MapReduce

a **scalable** distributed programming model to **process** Big Data

MapReduce

- Published in **2004** by **Google**
  - J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004
  - 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**
  - 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 (**doc**ument) → 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:

```
function(doc) {
        emit(key1, value1); // key1 = fk1(doc); value1 = fv1(doc)
        emit(key2, value2); // key2 = fk2(doc); value2 = fv2(doc)
}
```

function(doc) {

emit(key_1, $value_1$); // $key_1 = f_{k1}(doc)$; $value_1 = f_{v1}(doc)$

emit(key_2, $value_2$); // $key_2 = f_{k2}(doc)$; $value_2 = f_{v2}(doc)$

}

- 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

Id: 1
Exam: Database
Student: s123456
AYear: 2015-16
Date: 31-01-2016
Mark=29
CFU=8

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

Map example (1)

- List of exams and corresponding marks

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

Key Value

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: 1
Exam: Database
Student: s123456
AYear: 2015-16
Date: 31-01-2016
Mark=29
CFU=8

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

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

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

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

Result:

doc.id	Key	Value
6	Bioinformatics	30
2	Computer architectures	24
3	Computer architectures	27
1	Database	29
4	Database	26
8	Database	25
5	Software engineering	21
7	Software engineering	18

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);
}
```

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: 1
Exam: Database
Student: s123456
AYear: 2015-16
Date: 31-01-2016
Mark=29
CFU=8

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

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

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

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

Result:

doc.id	Key	Value
6	[Bioinformatics, 2015-16]	30
2	[Computer architectures, 2015-16]	24
3	[Computer architectures, 2015-16]	27
4	[Database, 2014-15]	26
8	[Database, 2014-15]	25
1	[Database, 2015-16]	29
5	[Software engineering, 2014-15]	21
7	[Software engineering, 2015-16]	18

Map example (3)

- Ordered list of students, with mark and CFU for each exam

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

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: 1
Exam: Database
Student: s123456
AYear: 2015-16
Date: 31-01-2016
Mark=29
CFU=8

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

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

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

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

Result:

doc.id	Key	Value
1	S123456	[29, 8]
2	S123456	[24, 10]
5	S123456	[21, 8]
6	S123456	[30, 6]
3	S654321	[27, 10]
4	S654321	[26, 8]
7	S654321	[18, 8]
8	s987654	[25, 8]

Reduce

- Documents (key-value pairs) emitted by the map function are **sorted by key**
  - 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;
}
```

id: 1 **DOC**
Exam: Database
Student: s123456
AYear: 2015-16
Date: 31-01-2016
Mark=29
CFU=8

The reduce function receives:

- **key**=Bioinformatics, **values**=[30]
- …
- **key**=Database, **values**=[29,26,25]
- …

Map

doc.id	Key	Value
6	Bioinformatics	30
2	Computer architectures	24
3	Computer architectures	27
1	Database	29
4	Database	26
8	Database	25
5	Software engineering	21
7	Software engineering	18

Reduce

Key	Value
Bioinformatics	30
Computer architectures	25.5
Database	26.67
Software engineering	19.5

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

id: 1 **DOC**
Exam: Database
Student: s123456
AYear: 2015-16
Date: 31-01-2016
Mark=29
CFU=8

The reduce function receives:

- **key**=[Database, 2014-15], **values**=[26,25]
- **key**=[Database, 2015-16], **values**=[29]
- …

Map

doc.id	Key	Value
6	Bioinformatics, 2015-16	30
2	Computer architectures, 2015-16	24
3	Computer architectures, 2015-16	27
4	Database, 2014-15	26
8	Database, 2014-15	25
1	Database, 2015-16	29
5	Software engineering, 2014-15	21
7	Software engineering, 2015-16	18

Reduce

Key	Value
[Bioinformatics, 2015-16]	30
[Computer architectures, 2015-16]	25.5
[Database, 2014-15]	25.5
[Database, 2015-16]	29
[Software engineering, 2014-15]	21
[Software engineering, 2015-16]	18

Rereduce in CouchDB

- Average mark the for each exam (**group level=1**) – **same Reduce** as before

DB

- Id: 1
Exam: Database
Student: s123456
AYear: 2015-16
Date: 31-01-2016
Mark=29
CFU=8
- Id: 8
Exam: Database
Student: s987654
AYear: 2014-15
Date: 28-06-2015
Mark=25
CFU=8
- Id: 6
Exam: Bioinformatics
Student: s123456
AYear: 2015-16
Date: 18-09-2016
Mark=30
CFU=6
- 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: 7
Exam: Software engineering
Student: s654321
AYear: 2015-16
Date: 28-06-2016
Mark=18
CFU=8
- Id: 3
Exam: Computer architectures
Student: s654321
AYear: 2015-16
Date: 26-01-2016
Mark=27
CFU=10
- Id: 2
Exam: Computer architectures
Student: s123456
AYear: 2015-16
Date: 03-07-2015
Mark=24
CFU=10

Map

doc.id	Key	Value
6	Bioinformatics, 2015-16	30
2	Computer architectures, 2015-16	24
3	Computer architectures, 2015-16	27
4	Database, 2014-1015	26
8	Database, 2014-15	25
1	Database, 2015-16	29
5	Software engineering, 2014-15	21
7	Software engineering, 2015-16	18

Reduce

Key	Value
[Bioinformatics, 2015-16]	30
[Computer architectures, 2015-16]	25.5
[Database, 2014-15]	25.5
[Database, 2015-16]	29
[Software engineering, 2014-15]	21
[Software engineering, 2015-16]	18

Rereduce

Key	Value
Bioinformatics	30
Computer architectures	25.5
Database	27.25
Software engineering	19.5

MapReduce example (3a)

Average CFU-weighted mark for each student

id: 1 **DOC**
Exam: Database
Student: s123456
AYear: 2015-16
Date: 31-01-2016
Mark=29
CFU=8

- Map

The reduce function receives:

- **key**=
  values=
- …
- **key**=
- **values**=

- Reduce

The reduce function results:

- **key**=
  values=
- …
- **key**=
- **values**=

Map

doc.id	Key	Value

Reduce

Key	Value

MapReduce example (3a)

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

```
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;
}
```

key = S123456,
values = [(29,8), (24,10), (21,8)...]
X = 29, 24, 21, ... → mark
Y = 8, 10, 8, ... →CFU

The reduce function receives:

- **key**=S123456, **values**=[(29,8), (24,10), (21,8)...]
- ...
- **key**=s987654, **values**=[(25,8)]

Map

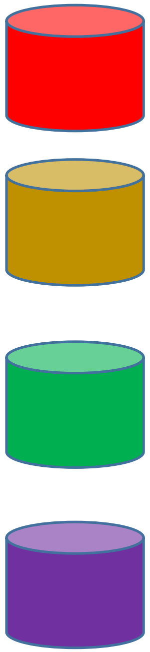
doc.id	Key	Value
1	S123456	[29, 8]
2	S123456	[24, 10]
5	S123456	[21, 8]
6	S123456	[30, 6]
3	S654321	[27, 10]
4	S654321	[26, 8]
7	S654321	[18, 8]
8	s987654	[25, 8]

Reduce

Key	Value
S123456	25.6
S654321	23.9
s987654	25

MapReduce example (3b)

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



DB

- Id: 1 Exam: Database Student: s123456 AYear: 2015-16 Date: 31-01-2016 Mark=29 CFU=8
- Id: 2 Exam: Computer architectures Student: s123456 AYear: 2015-16 Date: 03-07-2015 Mark=24 CFU=10
- 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: 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: 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

Map

doc.id	Key	Value
1	S123456	[29, 1]
2	S123456	[24, 1]
5	S123456	[21, 1]
6	S123456	[30, 1]
3	S654321	[27, 1]
4	S654321	[26, 1]
7	S654321	[18, 1]
8	s987654	[25, 1]

Reduce

Key	Value
S123456	3
S123456	1
S654321	3
s987654	1

Rereduce

Key	Value
S123456	4
S654321	3
s987654	1

Map Reduce

mongoDB

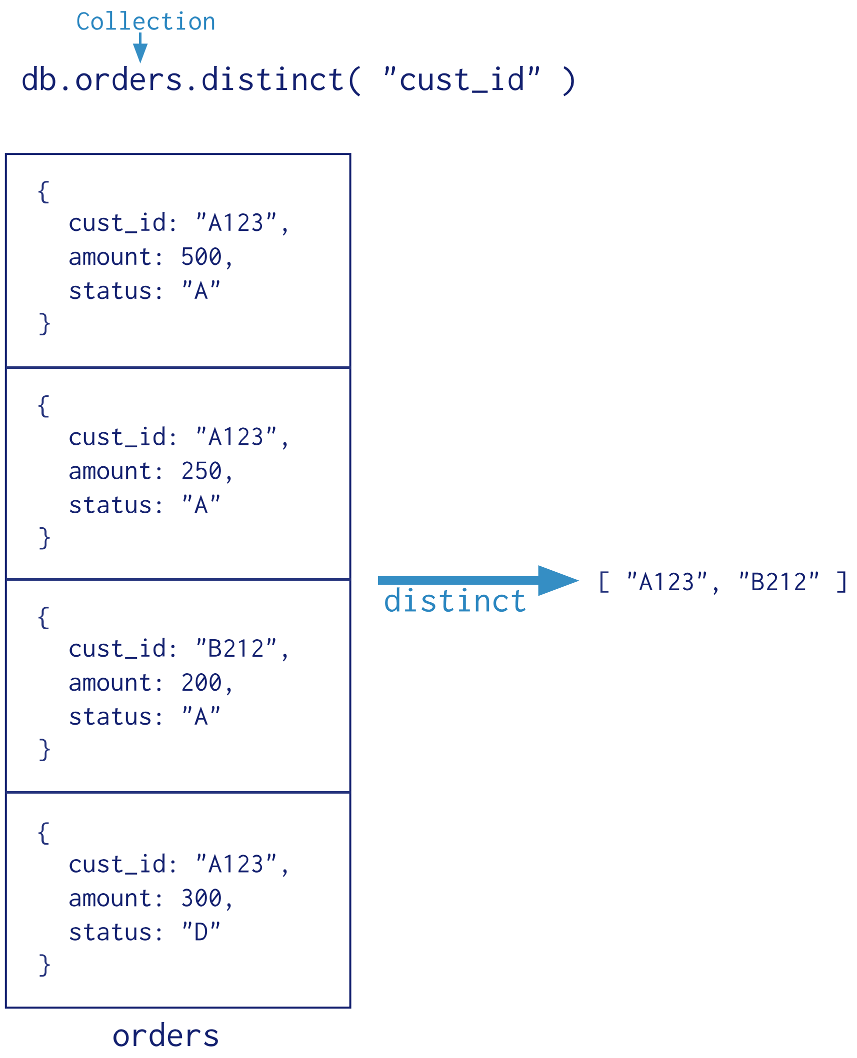
Aggregation operations in MongoDB

- Aggregation operations
  - **group** values from multiple documents together
  - can perform a variety of **operations** on the grouped data
  - return an **aggregated result**
- MongoDB provides three ways to perform aggregation:
  - the **aggregation pipeline**
    - exploits native operations within MongoDB,
    - is the preferred method for data aggregation in MongoDB
  - the **map-reduce function**
  - single-purpose aggregation **methods**

https://docs.mongodb.com/manual/aggregation/

Single-Purpose Aggregation Operations

- Commands
  - db.collection.**estimatedDocumentCount**(),
  - db.collection.**count**()
  - db.collection.**distinct**()
- Features
  - aggregate documents from a **single collection**
  - **simple** access to common aggregation processes
  - less **flexible** and **powerful** than aggregation pipeline and map-reduce

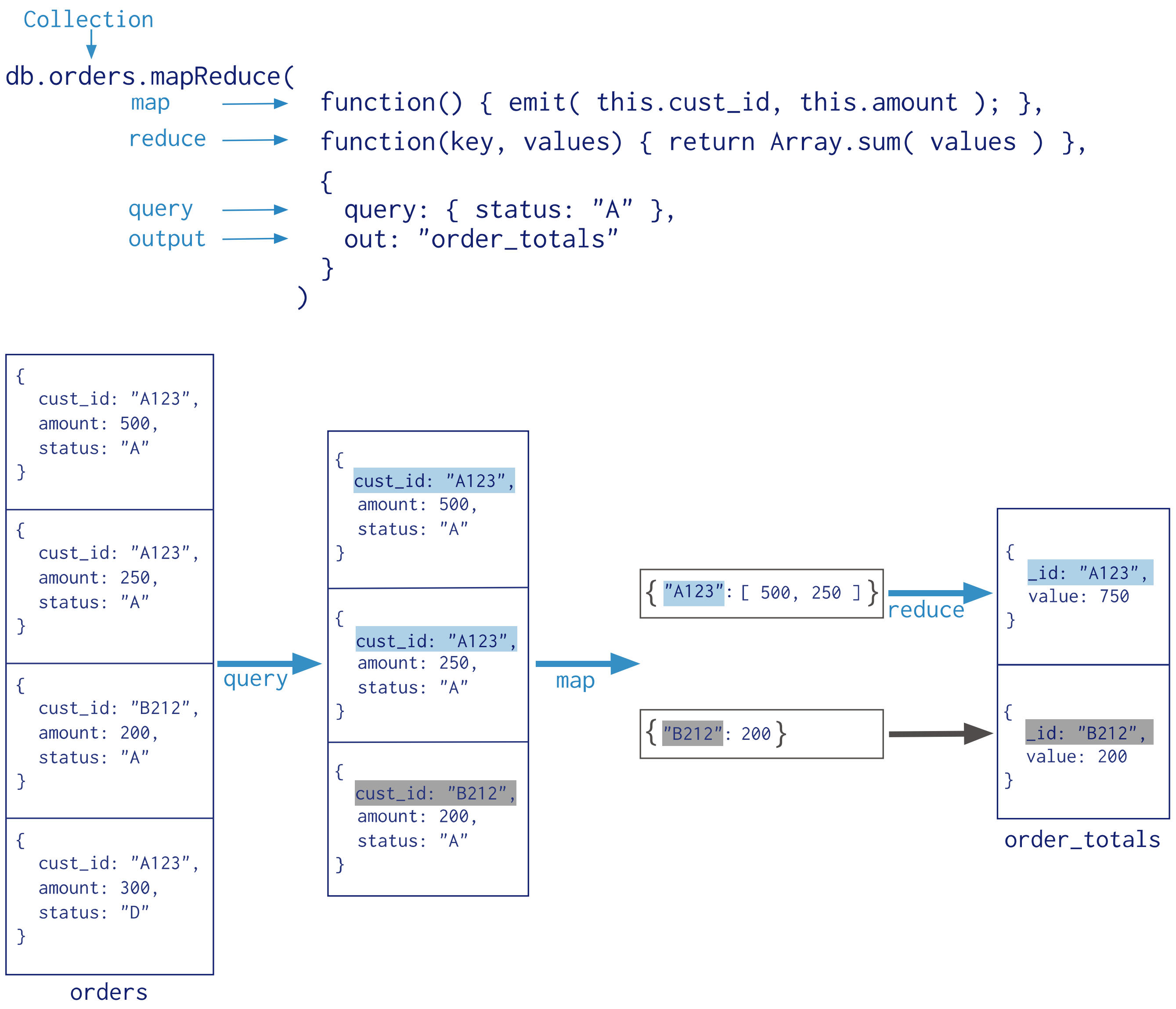


Comparison of aggregation operations

- Aggregation pipeline
  - **Performance** and usability
  - Virtually **infinite** pipeline of transformations
  - Limited to the operators and **expressions** supported
- Map Reduce
  - Besides grouping operations, can perform **complex aggregation tasks**
    - Custom map, reduce and finalize JavaScript functions offer flexibility
  - **Incremental** aggregation on continuously growing datasets
- For most aggregation operations, the Aggregation Pipeline provides better performance and more coherent interface
- However, map-reduce operations provide some flexibility that is not presently available in the aggregation pipeline

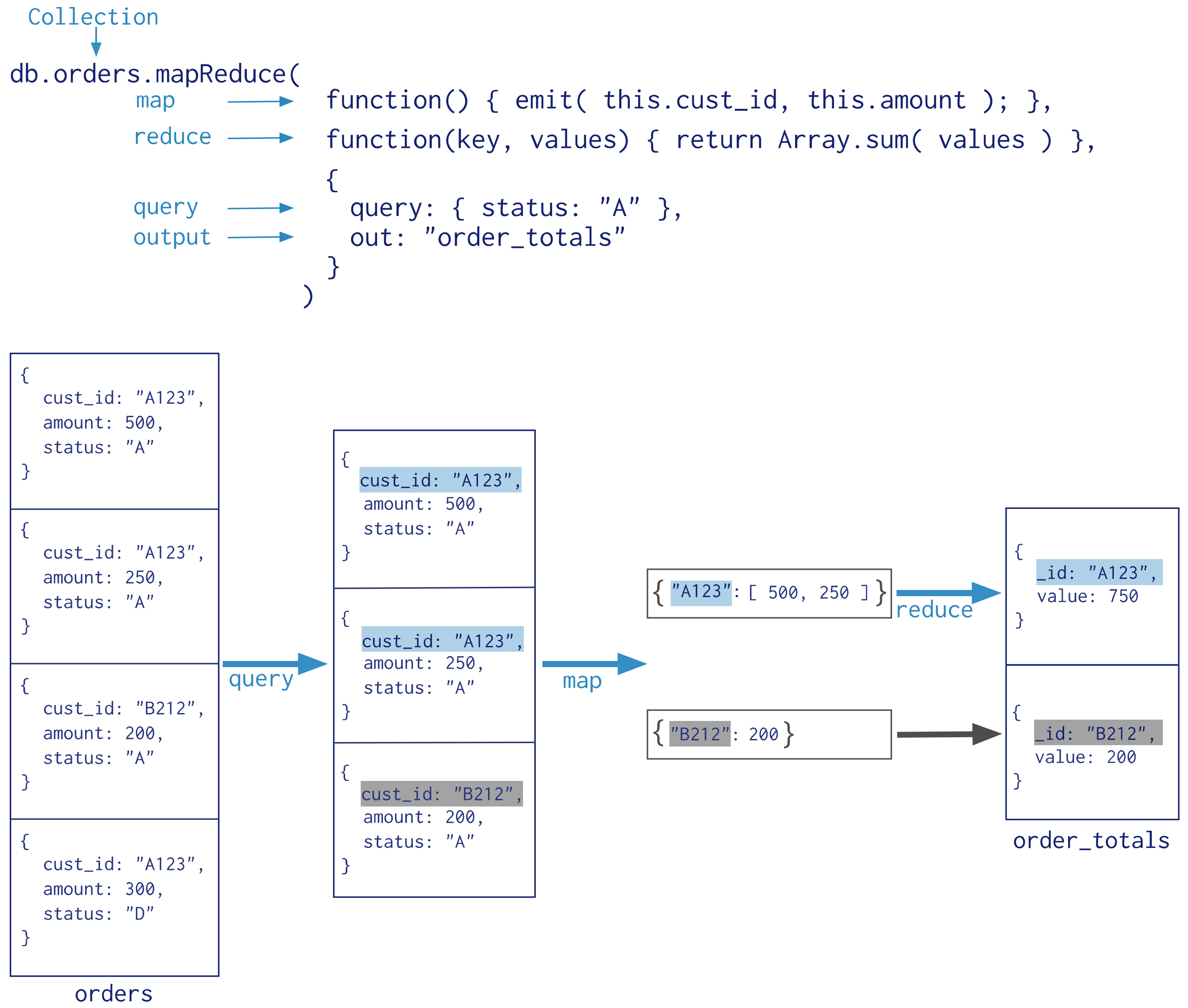
MongoDB: Map-Reduce

- custom JavaScript functions
- db.collection.**mapReduce**({
  - <map>,
  - <reduce>,
  - <finalize>,
  - <query>,
  - <out>,
  - <sort>,
  - <limit>,
  - …})



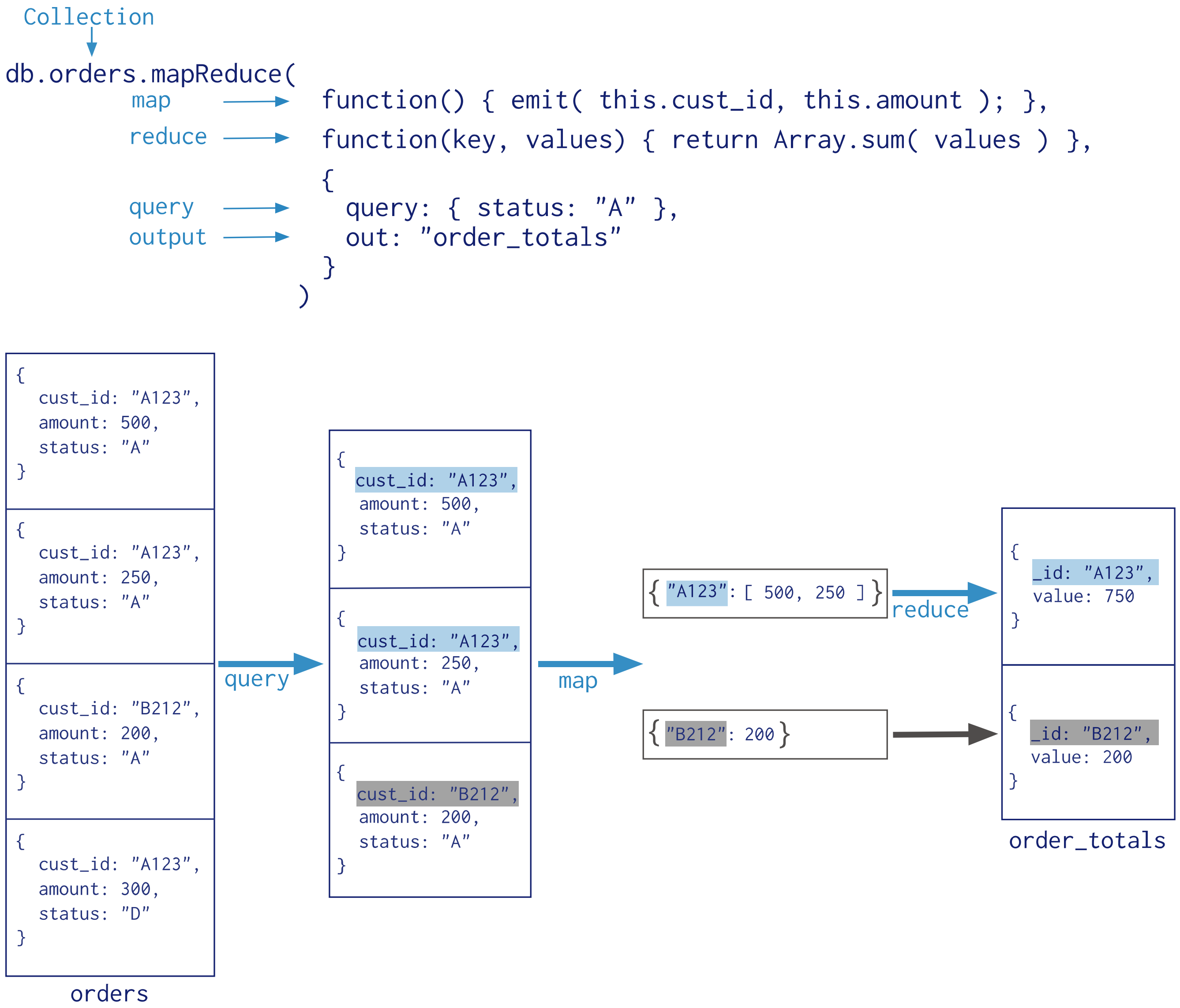
MongoDB: Map-Reduce

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



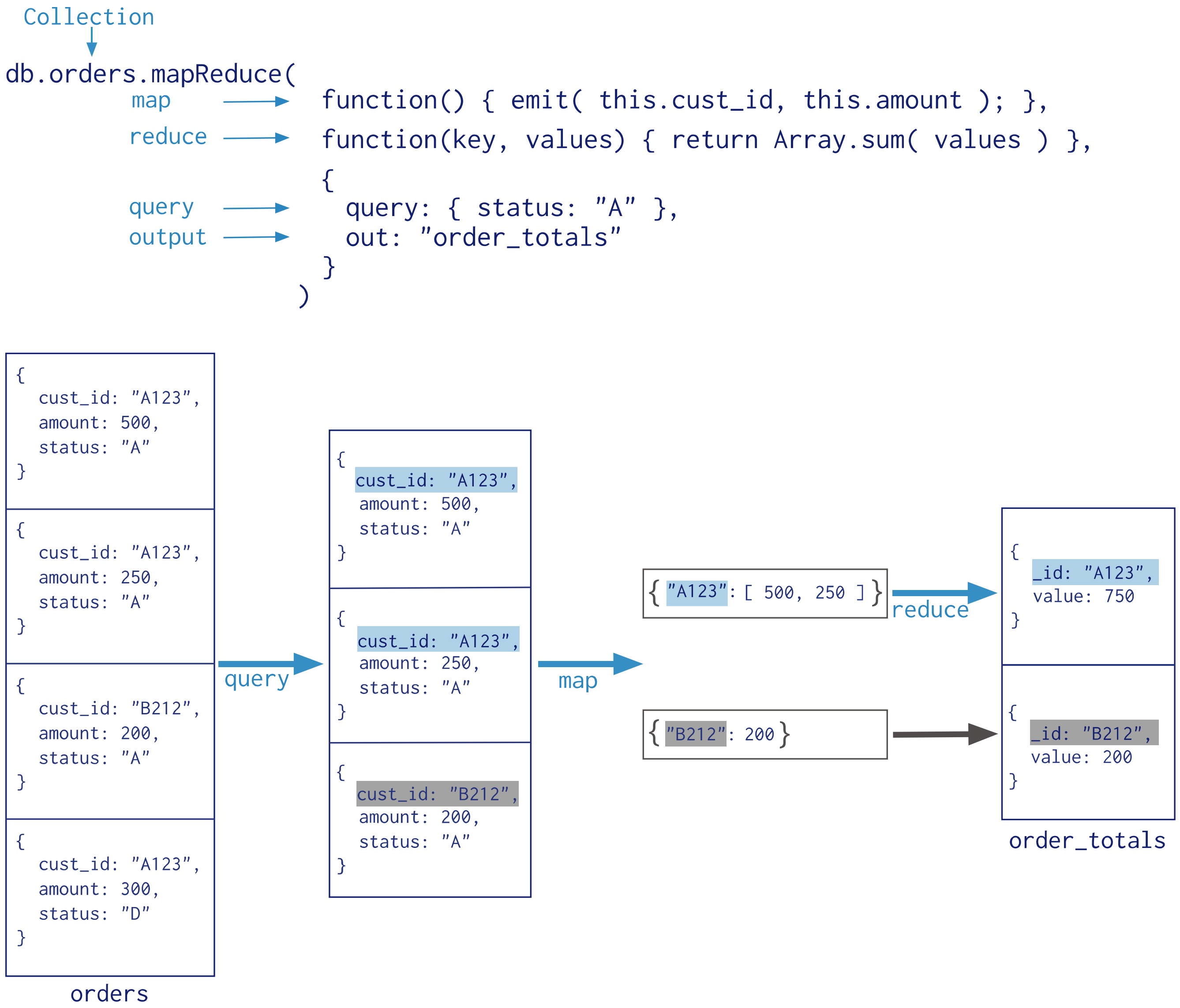
MongoDB: Map-Reduce

- **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

MongoDB: Map-Reduce example

- E.g.,

```
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

Map function

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

Reduce function

- Only for orders with status: “A”
- for each cust_id,
  - sum all the orders values
  - into the “order_totals” collection

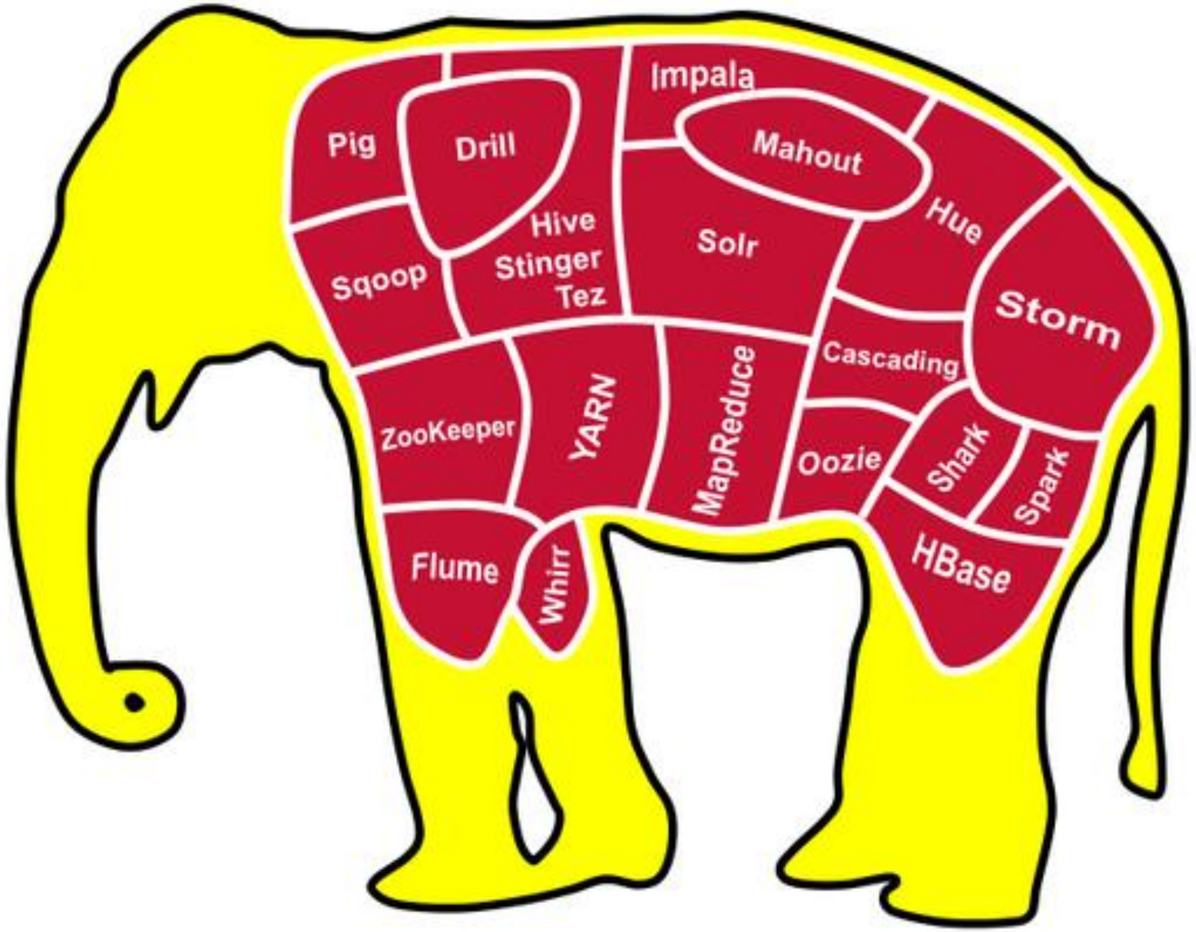
MongoDB: Map-Reduce features

- All map-reduce functions in MongoDB are **JavaScript** and run within the mongod process
- Map-reduce operations
  - take the documents of a single collection as the *input*
  - perform any arbitrary sorting and limiting before beginning the map stage
  - 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**,
  - 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**,
  - the result documents must be within the BSON Document Size limit, currently **16 megabytes**

Hadoop

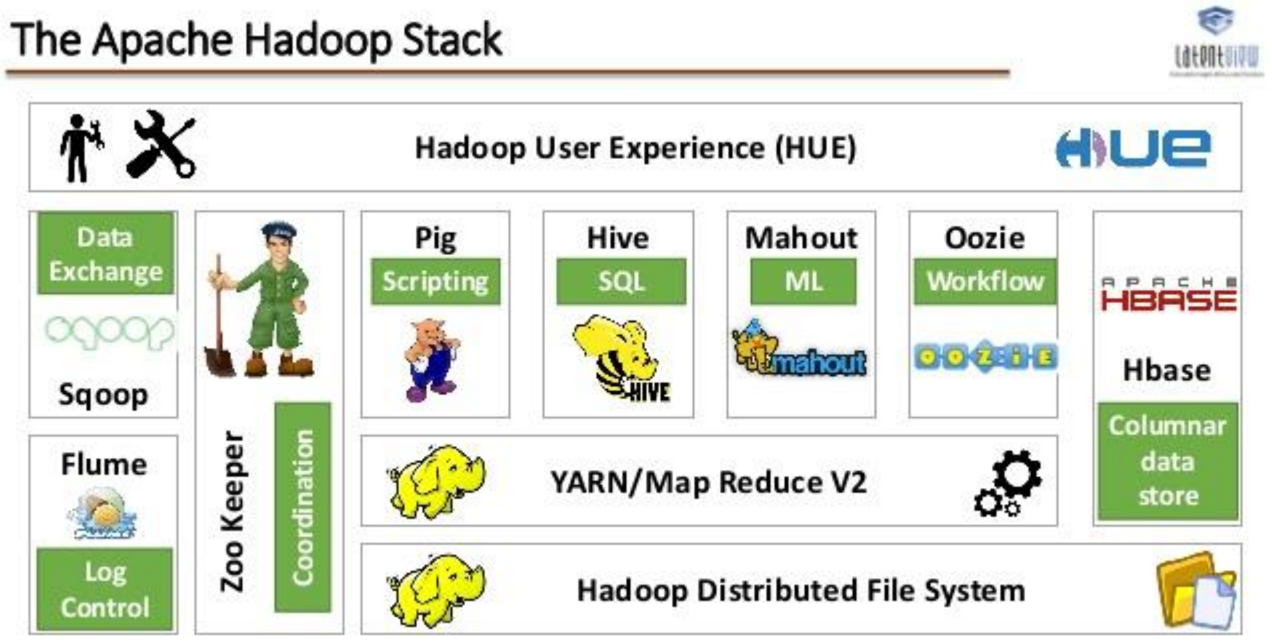
The de facto standard
Big Data platform

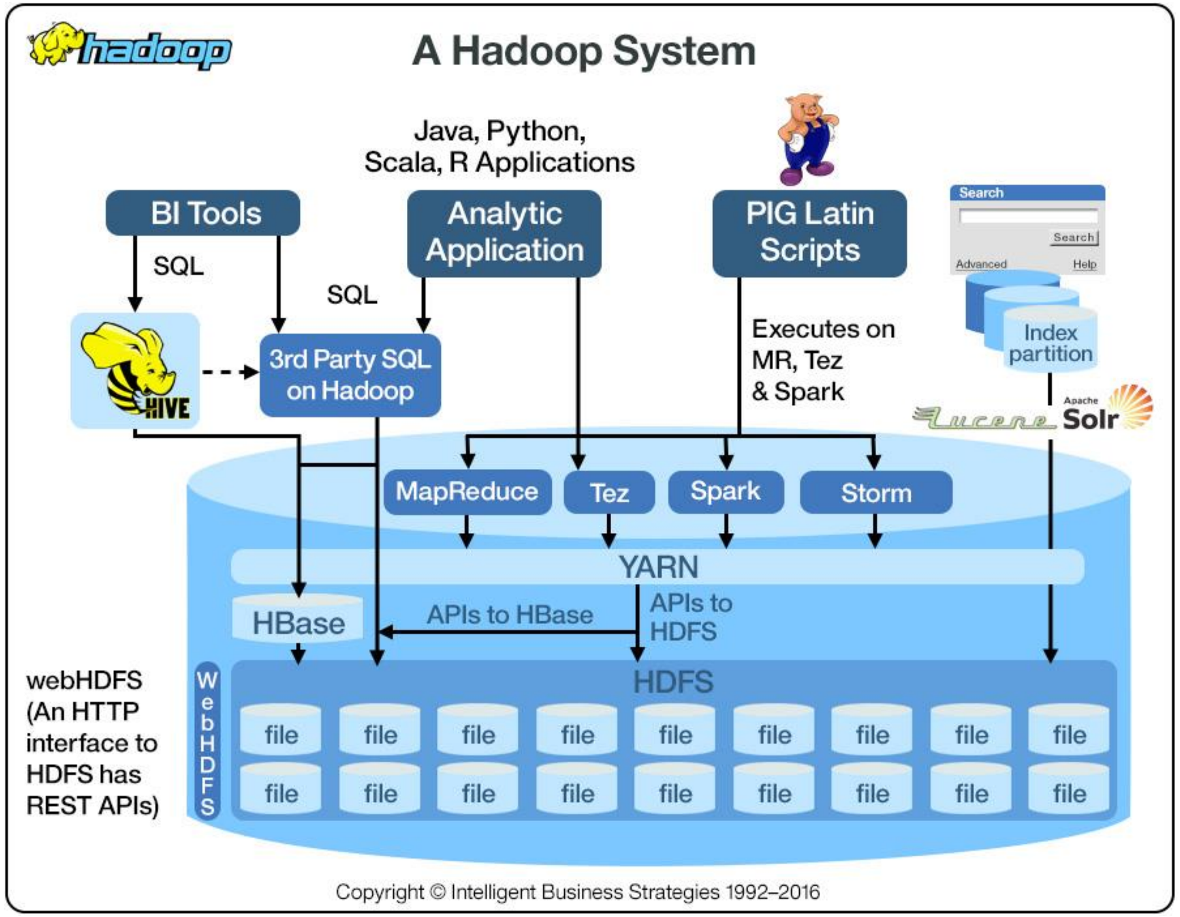
Hadoop, a Big-Data-everything platform



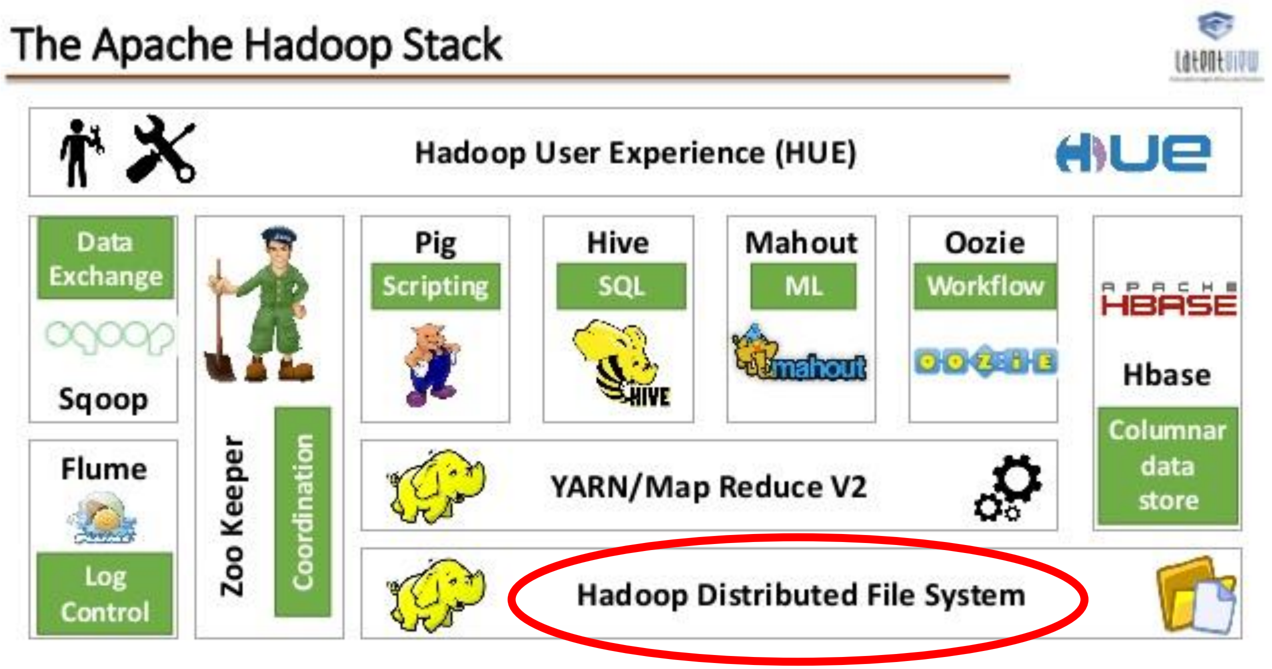
- **2003**: **Google** File System
- **2004**: MapReduce by **Google** (Jeff Dean)
- **2005**: **Hadoop**, funded by Yahoo, to power a search engine project
- **2006**: **Hadoop** migrated to Apache Software Foundation
- **2006**: **Google** BigTable
- **2008**: **Hadoop** wins the Terabyte Sort Benchmark, 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

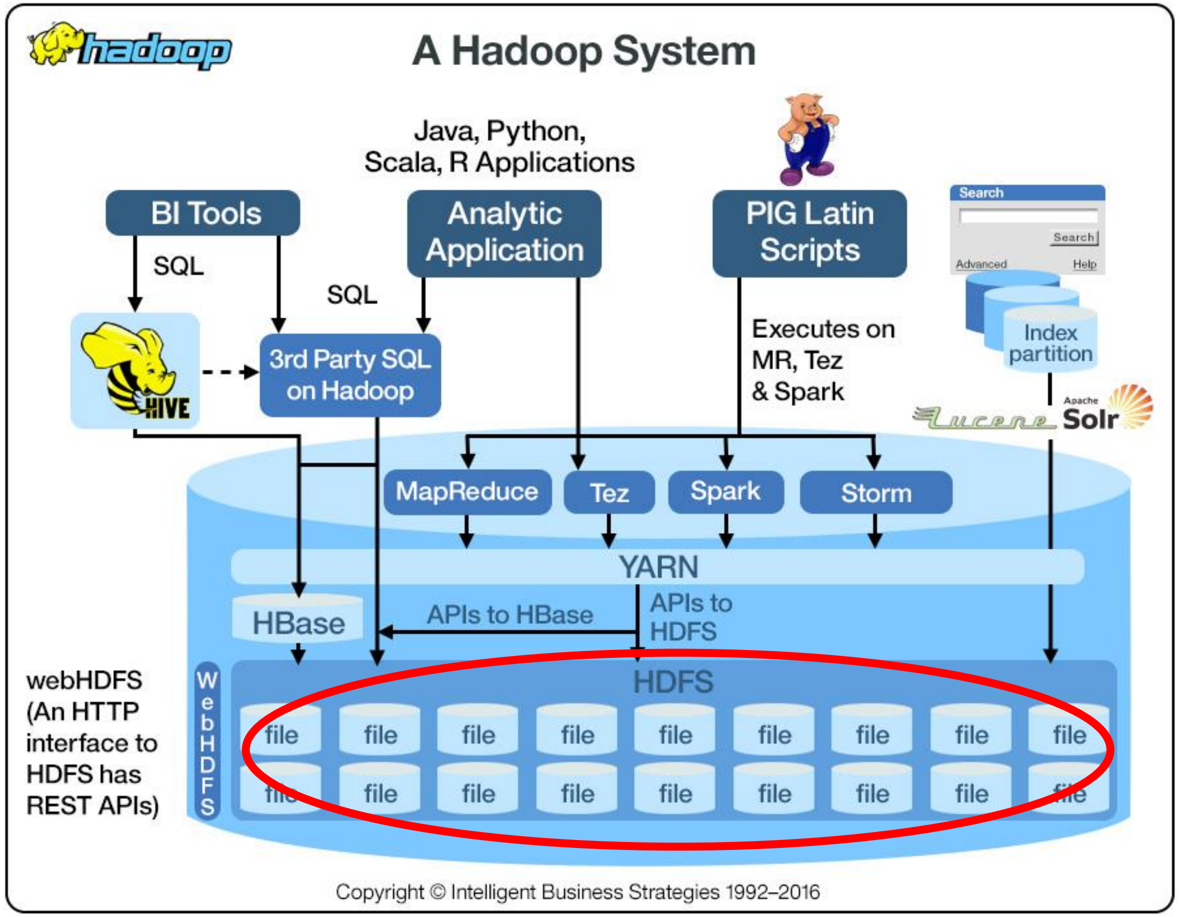
Hadoop, platform overview



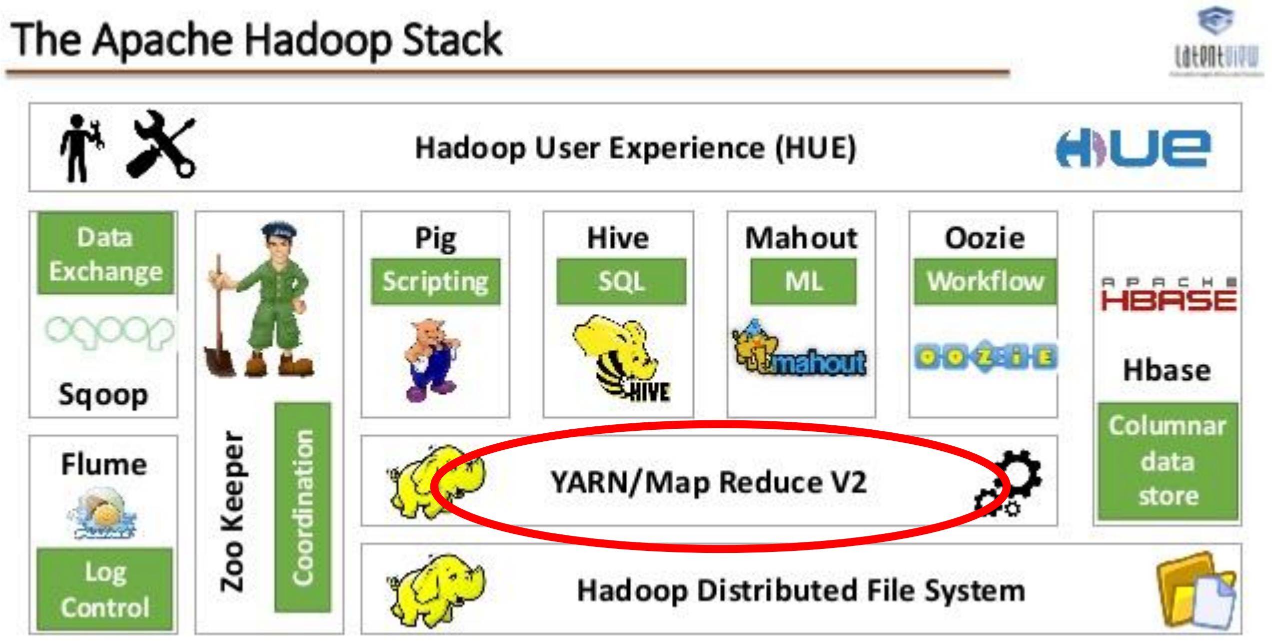


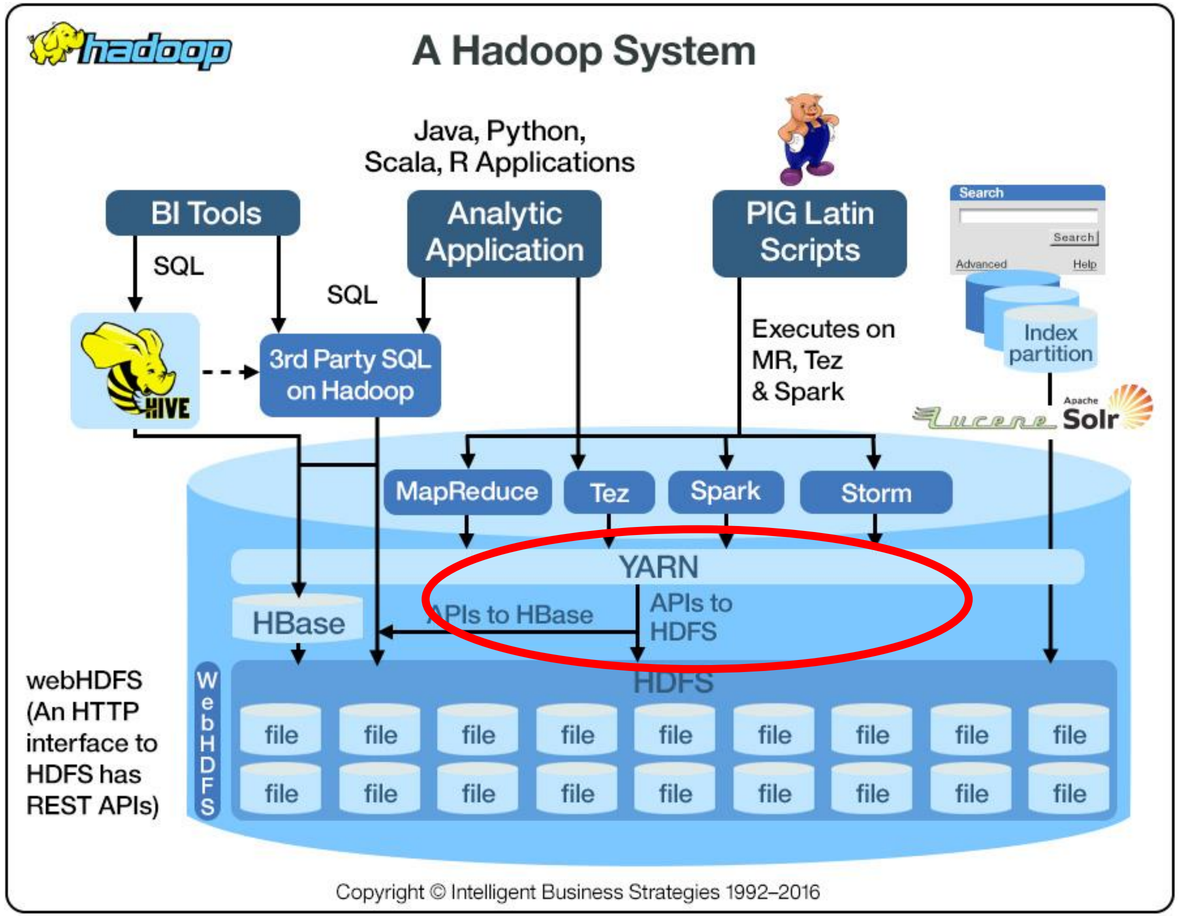
Hadoop, platform overview



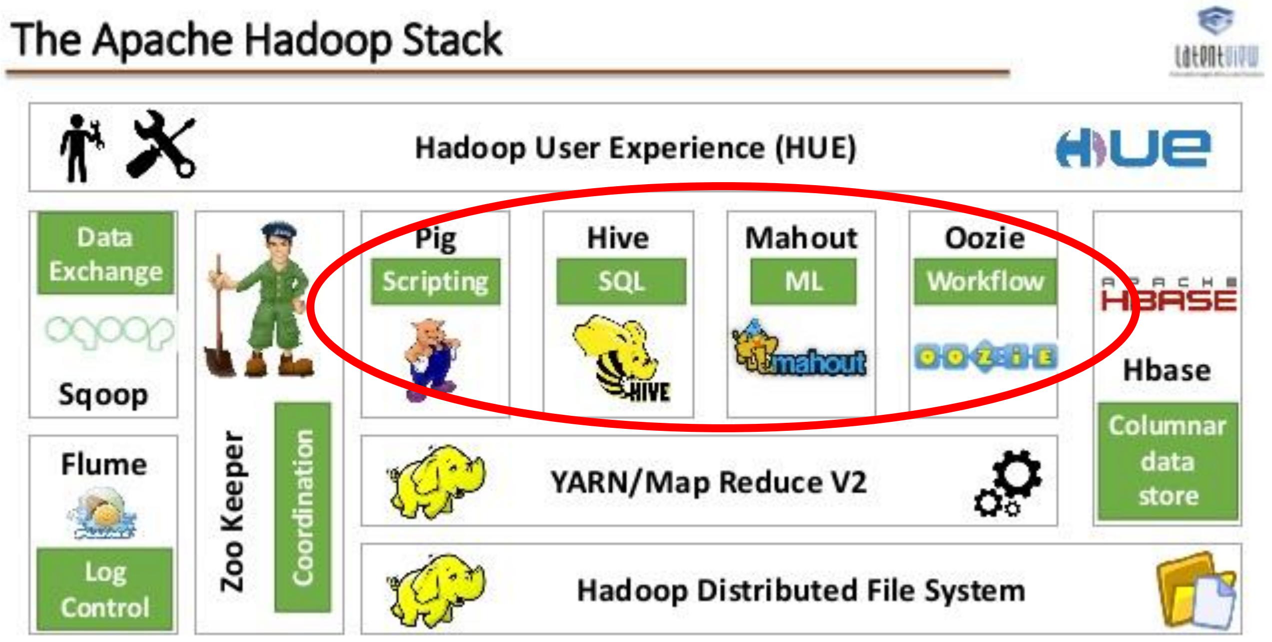


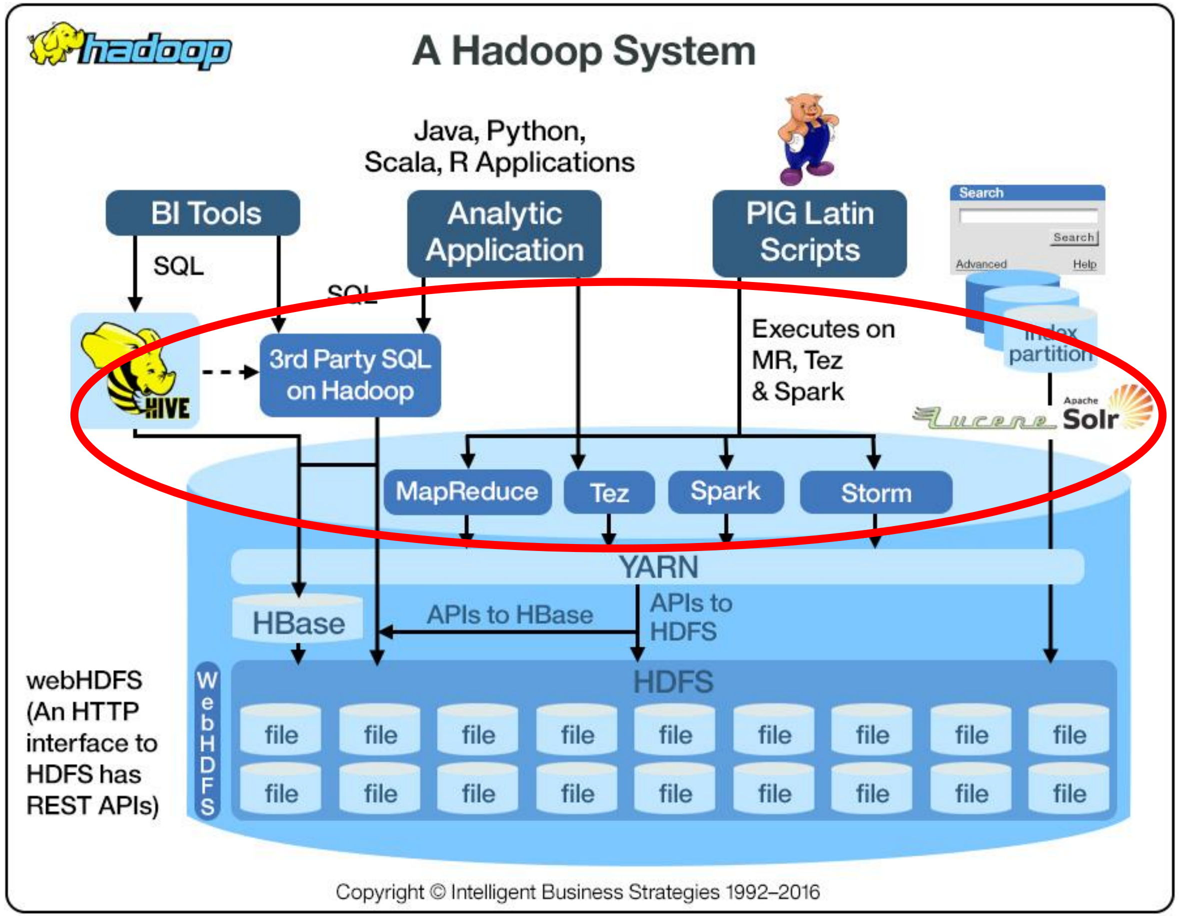
Hadoop, platform overview





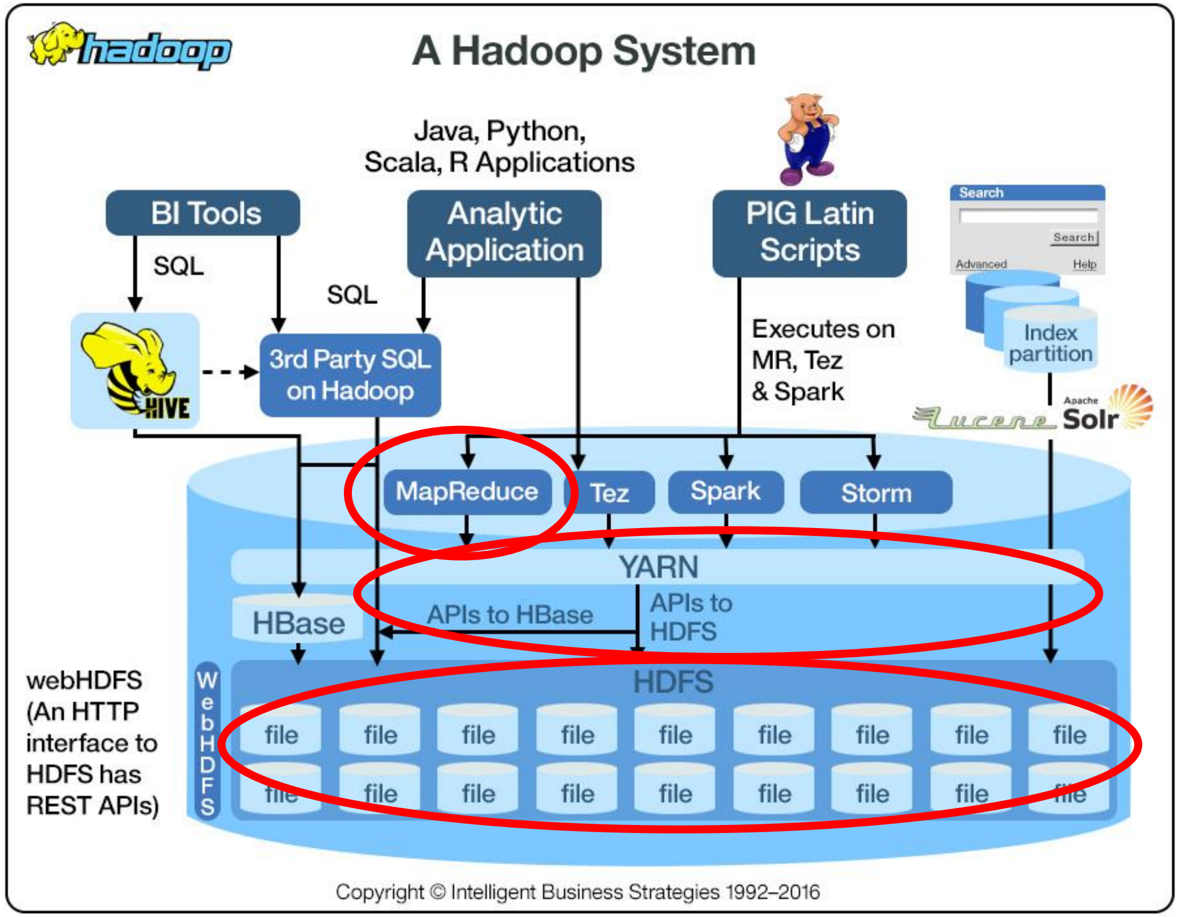
Hadoop, platform overview





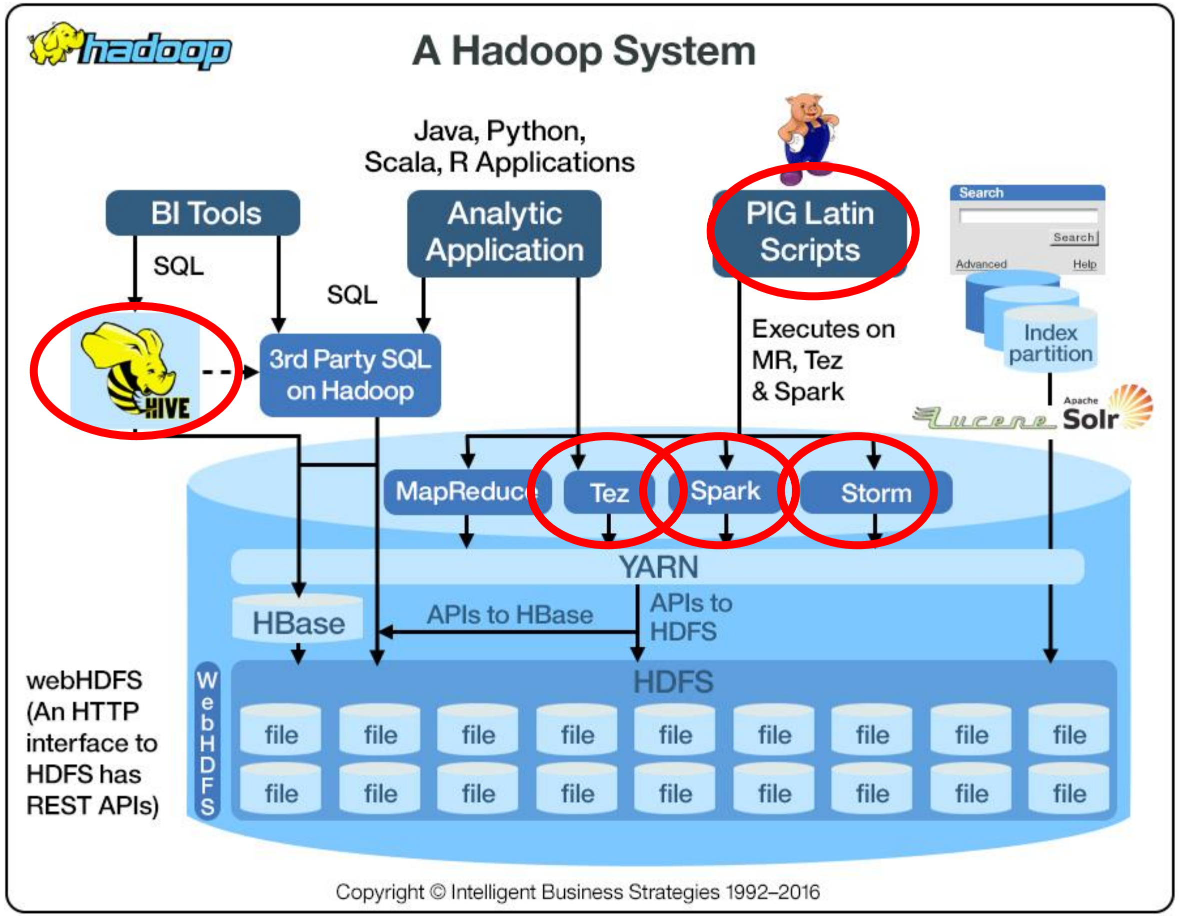
Apache Hadoop, core components

- **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.



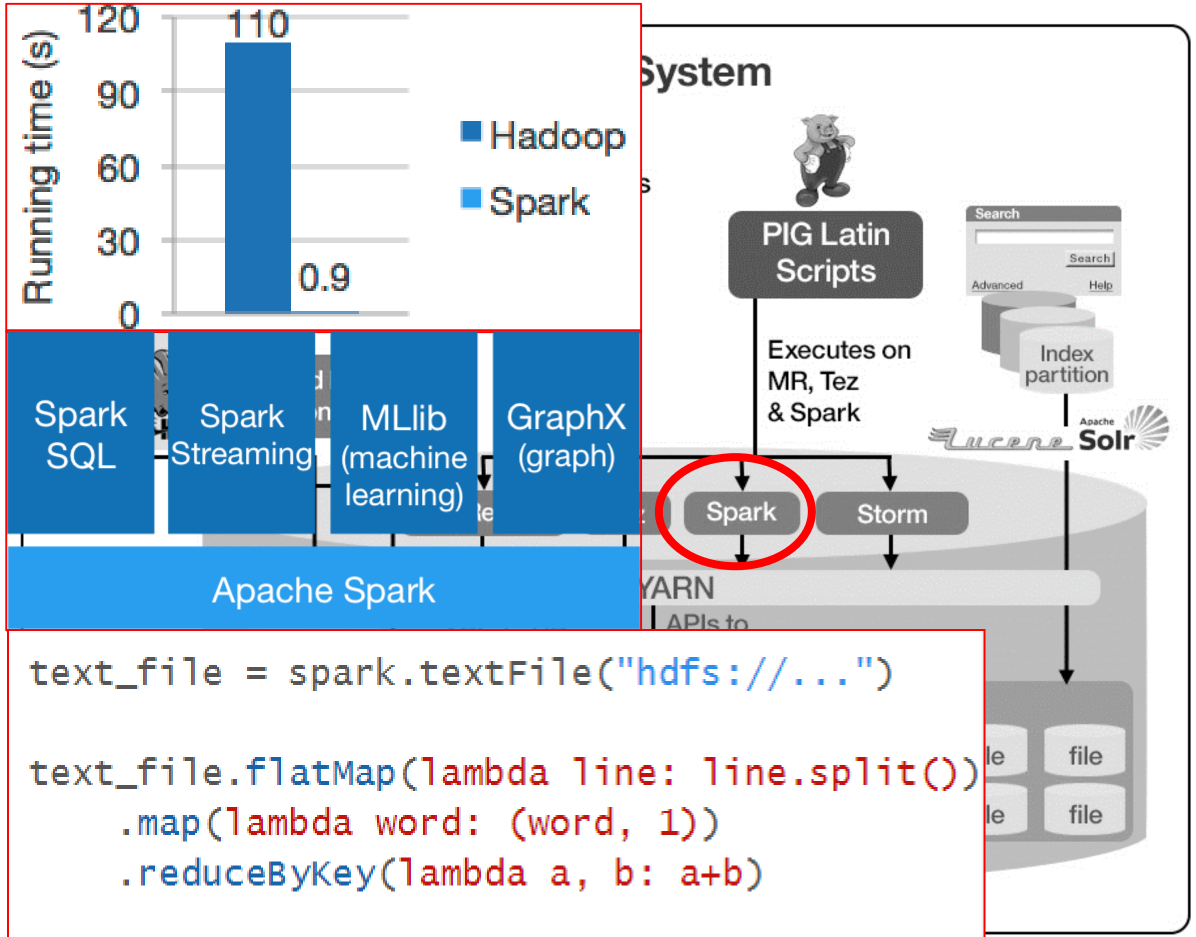
Hadoop-related projects at Apache

- 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 alongwith 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.



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.




```
text_file = spark.textFile("hdfs://...")

text_file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a+b)
```

Word count in Spark's Python API

Hadoop - why

- **Storage**
  - distributed,
  - fault-tolerant,
  - heterogenous,
  - Huge-data storage engine.
- **Processing**
  - Flexible (multi-purpose),
  - parallel and scalable,
  - high-level programming (Java, Python, Scala, R),
  - batch and real-time, historical and streaming data processing,
  - complex modeling and basic KPI analytics.
- **High availability**
  - Handle failures of nodes by design.
- **High scalability**
  - Grow by adding low-cost nodes, not by replacement with higher-powered computers.
- **Low cost**.
  - Lots of commodity-hardware nodes instead of expensive super-power computers.