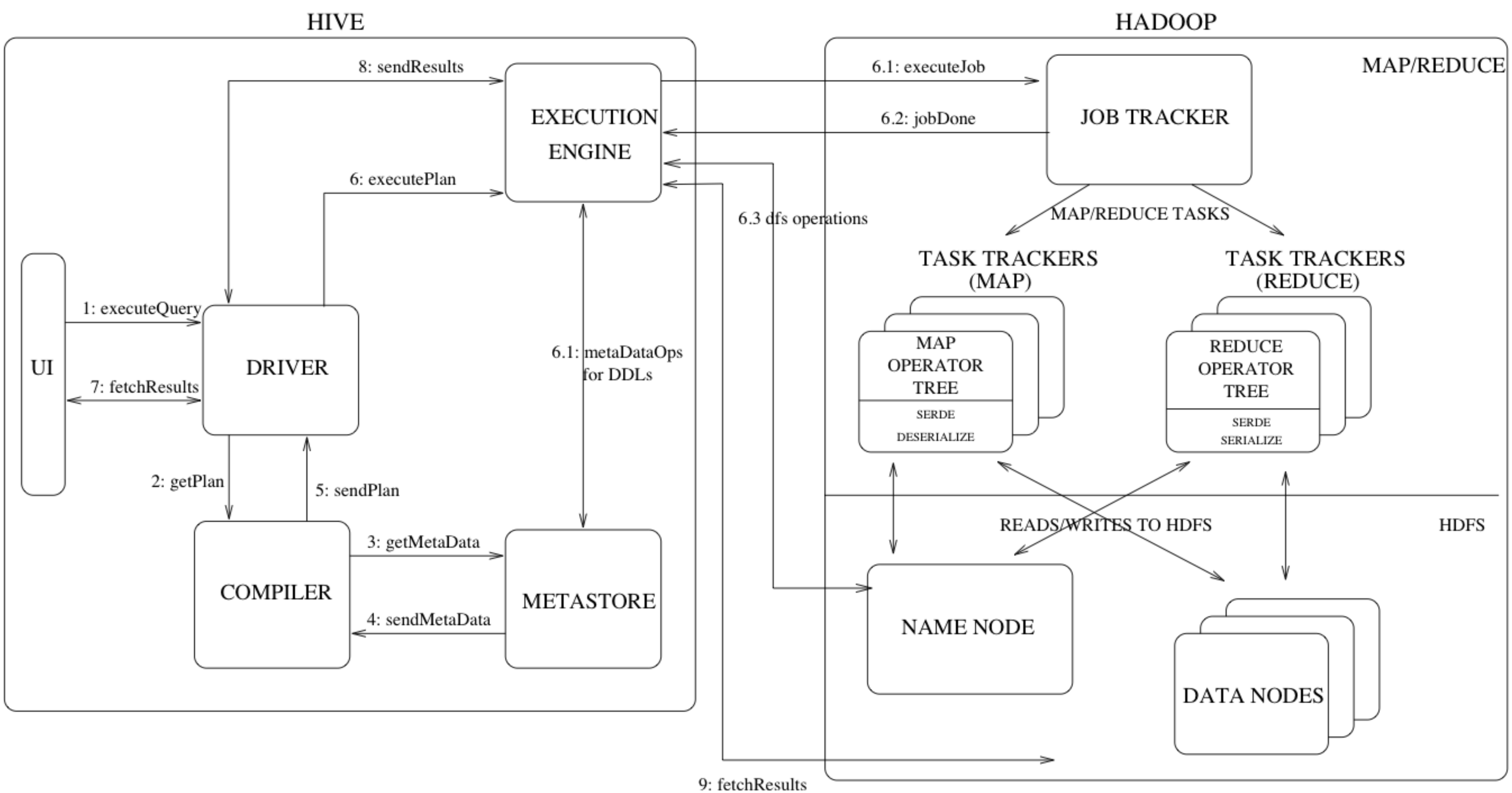
What is Apache HIVE?

- Open source data warehouse system
- Rich data types: structs, lists, maps
- SQL programming tool
- Rich meta data to allow data discovery and for optimization
- For large dataset batch queries
- Support for data partitioning

Applications

- Summaries: daily aggregation of click counts
- Data mining (assembling training data): user engagement as a function of user attributes
- Spam detection: anomalous usage patterns of APIs
- Ad hoc analysis: how many group admins broken down by state/country

System overview



Daily data statistics

- 3200 jobs/day with 800K map reduce tasks
- 55TB compressed data scanned
- 15TB of compressed output data to hdfs
- 80 M compute minutes

Database creation and data import

- Create database

```
create database customers;
```

- Create invoices table

```
create table invoices(id INT, customer INT, product STRING,
cost DOUBLE) row format delimited fields terminated by ‘,’
stored as textfile;
```

- Insertar data from HDFS file

```
LOAD DATA INPATH ‘invoices.txt’  OVERWRITE INTO TABLE
invoices;
```

Using Hive partitions

```
CREATE TABLE web_url(url STRING, website STRING, date
DATETIME)

PARTITIONED BY (country STRING)

STORED AS TEXTFILE;
```

- Each partition corresponds to a particular country value
- It is stored in a separated HDFS folder

Using Hive partitions

```
SELECT *

FROM web_url

WHERE date >=‘2017-12-24’ AND country= ‘UK’ AND

web_url.site like ‘%example.com’
```