www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

DMV - MONGODB1

18 January 2021 16:23

Fundamentals:

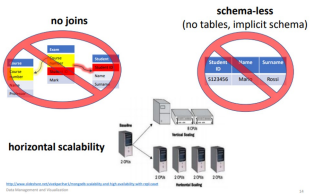
- Data structures are broken into the smallest units (normalization of the database shcmea)
- queries merge the data from different tables
- write operations are simple
- strong guarantees for transactional processing

Since the schema is tailored for a specific application the weite operations might be slower but read will be faster ---> effisicent and scalable.
Each NoSQL DB type has a different value model

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,...

NoSQL main features



Pros

- Work with semi-structured data (JSON, XML)
- 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
- Speed (join avoidance)

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
- Weak SQL
- Slow mass updates
- Use more disk space (replicated denormalized records, 10-50x)
- Difficulty in tracking "schema" (set of attribute) changes over time

No SQL is **scalable, allows for a dynamic schema, efeicient reading and cost saving**

Different models: rows, graph, document-based

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":"abelf879rft"}],
"shippingAddress":{"city":"Chicago"}
}
```

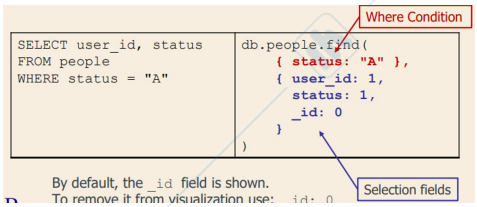
Document basics

- 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 Data design

- high level business ready data representation
- flexible and rich syntax adapting to most use cases
- mapping into developer language objects

MYSQL clause	MONGODB operator
SELECT	find()
SELECT <fields of interes> FROM <collection name> WHERE <conditions>	db.<collection name>.find({<conditions>}, {<fields of interest})




```
db.people.find({ age: { $gt: 25, $lte: 50 } })
```

- Age greater than 25 and less than or equal to 50
  - Returns all documents having **age > 25 and age <= 50**

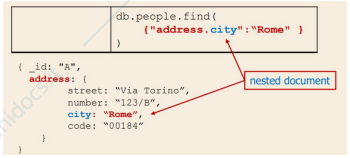
```
db.people.find({$or:[{status: "A"},{age: 55}]})
```

- Status = "A" or age = 55
  - Returns all documents having **status="A" or age=55**

```
db.people.find({ status: {$in:["A", "B"]}})
```

- Status = "A" or status = B
  - Returns all documents where the **status** field value is **either "A" or "B"**

- 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.



www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Status = "A" or status = B
- Returns all documents where the **status** field value is **either "A" or "B"**

reflects the order of documents on the disk.

MySQL	MongoDB	Description
>	$gt	greater than
>=	$gte	greater equal then
<	$lt	less than
<=	$lte	less equal then
=	$eq	equal to
!=	$neq	not equal to

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

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
$nin	Matches none of the values specified in an array

- `cursor.sort()`
- `cursor.count()`
- `cursor.forEach() //shell method`
- `cursor.limit()`
- `cursor.max()`
- `cursor.min()`
- `cursor.pretty()`

Cursor examples:

```
db.people.find({ status: "A"}).count()
```

- Select documents with status="A" and count them.

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

- `forEach` applies a JavaScript function to apply to each document from the cursor.
  - Select documents with status="A" and print the document name.

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

E.g.,

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

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

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

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

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

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();
```

MongoDB	Relational database
`db.users.find()`	`SELECT * FROM users`
`db.users.insert({ user_id: 'bcd001', age: 45, status: 'A'})`	`INSERT INTO users (user_id, age, status) VALUES ('bcd001', 45, 'A')`
`db.users.update( { age: { $gt: 25 } }, { $set: { status: 'C'}}, { multi: true })`	`UPDATE users SET status = 'C' WHERE age > 25`

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"
  } );
```

Field name

Field value

Insert multiple documents in a single statement: operator `insertMany()`

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

Documents can be updated by 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

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

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

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

<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

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

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

```
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

Replication

Same data in different places with the goal of redundancy (help survive failures)

- Master slave replicationù
  - a master takes all the input and write operations, the information is then replicated on one or more slaves which handle the read functions
  - synchronous slave replication waits for all the slaves to commit ---> performance killer
  - Asynchronous replication --> the master commits locally and than eachj slaves replicate wheneve he's ready
    - faster but unreliable

Distributed databases deal with 3 typical problems:

- Consistency (all the DB provide the same data)
- Aviability (database failures do not prevent operation)
- partition tolerance
  - CAP theorem says you can't satisfy all the guarantees simultaneously

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**.

- Usually partitions are rare so there is little reason to forfeit C or A when the system is not partitioning
- The choiche between C and A happens at different granularities
- The three properties are more continuous than binary

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 preserves 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

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

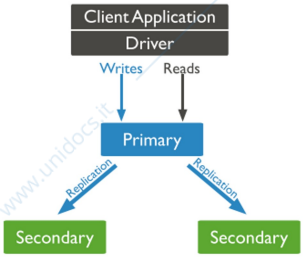
ACID versus BASE

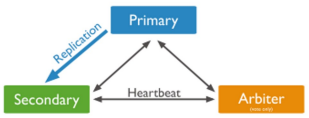
- ACID and BASE represent two design philosophies at opposite ends of the consistency-availability spectrum
- ACID properties focus on **consistency** and are the traditional approach of databases
- BASE properties focus on high **availability** and to make explicit both the choice and the spectrum
-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

Distributed query optimization allows for a quesry to be performed on multiple DBMS systems at the same time with the original dbms coordinating operations and distributing queries

- 2 phase commit protocol to coordinate the conclusion of a transaction
  - one coordinates, the other agree to the pact

KEY CONCEPTS





- A replica set is a group of *mongod* instances that maintain the same data set:
  - 1 primary node
  - several secondary node
  - 1 arbiter
- Primary node
  - receives all write operations
  - confirming writes with { *w: "majority"* } write concern
- 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 priority 0
- Arbiters
  - do not maintain a data set
  - maintain a quorum in a replica set by responding to heartbeat and election requests by other replica set members

- 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

- 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

- 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

Aggregation Pipeline

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

EXAMPLE

```
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)

PIPELINE

- Aggregate functions can be applied to collections to group documents

```
db.collection.aggregate( [ <list 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

$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.

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

$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)
$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.
$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).
$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).
$project	Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.
$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.
$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.
$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.

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.

- 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**

- **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

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari