Map Reduce
A scalable distributed programming model to process Big Data
MapReduce

• Published in 2004 by Google
  o used to rewrite the production indexing system with 24 MapReduce operations (in August 2004 alone, 3288 TeraBytes read, 80k machine-days used, jobs of 10’ avg)

• Distributed programming model

• Process large data sets with parallel algorithms on a cluster of common machines, e.g., PCs

• Great for parallel jobs requiring pieces of computations to be executed on all data records

• Move the computation (algorithm) to the data (remote node, PC, disk)

• Inspired by the map and reduce functions used in functional programming
  o In functional code, the output value of a function depends only on the arguments that are passed to the function, so calling a function $f$ twice with the same value for an argument $x$ produces the same result $f(x)$ each time; this is in contrast to procedures depending on a local or global state, which may produce different results at different times when called with the same arguments but a different program state.
MapReduce: working principles

• Consists of two functions, a **Map** and a **Reduce**
  - The Reduce is optional
  - Additional shuffling / finalize steps, implementation specific

• **Map** function
  - Process each record (document) → INPUT
  - Return a list of key-value pairs → OUTPUT

• **Reduce** function
  - for each key, reduces the list of its values, returned by the map, to a “single” value
  - Returned value can be a complex piece of data, e.g., a list, tuple, etc.
Map

• Map functions are called once for each document:

```javascript
function(doc) {
    emit(key1, value1); // key1 = f_k1(doc); value1 = f_v1(doc)
    emit(key2, value2); // key2 = f_k2(doc); value2 = f_v2(doc)
}
```

• The map function can choose to skip the document altogether or emit one or more key/value pairs

• Map function may **not** depend on any information outside the document
  - This independence is what allows map-reduces to be generated incrementally and **in parallel**
  - Some implementations allow global / scope variables
## Map example

- Example database, a collection of docs describing university exam records

<table>
<thead>
<tr>
<th>Id</th>
<th>Exam</th>
<th>Student</th>
<th>AYear</th>
<th>Date</th>
<th>Mark</th>
<th>CFU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Database</td>
<td>s123456</td>
<td>2015-16</td>
<td>31-01-2016</td>
<td>29</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Computer architectures</td>
<td>s123456</td>
<td>2015-16</td>
<td>03-07-2015</td>
<td>24</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>Computer architectures</td>
<td>s654321</td>
<td>2015-16</td>
<td>26-01-2016</td>
<td>27</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>Database</td>
<td>s654321</td>
<td>2014-15</td>
<td>26-07-2015</td>
<td>26</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>Software engineering</td>
<td>s123456</td>
<td>2014-15</td>
<td>14-02-2015</td>
<td>21</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>Bioinformatics</td>
<td>s123456</td>
<td>2015-16</td>
<td>18-09-2016</td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>Software engineering</td>
<td>s654321</td>
<td>2015-16</td>
<td>28-06-2016</td>
<td>18</td>
<td>8</td>
</tr>
</tbody>
</table>
Map example (1)

- List of exams and corresponding marks

Function(doc){
    emit(doc.exam, doc.mark);
}

Result:

<table>
<thead>
<tr>
<th>doc.id</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Bioinformatics</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>Computer architectures</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>Computer architectures</td>
<td>27</td>
</tr>
<tr>
<td>1</td>
<td>Database</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>Database</td>
<td>26</td>
</tr>
<tr>
<td>8</td>
<td>Database</td>
<td>25</td>
</tr>
<tr>
<td>5</td>
<td>Software engineering</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>Software engineering</td>
<td>18</td>
</tr>
</tbody>
</table>

Id: 2
Exam: Computer architectures
Student: s123456
AYear: 2015-16
Date: 03-07-2015
Mark=24
CFU=10

Id: 3
Exam: Computer architectures
Student: s654321
AYear: 2015-16
Date: 26-01-2016
Mark=27
CFU=10

Id: 4
Exam: Database
Student: s654321
AYear: 2014-15
Date: 26-07-2015
Mark=26
CFU=8

Id: 5
Exam: Software engineering
Student: s123456
AYear: 2014-15
Date: 14-02-2015
Mark=21
CFU=8

Id: 6
Exam: Bioinformatics
Student: s123456
AYear: 2015-16
Date: 18-09-2016
Mark=30
CFU=6

Id: 7
Exam: Software engineering
Student: s654321
AYear: 2015-16
Date: 28-06-2016
Mark=18
CFU=8

Id: 8
Exam: Database
Student: s987654
AYear: 2014-15
Date: 28-06-2015
Mark=25
CFU=8

DATA MANAGEMENT AND VISUALIZATION
Map example (2)

*Ordered list of exams, academic year, and date, and select their mark*

Function(doc) {
    key = [doc.exam, doc.AYear]
    value = doc.mark
    emit(key, value);
}

```
<table>
<thead>
<tr>
<th>Id</th>
<th>Exam</th>
<th>Student</th>
<th>AYear</th>
<th>Date</th>
<th>Mark</th>
<th>CFU</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>[Bioinformatics, 2015-16]</td>
<td>s123456</td>
<td>2015-16</td>
<td>18-09-2016</td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>[Database, 2015-16]</td>
<td>s654321</td>
<td>2015-16</td>
<td>31-01-2016</td>
<td>29</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>[Software engineering, 2015-16]</td>
<td>s654321</td>
<td>2015-16</td>
<td>28-06-2016</td>
<td>18</td>
<td>8</td>
</tr>
</tbody>
</table>
```
Ordered list of students, with mark and CFU for each exam

```javascript
Function(doc) {
    key = doc.student
    value = [doc.mark, doc.CFU]
    emit(key, value);
}
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Exam</th>
<th>Student</th>
<th>AYear</th>
<th>Date</th>
<th>Mark</th>
<th>CFU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Database</td>
<td>s123456</td>
<td>2015-16</td>
<td>31-01-2016</td>
<td>29</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Computer architectures</td>
<td>s123456</td>
<td>2015-16</td>
<td>03-07-2015</td>
<td>24</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>Computer architectures</td>
<td>s654321</td>
<td>2015-16</td>
<td>26-01-2016</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Database</td>
<td>s654321</td>
<td>2014-15</td>
<td>26-07-2015</td>
<td>26</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>Software engineering</td>
<td>s123456</td>
<td>2014-15</td>
<td>14-02-2015</td>
<td>21</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>Software engineering</td>
<td>s654321</td>
<td>2015-16</td>
<td>28-06-2015</td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>Bioinformatics</td>
<td>s123456</td>
<td>2015-16</td>
<td>18-09-2016</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

**Result:**

<table>
<thead>
<tr>
<th>doc.i.d</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s123456</td>
<td>[29, 8]</td>
</tr>
<tr>
<td>2</td>
<td>s123456</td>
<td>[24, 10]</td>
</tr>
<tr>
<td>5</td>
<td>s123456</td>
<td>[21, 8]</td>
</tr>
<tr>
<td>6</td>
<td>s123456</td>
<td>[30, 8]</td>
</tr>
<tr>
<td>3</td>
<td>s654321</td>
<td>[27, 10]</td>
</tr>
<tr>
<td>4</td>
<td>s654321</td>
<td>[26, 8]</td>
</tr>
<tr>
<td>7</td>
<td>s654321</td>
<td>[18, 8]</td>
</tr>
<tr>
<td>8</td>
<td>s987654</td>
<td>[25, 8]</td>
</tr>
</tbody>
</table>
Reduce

• Documents (key-value pairs) emitted by the map function are sorted by key
  o some platforms (e.g. Hadoop) allow you to specifically define a shuffle phase to manage the distribution of map results to reducers spread over different nodes, thus providing a fine-grained control over communication costs

• Reduce inputs are the map outputs: a list of key-value documents

• Each execution of the reduce function returns one key-value document

• The most simple SQL-equivalent operations performed by means of reducers are «group by» aggregations, but reducers are very flexible functions that can execute even complex operations

• Re-reduce: reduce functions can be called on their own results (in some implementations)
MapReduce example (1)

- **Map - List of exams and corresponding mark**
  
  Function(doc){
    emit(doc.exam, doc.mark);
  }

- **Reduce - Compute the average mark for each exam**
  
  Function(key, values){
    S = sum(values);
    N = len(values);
    AVG = S/N;
    return AVG;
  }

The reduce function receives:
- **key**=Bioinformatics, **values**=[30]
- ...  
- **key**=Database, **values**=[29,26,25]
- ...

### Map

<table>
<thead>
<tr>
<th>doc.id</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Bioinformatics</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>Computer architectures</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>Computer architectures</td>
<td>27</td>
</tr>
<tr>
<td>1</td>
<td>Database</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>Database</td>
<td>26</td>
</tr>
<tr>
<td>8</td>
<td>Database</td>
<td>25</td>
</tr>
<tr>
<td>5</td>
<td>Software engineering</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>Software engineering</td>
<td>18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bioinformatics</td>
<td>30</td>
</tr>
<tr>
<td>Computer architectures</td>
<td>25.5</td>
</tr>
<tr>
<td>Database</td>
<td>26.67</td>
</tr>
<tr>
<td>Software engineering</td>
<td>19.5</td>
</tr>
</tbody>
</table>
MapReduce example (2)

• Map - List of exams and corresponding mark
  
  Function(doc){
    emit(
      [doc.exam, doc.AYear],
      doc.mark
    );
  }

• Reduce - Compute the average mark for each exam and academic year
  
  Function(key, values){
    S = sum(values);
    N = len(values);
    AVG = S/N;
    return AVG;
  }
  
  Reduce is the same as before

The reduce function receives:
• key=[Database, 2014-15], values=[26,25]
• key=[Database, 2015-16], values=[29]
• …
# Rereduce in CouchDB

Average mark the for each exam (group level=1)

<table>
<thead>
<tr>
<th>Doc.i.d</th>
<th>Key</th>
<th>Value</th>
<th>Reduce Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Bioinformatics, 2015-16</td>
<td>30</td>
<td>[Bioinformatics, 2015-16]</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>Computer architectures, 2015-16</td>
<td>27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Database, 2015-16</td>
<td>29</td>
<td>[Database, 2015-16]</td>
<td>29</td>
</tr>
<tr>
<td>7</td>
<td>Software engineering, 2015-16</td>
<td>18</td>
<td>[Software engineering, 2015-16]</td>
<td>18</td>
</tr>
</tbody>
</table>
MapReduce example (3a)

Average CFU-weighted mark for each student

- **Map**
  - The reduce function receives:
    - `key=`
    - `values=`
    - `...`
    - `key=`
    - `values=`

- **Reduce**
  - The reduce function results:
    - `key=`
    - `values=`
    - `...`
    - `key=`
    - `values=`

<table>
<thead>
<tr>
<th>docId</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table Example**

<table>
<thead>
<tr>
<th>id: 1</th>
<th>DOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exam: Database</td>
<td></td>
</tr>
<tr>
<td>Student: s123456</td>
<td></td>
</tr>
<tr>
<td>AYear: 2015-16</td>
<td></td>
</tr>
<tr>
<td>Date: 31-01-2016</td>
<td></td>
</tr>
<tr>
<td>Mark=29</td>
<td></td>
</tr>
<tr>
<td>CFU=8</td>
<td></td>
</tr>
</tbody>
</table>
MapReduce example (3a)

**Map -** Ordered list of students, with mark and CFU for each student

Function(doc) {
    key = doc.student
    value = [doc.mark, doc.CFU]
    emit(key, value);
}

**Reduce -** Average CFU-weighted mark for each student

Function(key, values){
    S = sum([ X*Y for X,Y in values ]);  
    N = sum([ Y for X,Y in values ]);  
    AVG = S/N;
    return AVG;
}

<table>
<thead>
<tr>
<th>doc.i d</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S123456</td>
<td>[29, 8]</td>
</tr>
<tr>
<td>2</td>
<td>S123456</td>
<td>[24, 10]</td>
</tr>
<tr>
<td>5</td>
<td>S123456</td>
<td>[21, 8]</td>
</tr>
<tr>
<td>6</td>
<td>S123456</td>
<td>[30, 6]</td>
</tr>
<tr>
<td>3</td>
<td>S654321</td>
<td>[27, 10]</td>
</tr>
<tr>
<td>4</td>
<td>S654321</td>
<td>[26, 8]</td>
</tr>
<tr>
<td>7</td>
<td>S654321</td>
<td>[18, 8]</td>
</tr>
<tr>
<td>8</td>
<td>s987654</td>
<td>[25, 8]</td>
</tr>
</tbody>
</table>

The reduce function receives:
- key=S123456, values=[(29, 8), (24,10), (21,8)...]
- ...
- key=s987654, values=[(25,8)]
MapReduce example (3b)

- Compute the number of exams for each student
- Technological view of data distribution among different nodes

<table>
<thead>
<tr>
<th>Doc.i d</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S123456</td>
<td>[29, 1]</td>
</tr>
<tr>
<td>2</td>
<td>S123456</td>
<td>[24, 1]</td>
</tr>
<tr>
<td>5</td>
<td>S123456</td>
<td>[21, 1]</td>
</tr>
<tr>
<td>6</td>
<td>S123456</td>
<td>[30, 1]</td>
</tr>
<tr>
<td>3</td>
<td>S654321</td>
<td>[27, 1]</td>
</tr>
<tr>
<td>4</td>
<td>S654321</td>
<td>[26, 1]</td>
</tr>
<tr>
<td>7</td>
<td>S654321</td>
<td>[18, 1]</td>
</tr>
<tr>
<td>8</td>
<td>987654</td>
<td>[25, 1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>S123456</td>
<td>3</td>
</tr>
<tr>
<td>S123456</td>
<td>1</td>
</tr>
<tr>
<td>S654321</td>
<td>3</td>
</tr>
<tr>
<td>987654</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>S123456</td>
<td>4</td>
</tr>
<tr>
<td>S654321</td>
<td>3</td>
</tr>
<tr>
<td>987654</td>
<td>1</td>
</tr>
</tbody>
</table>
Map Reduce in MongoDB
Aggregation operations in MongoDB

• Aggregation operations
  o **group** values from multiple documents together
  o can perform a variety of **operations** on the grouped data
  o return an **aggregated result**

• MongoDB provides three ways to perform aggregation:
  o the **aggregation pipeline**
    ▪ exploits native operations within MongoDB,
    ▪ is the preferred method for data aggregation in MongoDB
  o the **map-reduce function**
    ▪ since MongoDB 5.0 the map-reduce operation is **deprecated**
  o single-purpose aggregation **methods**

https://docs.mongodb.com/manual/aggregation/
Single-Purpose Aggregation Operations

• Commands
  o `db.collection.estimatedDocumentCount()`,
  o `db.collection.count()`,
  o `db.collection.distinct()`

• Features
  o aggregate documents from a **single collection**
  o **simple** access to common aggregation processes
  o less **flexible** and **powerful** than aggregation pipeline and map-reduce

```
Collection
db.orders.distinct("cust_id")
```

```
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
{
  cust_id: "A123",
  amount: 300,
  status: "D"
}
```

distinct [ "A123", "B212" ]

orders
Comparison of aggregation operations

• Map Reduce
  o Besides grouping operations, can perform complex aggregation tasks
    ▪ Custom map, reduce and finalize JavaScript functions offer flexibility
  o Incremental aggregation on continuously growing datasets

• Aggregation pipeline
  o Performance and usability
  o Virtually infinite pipeline of transformations
  o Map-reduce operations can be rewritten using aggregation pipeline operators, e.g., $group, $merge
  o For map-reduce operations that require custom functionality, MongoDB provides the $accumulator and $function aggregation operators starting in version 4.4. Use these operators to define custom aggregation expressions in JavaScript.

• For most aggregation operations, the Aggregation Pipeline provides better performance and more coherent interface
custom JavaScript functions

db.collection.mapReduce( { 
  map: function() { emit( this.cust_id, this.amount ); },
  reduce: function(key, values) { return Array.sum( values ); },
  query: { status: "A" },
  output: "order_totals"
})
1. MongoDB applies the map phase to each input document (i.e. the documents in the collection that match the query condition)

2. The map function emits key-value pairs

3. For those keys that have multiple values, MongoDB applies the reduce phase, which collects and condenses the aggregated data

4. MongoDB then stores the results in a collection
MongoDB: Map-Reduce

- **Map**
  - Requires `emit(key, value)` to map each value with a key
  - It refers to the current document as `this`

- **Reduce**
  - Groups all document with the same key
  - These functions must be associative and commutative and must return an object of the same type of value emitted by **Map** (multiple calls to reduce function on the same key)

- **Out**
  - Specifies where to output the map-reduce query results
    - Either a collection
    - Or an inline result

```javascript
db.orders.mapReduce(
    function() { emit( this.cust_id, this.amount ); },
    function(key, values) { return Array.sum( values ) },
    { query: { status: "A" },
      out: "order_totals"
  )
```
MongoDB: Map-Reduce

- **Finalize** (optional)
  - Follows the *reduce* method and modifies the output

- **Query** (optional)
  - Specifies the selection criteria for selecting the input documents to the *map* function

- **Sort** (optional)
  - Specifies the sort criteria for the input documents
  - Useful for optimization, e.g., specify the sort key to be the same as the emit key so that there are fewer reduce operations.
  - The sort key must be in an existing index

- **Limit** (optional)
  - Specifies the maximum number of input documents
E.g.,

```javascript
db.orders.mapReduce(
    function() {
        emit(this.cust_id, this.amount);
    },
    function(key, values) {
        return Array.sum(values)
    },
    {
        query: { status: "A" },
        out: "order_totals"
    }
)
```

MongoDB: Map-Reduce example
MongoDB: Map-Reduce

db.orders.mapReduce(
  function() {emit(this.cust_id, this.amount);},
  function(key, values) {return Array.sum(values)};
  {query: {status: "A"},
   out: "order_totals"}
)

- Only for orders with status: "A"
- for each cust_id,
  - sum all the orders values
  - into the "order_totals" collection
MongoDB: Map-Reduce features

• All map-reduce functions in MongoDB are JavaScript and run within the mongod process

• Map-reduce operations
  o take the documents of a single collection as the input
  o perform any arbitrary sorting and limiting before beginning the map stage
  o return the results as a document or into a collection

• When processing a document, the map function can create more than one key and value mapping or no mapping at all

• If you write map-reduce output to a collection,
  o you can perform subsequent map-reduce operations on the same input collection that merge replace, merge, or reduce new results with previous results (incremental Map Reduce)

• When returning the results of a map-reduce operation inline,
  o the result documents must be within the BSON Document Size limit, currently 16 megabytes
Hadoop
The de facto standard Big Data Platform
Hadoop, a Big-Data-everything platform

- **2003**: Google File System
- **2004**: MapReduce by Google (Jeff Dean)
- **2005**: Hadoop, funded by Yahoo, to power a search engine project
- **2006**: Hadoop migrated to Apache Software Foundation
- **2006**: Google BigTable
- **2008**: Hadoop wins the Terabyte Sort Benchmark, sorted 1 Terabyte of data in 209 seconds, previous record was 297 seconds
- **2009**: additional components and sub-projects started to be added to the Hadoop platform
Hadoop, platform overview
Hadoop, platform overview

The Apache Hadoop Stack

- Data Exchange
- Sqoop
- Pig
- Hbase
- Mahout
- Oozie
- YARN/Map Reduce V2
- ZooKeeper
- Log Control

Hadoop User Experience (HUE)

- SQL
- Hive
- Mahout
- Workflow
- Columnar data store

Hadoop Distributed File System
Hadoop, platform overview
Hadoop, platform overview

The Apache Hadoop Stack

Hadoop User Experience (HUE)

- Data Exchange
- Sqoop
- Flume
- Log Control
- ZooKeeper Coordination
- Pig Scripting
- Hive SQL
- Mahout ML
- Oozie Workflow
- Hbase Columnar data store
- YARN/Map Reduce V2
- Hadoop Distributed File System

A Hadoop System

- BI Tools
  - SQL
- Analytic Application
  - 3rd Party SQL on Hadoop
  - SQL
- PIG Latin Scripts
  - Executes on MR, Tez & Spark
- Java, Python, Scala, R Applications
- MapReduce
  - Tez
  - Spark
  - Storm
- HBase
  - APIs to HBase
  - APIs to HDFS
- webHDFS
  - (An HTTP interface to HDFS has REST APIs)

Copyright © Intelligent Business Strategies 1992-2016
Apache Hadoop, core components

• **Hadoop Common**: The common utilities that support the other Hadoop modules.

• **Hadoop Distributed File System (HDFS™)**: A distributed file system that provides high-throughput access to application data.

• **Hadoop YARN**: A framework for job scheduling and cluster resource management.

• **Hadoop MapReduce**: A YARN-based system for parallel processing of large data sets.
**Hadoop-related projects at Apache**

- **Ambari™**: A web-based tool for provisioning, managing, and monitoring Apache Hadoop clusters which includes support for Hadoop HDFS, Hadoop MapReduce, Hive, HCatalog, HBase, ZooKeeper, Oozie, Pig and Sqoop. Ambari also provides a dashboard for viewing cluster health such as heatmaps and ability to view MapReduce, Pig and Hive applications visually along with features to diagnose their performance characteristics in a user-friendly manner.

- **Avro™**: A data serialization system.

- **Cassandra™**: A scalable multi-master database with no single points of failure.

- **Chukwa™**: A data collection system for managing large distributed systems.

- **HBase™**: A scalable, distributed database that supports structured data storage for large tables.

- **Hive™**: A data warehouse infrastructure that provides data summarization and ad hoc querying.

- **Mahout™**: A Scalable machine learning and data mining library.

- **Pig™**: A high-level data-flow language and execution framework for parallel computation.

- **Spark™**: A fast and general compute engine for Hadoop data. Spark provides a simple and expressive programming model that supports a wide range of applications, including ETL, machine learning, stream processing, and graph computation.

- **Tez™**: A generalized data-flow programming framework, built on Hadoop YARN, which provides a powerful and flexible engine to execute an arbitrary DAG of tasks to process data for both batch and interactive use-cases. Tez is being adopted by Hive™, Pig™ and other frameworks in the Hadoop ecosystem, and also by other commercial software (e.g. ETL tools), to replace Hadoop™ MapReduce as the underlying execution engine.

- **ZooKeeper™**: A high-performance coordination service for distributed applications.

---

**DATA MANAGEMENT AND VISUALIZATION**

34
Apache Spark

- **A fast and general engine for large-scale data processing**
  - Speed
    - Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.
    - Apache Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing.
  - Ease of Use
    - Write applications quickly in Java, Scala, Python, R.
    - Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it interactively from the Scala, Python and R shells.
  - Generality
    - Combine SQL, streaming, and complex analytics.
    - Spark powers a stack of libraries including SQL and DataFrames, MLlib for machine learning, GraphX, and Spark Streaming. You can combine these libraries seamlessly in the same application.
  - Runs Everywhere
    - Spark runs on Hadoop, Mesos, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.
Hadoop - why

• **Storage**
  o distributed,
  o fault-tolerant,
  o heterogenous,
  o Huge-data storage engine.

• **Processing**
  o Flexible (multi-purpose),
  o parallel and scalable,
  o high-level programming (Java, Python, Scala, R),
  o batch and real-time, historical and streaming data processing,
  o complex modeling and basic KPI analytics.

• **High availability**
  o Handle failures of nodes by design.

• **High scalability**
  o Grow by adding low-cost nodes, not by replacement with higher-powered computers.

• **Low cost.**
  o Lots of commodity-hardware nodes instead of expensive super-power computers.