Politecnico di Torino

DBMG

Non-relational databases for data management

DANIELE APILETTI

POLITECNICO DI TORINO

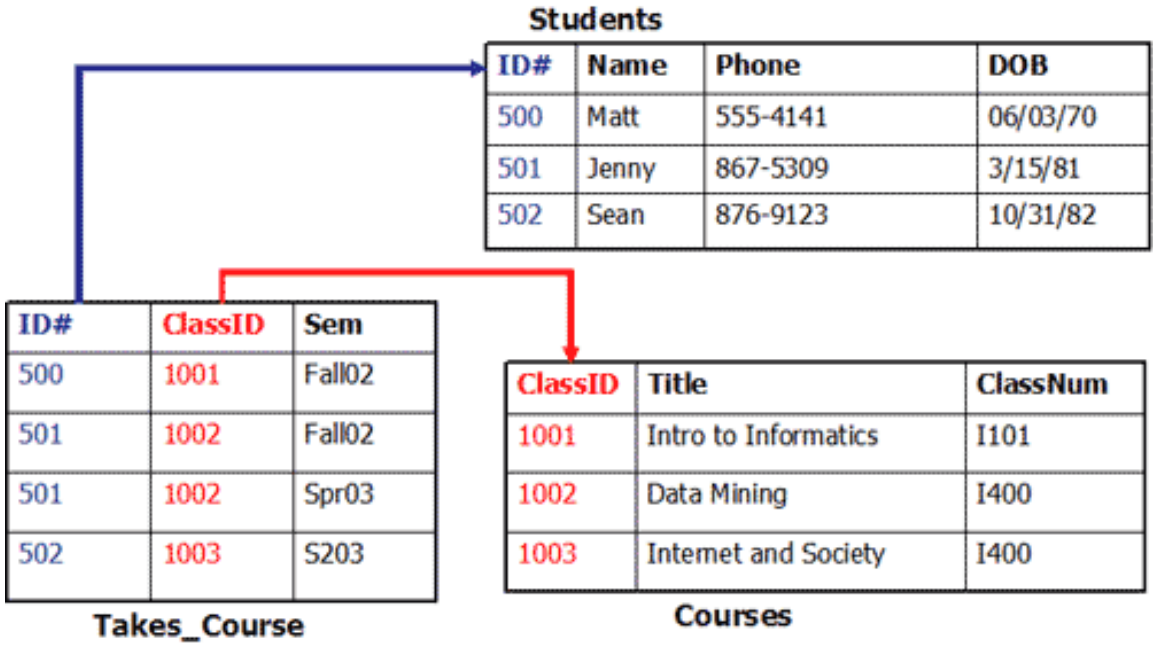
Technologies for Data management

- Data Warehouses
- NoSQL databases
- MapReduce
  - and other models for distributed programming
- Distributed file systems (GFS, HDFS, etc.)
- Grid computing, cloud computing, HPC computing
- Large-scale machine learning, Big Data

Relational Database Management Systems

- RDBMS are predominant database technologies
  - first defined in 1970 by Edgar Codd of IBM's Research Lab
- Data modeled as relations (tables)
  - object = tuple of attribute values
    - each attribute has a certain domain
  - a table is a set of objects (tuples, rows) of the same type
    - relation is a subset of cartesian product of the attribute domains
  - each tuple identified by a primary key
    - field (or a set of fields) that uniquely identifies a row
  - tables and objects "interconnected" via foreign keys
- SQL query language

RDBMS Example



Students

ID#	Name	Phone	DOB
500	Matt	555-4141	06/03/70
501	Jenny	867-5309	3/15/81
502	Sean	876-9123	10/31/82

Takes_Course

ID#	ClassID	Sem
500	1001	Fall02
501	1002	Fall02
501	1002	Spr03
502	1003	S203

Courses

ClassID	Title	ClassNum
1001	Intro to Informatics	I101
1002	Data Mining	I400
1003	Internet and Society	I400

SELECT Name
FROM Students S, Takes_Course T
WHERE S.ID=T.ID AND ClassID = 1001

source: https://github.com/talhafazal/DataBase/wiki/Home-Work-%23-3-Relational-Data-vs-Non-Relational-Databases

Fundamentals of RDBMS

Relational Database Management Systems (RDMBS)

1. Data structures are broken into the smallest units
   - normalization of database schema
     - because the data structure is known in advance
     - and users/applications query the data in different ways
   - database schema is rigid
2. Queries merge the data from different tables
3. Write operations are simple, search can be slower
4. Strong guarantees for transactional processing

From RDBMS to NoSQL

Efficient implementations of

- **table joins and of**
- **transactional processing**

require centralized system.

- NoSQL Databases:
- Database schema tailored for specific application
  - keep together data pieces that are often accessed together
- Write operations might be slower but read is fast
- Weaker consistency guarantees

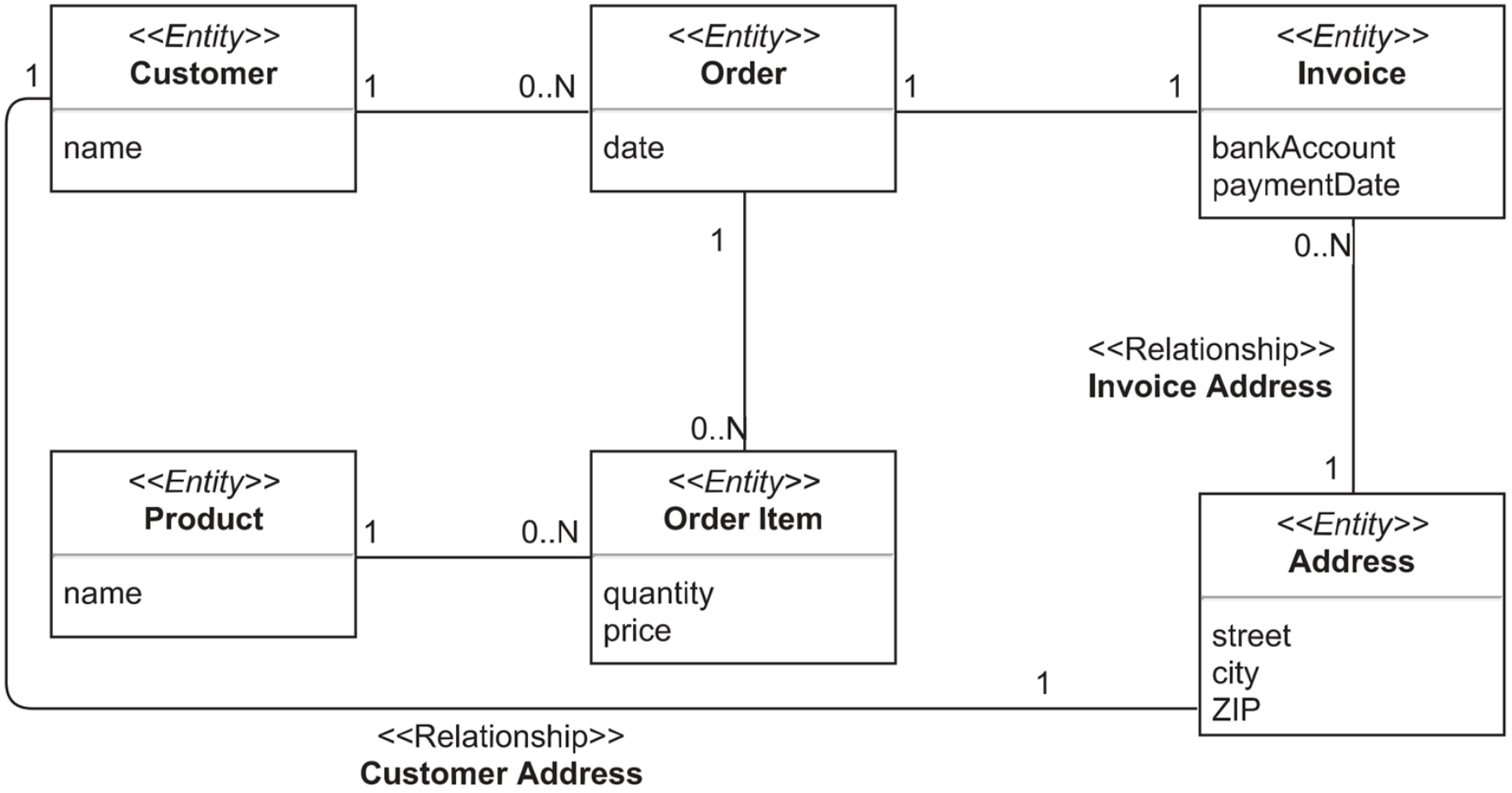
Data model

- The model by which the database **organizes data**

- Each NoSQL DB type has a different data model
  - Key-value, document, column-family, graph
  - The first three are oriented on **aggregates**

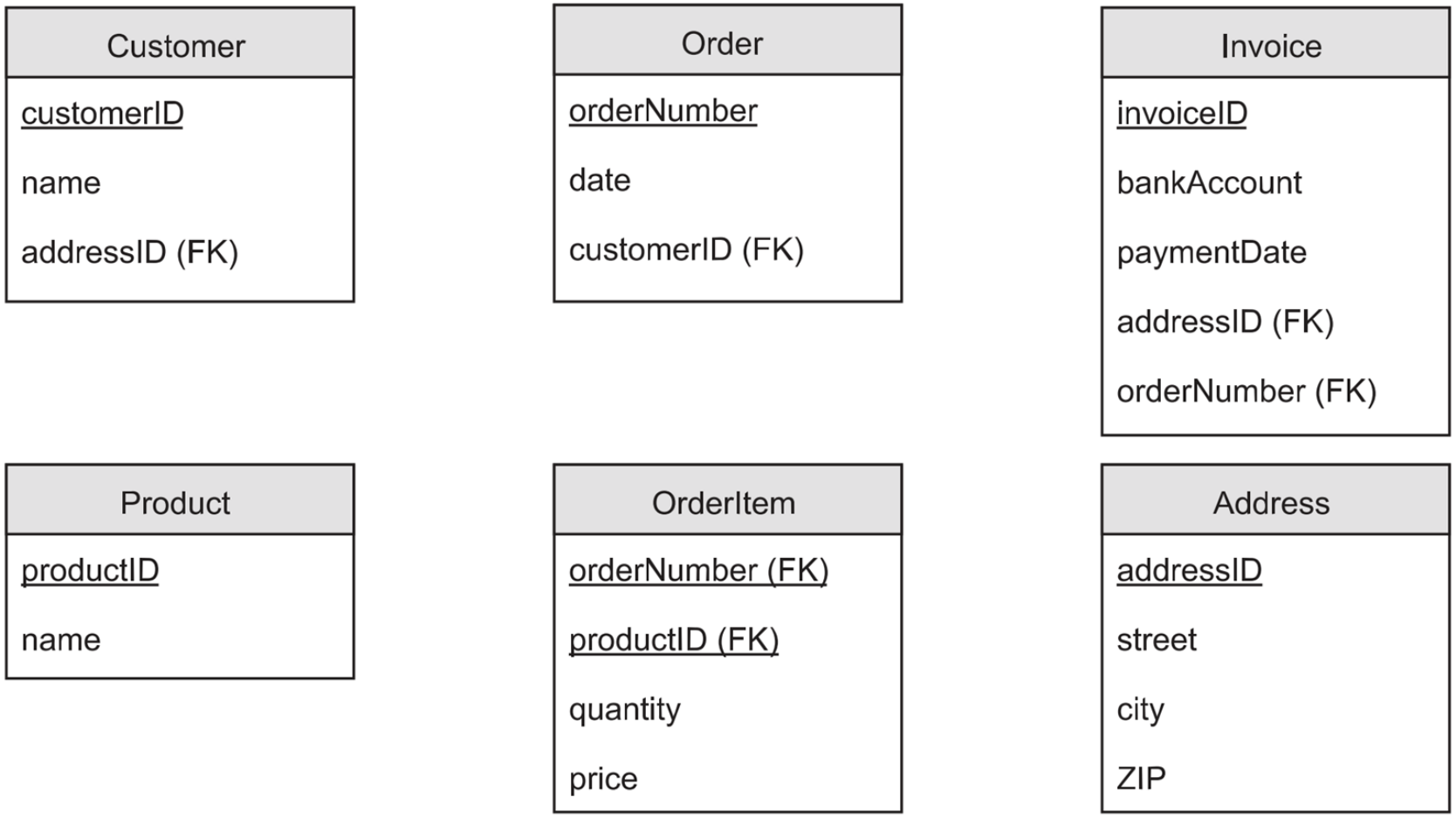
- Let us have a look at the classic relational model

Example: UML Model



source: Holubová, Kosek, Minařík, Novák. Big Data a NoSQL databáze. 2015.

Example: Relational Model



source: Holubová, Kosek, Minařík, Novák. Big Data a NoSQL databáze. 2015.

The Value of Relational Databases

- A (mostly) standard data model
- Many well-developed technologies
  - physical organization of the data, search indexes, query optimization, search operator implementations
- Good concurrency control (ACID)
  - transactions: atomicity, consistency, isolation, durability
- Many reliable integration mechanisms
  - "shared database integration" of applications
- Well-established: familiar, mature, support,…

RDBMS for Data Management

- Relational schema
  - data in tuples
  - a priori known schema
- Schema normalization
  - data split into tables
  - queries merge the data
- Transaction support
  - trans. management with ACID
  - Atomicity, Consistency, Isolation, Durability
  - safety first

- However, real data are
  - naturally flexible and
  - rapidly changing
- You don't know in advance..

- Inefficient for large data
- Slow in distributed environment

- Full transactions very inefficient in distributed environments

«NoSQL» birth

- In 1998 Carlo Strozzi's lightweight, open-source relational database that did not expose the standard SQL interface

- In 2009 Johan Oskarsson's (Last.fm) organizes an event to discuss recent advances on non-relational databases.
  - A new, unique, short hashtag to promote the event on Twitter was needed: #NoSQL

What is «NoSQL»?

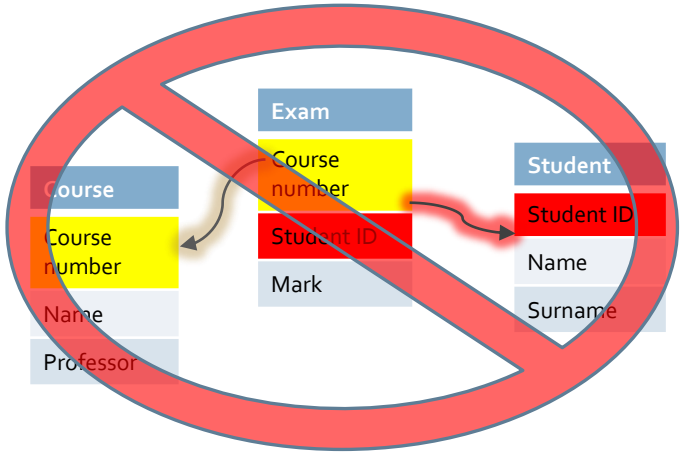
- Term used in late 90s for a different type of technology
  - Carlo Strozzi: http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/NoSQL/
- "Not Only SQL"?
  - but many RDBMS are also "not just SQL"

- "NoSQL is an accidental term with no precise definition"
  - first used at an informal meetup in 2009 in San Francisco (presentations from Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB, and MongoDB)

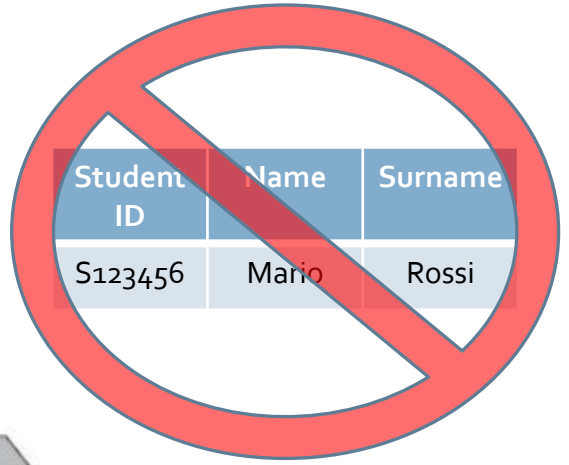
[Sadalage & Fowler: NoSQL Distilled, 2012]

NoSQL main features

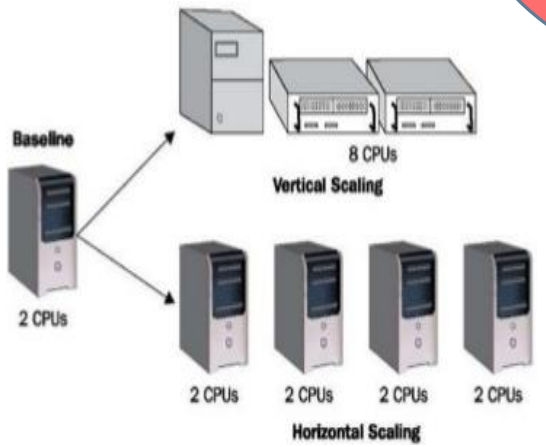
no joins



schema-less
(no tables, implicit schema)



horizontal scalability



http://www.slideshare.net/vivekparihar1/mongodb-scalability-and-high-availability-with-replicaset

Comparison

Relational databases	Non-Relational databases
Table-based, each record is a structured row	**Specialized storage solutions**, e.g, document-based, key-value pairs, graph databases, columnar storage
Predefined **schema** for each table, changes allowed but usually blocking (expensive in distributed and live environments)	**Schema-less**, schema-free, schema change is dynamic for each document, suitable for semi-structured or **un-structured data**
Vertically scalable, i.e., typically scaled by increasing the power of the hardware	**Horizontally** scalable, NoSQL databases are scaled by increasing the databases servers in the pool of resources to reduce the load

Comparison

Relational databases	Non-Relational databases
Use **SQL** (Structured Query Language) for defining and manipulating the data, very powerful	**Custom query** languages, focused on collection of documents, graphs, and other specialized data structures
Suitable for **complex queries**, based on data **joins**	**No standard** interfaces to perform complex queries, **no joins**
Suitable for **flat** and structured data storage	Suitable for complex (e.g., **hierarchical**) data, similar to JSON and XML
Examples: MySQL, Oracle, Sqlite, Postgres and Microsoft SQL Server	Examples: MongoDB, BigTable, Redis, Cassandra, HBase and CouchDB

Non-relational/NoSQL DBMSs

- Pros
  - Work with semi-structured data (JSON, XML), typical for web/online applications
  - Scale out (horizontal scaling – parallel query performance, replication)
  - High concurrency, high-volume random reads and writes
  - Massive data stores
  - Schema-free, schema-on-read
  - Support records/documents with different fields
  - High availability by design
  - Speed (join avoidance)

Non-relational/NoSQL DBMSs

- Cons
  - Do not support strict ACID transactional consistency
  - Data is de-normalized
    - requiring mass updates (e.g., product name change)
  - Missing built-in data integrity (do-it-yourself in your code)
  - No relationship enforcement (e.g., foreign keys)
  - Weak SQL
  - Slow mass updates
  - Use more disk space (replicated denormalized records, 10-50x)
  - Difficulty in tracking "schema" (set of attribute) changes over time

Just Another Temporary Trend?

- There have been other trends here before
  - object databases, XML databases, etc.

- But NoSQL databases:
  - are answer to real practical problems big companies have
  - are often developed by the biggest players
  - outside academia but based on solid theoretical results
    - e.g. old results on distributed processing
  - widely used

NoSQL Properties

1. Good scalability
   - horizontal scalability instead of vertical
2. Dynamic schema of data
   - different levels of flexibility for different types of DB
3. Efficient reading
   - spend more time storing the data, but read fast
   - keep relevant information together
4. Cost saving
   - designed to run on commodity hardware
   - typically open-source (with a support from a company)

Challenges of NoSQL Databases

1. Maturity of the technology
   - it's getting better, but RDBMS had a lot of time
2. User support
   - rarely professional support as provided by, e.g. Oracle
3. Administration
   - massive distribution requires advanced administration
4. Standards for data access
   - RDBMS have SQL, but the NoSQL world is more wild
5. Lack of experts
   - not enough DB experts on NoSQL technologies

The End of Relational Databases?

- Relational databases are not going away
  - are ideal for a lot of structured data, reliable, mature, etc.
- RDBMS became one option for data storage

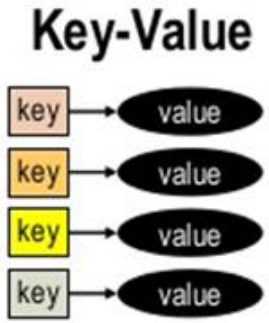
Polyglot persistence – using different data stores in different circumstances

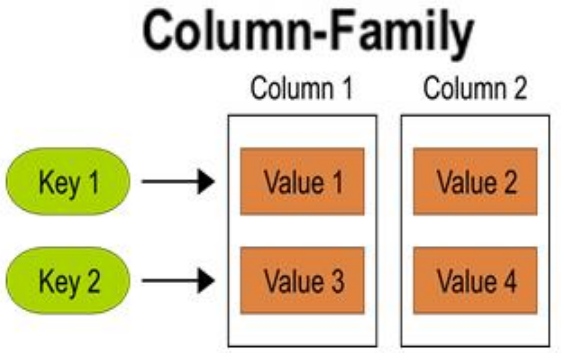
[Sadalage & Fowler: NoSQL Distilled, 2012]

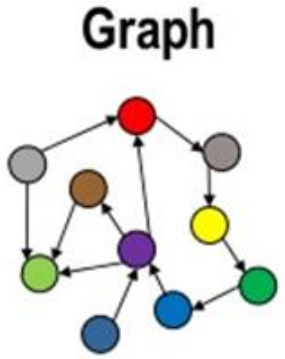
Two trends

1. NoSQL databases implement standard RDBMS features
2. RDBMS are adopting NoSQL principles

Types of NoSQL databases



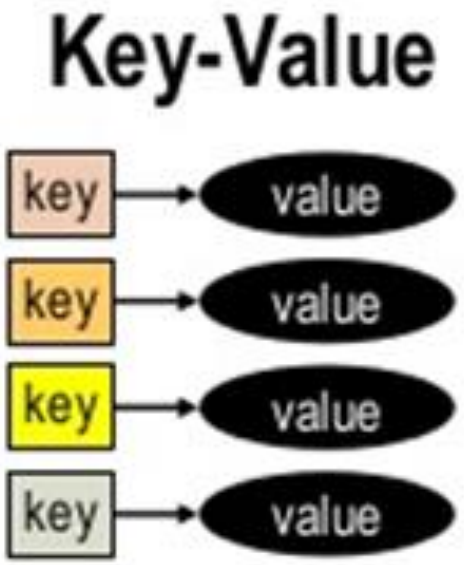




http://www.slideshare.net/Couchbase/webinar-making-sense-of-nosql-applying-nonrelational-databases-to-business-needs

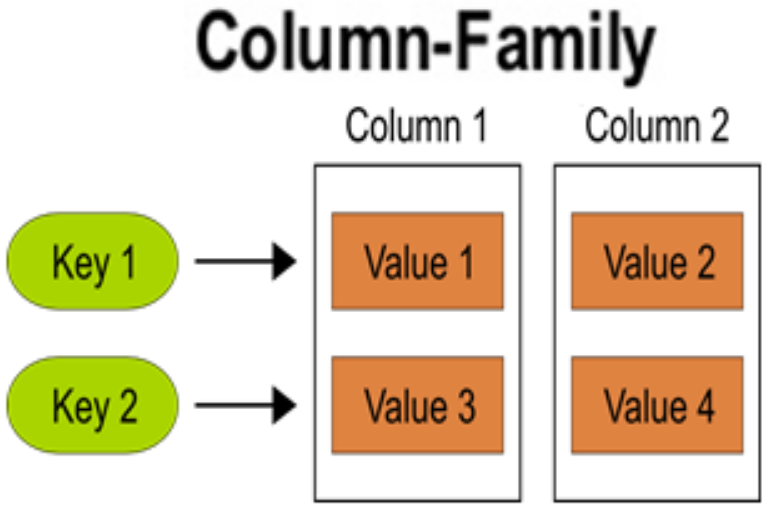
1) Key-values databases

- **Simplest** NoSQL data stores
- Match keys with values
- No structure
- Great **performance**
- Easily scaled
- Very fast
- Examples: Redis, Riak, Memcached



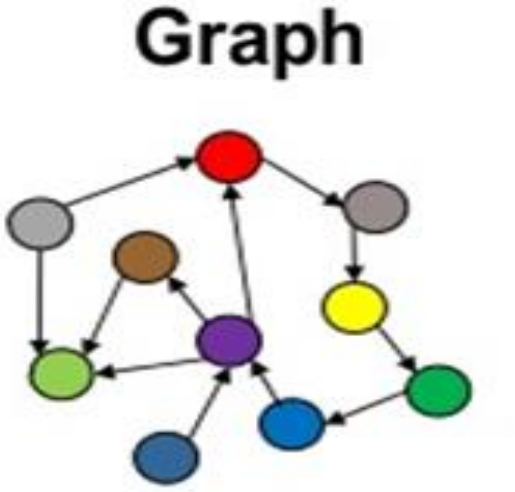
2) Column-oriented databases

- Store data in **columnar** format
  - Name = "Daniele":row1,row3; "Marco":row2,row4; …
  - Surname = "Apiletti":row1,row5; "Rossi":row2,row6,row7…
- A column is a (possibly-complex) **attribute**
- Key-value pairs stored and retrieved on key in a paralle system (similar to **indexes**)
- **Rows** can be constructed from column values
- Column stores can produce row output (**tables**)
- Completely transparent to application
- Examples: Cassandra, Hbase, Hypertable, Amazon DynamoDB



3) Graph databases

- Based on graph theory
- Made up by **Vertices** and **unordered** Edges or ordered **Arcs** between each Vertex pair
- Used to store information about **networks**
- Good fit for several real world applications
- Examples: Neo4J, Infinite Graph, OrientDB



4) Document databases

- Database stores and retrieves documents
- Keys are mapped to documents
- Documents are self-describing

  (attribute=value)
- Has hierarchical-tree nested data structures

  (e.g., maps, lists, datetime, …)
- **Heterogeneous** nature of documents
- Examples: **MongoDB**, CouchDB, RavenDB.

Document-based model

- Strongly aggregate-oriented
  - Lots of aggregates
  - Each aggregate has a key
  - Each aggregate is a document
- Data model
  - A **set of <key,value> pairs**
  - Document: an aggregate instance of <key,value> pairs
- Access to an aggregate
  - Queries based on the fields in the aggregate

```
# Customer object
{
"customerId": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"payment": [
  {"type": "debit",
  "ccinfo": "1000-1000-1000-1000"}
  ]
}
```

```
# Order object
{
"orderId": 99,
"customerId": 1,
"orderDate":"Nov-20-2011",
"orderItems":[{"productId":27, "price": 32.45}],
"orderPayment":[{"ccinfo":"1000-1000-1000-1000",
        "txnId":"abelif879rft"}],
"shippingAddress":{"city":"Chicago"}
}
```

Document basics

- Basic concept of data: Document
- Documents are self-describing pieces of data
  - Hierarchical tree data structures
  - Nested associative arrays (maps), collections, scalars
  - XML, JSON (JavaScript Object Notation), BSON, …
- Documents in a collection should be "similar"
  - Their schema can differ
- Documents stored in the value part of key-value
  - Key-value stores where the values are examinable
  - Building search indexes on various keys/fields

Document Example

- key=3 ->
```
{ "personID": 3,
  "firstname": "Martin",
  "likes": [ "Biking","Photography" ],
  "lastcity": "Boston",
  "visited": [ "NYC", "Paris" ] }
```

- key=5 ->
```
{ "personID": 5,
  "firstname": "Pramod",
  "citiesvisited": [ "Chicago", "London","NYC" ],
  "addresses": [
     { "state": "AK",
       "city": "DILLINGHAM" },
     { "state": "MH",
       "city": "PUNE" }  ],
  "lastcity": "Chicago" }
```

source: Sadalage & Fowler: NoSQL Distilled, 2012

Queries on Documents

Example in MongoDB syntax

- Query language expressed via JSON
- clauses: where, sort, count, sum, etc.

SQL: `SELECT * FROM users`

MongoDB: `db.users.find()`

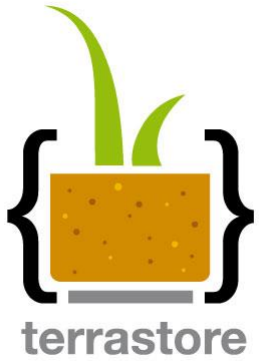
Example 1

```
SELECT *
FROM users
WHERE personID = 3
db.users.find( { "personID": 3 } )
```

Example 2

```
SELECT firstname, lastcity
FROM users
WHERE personID = 5
db.users.find( { "personID": 5},
{firstname:1, lastcity:1} )
```

Document Databases: Representatives



MS Azure DocumentDB

Ranked list: http://db-engines.com/en/ranking/document+store

Credits and sources

- P. Atzeni, R. Torlone
  - Dipartimento di Ingegneria, Sezione di Informatica e Automazione, Roma 3
- Martin Svoboda
  - Charles University, Faculty of Mathemacs and Physics Czech Technical
  - Universityin Prague,Faculty of Electrical Engineerin
- David Novak
  - FI, Masaryk University, Brno

Politecnico di Torino

Introduction to MongoDB

DANIELE APILETTI

POLITECNICO DI TORINO

Introduction

- The leader in the NoSQL Document-based databases
- Full of features, beyond NoSQL:
  - High performance
  - High availability
  - Native scalability
  - High flexibility
  - Open source

Terminology – Approximate mapping

Relational database	MongoDB
Table	Collection
Record	Document
Column	Field

Document Data Design

- High-level, business-ready representation of the data
  - Records are stored into BSON Documents
    - BSON is a binary representation of JSON documents
    - field-value pairs
    - may be nested

```
{
_id: <ObjectID1>,
username: "123xyz",
contact: {
          phone: 1234567890,
          email: "xyz@email.com",
         }
access:  {
          level: 5,
          group: "dev",
         }
}
```

Embedded Sub-Document

Embedded Sub-Document

```
{
   _id: ObjectId("5099803df3f4948bd2f98391"),
   name: { first: "Alan", last: "Turing" },
   birth: new Date('Jun 23, 1912'),
   death: new Date('Jun 07, 1954'),
   contribs: [ "Turing machine", "Turing test", "Turingery" ],
   views : NumberLong(1250000)
}
```

Document Data Design

- High-level, business-ready representation of the data
- Mapping into developer-language objects
  - date, timestamp, array, sub-documents, etc.
- Field names
  - The field name **_id** is reserved for use as a **primary key**; its value must be unique in the collection, is **immutable**, possibly autogenerated, and may be of any type other than an array.
  - Field names **cannot** contain the **null** character.
  - The server **permits** storage of field names that contain dots (.) and dollar signs ($)
  - BSON documents may have more than one field with **the same name**. Most MongoDB interfaces, however, represent MongoDB with a structure (e.g., a hash table) that does not support duplicate field names.
  - The maximum BSON document size is **16 megabytes**. To store documents larger than the maximum size, MongoDB provides GridFS.
  - Unlike JavaScript objects, the fields in a BSON document are **ordered.**

Politecnico di Torino

DBMG

Databases and collections.

Create and delete operations (1)

Databases and Collections

- Each **instance** of MongoDB can manage multiple **databases**
- Each database is composed of a set of **collections**
- Each collection contains a set of documents
  - The documents of each collection represent **similar** "objects"
  - However, remember that MongoDB is **schema-less**
  - You are not required to define the schema of the documents a-priori and objects of the same collections can be characterized by different fields
  - Starting in MongoDB 3.2, you can enforce **document validation** rules for a collection during update and insert operations.

Databases and Collections

- Show the list of available databases

```
show databases
```

- Select the database you are interested in

```
use <database-name>
```

- E.g.
  - `use deliverydb`

Databases and Collections

- Create a database and a collection inside the database
  - Select the database by using the command "use <database name>"
  - Then, create a collection
    - MongoDB creates a collection implicitly when the collection is first referenced in a command
- Delete/Drop a database
  - Select the database by using "use <database name>"
  - Execute the command

```
db.dropDatabase()
```

E.g.,

```
use deliverydb;
db.dropDatabase();
```

Databases and Collections

- A collection stores documents, uniquely identified by a document **"_id"**
- Create collections

```
db.createCollection(<collection name>, <options>);
```

  - The collection is associated with the current database. Always select the database before creating a collection.
  - Options related to the collection size and indexing, e.g., to create a capped collection, or to create a new collection that uses document validation

- E.g.,
  - `db.createCollection("authors", {capped: true});`

Databases and Collections

- Show collections

```
show collections
```

- Drop collections

```
db.<collection_name>.drop()
```

- E.g.
  - `db.authors.drop()`

C.R.U.D. Operations

- **C**reate

```
db.users.insertOne(                 ◄ collection
   {
      name: "sue",                  ◄ field: value
      age: 26,                      ◄ field: value
      status: "pending"             ◄ field: value
   }                                  } document
)
```

- **R**ead

```
db.users.find(                                  ◄ collection
   { age: { $gt: 18 } },                        ◄ query criteria
   { name: 1, address: 1 }                      ◄ projection
).limit(5)                                      ◄ cursor modifier
```

- **U**pdate

```
db.users.updateMany(                            ◄ collection
   { age: { $lt: 18 } },                        ◄ update filter
   { $set: { status: "reject" } }               ◄ update action
)
```

- **D**elete

```
db.users.deleteMany(                ◄ collection
   { status: "reject" }             ◄ delete filter
)
```

Create: insert **one** document

- Insert a single document in a collection

```
db.<collection name>.insertOne( {<set of the field:value pairs of the new document>} );
```

- E.g.,

```
db.people.insertOne( {
        user_id: "abc123",
        age: 55,
        status: "A"
    } );
```

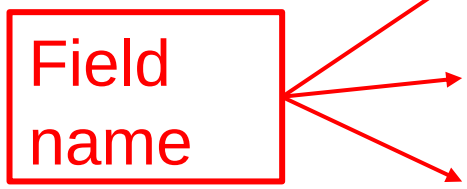
Create: insert **one** document

- Insert a single document in a collection

```
db.<collection name>.insertOne( {<set of the field:value pairs of the new document>} );
```

- E.g.,

```
db.people.insertOne( {
        user_id: "abc123",
        age: 55,
        status: "A"
    } );
```



Create: insert **one** document

- Insert a single document in a collection

```
db.<collection name>.insertOne( {<set of the field:value pairs of the new document>} );
```

- E.g.

```
db.people.insertOne( {
        user_id: "abc123",
        age: 55,
        status: "A"
    } );
```

Create: insert **one** document

- Insert a single document in a collection

```
db.<collection name>.insertOne( {<set of the field:value pairs of the new document>} );
```

Now people contains a new document representing a user with:

```
user_id: "abc123",
age: 55
status: "A"
```

Create: insert **one** document

- E.g.,

```
db.people.insertOne( {
        user_id: "abc124",
        age: 45,
        favorite_colors: ["blue", "green"]
} );
```

Now people contains a new document representing a user with:

`user_id: "abc124", age: 45` and an array `favorite_colors` containing the values `"blue"` and `"green"`

Create: insert **one** document

- E.g.,

```
db.people.insertOne( {
        user_id: "abc124",
        age: 45,
        address: {
            street: "my street",
            city: "my city"
        }
} );
```

Nested document

Example of a document containing a nested document

Create: insert **many** documents

- Insert multiple documents in a single statement:

```
db.<collection name>.insertMany([ <comma separated list of documents> ]);
```

```
db.products.insertMany( [
      { user_id: "abc123", age: 30, status: "A"},
      { user_id: "abc456", age: 40, status: "A"},
      { user_id: "abc789", age: 50, status: "B"}
   ] );
```

Create: insert **many** documents

- Insert many documents with one single command

```
db.<collection name>.insertMany([ <comma separated list of documents> ]);
```

- E.g.,

```
db.people.insertMany([
  {user_id: "abc123", age: 55, status: "A"},
  {user_id: "abc124", age: 45, favorite_colors: ["blue", "green"]}
] );
```

Delete

- Delete existing data, in MongoDB corresponds to the deletion of the associated document.
  - Conditional delete
  - Multiple delete

MySQL clause	MongoDB operator
`DELETE FROM`	`deleteMany()`

Delete

MySQL clause	MongoDB operator
`DELETE FROM`	`deleteMany()`

```
DELETE FROM people
WHERE status = "D"
```

```
db.people.deleteMany(
   { status: "D" }
)
```

Delete

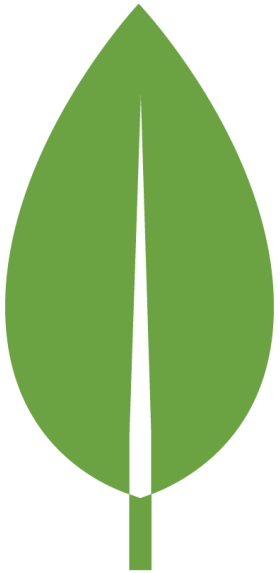
MySQL clause	MongoDB operator
`DELETE FROM`	`deleteMany()`

`DELETE FROM people`
`WHERE status = "D"`
	`db.people.deleteMany(`
	`  { status: "D" }`
	`)`

`DELETE FROM people`	`db.people.deleteMany({})`

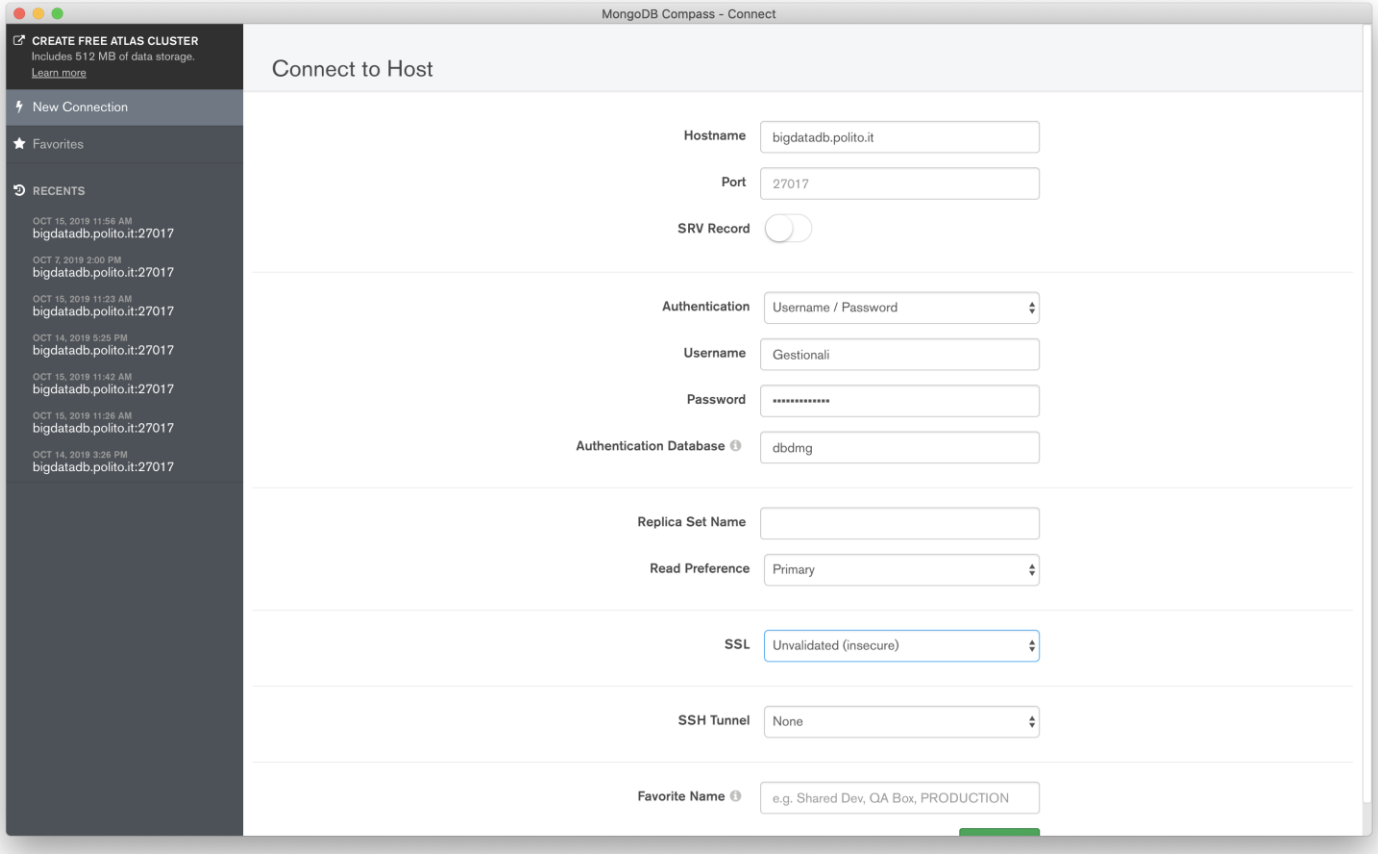
GUI for MongoDB

MongoDB Compass



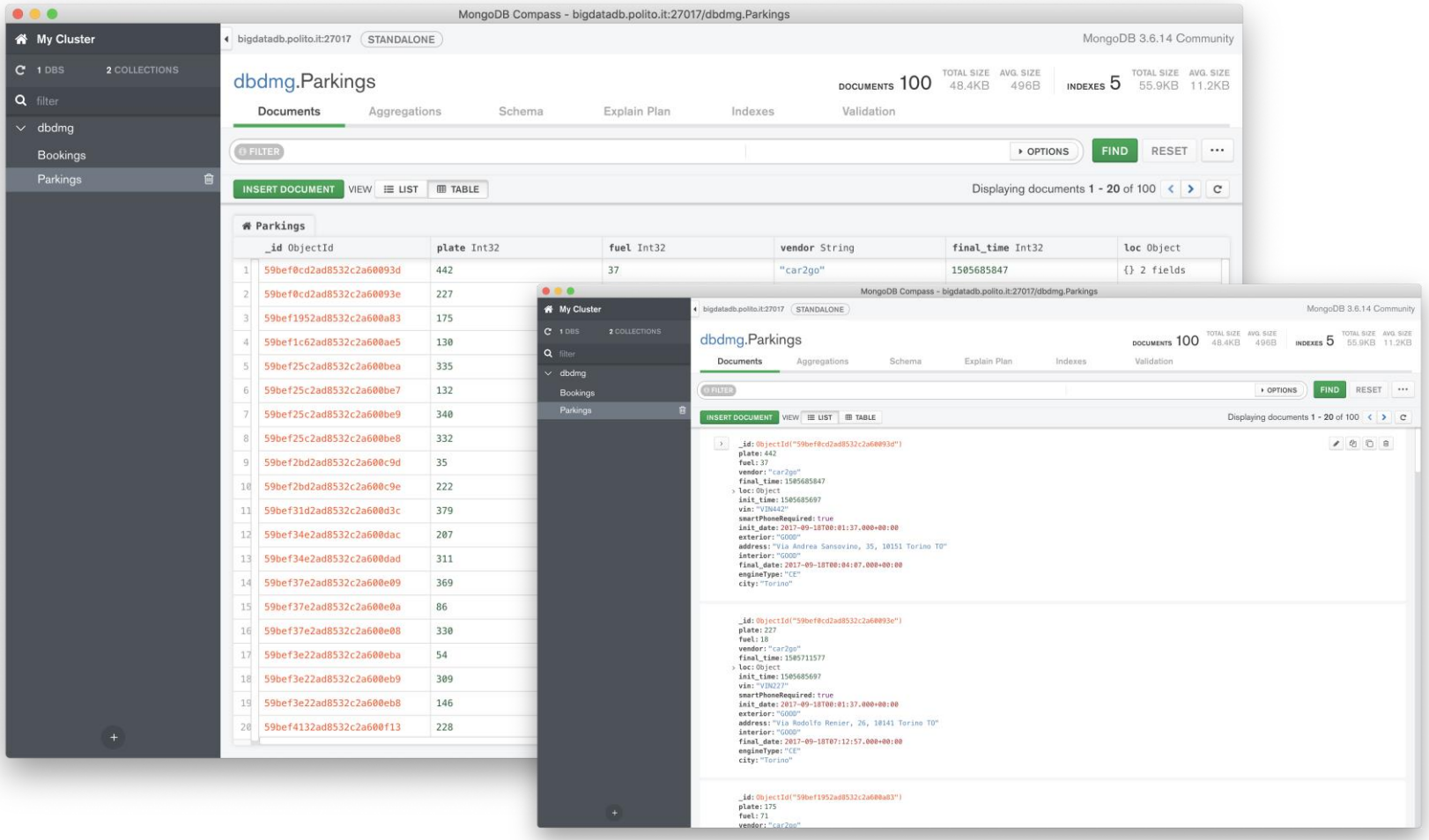
- Visually explore data.
- Available on Linux, Mac, or Windows.
- MongoDB Compass analyzes documents and displays rich structures within collections.
- Visualize, understand, and work with your geospatial data.

MongoDB Compass



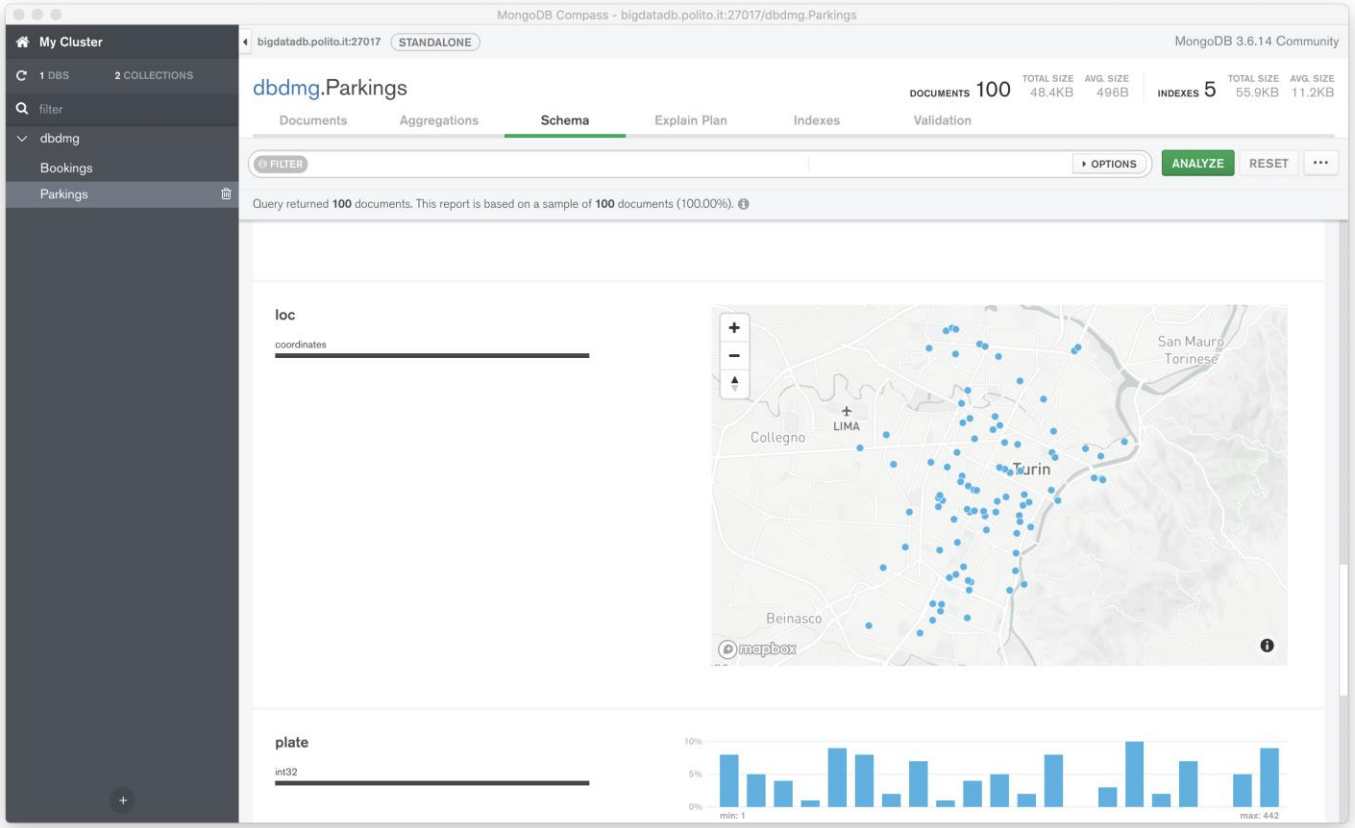
- Connect to local or remote instances of MongoDB.

MongoDB Compass



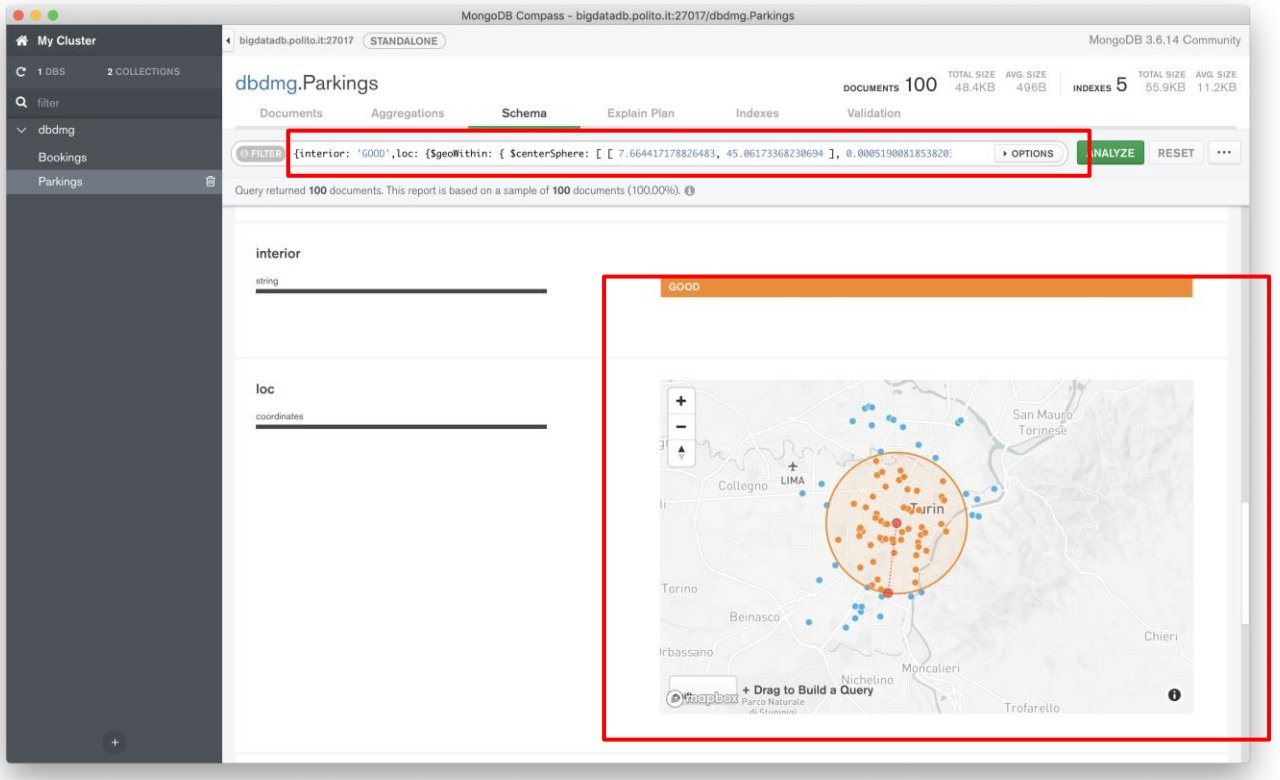
- Get an overview of the data in list or table format.

MongoDB Compass



- Analyze the documents and their fields.
- Native support for geospatial coordinates.

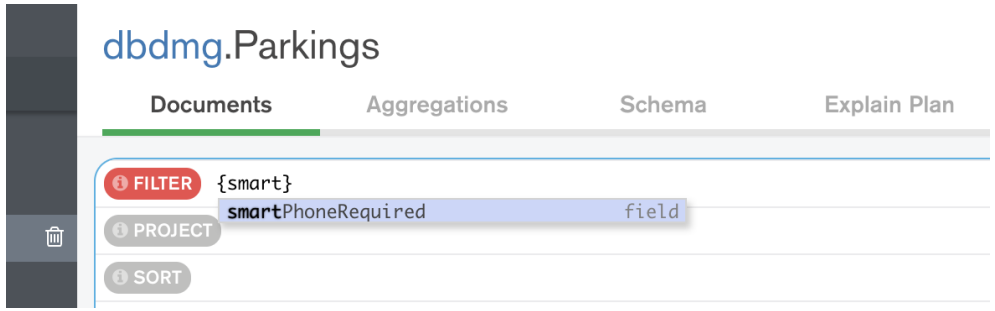
MongoDB Compass



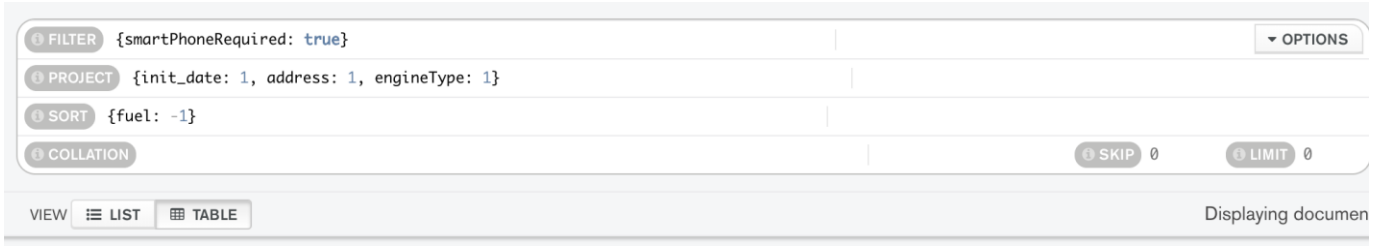
- Visually build the query conditioning on analyzed fields.

MongoDB Compass

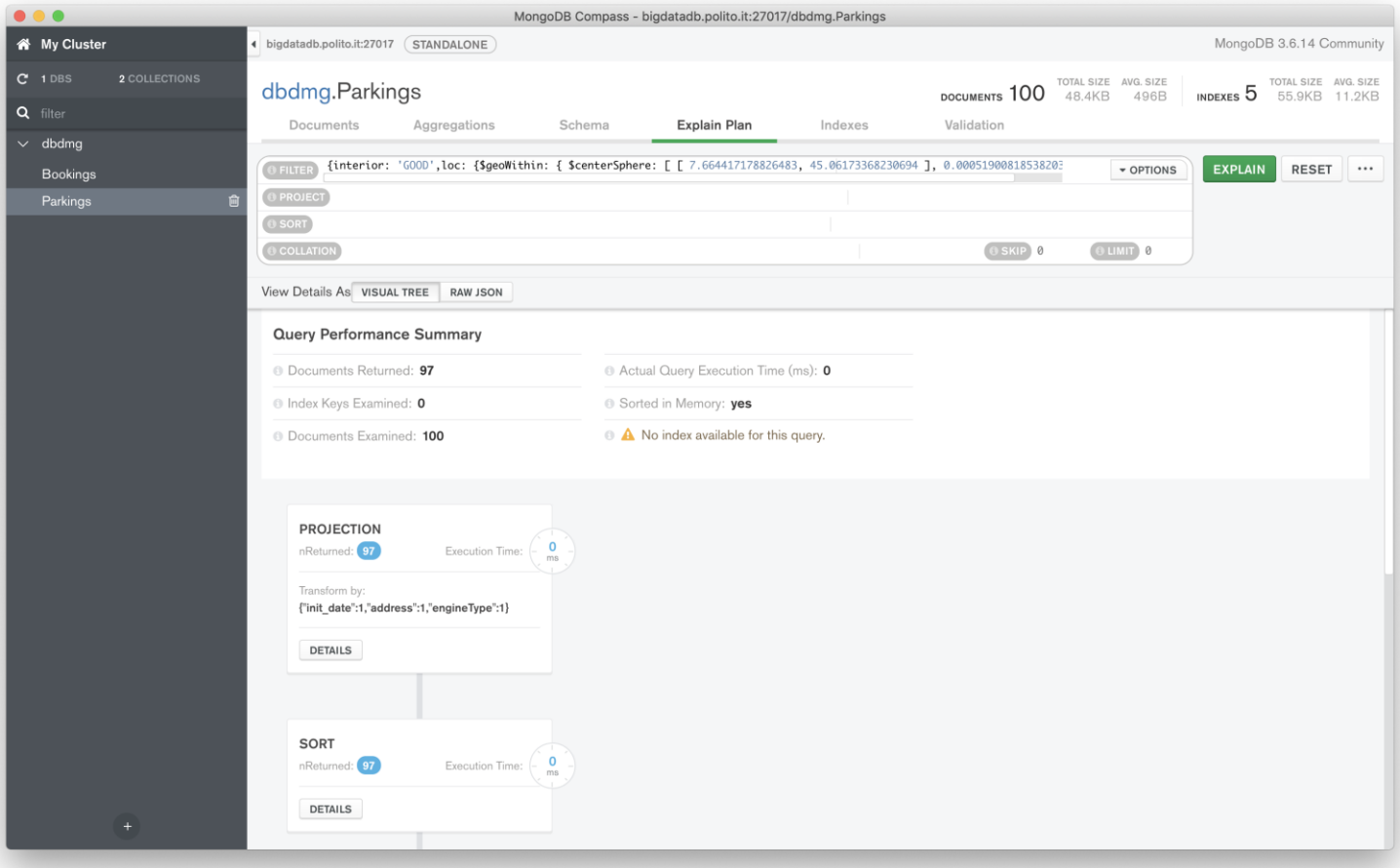
- Autcomplete enabled by default



- Construct the query step by step.

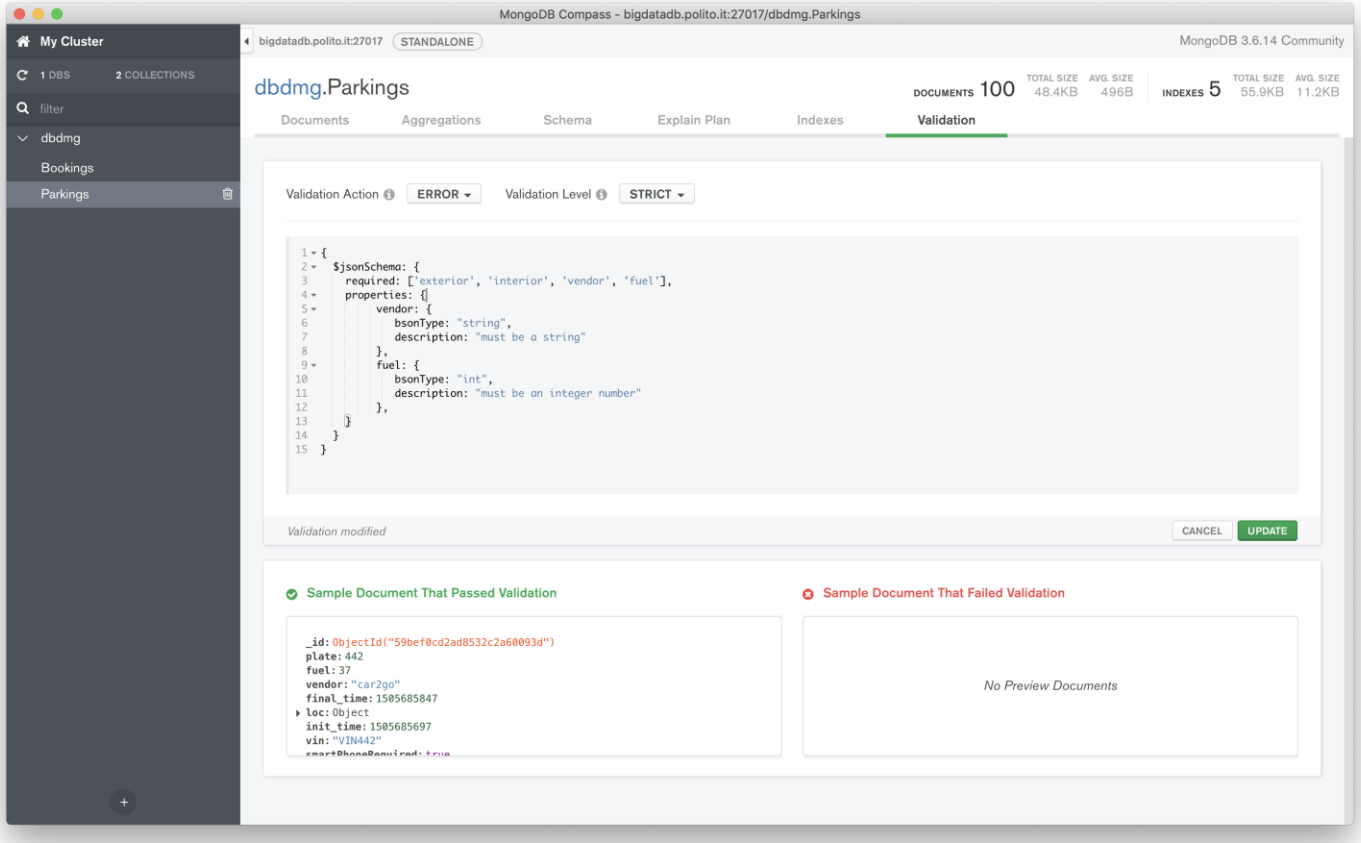


MongoDB Compass



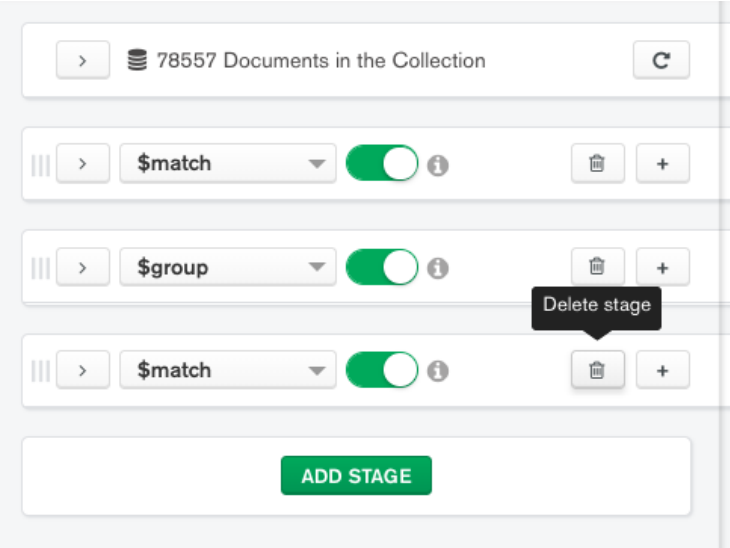
- Analyze query performance and get hints to speed it up.

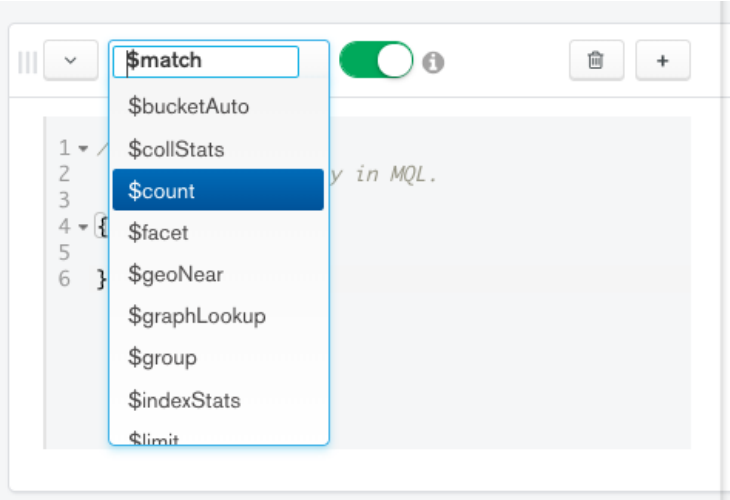
MongoDB Compass



- Specify contraints to validate data
- Find unconsistent documents.

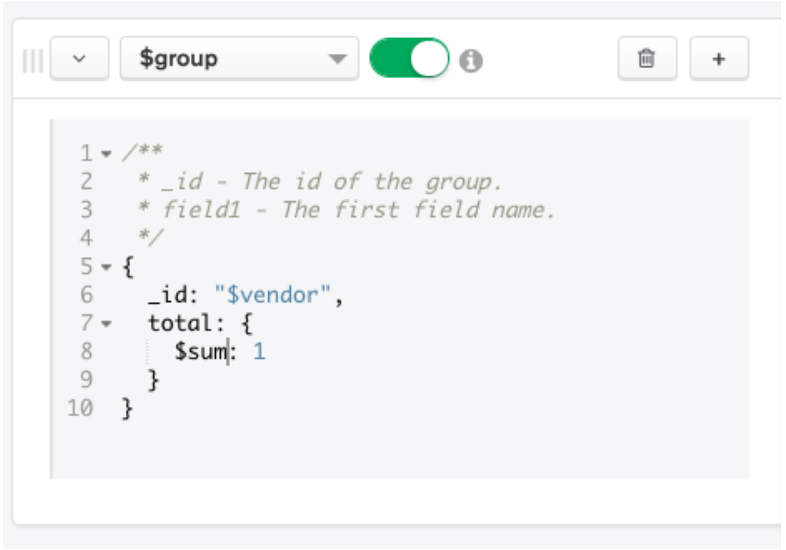
MongoDB Compass: Aggregation



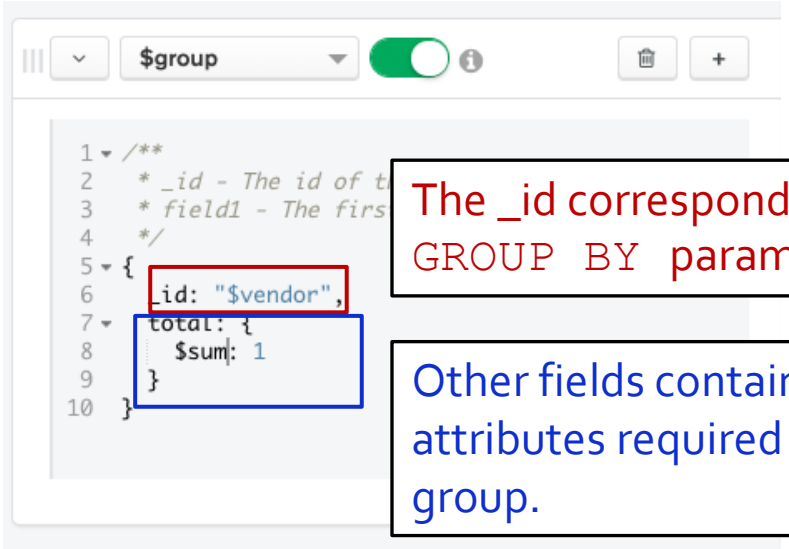


- Build a pipeline consisting of multiple aggregation stages
- Define the filter and aggregation attributes for each operator.

MongoDB Compass: Aggregation stages



MongoDB Compass: Aggregation stages

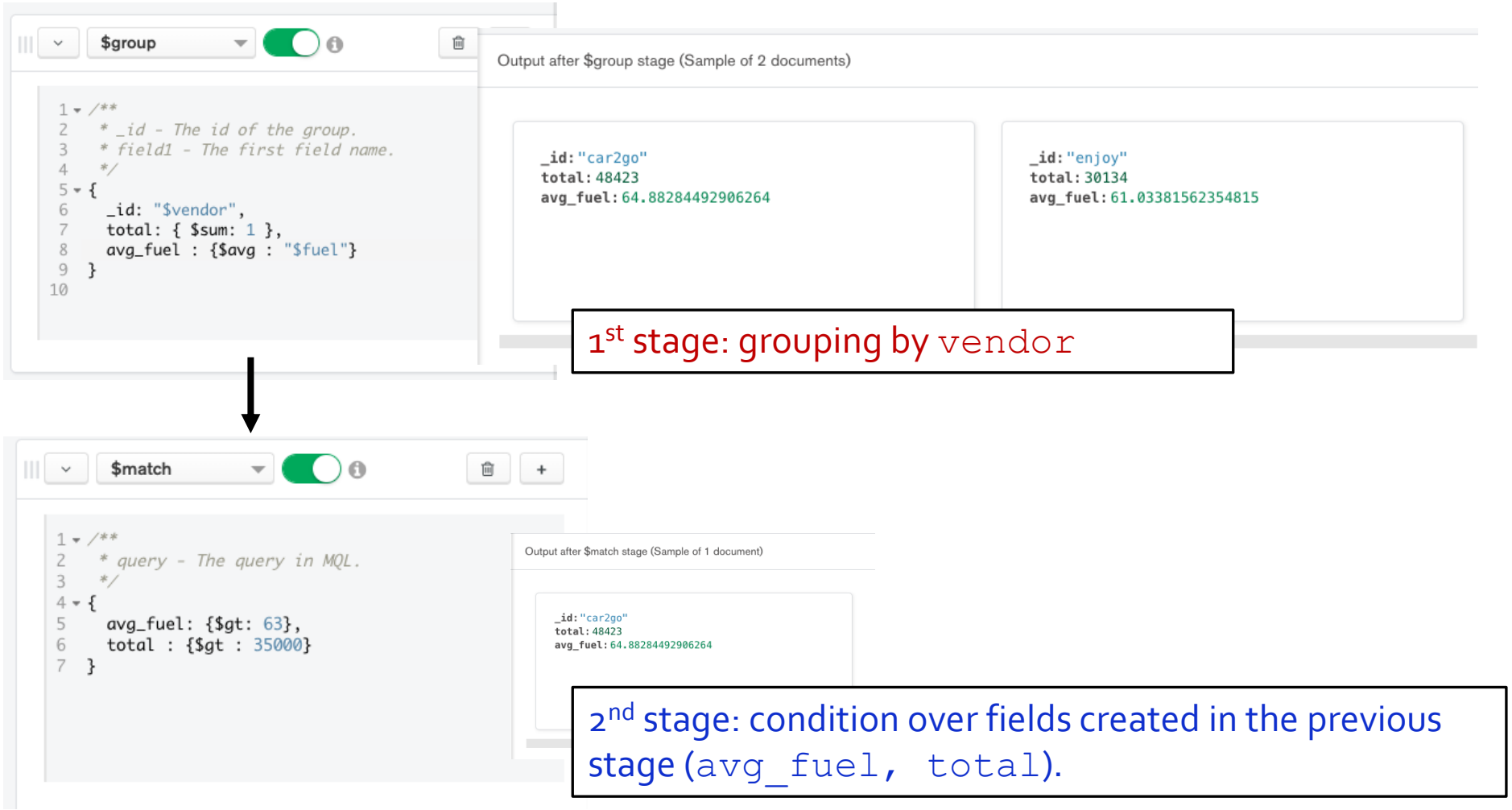


The _id corresponds to the `GROUP BY` parameter in SQL

Other fields contain the attributes required for each group.

One group for each "vendor".

MongoDB Compass: Pipelines



Politecnico di Torino

NoSQL Databases

Introduction to MongoDB

DANIELE APILETTI

POLITECNICO DI TORINO

Querying data – find() operation

Query language

- Most of the operations available in SQL language can be expressend in MongoDB language

MySQL clause	MongoDB operator
`SELECT`	`find()`

`SELECT` `*` `FROM people`	`db.people.`**`find()`**

Read data from documents

- Select documents

```
db.<collection name>.find( {<conditions>}, {<fields of interest>} );
```

Read data from documents (Filter conditions)

- Select documents

```
db.<collection name>.find( {<conditions>}, {<fields of interest>} );
```

- Select the documents satisfying the specified conditions and specifically only the fields specified in fields of interest
  - `<conditions>` are optional
    - conditions take a document with the form:

      `{field1 : <value>, field2 : <value> ... }`

    - Conditions may specify a value or a regular expression

Read data from documents (Project fields)

- Select documents

```
db.<collection name>.find( {<conditions>}, {<fields of interest>} );
```

- Select the documents satisfying the specified conditions and specifically only the fields specified in fields of interest
  - `<fields of interest>` are optional
    - projections take a document with the form:

      ```
      {field1 : <value>, field2 : <value> ... }
      ```

    - **1**/true to include the field, **0**/false to exclude the field

find() operator (1)

```
SELECT id,
       user_id,
       status
FROM people
```

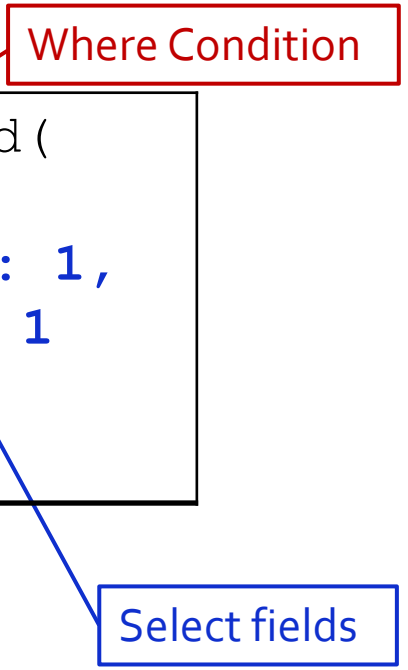
```
db.people.find(
    { },
    { user_id: 1,
      status: 1
    }
)
```

find() operator (2)

MySQL clause	MongoDB operator
SELECT	find()

```
SELECT id,
       user_id,
       status
FROM people
```

```
db.people.find(
    { },
    { user_id: 1,
      status: 1
    }
)
```



find() operator (3)

MySQL clause	MongoDB operator
SELECT	find()
WHERE	find({<WHERE CONDITIONS>})

SELECT * FROM people WHERE status = "A"	db.people.find(**{ status: "A" }**)

find() operator (4)

MySQL clause	MongoDB operator
SELECT	find()
WHERE	find({<WHERE CONDITIONS>})

```
SELECT user_id, status
FROM people
WHERE status = "A"
```

```
db.people.find(
    { status: "A" },
    { user_id: 1,
      status: 1,
      _id: 0
    }
)
```

Where Condition

Selection fields

By default, the `_id` field is always returned.
To remove it, you must explicitly indicate `_id: 0`

find() operator (5)

MySQL clause	MongoDB operator
SELECT	find()
WHERE	find({<WHERE CONDITIONS>})

```
db.people.find(
    {"address.city":"Rome" }
)
```

```
{ _id: "A",
  address: {
        street: "Via Torino",
        number: "123/B",
        city: "Rome",
        code: "00184"
    }
}
```

Read data from one document

- Select a single document

```
db.<collection name>.findOne( {<conditions>}, {<fields of interest>} );
```

- Select one document that satisfies the specified query criteria.
  - If multiple documents satisfy the query, it returns the first one according to the natural order which reflects the order of documents on the disk.

(No) joins

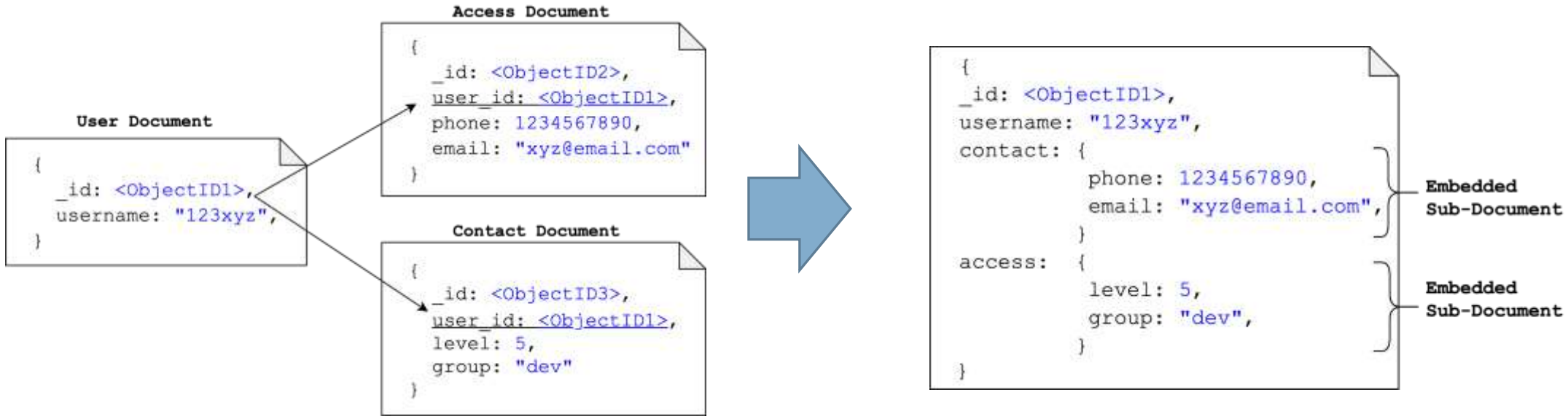
- No join operator exists (but `$lookup`)
  - You must write a program that
    - Selects the documents of the first collection you are interested in
    - Iterates over the documents returned by the first step, by using the loop statement provided by the programming language you are using
    - Executes one query for each of them to retrieve the corresponding document(s) in the other collection

https://docs.mongodb.com/manual/reference/operator/aggregation/lookup

(No) joins

- **(no) joins**
  - Relations among documents/records are provided by
    - Object_ID (_id), named "**Manual reference**" in MongoDB, a second query is required
    - **DBRef**, a standard approach across collections and databases (check the driver compatibility)

```
{ "$ref" : <value>, "$id" : <value>, "$db" : <value> }
```



https://docs.mongodb.com/manual/reference/database-references/

Comparison query operators

Name	Description
`$eq or :`	Matches values that are equal to a specified value
`$gt`	Matches values that are greater than a specified value
`$gte`	Matches values that are greater than or equal to a specified value
`$in`	Matches any of the values specified in an array
`$lt`	Matches values that are less than a specified value
`$lte`	Matches values that are less than or equal to a specified value
`$ne`	Matches all values that are not equal to a specified value, **including documents that do not contain the field**.
`$nin`	Matches none of the values specified in an array

Comparison operators (>)

MySQL	MongoDB	Description
>	**$gt**	**greater than**

```
SELECT *
FROM people
WHERE age > 25
```

```
db.people.find(
    { age: { $gt: 25 } }
)
```

Comparison operators (>=)

MySQL	MongoDB	Description
>	$gt	greater than
>=	**$gte**	**greater equal then**

```
SELECT *
FROM people
WHERE age >= 25
```

```
db.people.find(
    { age: { $gte: 25 } }
)
```

Comparison operators (<)

MySQL	MongoDB	Description
>	$gt	greater than
>=	$gte	greater equal then
<	**$lt**	**less than**

```
SELECT *
FROM people
WHERE age < 25
```

```
db.people.find(
    { age: { $lt: 25 } }
)
```

Comparison operators (<=)

MySQL	MongoDB	Description
>	$gt	greater than
>=	$gte	greater equal then
<	$lt	less than
<=	**$lte**	**less equal then**

```
SELECT *
FROM people
WHERE age <= 25
```

```
db.people.find(
    { age: { $lte: 25 } }
)
```

Comparison operators (=)

MySQL	MongoDB	Description
>	$gt	greater than
>=	$gte	greater equal then
<	$lt	less than
<=	$lte	less equal then
=	**$eq**	**equal to** The $eq expression is equivalent to { field: <value> }.

```
SELECT *
FROM people
WHERE age = 25
```

```
db.people.find(
    { age: { $eq: 25 } }
)
```

Comparison operators (!=)

MySQL	MongoDB	Description
>	$gt	greater than
>=	$gte	greater equal then
<	$lt	less than
<=	$lte	less equal then
=	$eq	equal to
!=	**$ne**	**Not equal to**

```
SELECT *
FROM people
WHERE age != 25
```

```
db.people.find(
    { age: { $ne: 25 } }
)
```

Conditional operators

- To specify multiple conditions, **conditional operators** are used
- MongoDB offers the same functionalities of MySQL with a different syntax.

MySQL	MongoDB	Description
AND	,	Both verified
OR	**$or**	At least one verified

Conditional operators (AND)

MySQL	MongoDB	Description
AND	**,**	**Both verified**

```
SELECT *
FROM people
WHERE status = "A"
AND age = 50
```

```
db.people.find(
    { status: "A",
      age: 50 }
)
```

Conditional operators (OR)

MySQL	MongoDB	Description
AND	,	Both verified
OR	**$or**	**At least one verified**

```
SELECT *
FROM people
WHERE status = "A"
OR age = 50
```

```
db.people.find(
{ $or:
   [ { status: "A" } ,
     { age: 50 }
   ]
}
)
```

Type of read operations (1)

- Count

```
db.people. count({ age: 32 })
```

- Comparison

```
db.people. find({ age: {$gt: 32 })          // or equivalently with $gte, $lt, $lte,
db.people.find({ age: {$in: [32, 40] })     // returns all documents having age either 32 or 40
db.people.find({ age: { $gt: 25, $lte: 50 } })   //returns all documents having age > 25 and age <= 50
```

- Logical

```
db.people.find({ name: {$not: {$eq: "Max" } } })
db.people.find({ $or: [ {age: 32}, {age: 33} ] } )
```

Type of read operations (2)

```
db.items.find({
    $and: [
        {$or: [{qty: {$lt: 15}}, {qty: {$gt: 50}} ]},
        {$or: [{sale: true}, {price: {$lt: 5}} ]}
    ]
```

This query returns documents (items) that satisfy **both** these conditions:

1. Quantity sold either less than 15 **or** greater than 50
2. Either the item is on sale (field "sale": true) **or** its price is less than 5

Type of read operations (3)

- Element

```
db.inventory.find( { item: null } )                    // equality filter

db.inventory.find( { item : { $exists: false } } )     // existence filter

db.inventory.find( { item : { $type: 10 } } )          // type filter
```

Note:

- Item: null → matches documents that either
  - contain the item field whose value is **null** or
  - that do **not** contain the item field
- Item: {$exists: false} → matches documents that do **not** contain the item field

- Aggregation → Slides on "Data aggregation"

Type of read operations (4)

- Embedded Documents

```
db.inventory.find( { size: { h: 14, w: 21, uom: "cm" } } )
```

Select all documents where the field size equals the **exact document** { h: 14, w: 21, uom: "cm" }

```
db.inventory.find( { "size.uom": "in" } )
```

To specify a query condition on fields in an embedded/nested document, use **dot notation**

```
db.inventory.find( { "size.h": { $lt: 15 } } )
```

Dot notation and comparison operator

Type of read operations (5)

- Array
  - Query for all documents where the field tags value is an array with exactly two specific elements

```
db.inventory.find( { tags: ["red", "black"] } )          → Item list order matters!
db.inventory.find( { tags: { $all: ["red", "black"] } } )  → List order does not matter
```

    - The following queries return **different** results, i.e., they are **not** equivalent

```
db.inventory.find( { tags: ["red", "black"] } )
db.inventory.find( { tags: ["black", "red"] } )
```

```
db.inventory.find( { tags: { $all: ["red", "black"] } } )
db.inventory.find( { tags: { $all: ["black", "red"] } } )
```

Type of read operations (6)

- Query for all documents where tags is an array that **contains** the string "red" as one of its elements

```
db.inventory.find( { tags: "red" } )
```

- Query an Array with **Compound Filter Conditions** on the Array Elements

```
db.inventory.find( { dim_cm: { $gt: 15, $lt: 20 } } )
```

- Query for an Array Element that **Meets Multiple Criteria**

```
db.inventory.find( { dim_cm: { $elemMatch: { $gt: 15, $lt: 20 } } } )
```

Attention:

- **Compound filter**: one element of the array can satisfy the greater than 15 condition and another element can satisfy the less than 20 condition, or alternatively a single element can satisfy both
- **elemMatch**: one single element of the array **must** satisfy **both**

Type of read operations (7)

- Query for an Element by the Array Index Position

```
db.inventory.find( { "dim_cm.0": { $gt: 25 } } )
```

- Query an Array by Array Length

```
db.inventory.find( { "tags": { $size: 3 } } )
```

Cursor

- `db.collection.find()` gives back a cursor. It can be used to iterate over the result or as input for next operations.
- E.g.,
  - `cursor.sort()`
  - `cursor.count()`
  - `cursor.forEach() //shell method`
  - `cursor.limit()`
  - `cursor.max()`
  - `cursor.min()`
  - `cursor.pretty()`

Cursor: sorting data

- Sort is a cursor method
- Sort documents
  - `sort( {<list of field:value pairs>} );`
  - field specifies which filed is used to sort the returned documents
  - value = -1 descending order
  - Value = 1 ascending order
- Multiple field: value pairs can be specified
  - Documents are sort based on the first field
  - In case of ties, the second specified field is considered

Cursor: sorting data

- Sorting data with respect to a given field in sort() operator

MySQL clause	MongoDB operator
`ORDER BY`	`sort()`

```
SELECT *
FROM people
WHERE status = "A"
ORDER BY age ASC
```

```
db.people.find(
  { status: "A" }
).sort( { age: 1 } )
```

- Returns all documents having status="A". The result is sorted in ascending age order

Cursor: sorting data

- Sorting data with respect to a given field in sort() operator

MySQL clause	MongoDB operator
`ORDER BY`	`sort()`

`SELECT *`
`FROM people`
`WHERE status = "A"`
`ORDER BY age ASC`
	`db.people.find(`
	`  { status: "A" }`
	`).sort( { age: 1 } )`

`SELECT *`
`FROM people`
`WHERE status = "A"`
`ORDER BY age DESC`
	`db.people.find(`
	`  { status: "A" }`
	`).sort( { age: -1 } )`

- Returns all documents having status="A". The result is sorted in ascending age order
- Returns all documents having status = "A". The result is sorted in descending age order

Cursor: counting

MySQL clause	MongoDB operator
`COUNT`	`count()or find().count()`

`SELECT COUNT(*)`
`FROM people`
	`db.people.count()`
	or
	`db.people.find().count()`

Cursor: counting

MySQL clause	MongoDB operator
COUNT	count()or find().count()

SELECT COUNT(*) FROM people	db.people.count() or db.people.find().count()
SELECT COUNT(*) WHERE status = "A" FROM people	db.people.count(status: "A")} or db.people.find({status: "A"}).count()

Cursor: counting

MySQL clause	MongoDB operator
COUNT	count()or find().count()

SELECT COUNT(*) FROM people	db.people.count() or db.people.find().count()
SELECT COUNT(*) WHERE status = "A" FROM people	db.people.count(status: "A")} or db.people.find({status: "A"}).count()
SELECT COUNT(*) FROM people WHERE **age > 30**	db.people.count(**{ age: { $gt: 30 } }**)

Similar to the find() operator, count() can embed conditional statements.

Cursor: forEach()

- `forEach` applies a JavaScript function to apply to each document from the cursor.

```
db.people.find({status: "A"}).forEach(
   function(myDoc){
         print( "user:"+myDoc.name );
   })
```

- Select documents with status="A" and print the document name.

Politecnico di Torino

Databases and collections.

Update operations (3)

Document update

- Back at the C.R.U.D. operations, we can now see how documents can be updated using:

```
db.collection.updateOne(<filter>, <update>, <options>)
```

```
db.collection.updateMany(<filter>, <update>, <options>)
```

  - `<filter>` = filter condition. It specifies which documents must be updated
  - `<update>` = specifies which fields must be updated and their new values
  - `<options>` = specific update options

Document update

- E.g.,

```
db.inventory.updateMany(
    { "qty": { $lt: 50 } },
    {
      $set: { "size.uom": "in", status: "P" },
      $currentDate: { lastModified: true }
    }
)
```

  - This operation updates all documents with qty<50
  - It sets the value of the size.uom field to "in", the value of the status field to "P", and the value of the lastModified field to the current date.

Updating data

- Tuples to be updated should be selected using the WHERE statements

MySQL clause	MongoDB operator
UPDATE <table> SET <statement> WHERE <condition>	db.<table>.updateMany({ <condition> }, { $set: {<statement>} })

Updating data

MySQL clause	MongoDB operator
UPDATE <table> SET <statement> WHERE <condition>	db.<table>.updateMany({ <condition> }, { $set: {<statement>}})
UPDATE people SET status = "C" WHERE age > 25	db.people.updateMany({age: { $gt: 25 } }, {$set: { status: "C"}})

Updating data

MySQL clause	MongoDB operator
UPDATE <table> SET <statement> WHERE <condition>	db.<table>.updateMany({ <condition> }, { $set: {<statement>}})
UPDATE people SET status = "C" WHERE age > 25	db.people.updateMany({age: { $gt: 25 } }, {$set: { status: "C"}})
UPDATE people SET age = **age + 3** WHERE status = "A"	db.people.updateMany({ status: "A" } , { **$inc: { age: 3 }** })

The $inc operator increments a field by a specified value

Data aggregation

General concepts

- Documents enter a multi-stage pipeline that transforms the **documents of a collection** into an aggregated result
- Pipeline **stages** can appear **multiple** times in the pipeline
  - exceptions *$out*, *$merge*, and *$geoNear* stages
- Pipeline expressions can **only** operate on the **current document** in the pipeline and cannot refer to data from other documents: expression operations provide in-memory transformation of documents (max 100 Mb of RAM per stage).
- Generally, expressions are **stateless** and are only evaluated when seen by the aggregation process with one exception: accumulator expressions used in the *$group* stage (e.g. totals, maximums, minimums, and related data).
- The aggregation pipeline provides an alternative to ***map-reduce*** and may be the preferred solution for aggregation tasks since MongoDB introduced the *$accumulator* and *$function* aggregation operators starting in version 4.4

Aggregation Framework

SQL	MongoDB
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
//LIMIT	$limit
SUM	$sum
COUNT	$sum

Aggregation pipeline

- Aggregate functions can be applied to collections to group documents

```
db.collection.aggregate( { <set of stages> })
```

  - Common stages: `$match, $group ..`
  - The aggregate function allows applying aggregating functions (e.g. sum, average, ..)
  - It can be combined with an initial definition of groups based on the grouping fields

Aggregation example (1)

```
db.people.aggregate( [
    { $group: { _id: null,
                mytotal: { $sum: "$age" },
                mycount: { $sum: 1 }
              }
    }
] )
```

- Considers all documents of people and
  - sum the values of their age
  - sum a set of ones (one for each document)
- The returned value is associated with a field called “mytotal” and a field “mycount”

Aggregation example (2)

```
db.people.aggregate( [
    { $group: { _id: null,
                myaverage: { $avg: "$age" },
                mytotal: { $sum: "$age" }
              }
    }
] )
```

- Considers all documents of people and computes
  - sum of age
  - average of age

Aggregation example (3)

```
db.people.aggregate( [
    { $match: {status: "A"} } ,
    { $group: { _id: null,
                count: { $sum: 1 }
              }
    }
] )
```

Where conditions

- Counts the number of documents in people with status equal to “A”

Aggregation in “Group By”

MySQL clause	MongoDB operator
GROUP BY	aggregate($group)

```
SELECT status,
       AVG(age) AS total
FROM people
GROUP BY status
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $avg: "$age" }
     }
   }
] )
```

Aggregation in “Group By”

MySQL clause	MongoDB operator
GROUP BY	aggregate($group)

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $sum: "$age" }
     }
   }
] )
```

Group field

Aggregation in “Group By”

MySQL clause	MongoDB operator
GROUP BY	aggregate($group)

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $sum: "$age" }
     }
   }
] )
```

Group field

Aggregation function

Aggregation in “Group By + Having”

MySQL clause	MongoDB operator
HAVING	aggregate($group, $match)

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
HAVING total > 1000
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $sum: "$age" }
     }
   },
  { $match: { total: { $gt: 1000 } } }
] )
```

Aggregation in "Group By + Having"

MySQL clause	MongoDB operator
HAVING	aggregate($group, $match)

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
HAVING total > 1000
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $sum: "$age" }
     }
   },
  { $match: { total: { $gt: 1000 } } }
] )
```

Group stage: Specify the aggregation field and the aggregation function

Aggregation in "Group By + Having"

MySQL clause	MongoDB operator
HAVING	aggregate($group, $match)

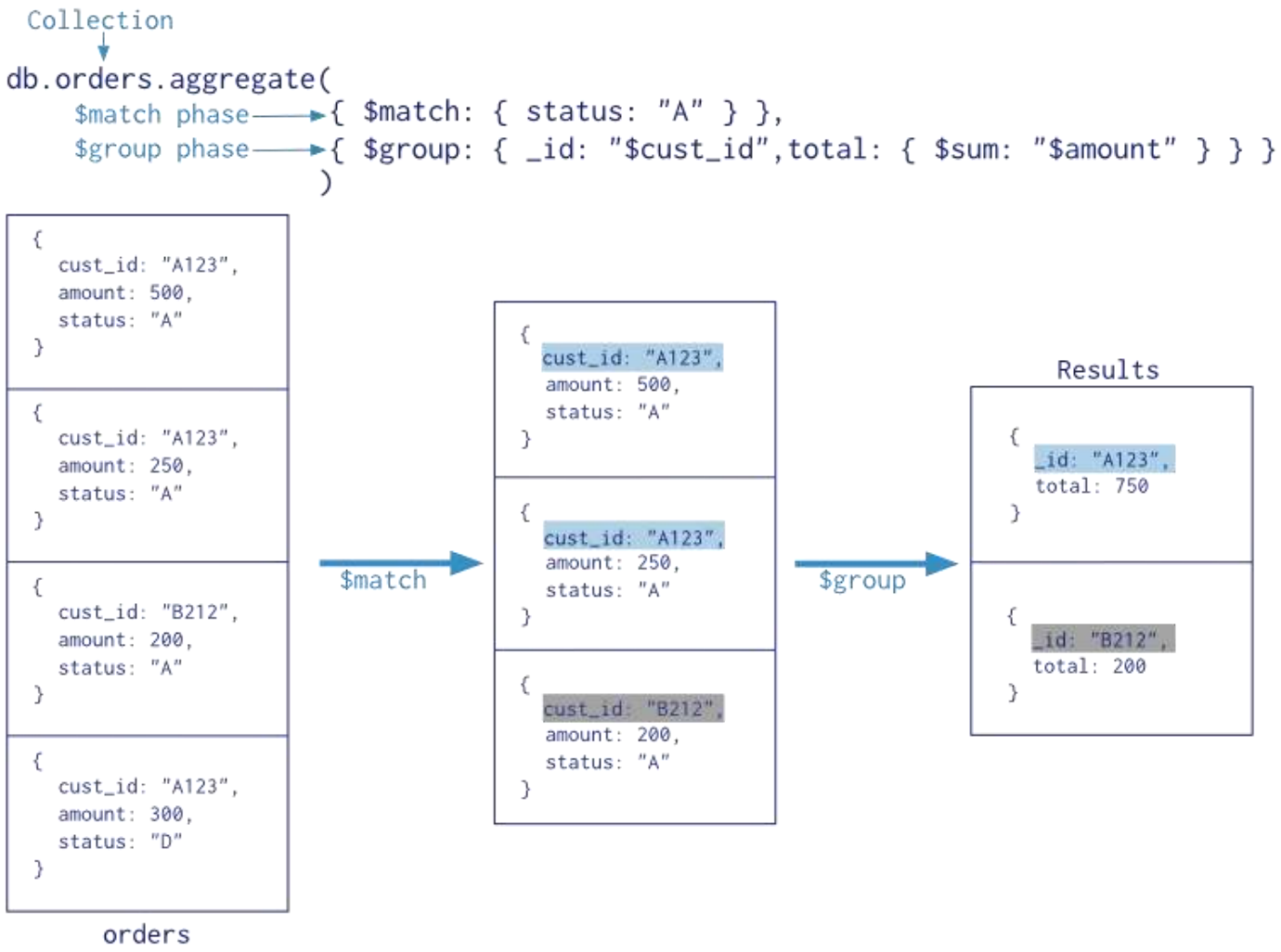
```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
HAVING total > 1000
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $sum: "$age" }
     }
   },
   { $match: { total: { $gt: 1000 } } }
] )
```

Group stage: Specify the aggregation field and the aggregation function

Match Stage: specify the condition as in `HAVING`

Aggregation at a glance



Pipeline stages (1)

Stage	Description
$addFields	Adds **new fields** to documents. Reshapes each document by adding new fields to output documents that will contain both the existing fields from the input documents and the newly added fields.
$bucket	Categorizes incoming documents **into groups,** called buckets, based on a specified expression and bucket boundaries. On the contrary**, $group** creates a "bucket" for each value of the group field.
$bucketAuto	Categorizes incoming documents into a specific number of groups, called buckets, based on a specified expression. Bucket boundaries are automatically determined in an attempt to **evenly distribute** the documents into the specified number of buckets.
$collStats	Returns statistics regarding a collection or view (it must be the **first stage**)
$count	Passes a document to the next stage that contains a count of the input **number of documents** to the stage (same as $group+$project)

Pipeline stages (2)

Stage	Description
$facet	Processes **multiple aggregation pipelines** within a single stage on the same set of input documents. Enables the creation of multi-faceted aggregations capable of characterizing data across multiple dimensions. Input documents are passed to the $facet stage only once, without needing multiple retrieval.
$geoNear	Returns an ordered stream of documents based on the **proximity** to a geospatial point. The output documents include an additional **distance** field. It must in the **first stage** only.
$graphLookup	Performs a **recursive search** on a collection. To each output document, adds a new array field that contains the traversal results of the recursive search for that document.

Example

```
db.employees.aggregate( [
  {
    $graphLookup: {
      from: "employees",
      startWith: "$reportsTo",
      connectFromField: "reportsTo",
      connectToField: "name",
      as: "reportingHierarchy"
    }
  }
] )
```

- The $graphLookup operation recursively matches on the **reportsTo** and **name** fields in the employees collection, returning the **reporting hierarchy** for each person.
- Returns a list of documents such as

```
{
  "_id" : 5,
  "name" : "Asya",
  "reportsTo" : "Ron",
  "reportingHierarchy" : [
    { "_id" : 1, "name" : "Dev" },
    { "_id" : 2, "name" : "Eliot", "reportsTo" : "Dev" },
    { "_id" : 3, "name" : "Ron", "reportsTo" : "Eliot" }
  ]
}
```

original document

Pipeline stages (3)

Stage	Description
$group	Groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group. Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields.
$indexStats	Returns statistics regarding the use of each index for the collection.
$limit	Passes the first n documents unmodified to the pipeline where n is the specified limit. For each input document, outputs either one document (for the first n documents) or zero documents (after the first n documents).
$lookup	Performs a **join** to another collection in the same database to filter in documents from the "joined" collection for processing. To each input document, the $lookup stage adds a new array field whose elements are the matching documents from the "joined" collection. The $lookup stage passes these reshaped documents to the next stage.

Pipeline stages (4)

Stage	Description
$match	Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. $match uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).
$merge	Writes the resulting documents of the aggregation pipeline to a collection. The stage can incorporate (insert new documents, merge documents, replace documents, keep existing documents, fail the operation, process documents with a custom update pipeline) the results into an output collection. To use the $merge stage, it must be the last stage in the pipeline.
$out	Writes the resulting documents of the aggregation pipeline to a collection. To use the $out stage, it must be the last stage in the pipeline.
$project	Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.

Pipeline stages (5)

Stage	Description
$sample	Randomly selects the specified number of documents from its input.
$set	Adds new fields to documents. Similar to $project, $set reshapes each document in the stream; specifically, by adding new fields to output documents that contain both the existing fields from the input documents and the newly added fields. $set is an alias for $**addFields** stage. If the name of the new field is the same as an existing field name (including _id), $set **overwrites** the existing value of that field with the value of the specified expression.
$skip	Skips the first n documents where n is the specified skip number and passes the remaining documents unmodified to the pipeline. For each input document, outputs either zero documents (for the first n documents) or one document (if after the first n documents).
$sort	Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.

Pipeline stages (6)

Stage	Description
$sortByCount	Groups incoming documents based on the value of a specified expression, then computes the count of documents in each distinct group.
$unset	Removes/excludes fields from documents.
$unwind	Deconstructs an array field from the input documents to output a document for each element. Each output document replaces the array with an element value. For each input document, outputs n documents where n is the number of array elements and can be zero for an empty array.

Politecnico di Torino

NoSQL Databases

Introduction to MongoDB

DANIELE APILETTI

POLITECNICO DI TORINO

Politecnico di Torino

Data aggregation examples

Data Model

- Given the following collection of books

```
{
 "title":"MongoDb Guide2",
 "tag":["mongodb","guide","database"],
 "n":200,
 "review_score": 2.2,
 "price":[ {"v": 22.22, "c": "€", "country": "IT"},
           {"v": 22.00, "c": "£", "country": "UK"}
         ],
 "author": {
    "_id": 1,
    "name":"Mario",
    "surname": "Rossi"}
}
{_id:ObjectId("5fb29b175b99900c3fa24293",
 title:”Developing with Python",
 tag:[”python”,”guide”,“programming”],
 n:352,
 review_score:4.6,
 price:[{v: 24.99, c: “€”,  country: “IT”},
        {v: 19.49, c: “£”, country:”UK”} ],
 author: {_id: 2,
          name:”John”,
          surname: “Black”}
}, …
```

price currency

price value

number of pages

Example 1

- For each country, select the average price and the average review_score.
- The review score should be rounded down.
- Show the first 20 results with a total number of books higher than 50.

$unwind

```
db.book.aggregate( [
  { $unwind: "$price" },
] )
```

Build a document for each entry of the **price** array

Result - **$unwind**

{ "_id" : ObjectId("5fb29ae15b99900c3fa24292"), "title" : "MongoDb guide", "tag" : ["mongodb", "guide", "database"], "n" : 100, "review_score" : 4.3, "price" : { "v" : 19.99, "c" : " € ", "country" : "IT" }, "author" : { "_id" : 1, "name" : "Mario", "surname" : "Rossi" } }

{ "_id" : ObjectId("5fb29ae15b99900c3fa24292"), "title" : "MongoDb guide", "tag" : ["mongodb", "guide", "database"], "n" : 100, "review_score" : 4.3, "price" : { "v" : 18, "c" : "£", "country" : "UK" }, "author" : { "_id" : 1, "name" : "Mario", "surname" : "Rossi" } }

{ "_id" : ObjectId("5fb29b175b99900c3fa24293"), "title" : " Developing with Python ", "tag" : ["python", "guide", "programming"], "n" : 352, "review_score" : 4.6, "price" : { "v" : 24.99, "c" : " € ", "country" : "IT" }, "author" : { "_id" : 2, "name" : "John", "surname" : "Black" } }

{ "_id" : ObjectId("5fb29b175b99900c3fa24293"), "title" : " Developing with Python ", "tag" : ["python", "guide", "programming"], "n" : 352, "review_score" : 4.6, "price" : { "v" : 19.49, "c" : "£", "country" : "UK" }, "author" : { "_id" : 2, "name" : "John", "surname" : "Black" } }

…

$group

```
db.book.aggregate( [
  { $unwind: "$price" } ,
  { $group: { _id: "$price.country"},
              avg_price: { $avg: " $price.v" ,
              bookcount: {$sum:1},
              review: {$avg: " $review_score"}
            }
  }
] )
```

dot notation to access the value of the **embedded** document fields

count the number of books (number of documents)

Result - `$group`

{ "_id" : "**UK**", "avg_price" : 18.75, "bookcount": **150**, "review": 4.3}

{ "_id" : "**IT**", "avg_price" : 22.49, "bookcount": **132**, "review": 3.9}

{ "_id" : "**US**", "avg_price" : 22.49, "bookcount": **49**, "review": 4.2}

…

$match

```
db.book.aggregate( [
  { $unwind: '$price' } ,
  { $group: { _id: '$price.country',
                avg_price: { $avg: '$price.v' },
                bookcount: {$sum:1},
                review: {$avg: '$review_score'}
              }
  },
 {$match: { bookcount: { $gte: 50 } } },
] )
```

Filter the documents where **bookcount** is greater than 50

Result - `$match`

```
{ "_id" : "UK", "avg_price" : 18.75, "bookcount": 150, "review": 4.3}
{ "_id" : "IT", "avg_price" : 22.49, "bookcount": 132, "review":  3.9}
...
```

$project

```
db.book.aggregate( [
  { $unwind: '$price' } ,
  { $group: { _id: '$price.country',
                avg_price: { $avg: '$price.v' },
                bookcount: {$sum:1},
                review: {$avg: '$review_score'}
              }
  },
 {$match: { bookcount: { $gte: 50 } } },
 {$project: {avg_price: 1, review: { $floor: '$review' }}},
] )
```

round down the review score

Result - `$project`

```
{ "_id" : "UK", "avg_price" : 18.75, "review": 4}
{ "_id" : "IT", "avg_price" : 22.49, "review" :  3}
...
```

$limit

```
db.book.aggregate( [
  { $unwind: '$price' } ,
  { $group: { _id: '$price.country',
                avg_price: { $avg: '$price.v' },
                bookcount: {$sum:1},
                review: {$avg: '$review_score'}
              }
  },
 {$match: { bookcount: { $gte: 50 } } },
 {$project: {avg_price: 1, review: { $floor: '$review' }}},
 {$limit:20}
] )
```

Limit the results to the first 20 documents

Example 2

- Compute the 95 percentile of the number of pages,
- only for the books that contain the tag “guide”.

$match

```
db.book.aggregate( [
   {$match: { tag : "guide" } }
] )
```

select documents containing “guide” in the tag array, compare with tag:[“guide”]

Result - `$match`

{ "_id" : ObjectId("5fb29b175b99900c3fa24293"), "title" : " Developing with Python", "tag" : ["python", "guide", "programming"], "n" : 352, "review_score" : 4.6, "price" : [{ "v" : 24.99, "c" : "€", "country" : "IT" }, { "v" : 19.49, "c" : "£", "country" : "UK" }], "author" : { "_id" : 1, "name" : "John", "surname" : "Black" } }

{ "_id" : ObjectId("5fb29ae15b99900c3fa24292"), "title" : "MongoDb guide", "tag" : ["mongodb", "guide", "database"], "n" : 100, "review_score" : 4.3, "price" : [{ "v" : 19.99, "c" : "€", "country" : "IT" }, { "v" : 18, "c" : "£", "country" : "UK" }], "author" : { "_id" : 1, "name" : "Mario", "surname" : "Rossi" } }

…

$sort

```
db.book.aggregate( [
 {$match: { tag : "guide"} },
{$sort : { n: 1} }
] )
```

sort the documents in ascending order according to the value of the **n** field, which stores the number of pages of each book

Result - `$sort`

{ "_id" : ObjectId("5fb29ae15b99900c3fa24292"), "title" : "MongoDb guide", "tag" : ["mongodb", "guide", "database"], **"n" : 100**, "review_score" : 4.3, "price" : [{ "v" : 19.99, "c" : "€", "country" : "IT" }, { "v" : 18, "c" : "£", "country" : "UK" }], "author" : { "_id" : 1, "name" : "Mario", "surname" : "Rossi" } }

{ "_id" : ObjectId("5fb29b175b99900c3fa24293"), "title" : " Developing with Python", "tag" : ["python", "guide", "programming"], **"n" : 352**, "review_score" : 4.6, "price" : [{ "v" : 24.99, "c" : "€", "country" : "IT" }, { "v" : 19.49, "c" : "£", "country" : "UK" }], "author" : { "_id" : 1, "name" : "John", "surname" : "Black" } }

…

$group + $push

```
db.book.aggregate( [
  {$match: { tag : "guide"} },
  {$sort : { n: 1} },
  {$group: {_id:null, value: {$push: "$n"}}}
] )
```

group all the records together inside a single document (**_id:null**), which contains an array with all the values of **n** of all the records

Result - `$group + $push`

```
{ "_id": null, "value": [100, 352, ...]}
```

$project + $arrayElemAt

```
db.book.aggregate( [
 {$match: { tag : "guide"} },
 {$sort : { n: 1} },
 {$group: {_id:null, value: {$push: "$n"}}},
 {$project:
            {"n95p": {$arrayElemAt:
                       ["$value",
                       {$floor: {$multiply: [0.95, {$size: "$value"}]}}
                      ]
            } }
}
] )
```

get the value of the array at a given index
with { $arrayElemAt: [<array>, <idx>] }

compute the index at 95% of the array length

Result - `$project + $arrayElemAt`

```
{ "_id" : null, "n95p" : 420 }
```

Example 3

- Compute the median of the review_score,
- only for the books having at least a price whose value is higher than 20.0.

Solution

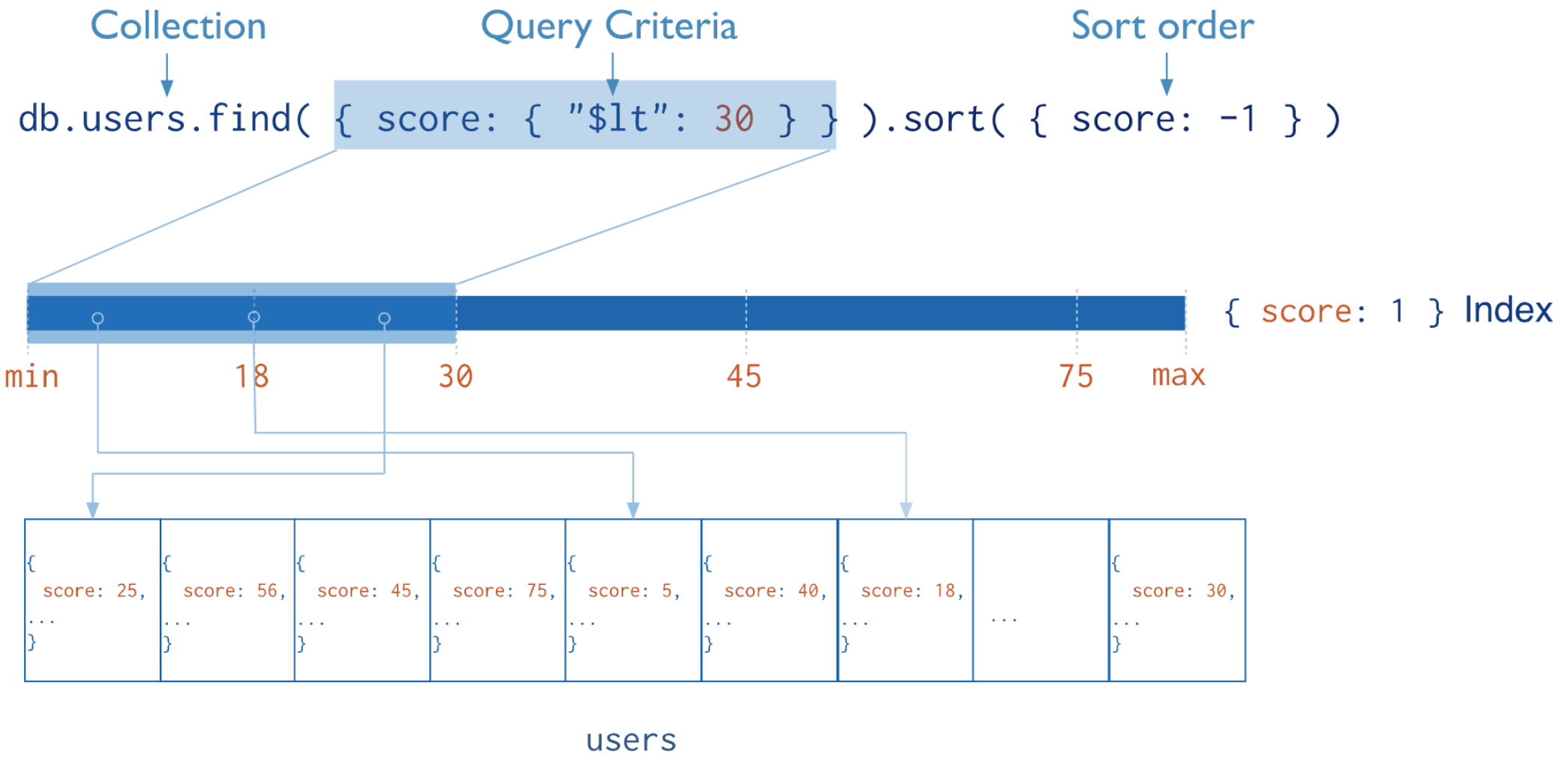
```
db.book.aggregate( [
 {$match: {'price.v' : { $gt: 20 }} },
 {$sort : {review_score: 1} },
 {$group: {_id:null, rsList: {$push: '$review_score'}}},
 {$project:
            {'median': {$arrayElemAt:
                        ['$rsList',
                        {$floor: {$multiply: [0.5, {$size: '$rsList'}]}}
                    ]
            } }
}
] )
```

Indexing

Indexes

- Without indexes, MongoDB must perform a **collection scan**, i.e. scan every document in a collection, to select those documents that match the query statement.
- Indexes are data structures that store a small portion of the collection's data set in a form easy to traverse.
- They store **ordered values of a specific field**, or set of fields, in order to efficiently support
  - equality matches,
  - range-based queries and
  - sorting operations.

Indexes


Collection
Query Criteria
Sort order
db.users.find({ score: { "$lt": 30 } }).sort({ score: -1 })
{ score: 1 } Index
min
18
30
45
75
max
{ score: 25, ... }
{ score: 56, ... }
{ score: 45, ... }
{ score: 75, ... }
{ score: 5, ... }
{ score: 40, ... }
{ score: 18, ... }
...
{ score: 30, ... }
users

Indexes

- MongoDB creates a unique index on the **_id** field during the creation of a collection.
- The **_id** index prevents clients from inserting two documents with the same value for the **_id** field.
- You cannot drop this index on the **_id** field.

Create new indexes

- Creating an index

```
db.collection.createIndex(<index keys>, <options>)
```

  - Before v. 3.0 use `db.collection.ensureIndex()`

- Options include:
  - `name` - a mnemonic name given by the user, you cannot rename an index once created, instead, you must drop and re-create the index with a new name
  - `unique` - whether to accept or not insertion of documents with duplicate keys,
  - `background`, `dropDups`, …

Indexes

- MongoDB provides different data-type indexes
  - Single field indexes
  - Compound field indexes
  - Multikey indexes (to index the content stored in **arrays**, MongoDB creates separate index entries for every element of the array)
  - Geospatial indexes (2d indexes with **planar** and 2dsphere with **spherical** geometry)
  - Text indexes (searching for **string** content in a collection, they do not store language-specific stop words, e.g., "the", "a", "or‟, and stem the words in a collection to only store root words
  - Hashed indexes (indexes the hash of the value of a field, they have a more random distribution of values along their range, but only support equality matches and cannot support range-based queries)

Indexes

- Single field indexes
  - Support user-defined ascending/descending indexes on a single field of a document
- E.g.,
  - `db.orders.createIndex( {orderDate: 1} )`
- Compound field indexes
  - Support user-defined indexes on a set of fields
- E.g.,
  - `db.orders.createIndex( {orderDate: 1,  zipcode: -1} )`

Indexes

- MongoDB supports efficient queries of geospatial data
- Geospatial data are stored as:
  - GeoJSON objects: embedded document { <type>, <coordinate> }
    - E.g., location: `{type: "Point", coordinates: [-73.856, 40.848]}`
  - Legacy coordinate pairs: array or embedded document
    - `point: [-73.856, 40.848]`
- Fields with 2dsphere indexes must hold geometry data in the form of coordinate pairs or GeoJSON data.
  - If you attempt to insert a document with non-geometry data in a 2dsphere indexed field, or build a 2dsphere index on a collection where the indexed field has non-geometry data, the operation will fail.

Indexes

- Geospatial indexes
  - Two type of geospatial indexes are provided: `2d` and `2dsphere`
- A `2dsphere` index supports queries that calculate geometries on an earth-like sphere
- Use a `2d` index for data stored as points on a two-dimensional plane.
- E.g.,
  - `db.places.createIndex( {location: "2dsphere"} )`
- Geospatial query operators
  - `$geoIntersects`, `$geoWithin`, `$near`, `$nearSphere`

Indexes

- `$near` syntax:

```
{
    <location field>: {
      $near: {
        $geometry: {
           type: "Point" ,
           coordinates: [ <longitude> , <latitude> ]
        },
        $maxDistance: <distance in meters>,
        $minDistance: <distance in meters>
      }
    }
}
```

Indexes

- E.g.,
  - `db.places.createIndex( {location: "2dsphere"} )`
- Geospatial query operators
  - `$geoIntersects`, `$geoWithin`, `$near`, `$nearSphere`
- Geopatial aggregation stage
  - `$near`

Indexes

- E.g.,
  - ```
    db.places.find({location:
        {$near:
            {$geometry: {
               type: "Point",
               coordinates: [ -73.96, 40.78 ] },
              $maxDistance: 5000}
        }})
    ```
  - Find all the places within 5000 meters from the specified GeoJSON point, sorted in order from nearest to furthest
    ```

Indexes

- Text indexes
  - Support efficient searching for string content in a collection
  - Text indexes store only *root words* (no language-specific *stop words* or *stem*)
- E.g.,

```
db.reviews.createIndex( {comment: "text"} )
```

  - Wildcard (`$**`) allows MongoDB to index every field that contains string data
  - E.g.,

```
db.reviews.createIndex( {"$**": "text"} )
```

VIEWS

- A queryable object whose contents are defined by an **aggregation** pipeline on other **collections** or **views**.
- MongoDB does not persist the view contents to disk. A view's content is **computed on-demand**.
- Starting in version 4.2, MongoDB adds the $merge stage for the aggregation pipeline to create on-demand **materialized views**, where the content of the output collection can be updated each time the pipeline is run.
- **Read-only** views from existing collections or other views. E.g.:
  - excludes private or confidential data from a collection of employee data
  - adds computed fields from a collection of metrics
  - joins data from two different related collections

```
db.runCommand( {
  create: <view>, viewOn: <source>, pipeline: <pipeline>, collation: <collation> } )
```

- Restrictions
  - immutable Name
  - you can modify a view either by dropping and recreating the view or using the *collMod* comman

Politecnico di Torino

Distributed Data Management

Introduction to data replication and the CAP theorem

DANIELE APILETTI

POLITECNICO DI TORINO

Replication

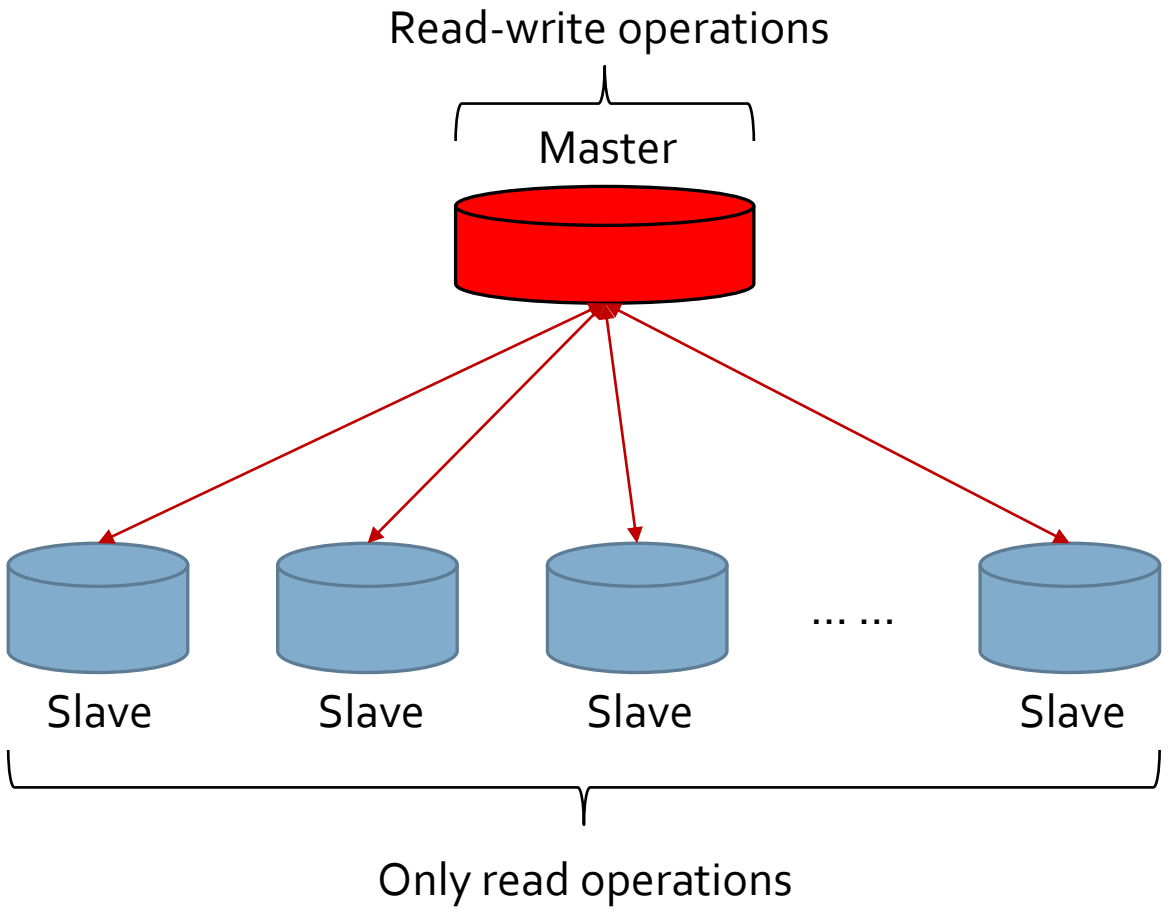
Same data
in **different** places

Replication

- **Same** data
  - portions of the whole dataset (chunks)
- in **different** places
  - local and/or remote servers, clusters, data centers
- Goals
  - Redundancy helps surviving failures (availability)
  - Better performance
- Approaches
  - Master-Slave replication
  - A-Synchronous replication

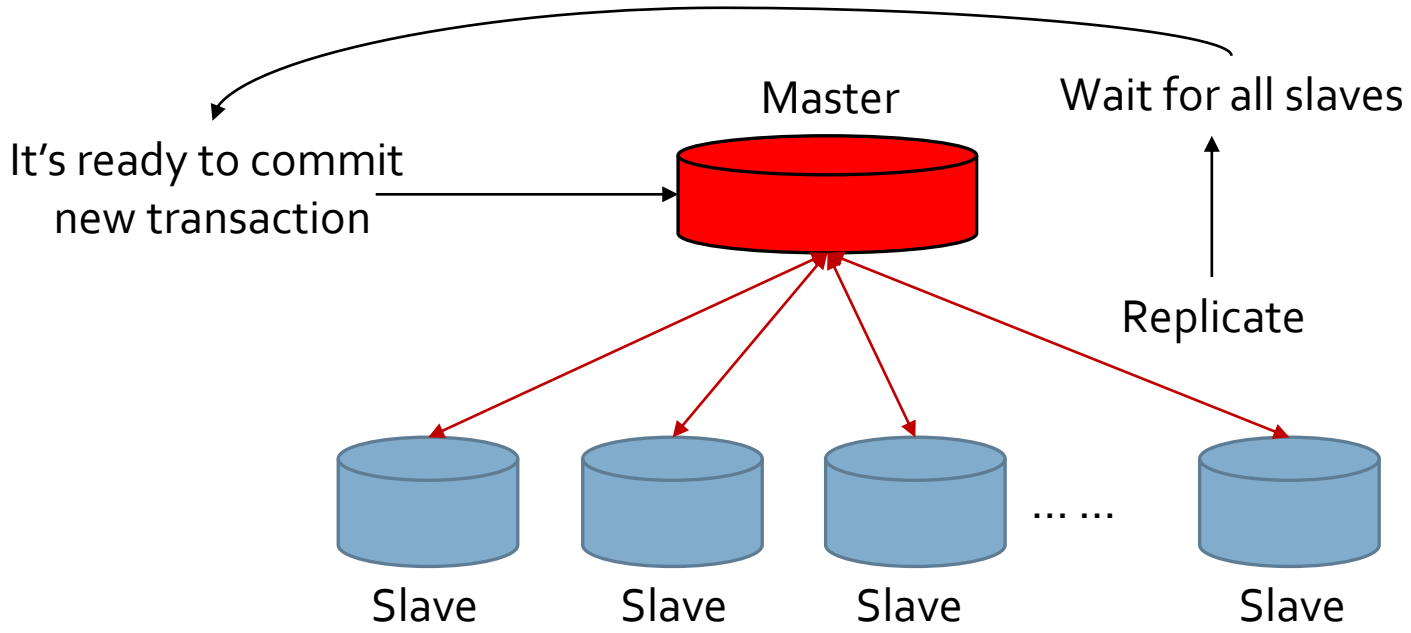
Master-Slave replication

- Master-Slave
  - A **master** server takes all the writes, updates, inserts
  - One or more **Slave** servers take all the reads (they can't write)
  - Only read **scalability**
  - The master is a single point of **failure**
- Some NoSQLs (e.g., CouchDB) support Master-Master replica



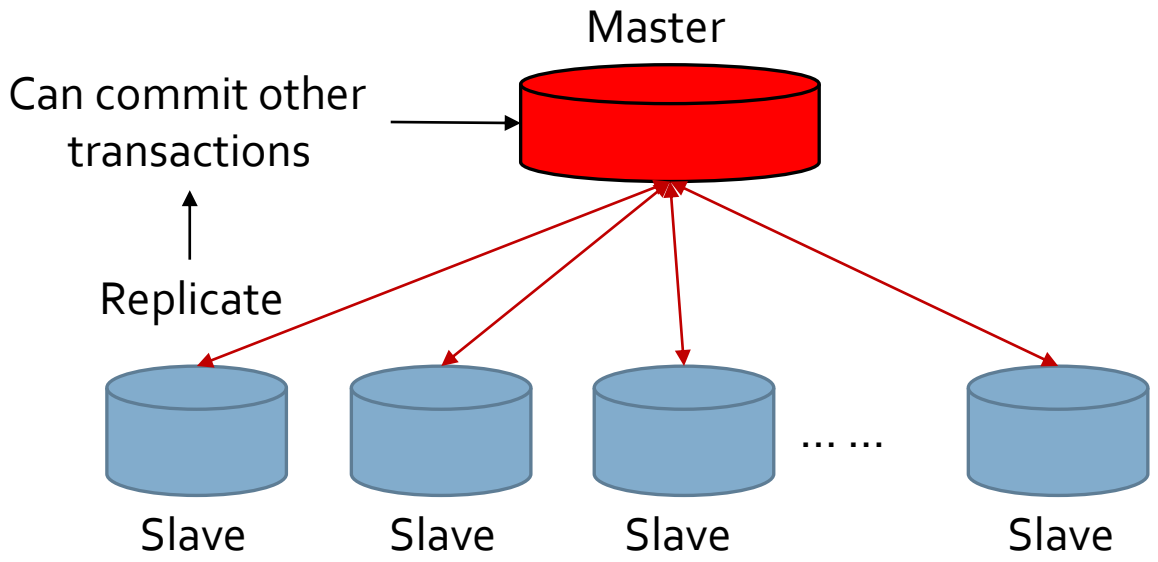
Synchronous replication

- Before committing a transaction, the Master **waits** for (all) the Slaves to commit
- Similar in concept to the **2-Phase Commit** in relational databases
- **Performance** killer, in particular for replication in the cloud
- Trade-off: wait for a subset of Slaves to commit, e.g., the **majority** of them



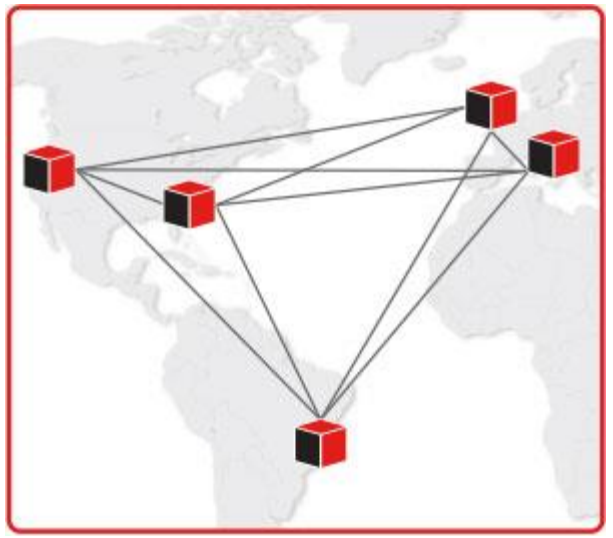
Asynchronous replication

- The Master commits **locally**, it does not wait for any Slave
- Each Slave independently fetches updates from Master, which may **fail**…
  - IF no Slave has replicated, then you've **lost the data** committed to the Master
  - IF some Slaves have replicated and some haven't, then you have to **reconcile**
- Faster and **un**reliable



Distributed databases

Different autonomous machines, working **together** to manage the same **dataset**

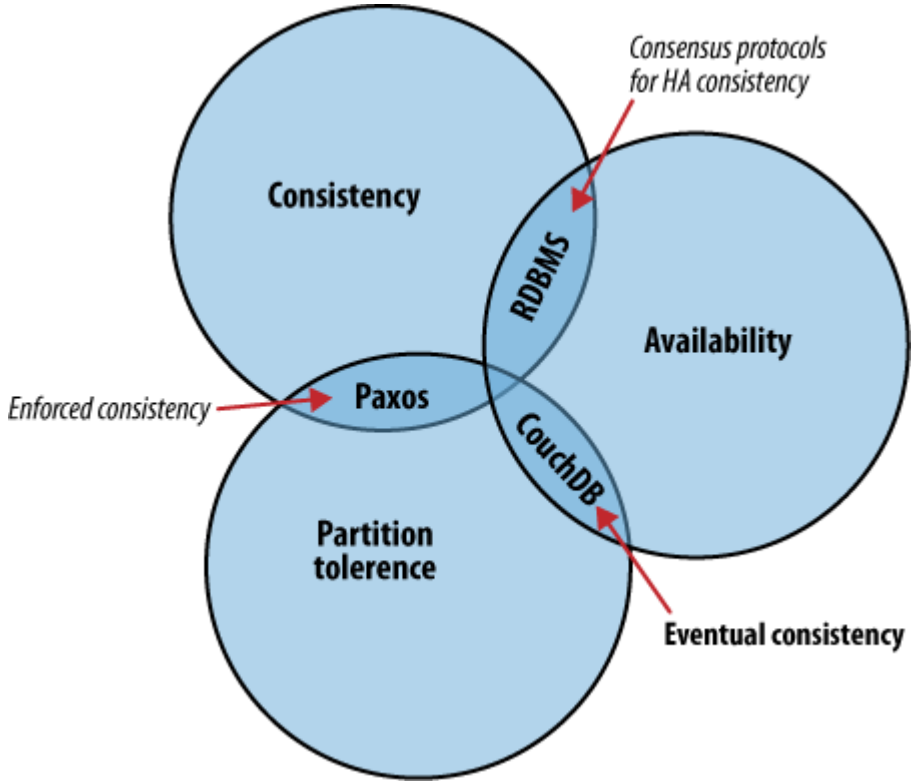


Key features of distributed databases

- There are 3 typical problems in distributed databases:
  - **C**onsistency
    - All the distributed databases provide the same data to the application
  - **A**vailability
    - Database failures (e.g., master node) do not prevent survivors from continuing to operate
  - **P**artition tolerance
    - The system continues to operate despite arbitrary message loss, when connectivity failures cause network partitions

CAP Theorem

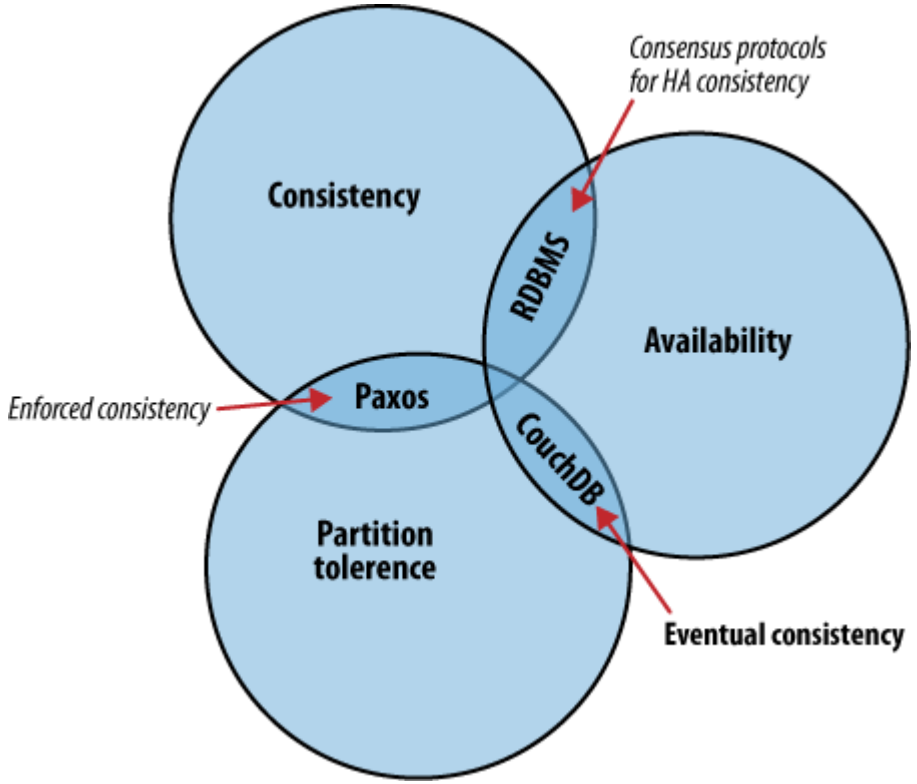
- The CAP theorem, also known as Brewer's theorem, states that it is **impossible** for a distributed system to **simultaneously** provide **all three** of the previous guarantees
- The theorem began as a **conjecture** made by University of California in 1999-2000
  - Armando Fox and Eric Brewer, "Harvest, Yield and Scalable Tolerant Systems", Proc. 7th Workshop Hot Topics in Operating Systems (HotOS 99), IEEE CS, 1999, pg. 174-178.
- In 2002 a formal proof was published, establishing it as a **theorem**
  - Seth Gilbert and Nancy Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services", ACM SIGACT News, Volume 33 Issue 2 (2002), pg. 51-59
- In 2012, a follow-up by Eric Brewer, "CAP twelve years later: How the "rules" have changed"
  - IEEE Explore, Volume 45, Issue 2 (2012), pg. 23-29.



http://guide.couchdb.org/editions/1/en/consistency.html#figure/1

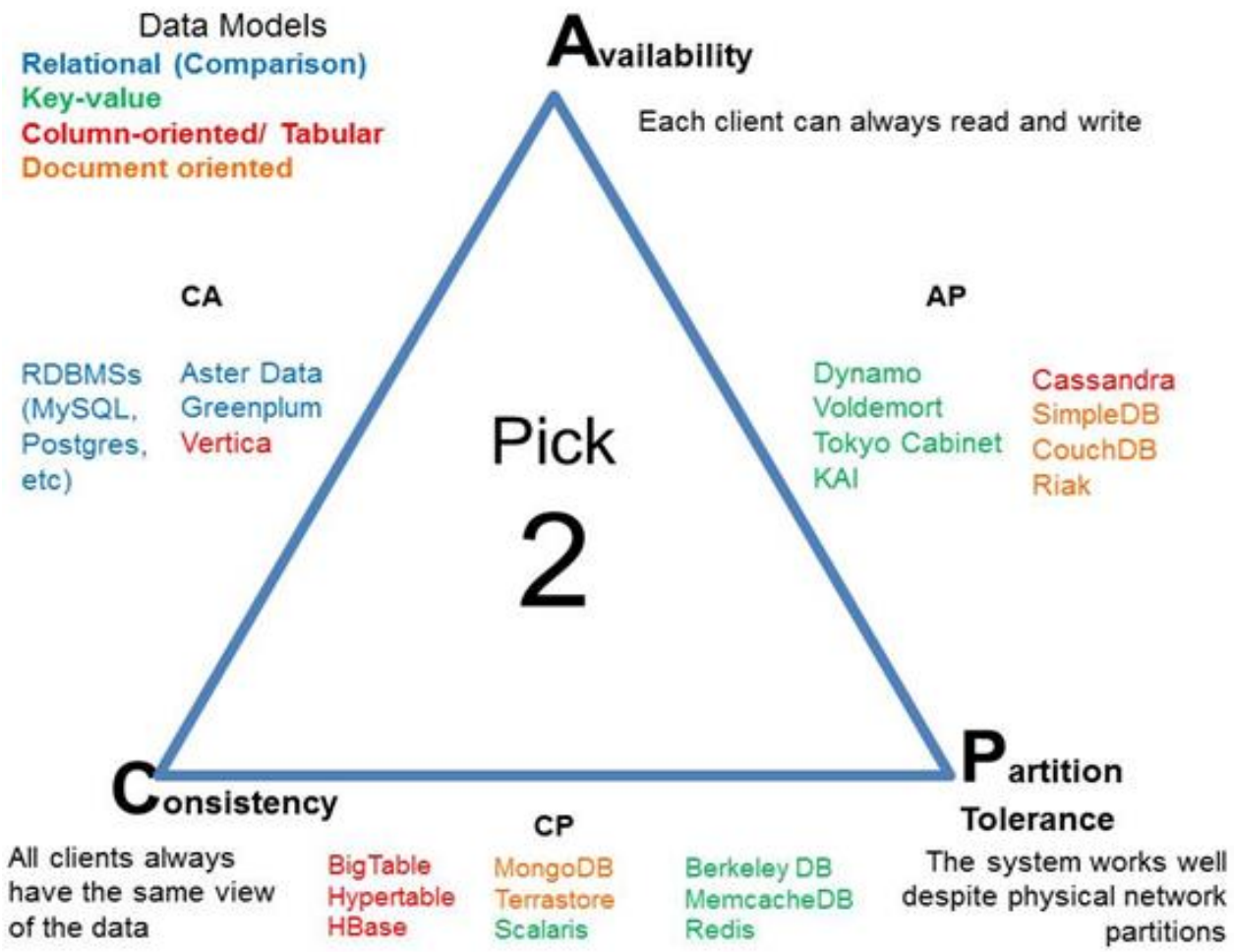
CAP Theorem

- The easiest way to understand CAP is to think of **two nodes** on opposite sides of a **partition**.
- Allowing at least one node to update state will cause the nodes to become **inconsistent**, thus forfeiting C.
- If the choice is to preserve consistency, one side of the partition must act as if it is **unavailable**, thus forfeiting A.
- Only when no network **partition** exists, is it possible to preserve both consistency and availability, thereby forfeiting P.
- The general belief is that for wide-area systems, **designers cannot forfeit P** and therefore have a difficult choice between C and A.



http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

CAP Theorem



http://blog.fluz7.com/blogs/nosql/cap-theorem-why-does-it-matter

CA without P (local consistency)

- **Partitioning** (communication breakdown) causes a failure.
- We can still have **Consistency** and **Availability** of the data shared by agents **within each Partition**, by ignoring other partitions.
  - Local rather than global consistency / availability
- Local consistency for a partial system, 100% availability for the partial system, and no partitioning does not exclude several partitions from existing with their own "internal" CA.
- So partitioning means having **multiple independent systems** with 100% CA that do not need to interact.

CP without A (transaction locking)

- A system is allowed to *not* answer requests at all (turn off "A").
- We claim to tolerate **partitioning/faults**, because we simply block all responses if a partition occurs, assuming that we cannot continue to function correctly without the data on the other side of a partition.
- Once the partition is healed and **consistency** can once again be verified, we can restore availability and leave this mode.
- In this configuration there are global consistency, and global correct behaviour in partitioning is to **block access to replica sets** that are not in synch.
- In order to tolerate P at any time, we must sacrifice A at any time for **global consistency**.
- This is basically the **transaction lock**.

AP without C (best effort)

- If we don't care about **global consistency** (i.e. simultaneity), then every part of the system can make available what it knows.
- Each part might be able to answer someone, even though the system as a whole has been broken up into incommunicable regions (**partitions**).
- In this configuration "without consistency" means without the assurance of **global** consistency **at all times**.

A consequence of CAP

"Each node in a system should be able to make decisions purely based on **local state**. If you need to do something under high load with **failures** occurring and you need to reach agreement, you're lost. If you're concerned about **scalability**, any algorithm that forces you to run agreement will eventually become your **bottleneck**. Take that as a given."

Werner Vogels, Amazon CTO and Vice President

Beyond CAP

- The "2 of 3" view is misleading on several fronts.
- First, because **partitions** are rare, there is little reason to forfeit C or A when the system is not partitioned.
- Second, the **choice between C and A** can occur many times within the same system at very fine granularity; not only can subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved.
- Finally, all three **properties are more continuous than binary**.
  - Availability is obviously continuous from 0 to 100 percent
  - There are also many levels of consistency
  - Even partitions have nuances, including disagreement within the system about whether a partition exists

How the rules have changed

- Any networked shared-data system can have **only 2 of 3** desirable properties at the **same time**
- Explicitly handling partitions, designers can optimize consistency and availability, thereby achieving some **trade-off of all three**
- CAP prohibits only a tiny part of the design space:
  - **perfect** availability (A) and consistency (C)
  - in the presence of partitions (P), which are **rare**
- Although designers need to choose between consistency and availability when partitions are present, there is an incredible range of **flexibility for handling partitions** and recovering from them
- Modern CAP goal should be to maximize combinations of consistency (C) and availability (A) that make sense for the **specific application**

ACID

- The four ACID properties are:
  - **Atomicity (A)** All systems benefit from atomic operations, the database transaction must completely succeed or fail, partial success is not allowed
  - **Consistency (C)** During the database transaction, the database progresses from a valid state to another. In ACID, the C means that a transaction pre-serves all the database rules, such as unique keys. In contrast, the C in CAP refers only to single copy consistency.
  - **Isolation (I)** Isolation is at the core of the CAP theorem: if the system requires ACID isolation, it can operate on at most one side during a partition, because a client's transaction must be isolated from other client's transaction
  - **Durability (D)** The results of applying a transaction are permanent, it must persist after the transaction completes, even in the presence of failures.

BASE

- **Basically Available**: the system provides availability, in terms of the CAP theorem
- **Soft state:** indicates that the state of the system may change over time, even without input, because of the eventual consistency model.
- **Eventual consistency:** indicates that the system will become consistent over time, given that the system doesn't receive input during that time
- Example: DNS – Domain Name Servers
  - DNS is not multi-master

ACID versus BASE

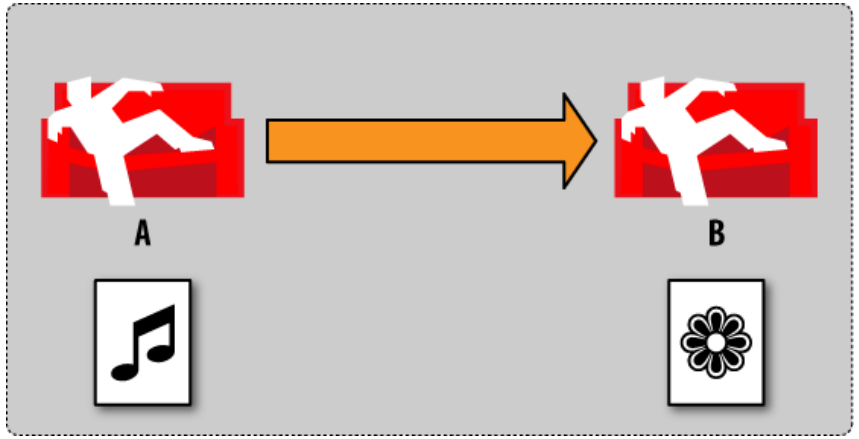
- ACID and BASE represent two design philosophies at opposite ends of the consistency-availability spectrum
- ACID properties focus on **consistency** and are the traditional approach of databases
- BASE properties focus on high **availability** and to make explicit both the choice and the spectrum
- **BASE**: Basically Available, Soft state, Eventually consistent, work well in the presence of **partitions** and thus promote **availability**

Conflict detection and resolution

An example from a notable NoSQL database

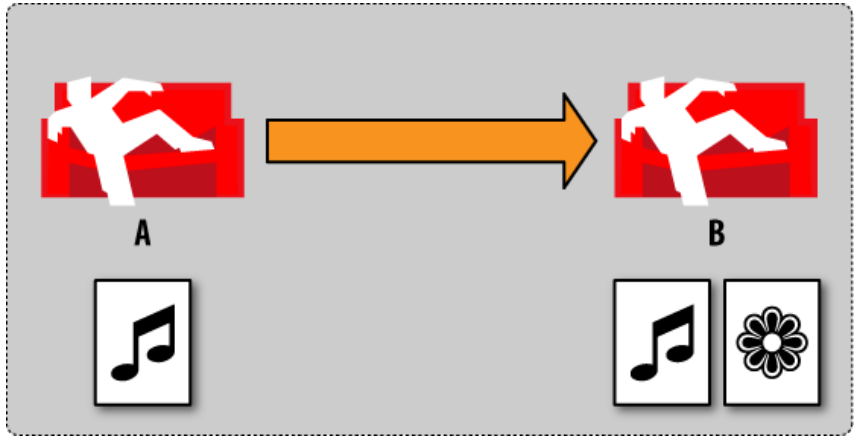
Conflict resolution problem

- There are two customers, **A** and **B**
- **A** books a hotel room, the last available room
- **B** does the same, on a different node of the system, which was **not consistent**



Conflict resolution problem

- The hotel room document is affected by two **conflicting updates**
- Applications should solve the conflict with custom logic (it's a business decision)
- The database can
  - **Detect** the conflict
  - Provide a local **solution**, e.g., latest version is saved as the winning version



Conflict

- CouchDB guarantees that **each instance** that sees the **same conflict** comes up with the **same winning** and losing **revisions**.
- It does so by running a **deterministic algorithm** to pick the winner.
  - The revision with the longest revision history list becomes the winning revision.
  - If they are the same, the **_rev** values are compared in ASCII sort order, and the highest wins.

o

Replication in MongoDB

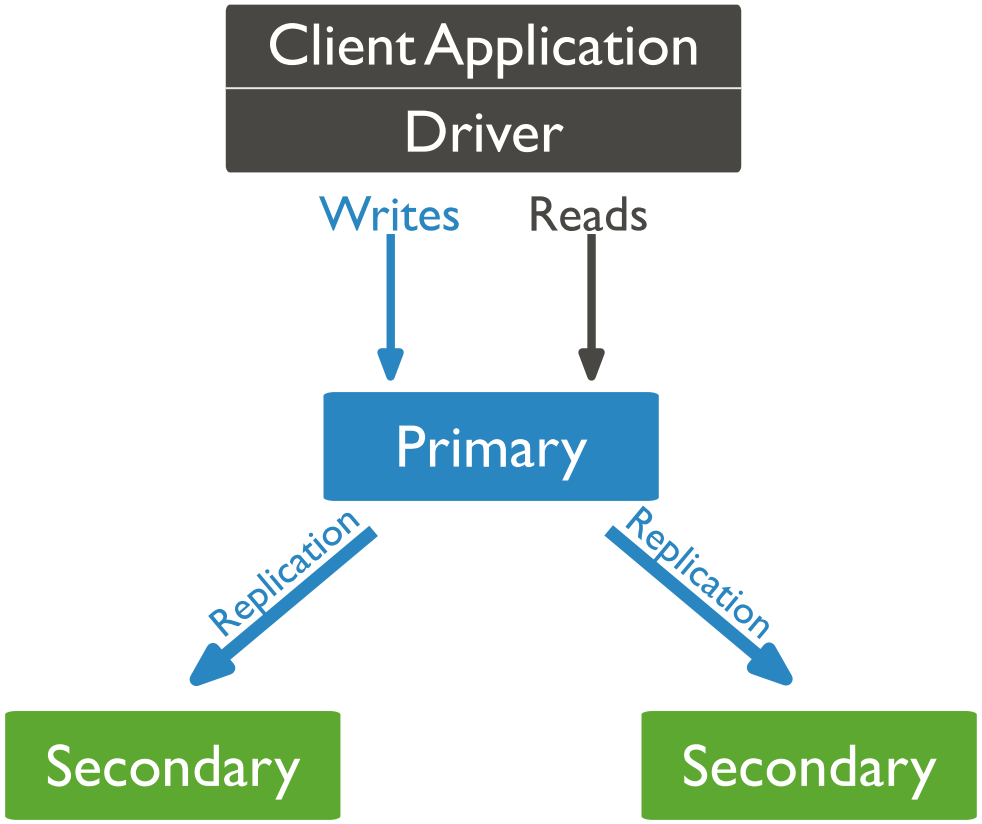
DANIELE APILETTI

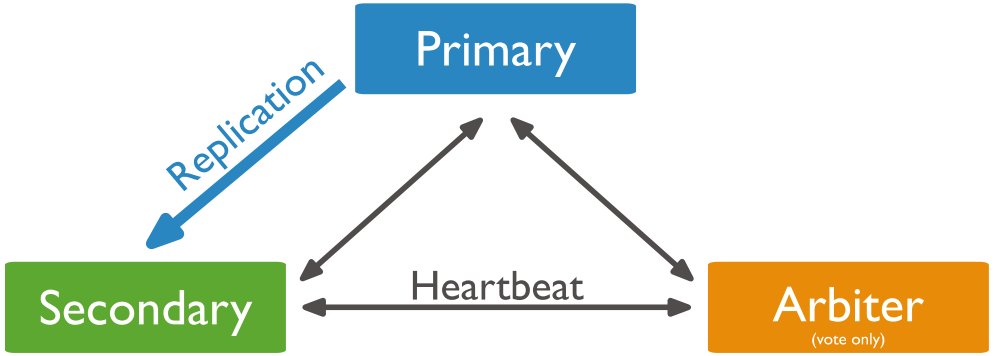
POLITECNICO DI TORINO

Key Concepts

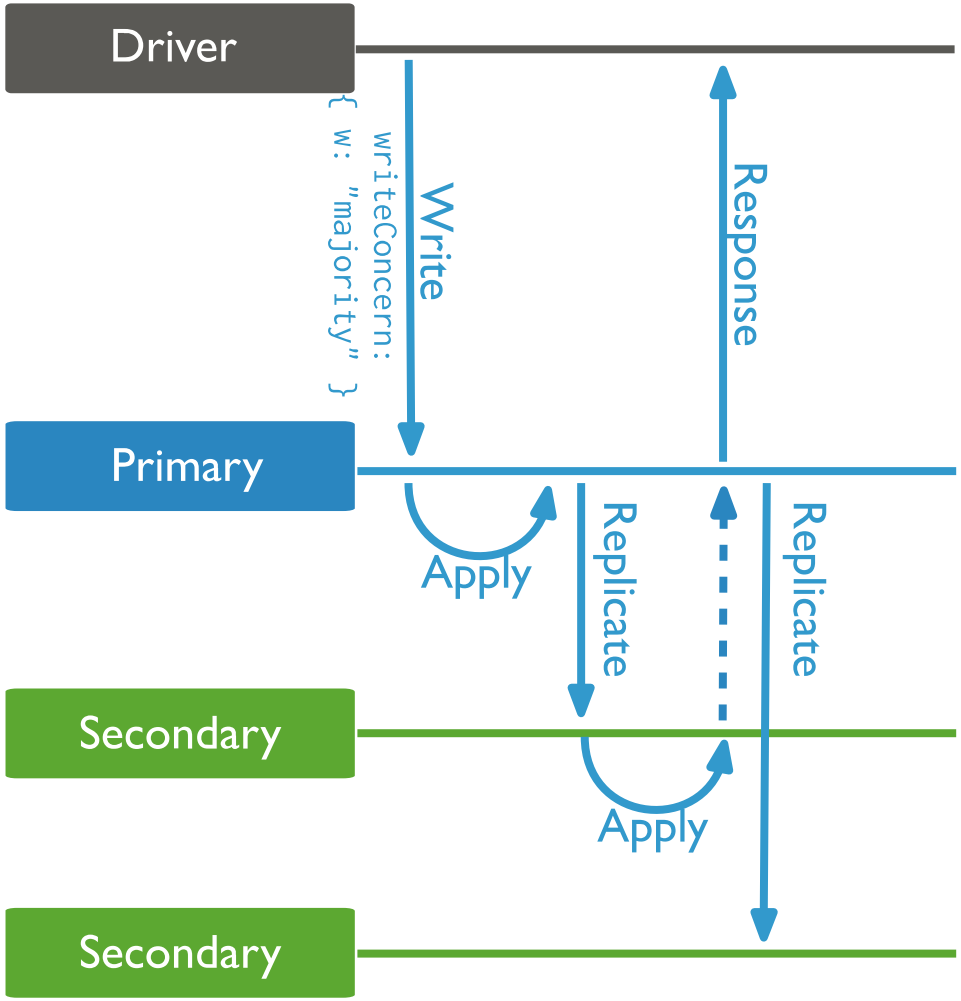
- A replica set is a group of *mongod* instances that maintain the same data set (rendundancy):
  - 1 **primary** node, and **only one**
  - **secondary** nodes (all the other nodes containing a copy of the **data**)
  - 1 **arbiter** (optional)
- Primary node
  - receives all **write** operations
  - confirming writes with *{ w: "majority" }* **write concern,** i.e., the number of data-bearing members (primary and secondaries, but not arbiters) that must acknowledge a write operation before the operation returns as successful
- Secondary node
  - replicates the **primary's oplog** and apply the operations to their data sets
  - if the primary is unavailable, an eligible secondary will hold an **election** to elect itself the new primary
  - secondaries may have additional configurations for special usage profiles. For example, secondaries may be **non-voting** or with different **priority** levels
- Arbiters
  - does **not** hold **data**! (will always be an arbiter, cannot be elected primary)
  - maintains a quorum in a replica set by responding **election** requests by other replica set members (and keeping the heartbeat)

Architecture





Write concern



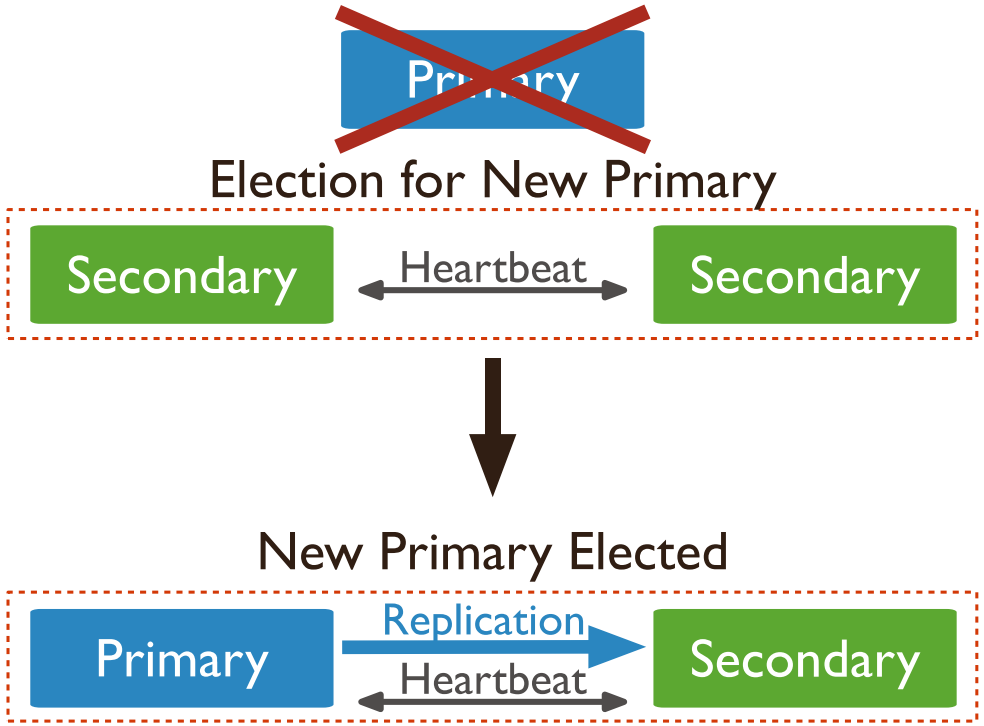
For write concern of
- w **>1** or
- w : "**majority**"

the primary waits until the required number of secondaries acknowledge the write before returning write concern acknowledgment.

For write concern of **w: 1,** the primary can return write concern acknowledgment as soon as it locally applies the write.

Automatic Failover

- When a primary does not communicate with the other members of the set for more than the configured *electionTimeoutMillis* period (**10 seconds** by default)
- The replica set cannot process **write** operations until the election completes successfully
- The replica set can continue to serve **read** queries if such queries are configured to run **on secondaries** while the primary is offline
- The median time for primary election should not typically exceed **12 seconds**

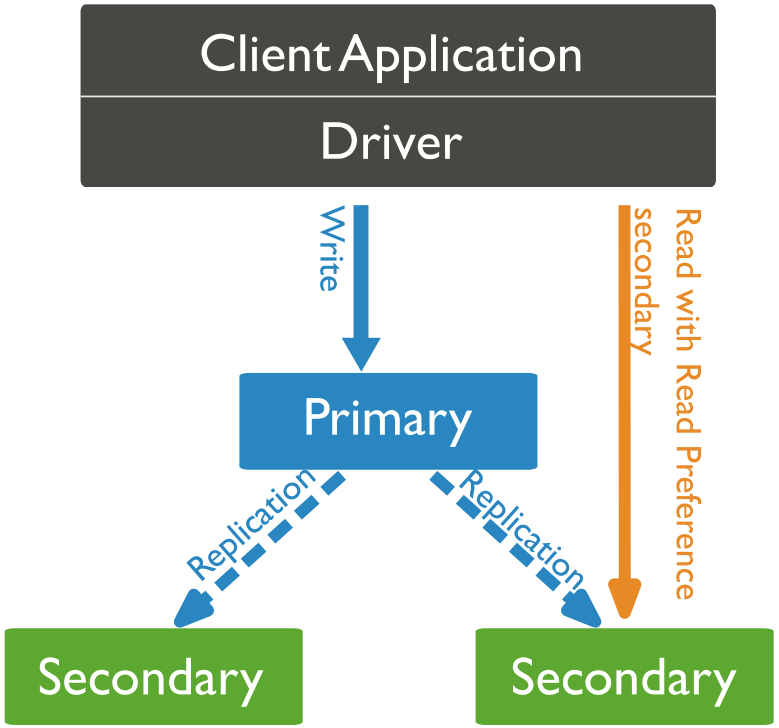


Fault tolerance

Number of Members	Majority Required to Elect a New Primary	Fault Tolerance
3	2	1
4	3	1
5	3	2
6	4	2

Read Operations

- By default, clients read from the primary
- **Asynchronous replication** to secondaries means that reads from secondaries may return data that does not reflect the state of the data on the primary
- Multi-document **transactions** that contain read operations must use read preference primary. All operations in a given transaction must route to the same member
- Until a transaction commits, the data changes made in the transaction are not visible outside the transaction



Deploy a replica set

- Three-member replica sets provide enough redundancy to survive most network partitions and other system failures
- These sets also have sufficient capacity for many distributed read operations
- Replica sets should always have an odd number of members to ensure that elections will proceed smoothly
- Maintain as much separation between members as possible by hosting the mongod instances on separate machines
- Place each mongod instance on a separate host server serviced by redundant power circuits and redundant network paths
- Install MongoDB on each system that will be part of your replica set

Considerations

- Architecture
  - deploy each member to its own machine
  - if possible, bind to the standard port *27017*
- Hostnames
  - use a logical DNS hostname instead of an ip address
- IP Binding
  - use the *bind_ip* option to ensure that MongoDB listens for connections from applications on configured addresses
  - *mongod --bind_ip localhost,My-Hostname*
- *Connectivity*
  - establish a virtual private network
  - configure access control
  - configure networking and firewall rules

Members Configuration

- Set *replication.replSetName* option to the replica set name
- Set *net.bindIp* option to the hostname/ip
- Set any other settings as appropriate for your deployment

Replica Set Member	Hostname
Member 0	mongodbo.example.net
Member 1	mongodb1.example.net
Member 2	mongodb2.example.net

```
mongod --replSet "rso" --bind_ip localhost,<hostname(s) | ip address(es)>
```

Deploy a Replica Set for Testing (1)

- Create the necessary data directories for each member

```
mkdir -p /srv/mongodb/rs0-0 /srv/mongodb/rs0-1 /srv/mongodb/rs0-2
```

- Start your mongod instances in their own shell windows

```
1) mongod --replSet rs0 --port 27017 --bind_ip localhost,<hostname(s)|ip address(es)> --dbpath /srv/mongodb/rs0-0  --oplogSize 128
2) mongod --replSet rs0 --port 27018 --bind_ip localhost,<hostname(s)|ip address(es)> --dbpath /srv/mongodb/rs0-1  --oplogSize 128
3) mongod --replSet rs0 --port 27019 --bind_ip localhost,<hostname(s)|ip address(es)> --dbpath /srv/mongodb/rs0-2 --oplogSize 128
```

Deploy a Replica Set for Testing (2)

- Connect to one of your mongod instances through the mongo shell

```
mongo --port 27017
```

- Initiate the replica set

```
rsconf = {
                _id: "rso",
                members: [
                                { _id: 0, host: "<hostname>:27017" },
                                { _id: 1, host: "<hostname>:27018" },
                                { _id: 2, host: "<hostname>:27019" }
                ]
}
rs.initiate( rsconf )
```

Add members

- Start the new mongod instance

```
mongod --dbpath /srv/mongodb/dbo --replSet rso  --bind_ip localhost,<hostname(s)|ip address(es)>
```

- Connect to the replica set's primary
- Add the new member to the replica set

```
rs.add( { host: "mongodb3.example.net:27017", priority: 0, votes: 0 } )
```

- Ensure that the new member has reached *SECONDARY* state
- Update the newly added member's priority and votes if needed

```
var cfg = rs.conf();
cfg.members[4].priority = 1
cfg.members[4].votes = 1
rs.reconfig(cfg)
```

Remove members

- Shut down the mongod instance for the member you wish to remove
- Connect to the replica set's current primary
- Use *rs.remove()*

```
rs.remove("mongod3.example.net:27017")
rs.remove("mongod3.example.net")
```

Verify replica set

- View the replica set configuration

```
rs.conf()
```

- Ensure that the replica set has a primary

```
rs.status()
```

Configuration info

```
{ "_id" : "rso",
"version" : 1,
"protocolVersion" : NumberLong(1),
"members" : [
                { "_id" : 0, "host" :
                "mongodbo.example.net:27017",
                "arbiterOnly" : false,
                "buildIndexes" : true,
                "hidden" : false,
                "priority" : 1, // higher values to make a member more eligible to become primary, zero is ineligible to become primary.
                "tags" : { },
                "slaveDelay" : NumberLong(0),
                "votes" : 1 },
                 ... ],
"settings" : {
                 "chainingAllowed" : true,
                "heartbeatIntervalMillis" : 2000,
                "heartbeatTimeoutSecs" : 10,
                "electionTimeoutMillis" : 10000,
                "catchUpTimeoutMillis" : -1,
                 "getLastErrorModes" : { },
                "getLastErrorDefaults" : { "w" : 1, "wtimeout" : 0 },
                "replicaSetId" : ObjectId("585ab9df685f726db2c6a840")
} }
```

Politecnico di Torino

Master course

DBMG

Distributed transactions in relational databases

DANIELE APILETTI

POLITECNICO DI TORINO

ACID properties

- Atomicity
  - It requires distributed techniques
    - 2 phase commit
- Consistency
  - Constraints are currently enforced only locally
- Isolation
  - It requires strict 2PL and 2 Phase Commit
- Durability
  - It requires the extension of local procedures to manage atomicity in presence of failure

Other issues

- *Distributed query optimization* is performed by the DBMS receiving the query execution request
  - It partitions the query in subqueries, each addressed to a single DBMS
  - It selects the execution strategy
    - order of operations and execution technique
    - order of operations on different nodes
      - transmission cost may become relevant
    - (optionally) selection of the appropriate replica
  - It coordinates operations on different nodes and information exchange

Atomicity

- All nodes (i.e., DBMS servers) participating to a distributed transaction must implement the same decision (commit or rollback)
  - Coordinated by 2 phase commit protocol
- Failure causes
  - Node failure
  - Network failure which causes lost messages
    - Acknowledgement of messages (ack)
    - Usage of timeout
  - Network partitioning in separate subnetworks

2 Phase Commit protocol

- Objective
  - Coordination of the conclusion of a distributed transaction
- Parallel with a wedding
  - Priest celebrating the wedding
    - Coordinates the agreement
  - Couple to be married
    - Participate to the agreement

2 Phase Commit protocol

- Distributed transaction
  - One coordinator
    - *Transaction Manager* (TM)
  - Several DBMS servers which take part to the transaction
    - *Resource Managers* (RM)
- Any participant may take the role of TM
  - Also the client requesting the transaction execution

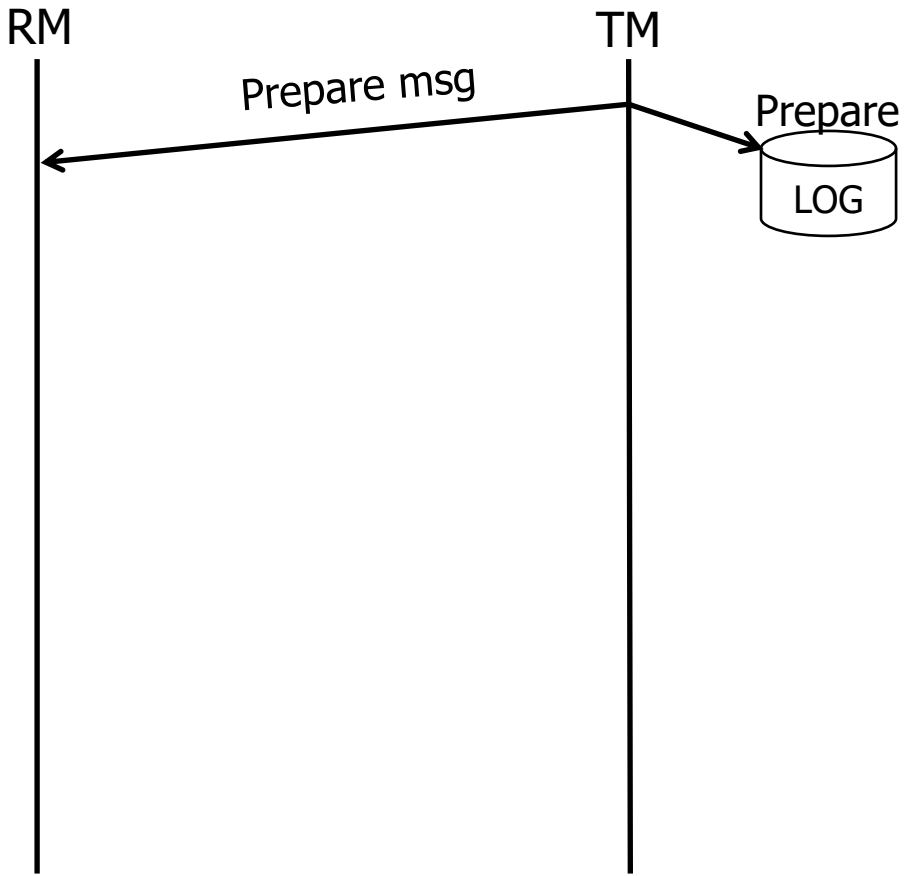
New log records

- TM and RM have *separate private* logs
- Records in the TM log
  - *Prepare*
    - it contains the identity of all RMs participating to the transaction (Node ID + Process ID)
  - *Global commit/abort*
    - final decision on the transaction outcome
  - *Complete*
    - written at the end of the protocol

New log records

- New records in the RM log
  - *Ready*
    - The RM is willing to perform commit of the transaction
    - The decision *cannot be changed* afterwards
    - The node has to be in a reliable state
      - WAL and commit precedence rules are enforced
      - Resources are locked
    - After this point the RM *loses its autonomy* for the current transaction

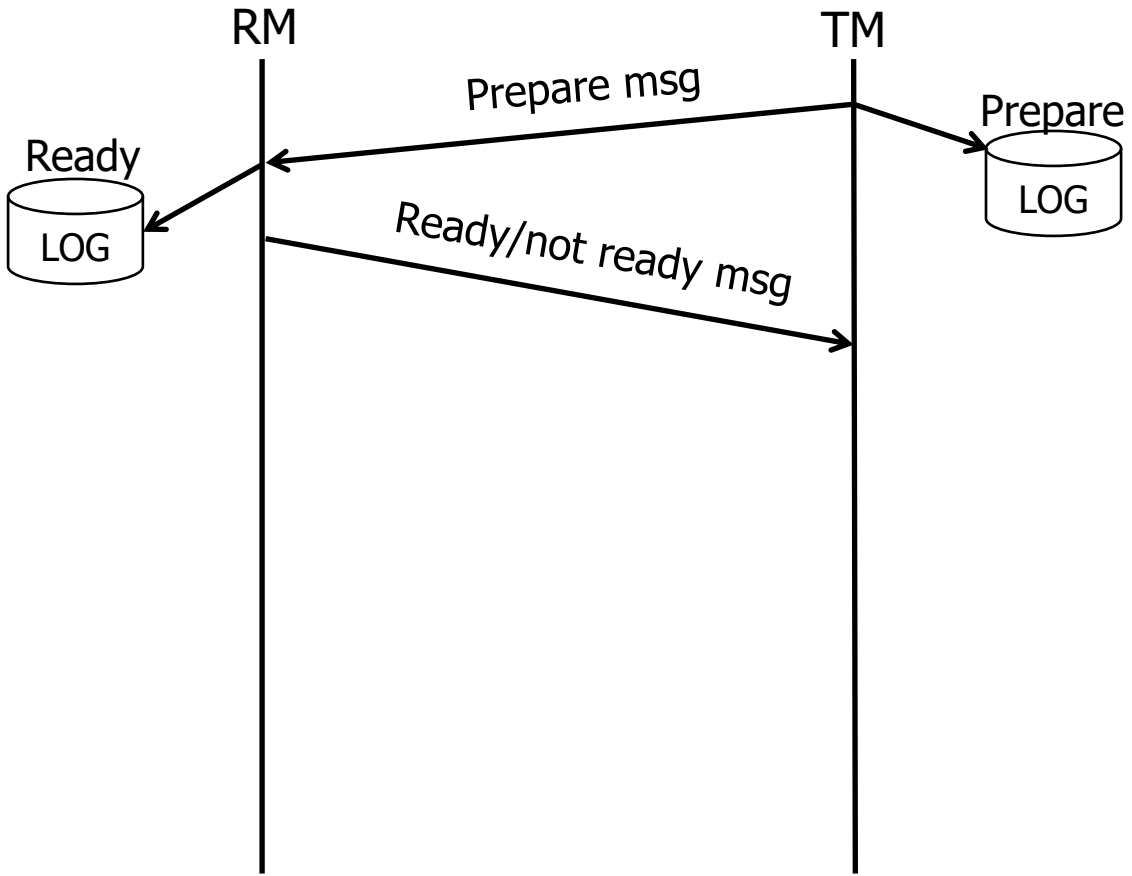
2 Phase Commit protocol



Phase I

1. The TM
   - Writes the prepare record in the log
   - Sends the prepare message to all RM (participants)
   - Sets a timeout, defining maximum waiting time for RM answer

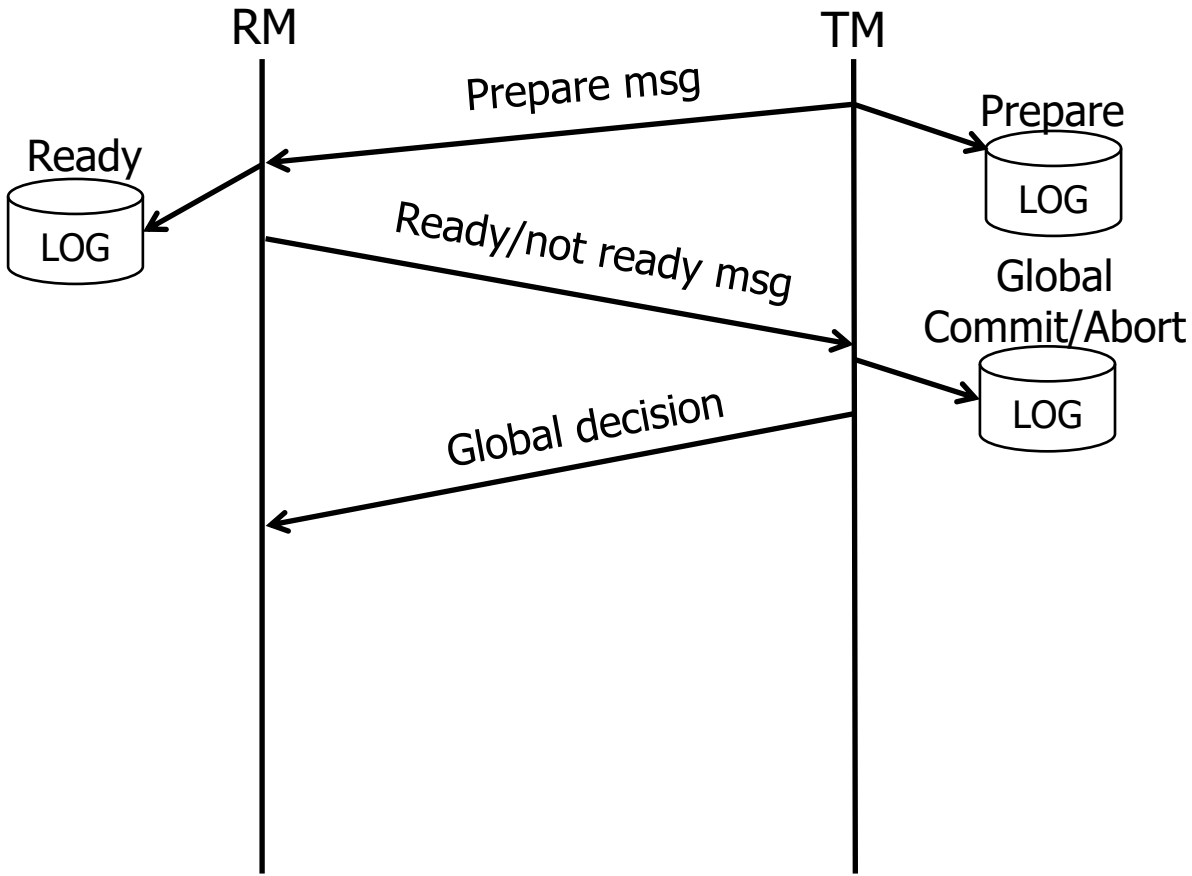
2 Phase Commit protocol



Phase I

- The RMs
  - Wait for the prepare message
  - When they receive it
    - If they are in a reliable state
      - Write the ready record in the log
      - Send the ready message to the TM
    - If they are not in a reliable state
      - Send a not ready message to the TM
      - Terminate the protocol
      - Perform local rollback
    - If the RM crashed
      - No answer is sent

2 Phase Commit protocol



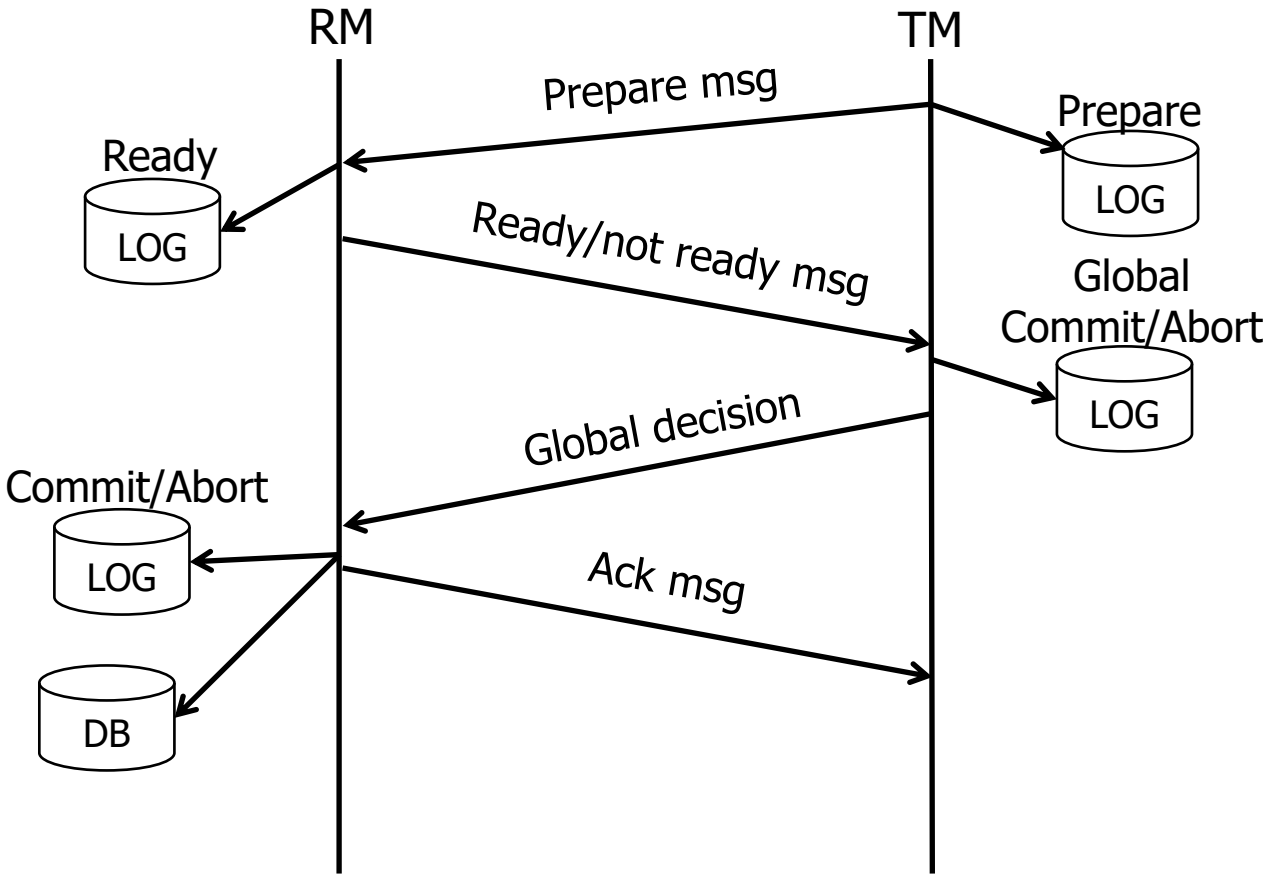
Phase I

3. The TM
   - Collects all incoming messages from the RMs
   - If it receives ready from *all* RMs
     - The commit global decision record is written in the log
   - If it receives one or more not ready or the timeout expires
     - The abort global decision record is written in the log

Phase II

1. The TM
   - Sends the global decision to the RMs
   - Sets a timeout for the RM answers

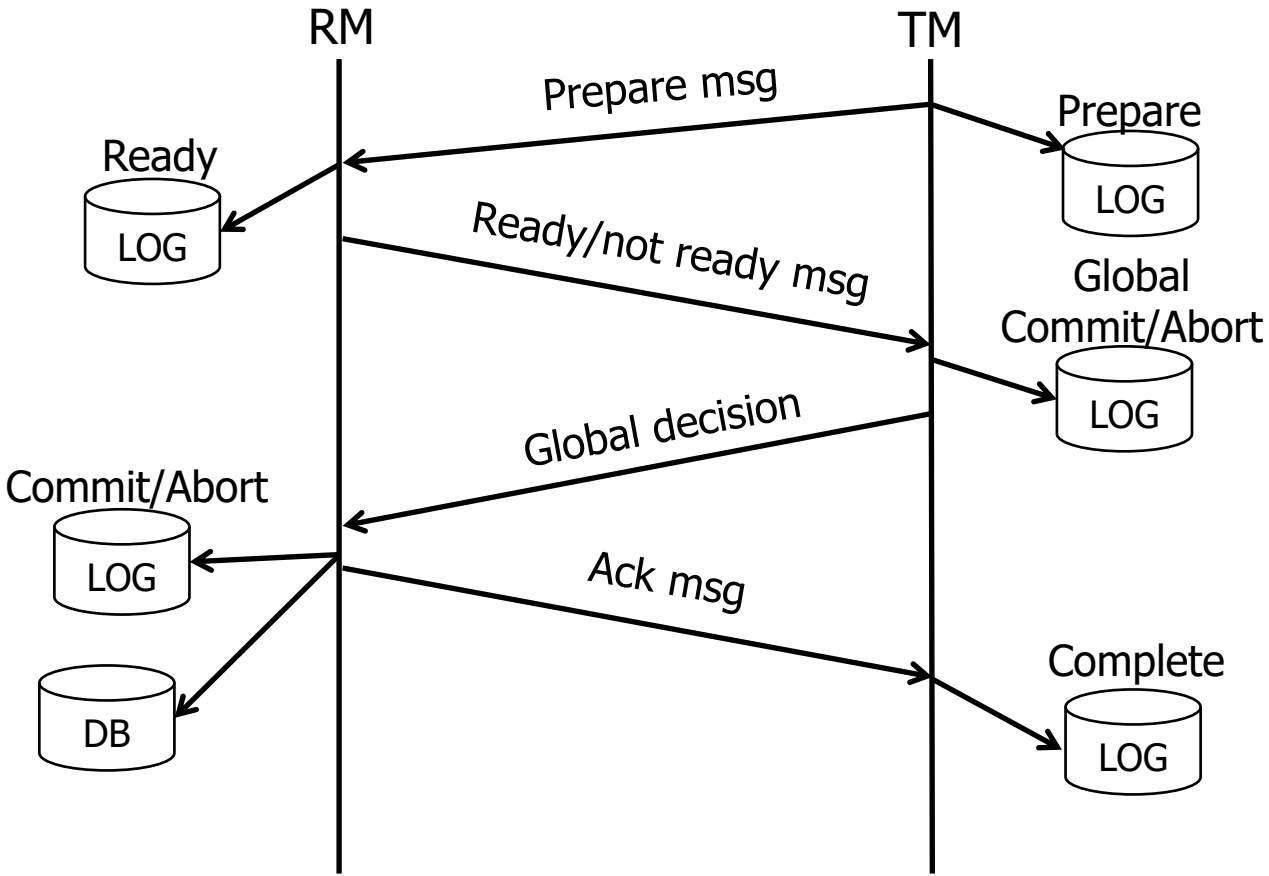
2 Phase Commit protocol



Phase II

2. The RM
   - Waits for the global decision
   - When it receives it
     - The commit/abort record is written in the log
     - The database is updated
     - An ACK message is sent to the TM

2 Phase Commit protocol

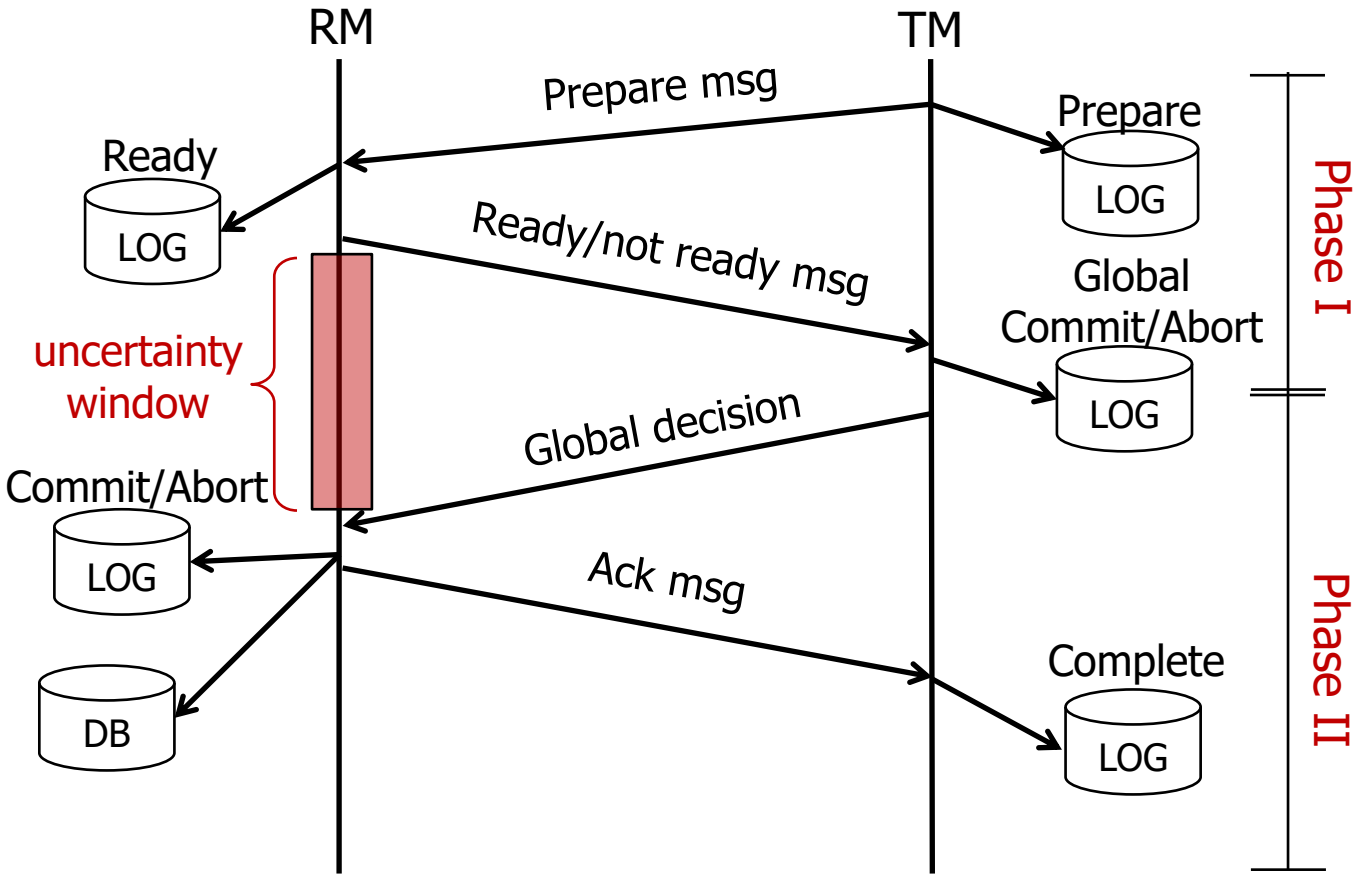


Phase II

3. The TM
   - Collects the ACK messages from the RMs
   - If *all* ACK messages are received
     - The complete record is written in the log
   - If the timeout expires and some ACK messages are missing
     - A new timeout is set
     - The global decision is resent to the RMs which did not answer

   until all answers are received

2 Phase Commit protocol



Uncertainty window

- Each RM is affected by an *uncertainty window*
  - Start after ready msg is sent
  - End upon receipt of global decision
- Local resources in the RM are locked during the uncertainty window
  - It should be small

Failure of a participant (RM)

- The warm restart procedure is modified with a new case
  - If the last record in the log for transaction T is "ready", then T does not know the global decision of its TM
- Recovery
  - READY list
    - new list collecting the IDs of all transactions in ready state
  - For all transactions in the ready list, the global decision is asked to the TM at restart
    - Remote recovery request

Failure of the coordinator (TM)

- Messages that can be lost
  - Prepare (outgoing)
  - Ready (incoming)

  } I Phase

  - Global decision (outgoing) } II Phase
- Recovery
  - If the last record in the TM log is prepare
    - The global abort decision is written in the log and sent to all participants
    - Alternative: redo phase I (not implemented)
  - If the last record in the TM log is the global decision
    - Repeat phase II

Network failures

- Any network problem in phase I causes global abort
  - The prepare or the ready msg are not received
- Any network problem in phase II causes the repetition of phase II
  - The global decision or the ACK are not received

Politecnico di Torino

DBMG

Map Reduce

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

DANIELE APILETTI

POLITECNICO DI TORINO

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(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

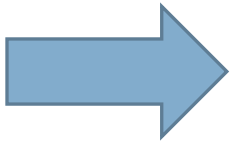
Id: 1
Exam: Database
Student: s123456
AYear: 2015-16
Date: 31-01-2016
Mark=29
CFU=8

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

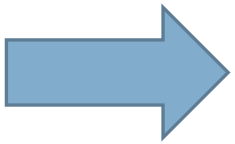
Id: 1
Exam: Database
Student: s123456
AYear: 2015-16
Date: 31-01-2016
Mark=29
CFU=8

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**)

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 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;
}
```

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

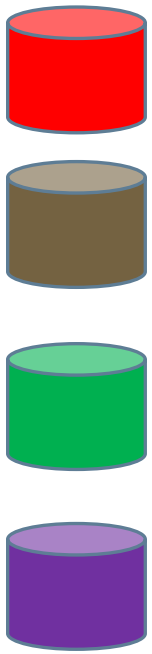
doc.id	Key	Value
1	S123456	[29, 8]
2	S123456	[24, 10]
5	S123456	[21, 8]
6	S123456	[30, 6]
3	S654321	[27, 10]
4	S654321	[26, 8]
7	S654321	[18, 8]
8	s987654	[25, 8]

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 in mongoDB

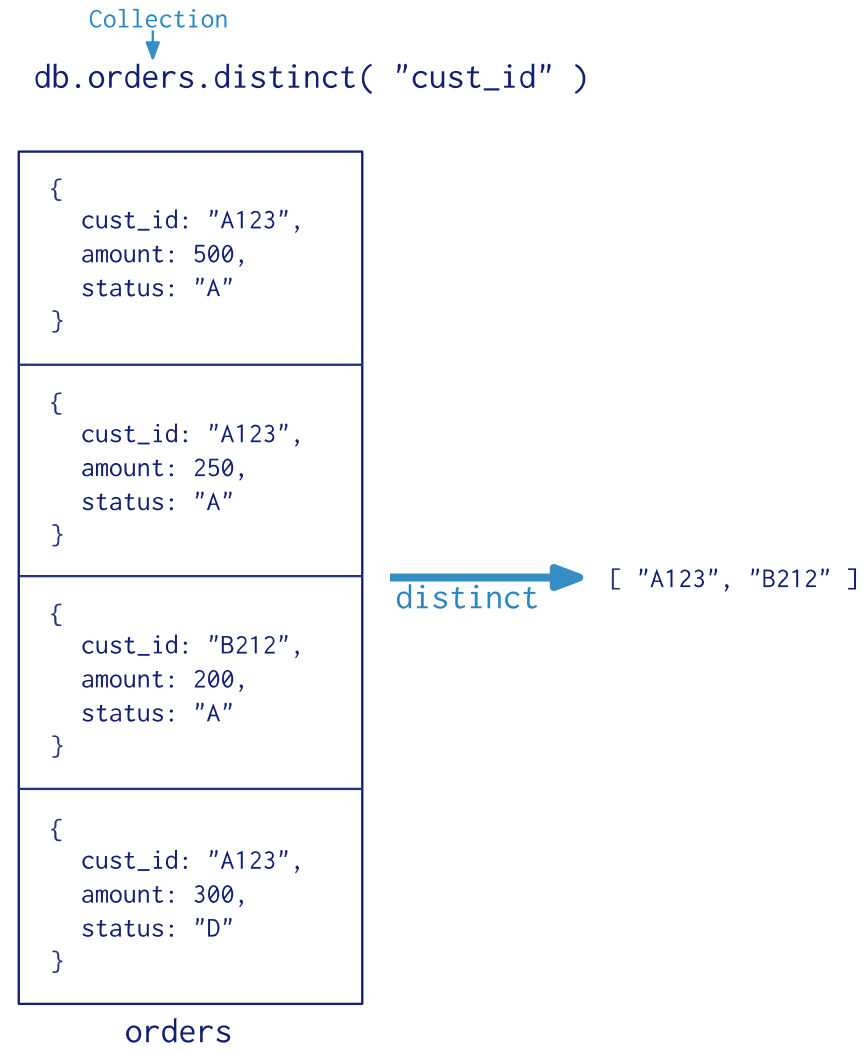
Aggregation operations in MongoDB

- 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**
    - since MongoDB 5.0 the map-reduce operation is **deprecated**
  - 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

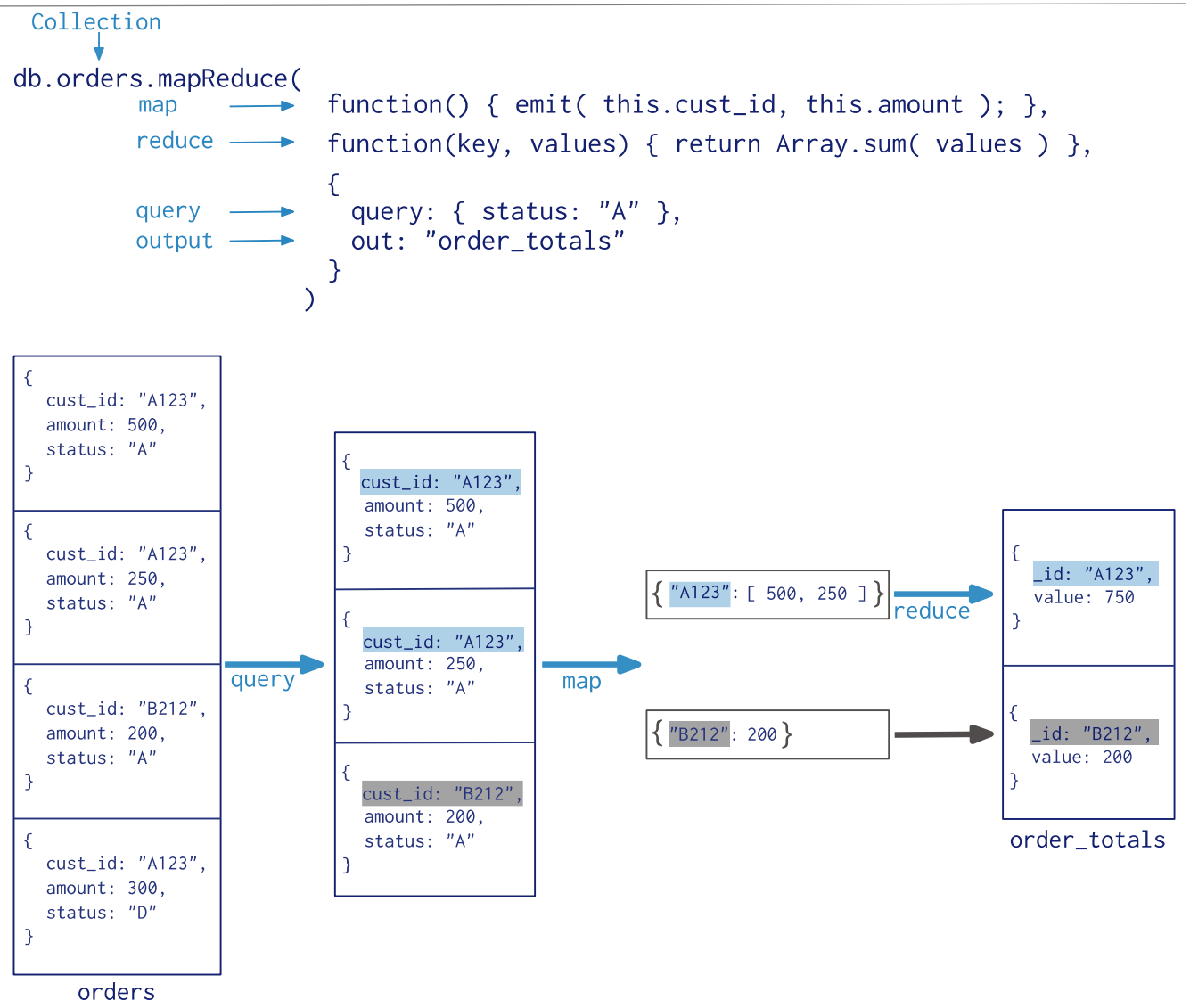


Comparison of aggregation operations

- 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
- Aggregation pipeline
  - **Performance** and usability
  - Virtually **infinite** pipeline of transformations
  - Map-reduce operations can be rewritten using aggregation pipeline operators, e.g., $group, $merge
  - 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

MongoDB: Map-Reduce

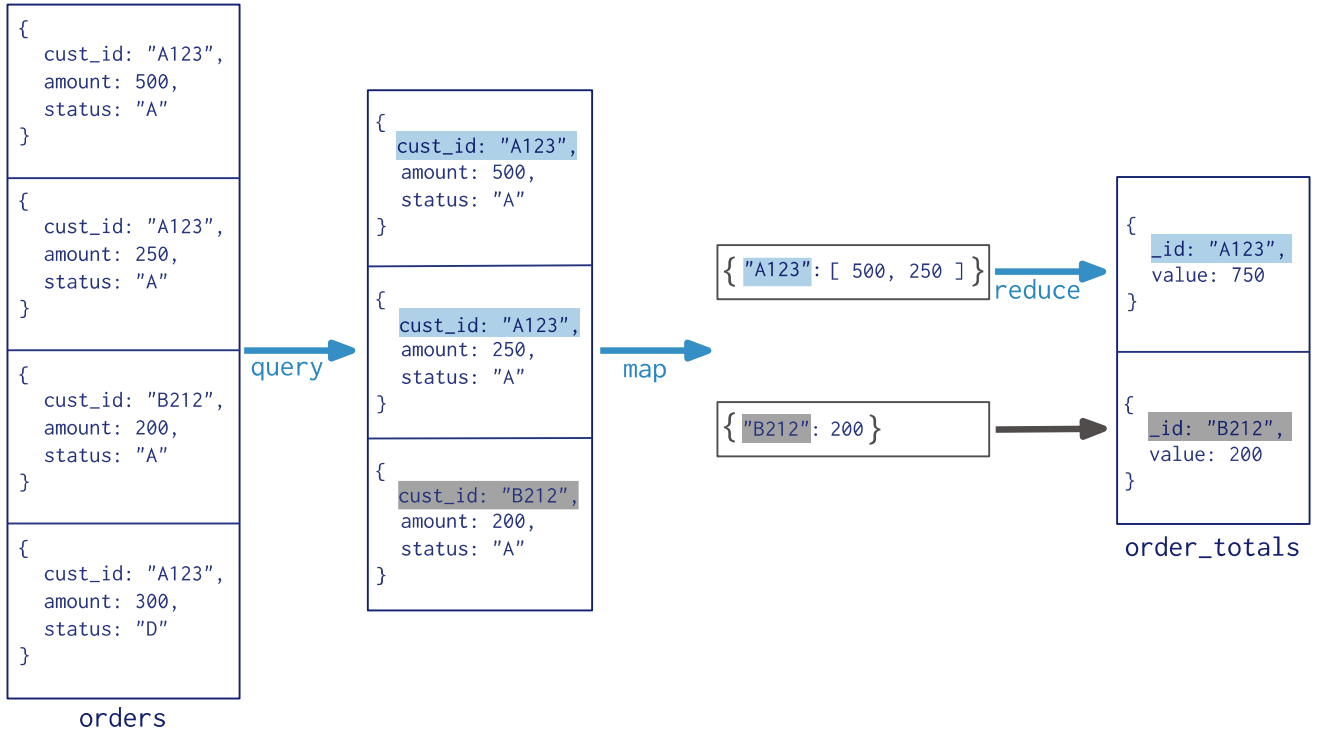
- custom JavaScript functions
- db.collection.**mapReduce**({
  - <map>,
  - <reduce>,
  - <finalize>,
  - <query>,
  - <out>,
  - <sort>,
  - <limit>,
  - ...})



MongoDB: Map-Reduce

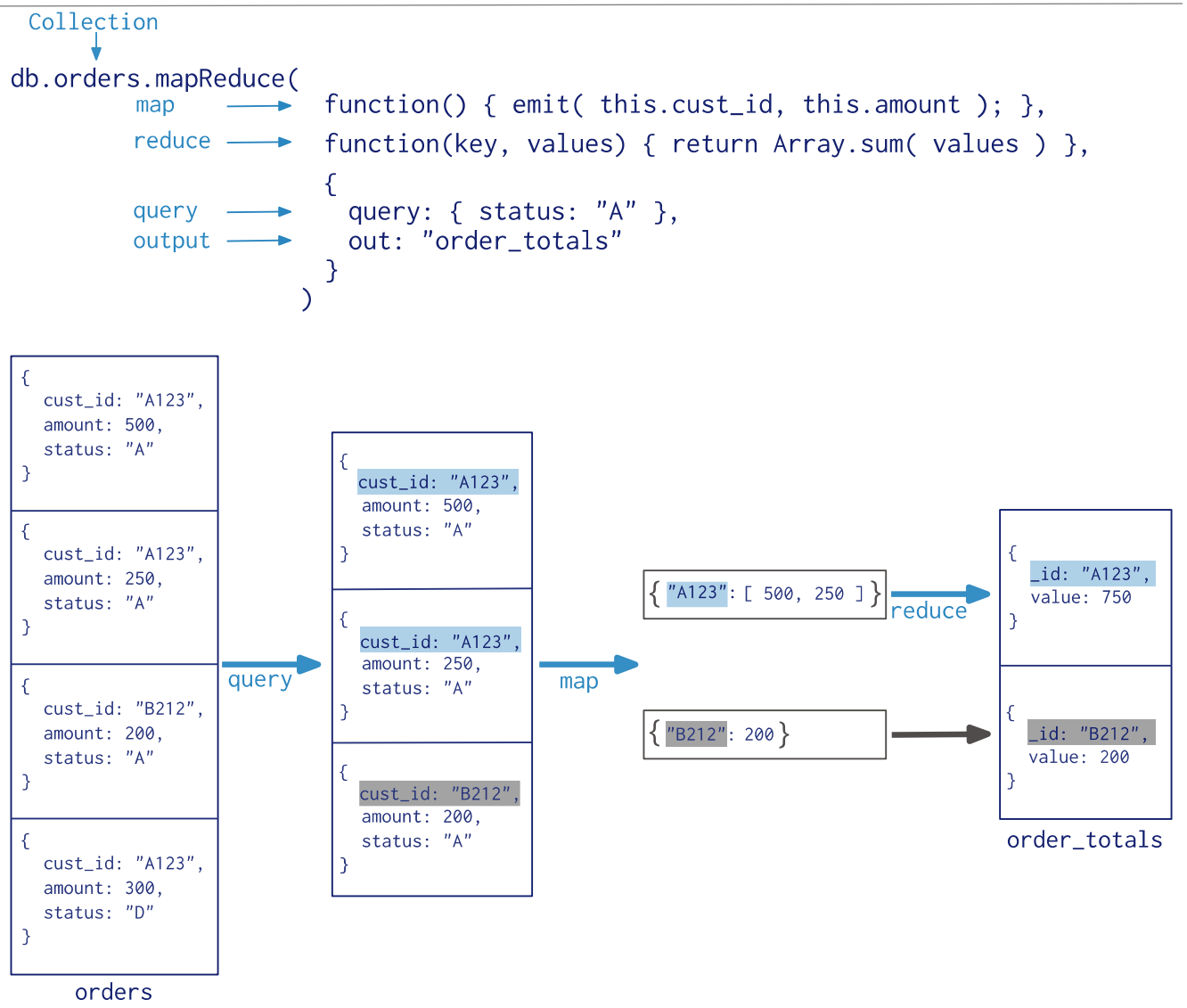
1. MongoDB applies the map phase **to each input document** (i.e. the documents in the collection that match the query condition)
2. The map function emits **key-value pairs**
3. For those keys that have multiple values, MongoDB applies the ***reduce* phase**, which collects and condenses the aggregated data
4. MongoDB then stores the **results** in a collection

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



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

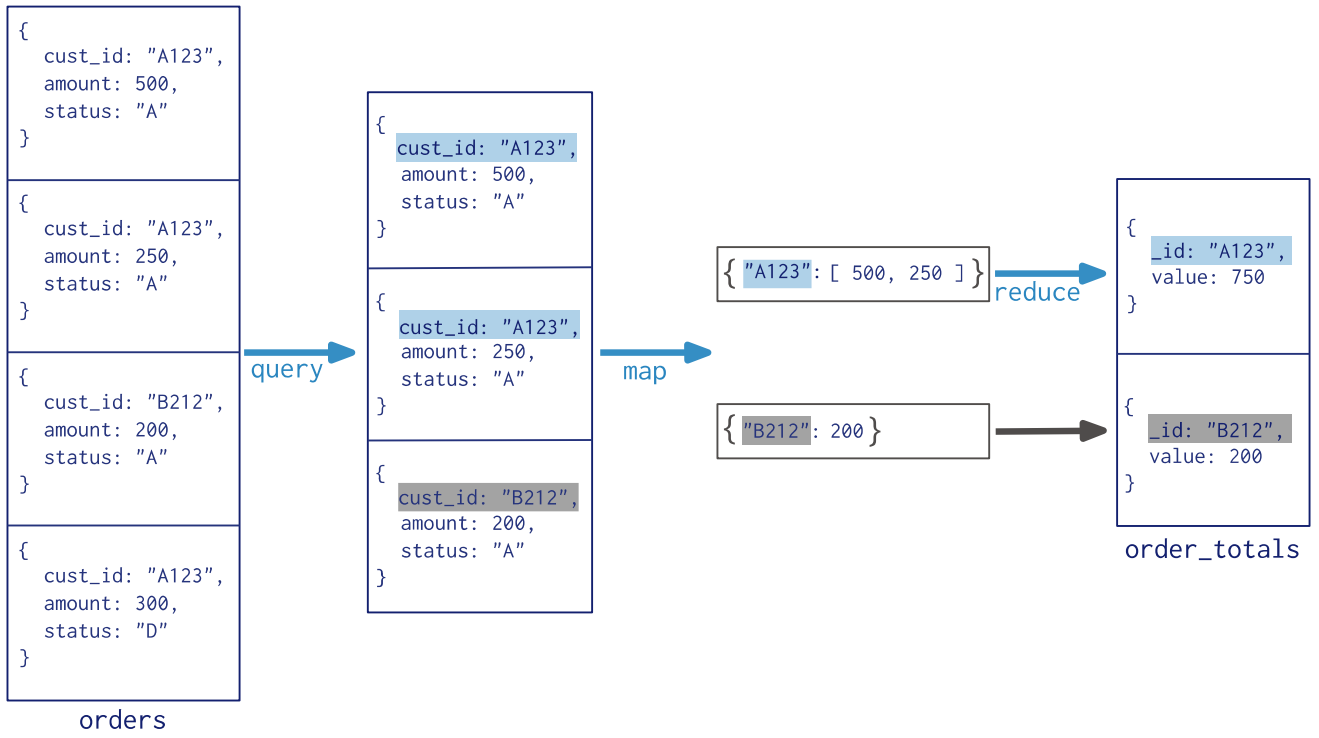
- **Finalize** (optional)
  - Follows the *reduce* method and modifies the output
- **Query** (optional)
  - specifies the selection criteria for selecting the input documents to the *map* function
- **Sort** (optional)
  - specifies the sort criteria for the input documents
  - useful for optimization, 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"
    }
)
```

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



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"
  }
)
```

Map function

Reduce function

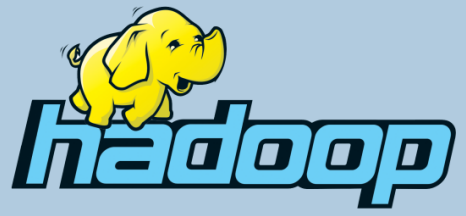
- Only for orders with status: "A"
- for each cust_id,
  - sum all the orders values
  - into the "order_totals" collection

MongoDB: Map-Reduce features

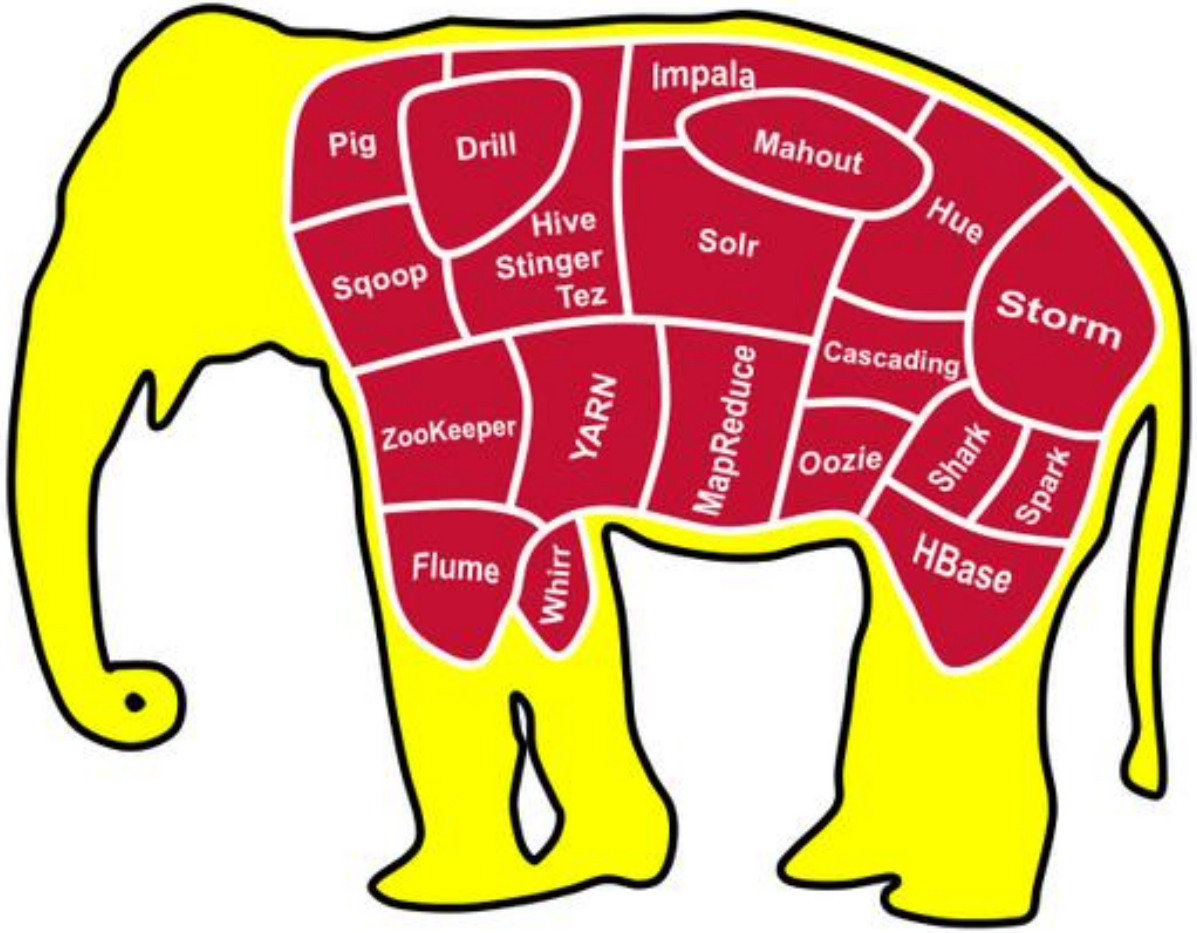
- All map-reduce functions in MongoDB are **JavaScript** and run within the mongod process
- Map-reduce operations
  - take the documents of a single collection as the *input*
  - perform any arbitrary sorting and limiting before beginning the map stage
  - return the results as a document or into a 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

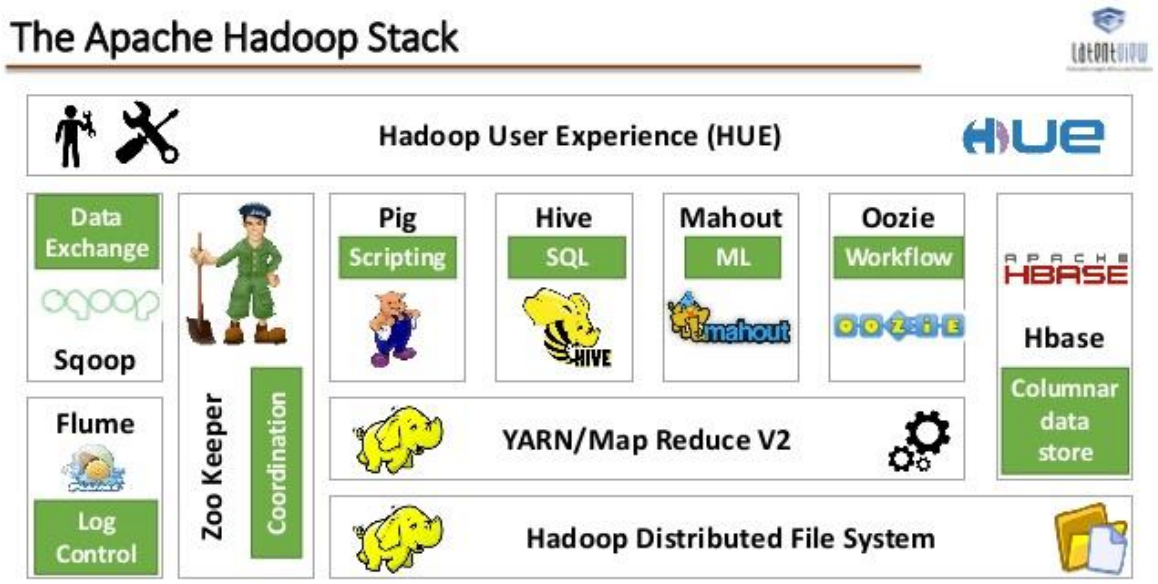


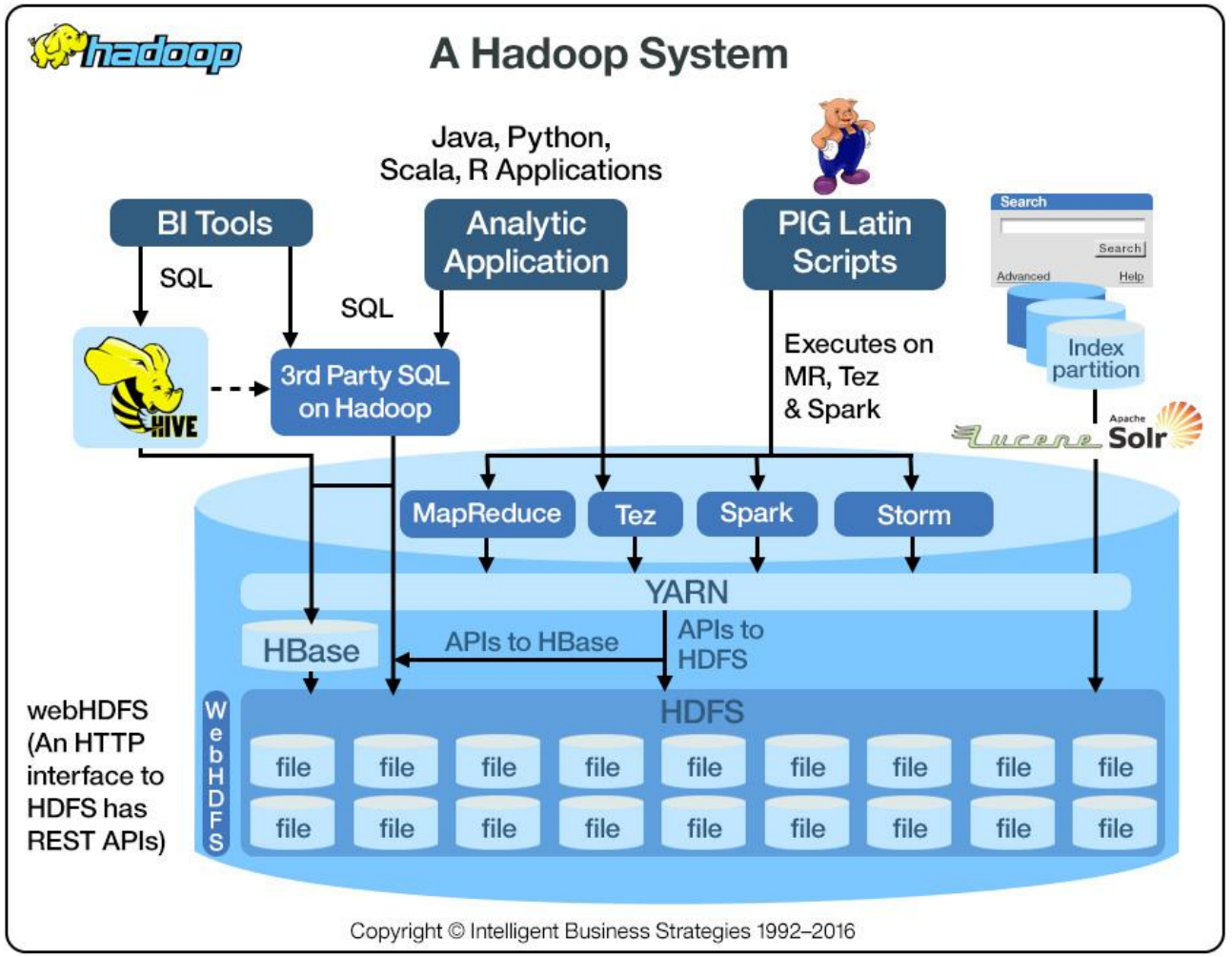
Hadoop, a Big-Data-everything platform



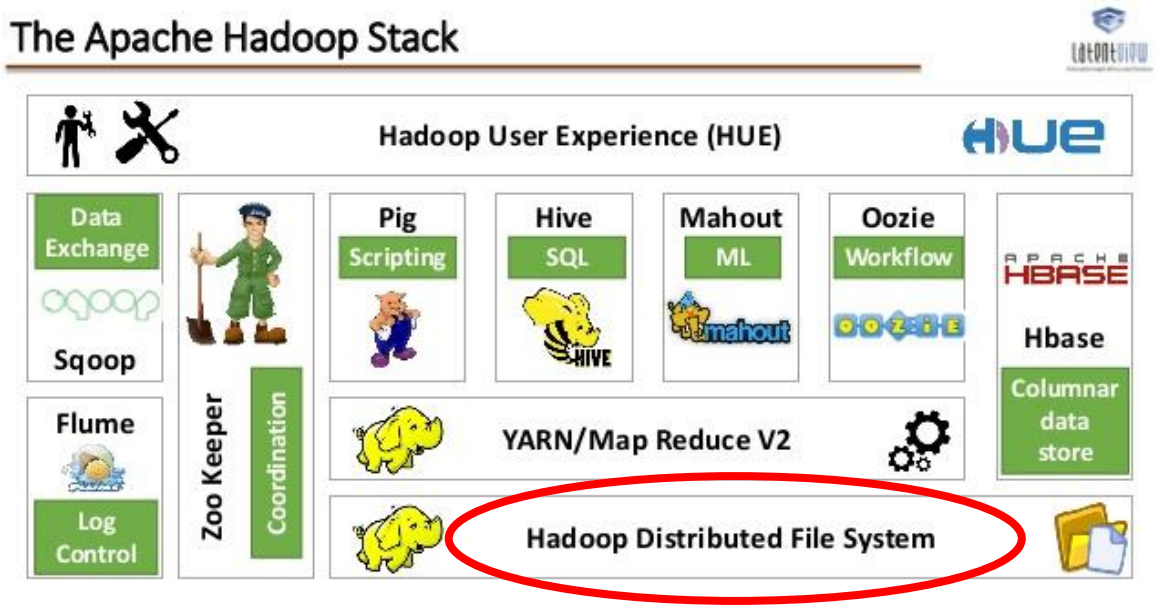
- **2003**: **Google** File System
- **2004**: MapReduce by **Google** (Jeff Dean)
- **2005**: **Hadoop**, funded by Yahoo, to power a search engine project
- **2006**: **Hadoop** migrated to Apache Software Foundation
- **2006**: **Google** BigTable
- **2008**: **Hadoop** wins the Terabyte Sort Benchmark,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

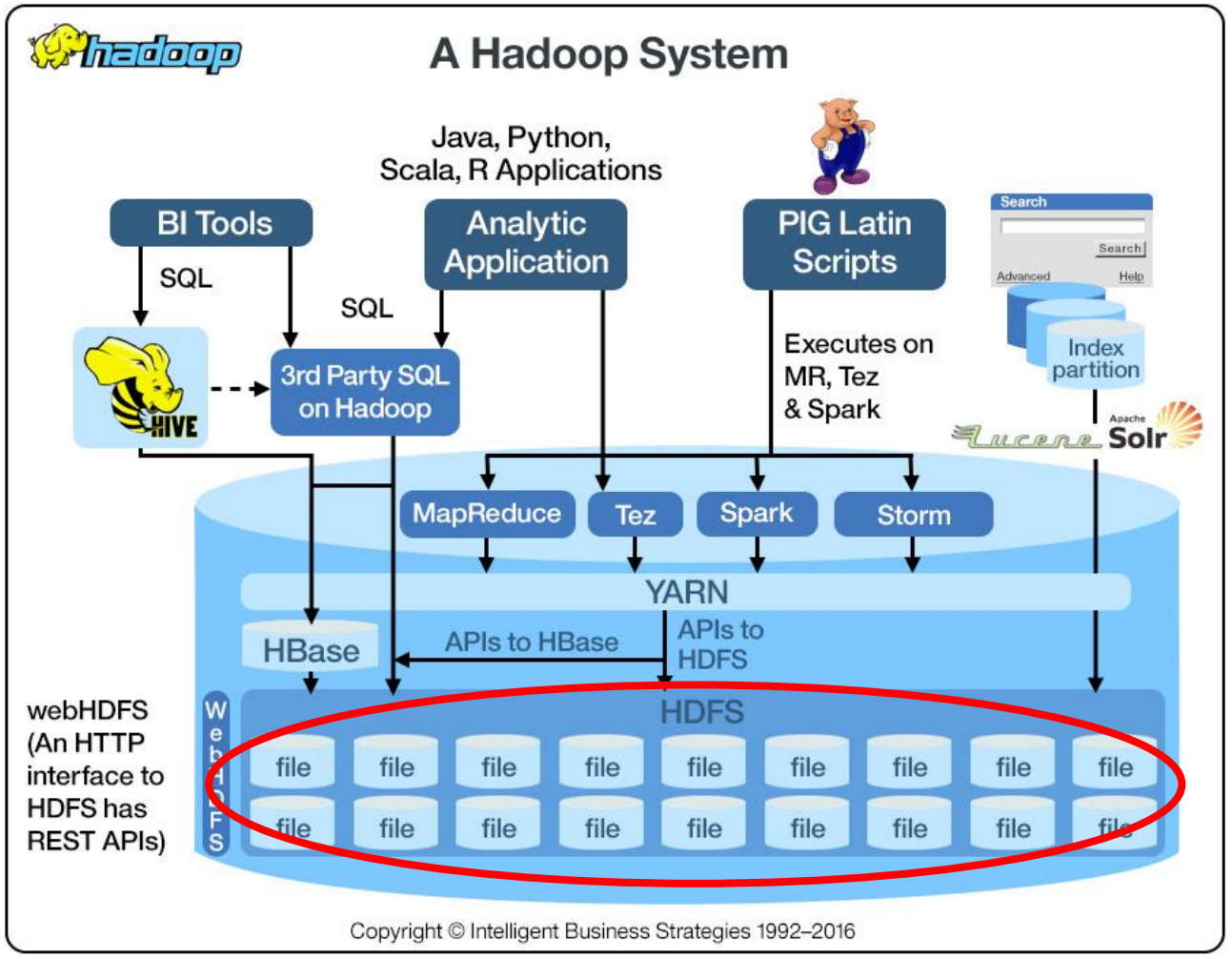
Hadoop, platform overview



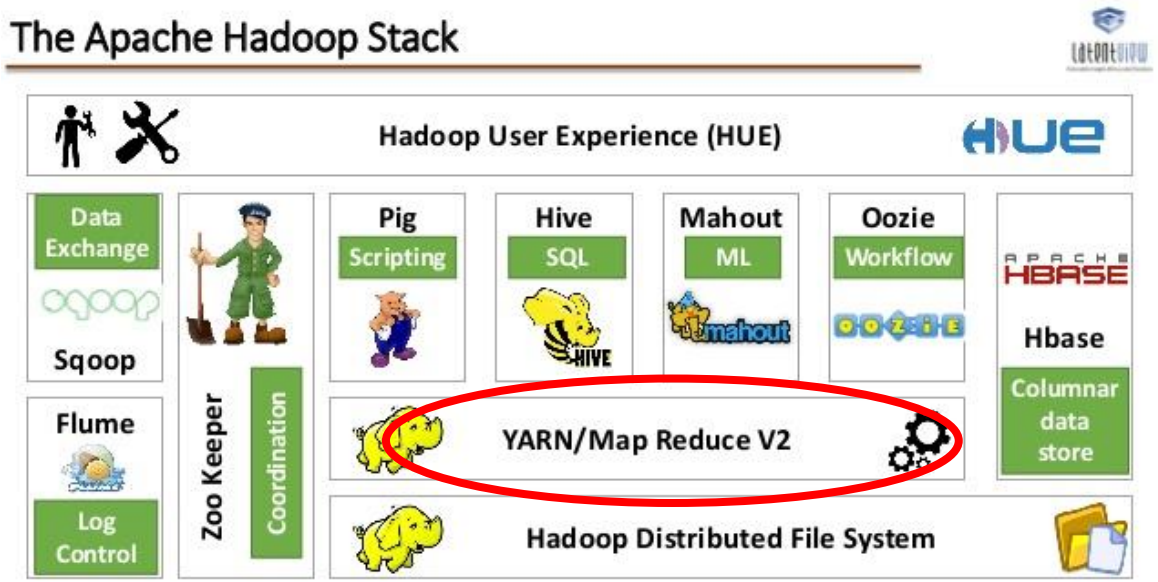


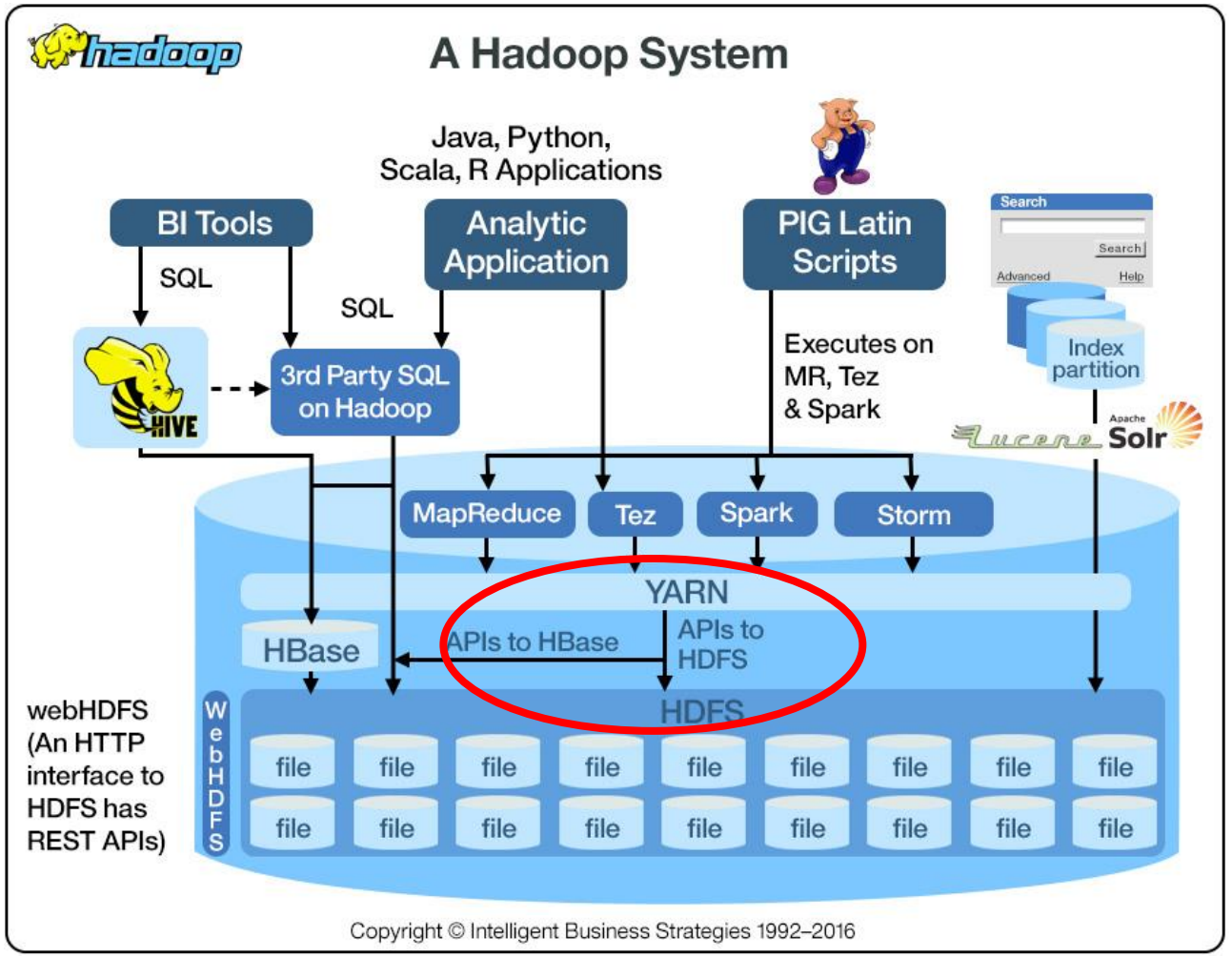
Hadoop, platform overview



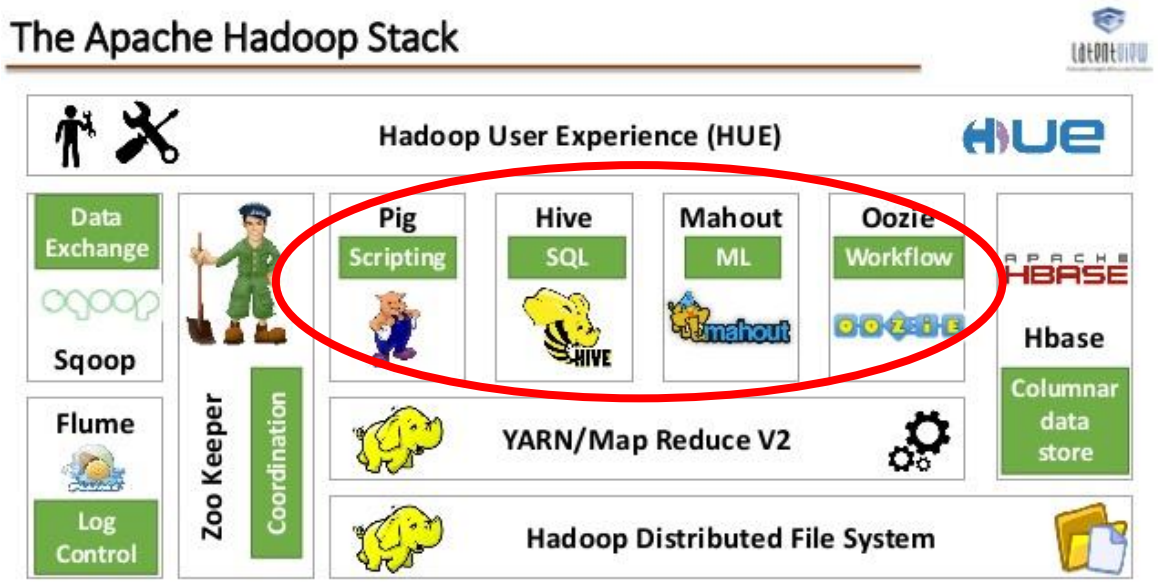


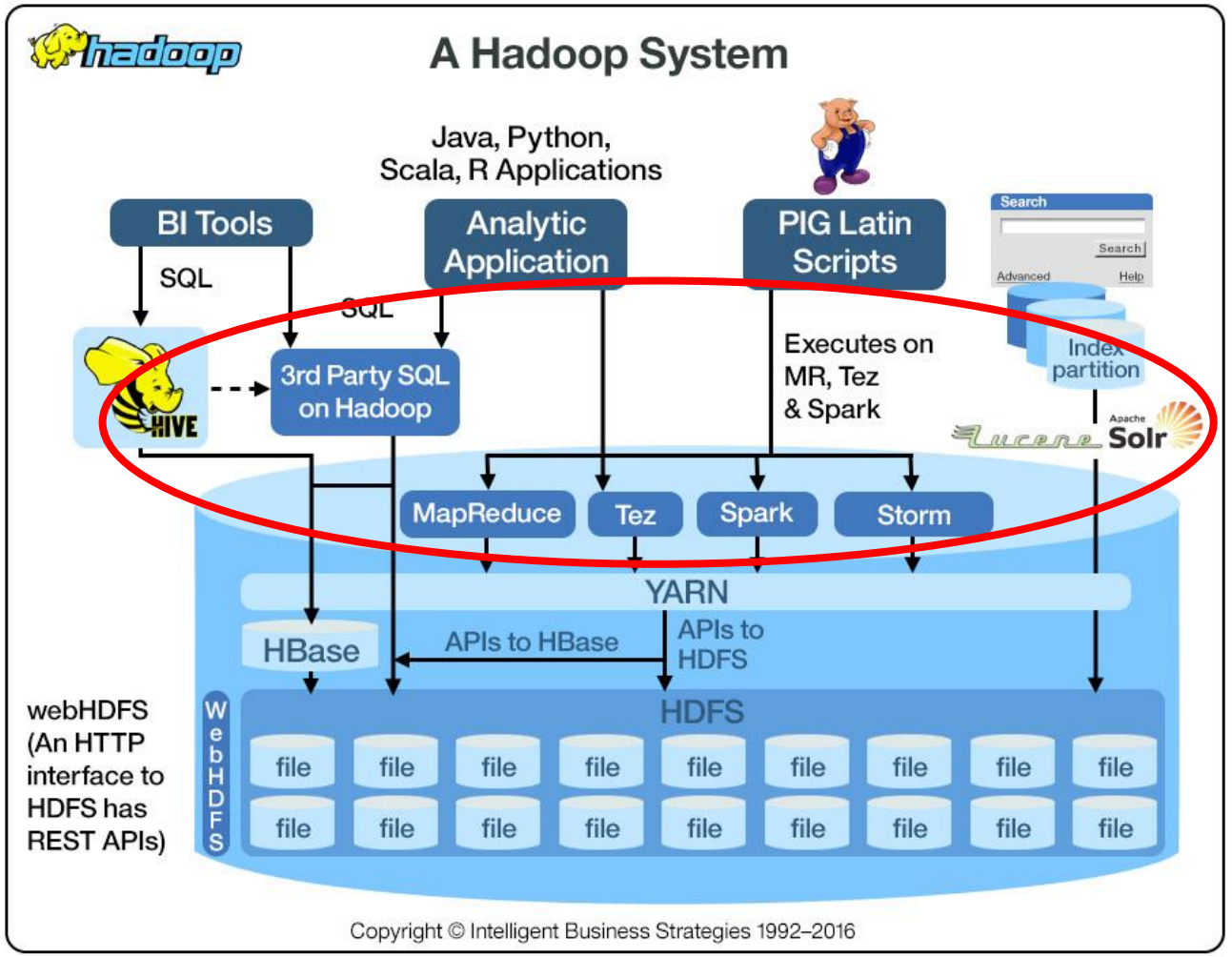
Hadoop, platform overview





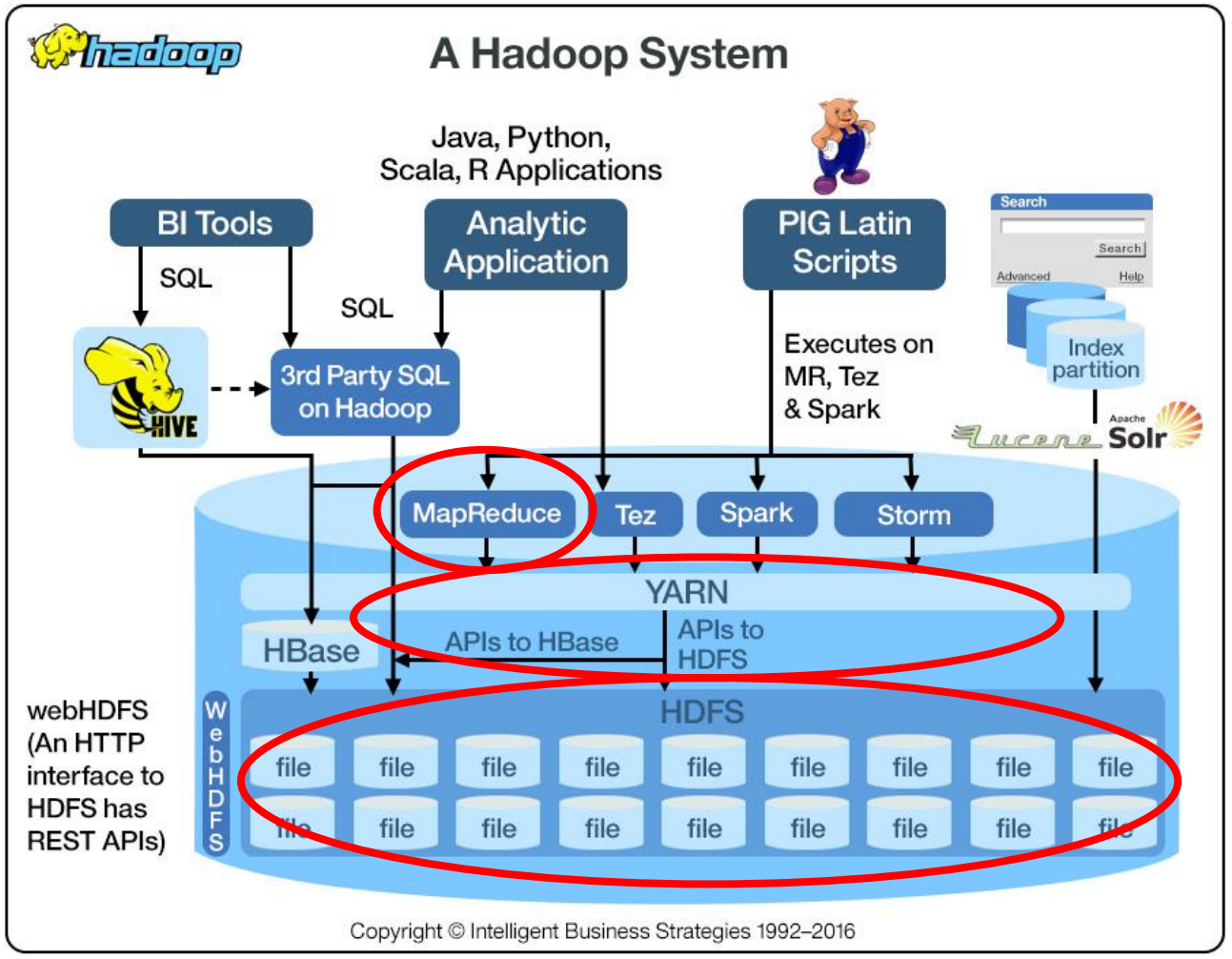
Hadoop, platform overview





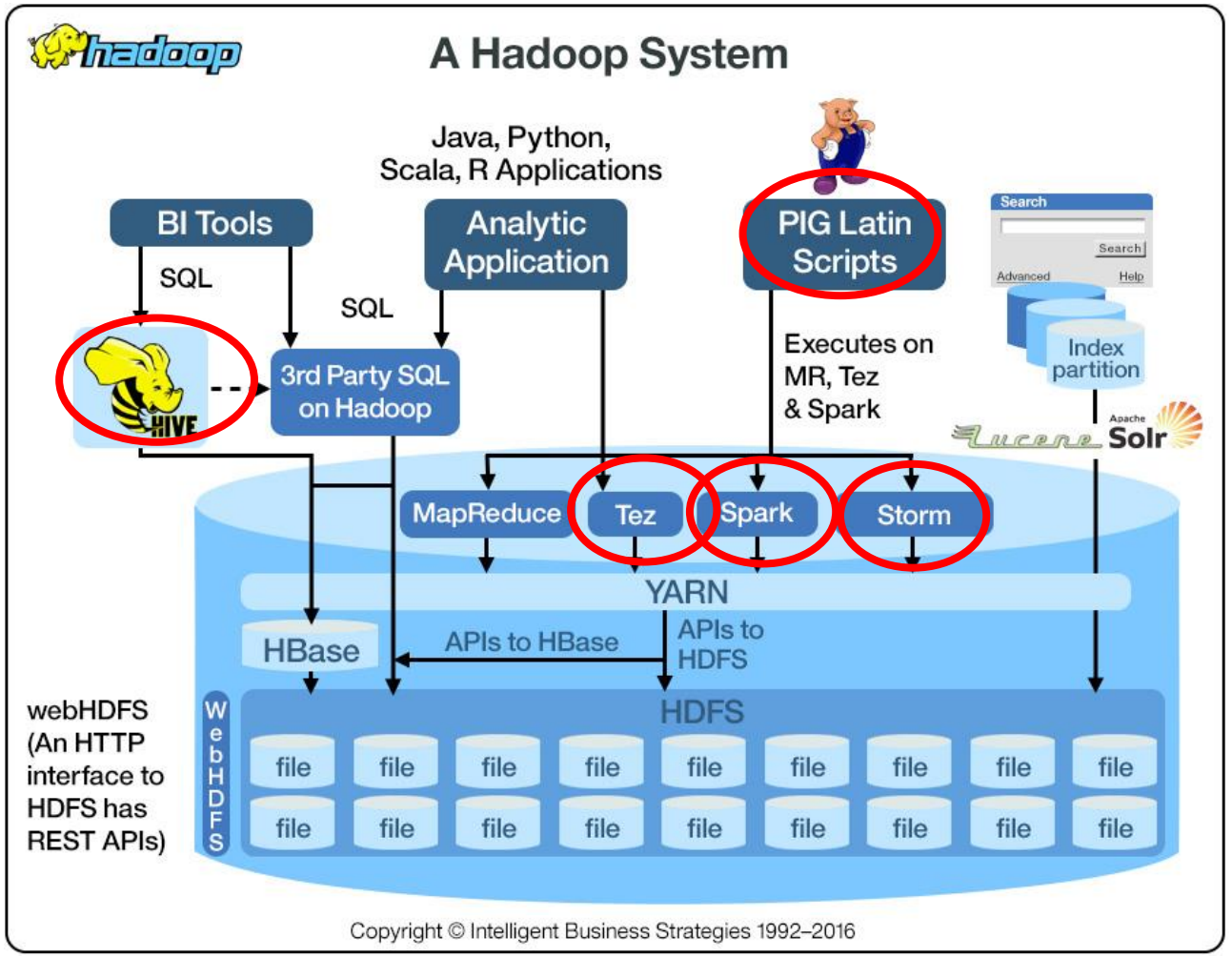
Apache Hadoop, core components

- **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.



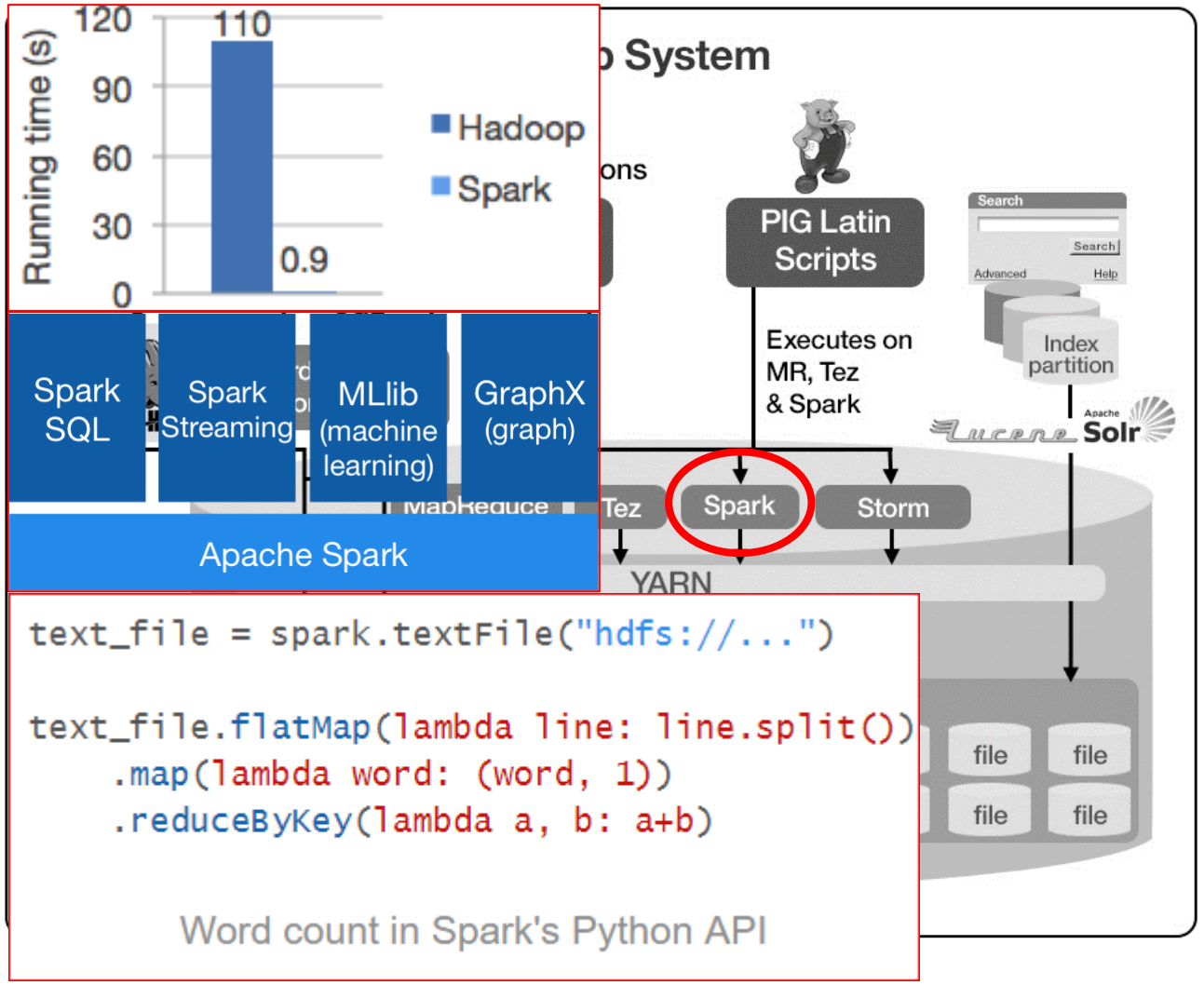
Hadoop-related projects at Apache

- 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.



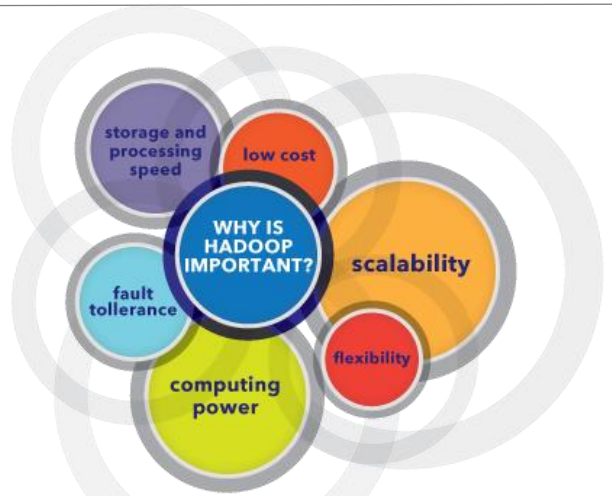
Apache Spark

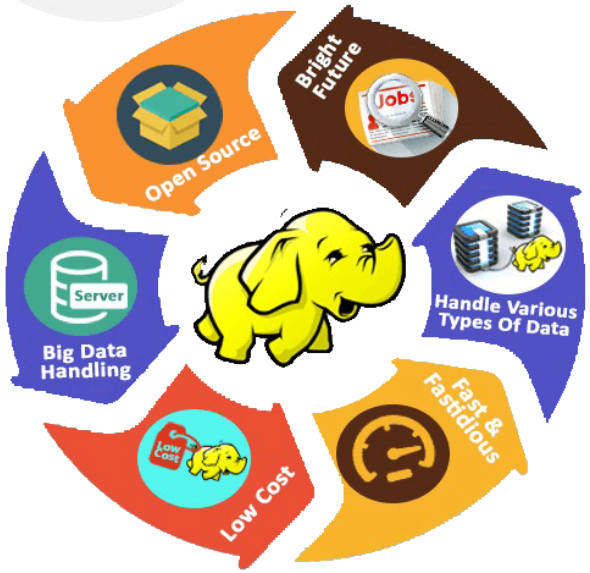
- A **fast** and general engine for **large-scale data processing**
- **Speed**
  - Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.
  - Apache Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing.
- Ease of Use
  - Write 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.



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.





Politecnico di Torino

DBMG

A design recipe

A notable example of NoSQL design for «distributed transactions»

DANIELE APILETTI

POLITECNICO DI TORINO

Design recipe: banking account

- Banks are serious businesses
- They need serious databases to store serious transactions and serious account information
- They can't lose or create money
- A bank **must** be in balance **all the time**

Design recipe: banking example

- Say you want to give $100 to your cousin Paul for Christmas.
- You need to:

decrease your account balance by 100$

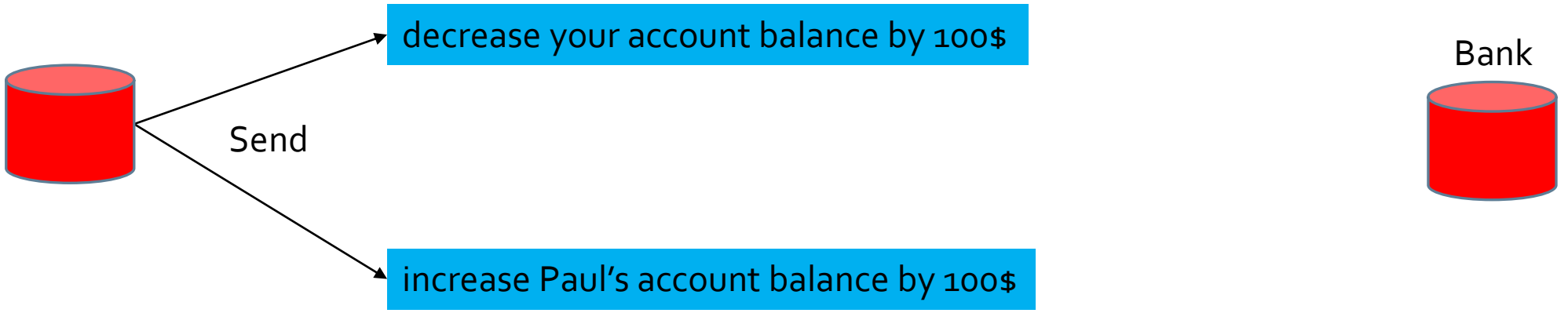
```
{
_id: "account_123456",
account:"bank_account_001",
balance: 900,
timestamp: 1290678353,45,
categories: ["bankTransfer"...],
...
}
```

increase Paul's account balance by 100$

```
{
_id: "account_654321",
account:"bank_account_002",
balance: 1100,
timestamp: 1290678353,46,
categories: ["bankTransfer"...],
...
}
```

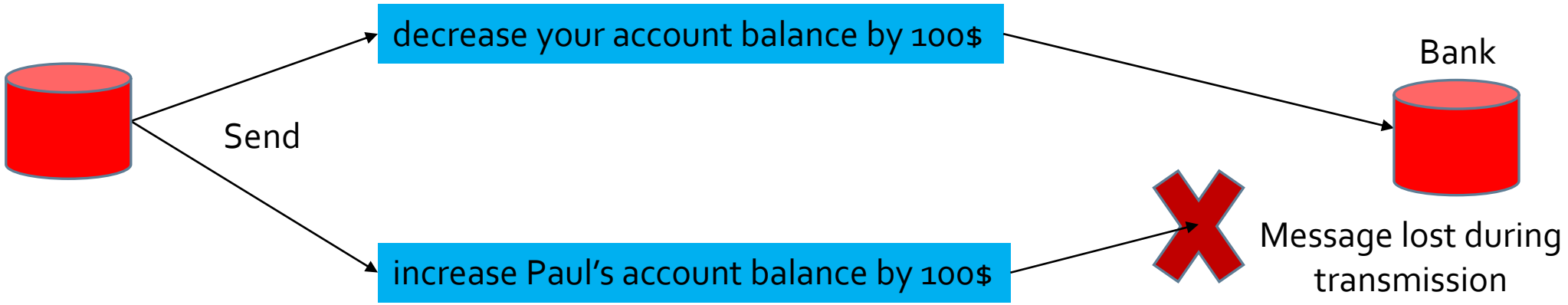
Design recipe: banking example

- What if some kind of failure occurs between the two separate updates to the two accounts?



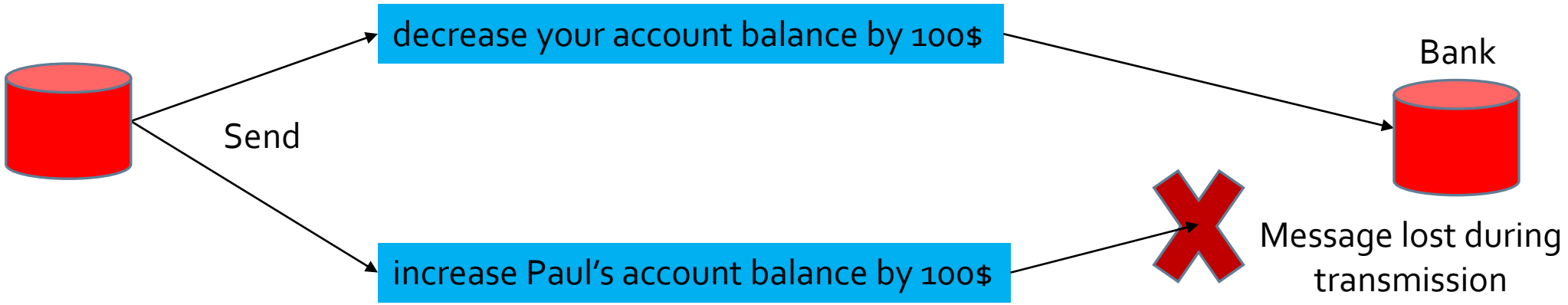
Design recipe: banking example

- What if some kind of failure occurs between the two separate updates to the two accounts?



Design recipe: banking example

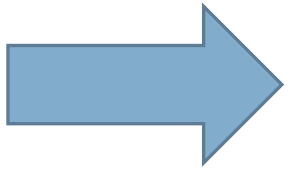
- What if some kind of failure occurs between the two separate updates to the two accounts?



- The NoSQL DB **cannot guarantee the bank balance**
- A different strategy (design) must be adopted

Banking recipe solution

- What if some kind of failure occurs between the two separate updates to the two accounts?
- A NoSQL database without 2-Phase Commit cannot guarantee the bank balance → a different strategy (design) must be adopted.



```
id:     transaction001
from:   "bank_account_001",
to:     "bank_account_002",
qty:    100,
when:   1290678353.45,
...
```

Design recipe: banking example

- How do we read the current account balance?
- Map

```
function(transaction){
 emit(transaction.from, transaction.amount*-1);
 emit(transaction.to, transaction.amount);
}
```

- Reduce

```
function(key, values){
 return sum(values);
}
```

- Result

{rows: [{key: "**bank_account_001**", value: **900**}]

{rows: [{key: "**bank_account_002**", value: **1100**}]

The reduce function receives:

- **key**= **bank_account_001**, **values**=[1000, -100]
- …
- **key**= **bank_account_002**, **values**=[1000, 100]
- …