POLITECNICO
MILANO 1863

Cassandra DB

Andrea Tocchetti
andrea.tocchetti@polimi.it

Cassandra - Introduction

Apache Cassandra is a highly scalable, high-performance distributed database designed to handle large amounts of data, providing high availability with no single point of failure. It’s a **column-based** NoSQL database created by **Meta** (ex. **Facebook**).

Cassandra - Introduction

Cassandra has the following features

- **Highly** and **Linearly Scalable**
- **No Single Point of Failure** (i.e., no single part of the system can stop the entire system from working)
- **Quick Response Time**
- **Flexible Data Storage** (i.e., supports structured, unstructured and semi-structured data)
- **Easy Data Distribution** (i.e., supports flexible data duplication)
- **BASE Properties**
- **Fast Writes**

Cassandra Query Language - Introduction

To query the data stored within Cassandra, a dedicated query language named **Cassandra Query Language (CQL)** was developed.

CQL offers a model similar to **MySQL** under many different aspects

- It is used to query data stored in **tables**
- Each table is made by **rows** and **columns**
- Most of the **operators** are the ones used in MySQL

CQL commands and queries can either be run in the console or by reading a textual file with the corresponding command.

Cassandra Query Language - CREATE a Keyspace

The first operation to perform before creating the table is creating the **keyspace**.
A **keyspace** is the outermost container in Cassandra.

Keyspaces are created using the **CREATE KEYSPACE** command.

```
cqlsh> CREATE KEYSPACE <identifier> WITH <properties>;
```

Let's create a keyspace with the name **population**.

```
cqlsh> CREATE KEYSPACE population
       WITH replication = {'class': 'SimpleStrategy',
                           'replication_factor': 3};
```

Cassandra Query Language - Partition and Clustering Key

A table can employ many different **Clustering** and/or **Partition Keys**.

```
PRIMARY KEY ((personal_id, ...),
              age, ... )
```

Partition Keys

Clustering Keys

When creating a table, clustering keys can be used to define an ordering.

```
cqlsh:population> CREATE TABLE person (...)
                  WITH CLUSTERING ORDER BY (age ASC, ...);
```

Cassandra Query Language - DESCRIBE & USE

The **DESCRIBE** command can be used to check whether a keyspace (or a table) has been correctly created. It can also be applied to other elements.

```
cqlsh> DESCRIBE keyspaces;
```

To be able to perform the operations on the tables (that we still have to create), we must choose in which keyspace we want to work. The command **USE** covers such need.

```
cqlsh> USE <keyspace_name>;
```

Let’s **USE** the keyspace we just created.

```
cqlsh> USE population;
```

Cassandra Query Language - ALTER & DROP

Keyspaces can be also modified (**ALTER**) and deleted (**DROP**) with the corresponding commands.

```
cqlsh> ALTER KEYSPACE <identifier> WITH <properties>;
```

```
cqlsh> DROP KEYSPACE <identifier>;
```

Cassandra Query Language - CREATE a Table

Let’s now learn how to **CREATE** a table. The command is the following.

```
cqlsh:keyspace> CREATE TABLE <table_name> (
                    <column_definition>,
                    <column_definition>,
                    …
                )
```

Optionally, some options can be included by using **WITH <options>**.

The definition of the columns is performed as follows.

```
<column_name> <column_type>
```

Cassandra Query Language - CREATE a Table

Let's try to create a simple table named **person** with name, age, birth date and gender.

```
cqlsh:population> CREATE TABLE person (
                     personal_id text,
                     name text,
                     age varint,
                     birth_date text,
                     gender text,
                     PRIMARY KEY (personal_id, age)
                  );
```

Let's check whether the table has been created or not.

```
cqlsh:population> DESCRIBE tables;
```

Cassandra Query Language - CREATE a Table

Now that the table has been created, let’s take a look at its description.

```
cqlsh:population> DESCRIBE person;
```

This command prints a lot of details about the table. All of these details can be customized when creating the table (as mentioned before).

Cassandra Query Language - Partition and Clustering Key

When creating the **PRIMARY KEY** of the table as the last definition within the **CREATE TABLE** operation, the columns that you put within the **PRIMARY KEY** statement have different meaning depending on the order and the brackets.

The first value (or set of values) is named **Partition Key(s)**. It defines the way in which the data is partitioned within the cassandra nodes.

The second value (or sets of values) is named **Clustering Key(s)**. It is used to define the way in which the data is stored within a partition (i.e., the sorting).

```
PRIMARY KEY (personal_id, age)
```

Partition Key　　Clustering Key

Cassandra Query Language - ALTER a Table

Tables can be also modified through the **ALTER** command.

```
cqlsh:keyspace> ALTER TABLE <table_name> <instructions>;
```

For example, we can add a new column to our table…

```
cqlsh:keyspace> ALTER TABLE <table_name> ADD
<column_definition>;
```

… or remove a column from it

```
cqlsh:keyspace> ALTER TABLE <table_name> DROP <column_name>;
```

Cassandra Query Language - ALTER a Table

Let's try add two new columns to the person table named address (text) and salary (float).

```
cqlsh:population> ALTER TABLE person
                  ADD address text;
```

```
cqlsh:population> ALTER TABLE person
                  ADD salary float;
```

Let's drop the salary attribute from the person table.

```
cqlsh:population> ALTER TABLE person
                  DROP salary;
```

Cassandra Query Language - DROP & TRUNCATE a Table

Tables can be also deleted through the **DROP** command.

```
cqlsh:keyspace> DROP TABLE <table_name>;
```

Rather than deleting the table, it is possible to empty it through the **TRUNCATE** command.

```
cqlsh:keyspace> TRUNCATE TABLE <table_name>;
```

N.B. by looking at the documentation, you may notice that the keyword **TABLE** can be interchanged with **COLUMNFAMILY**. There is no difference between them. Indeed, **COLUMNFAMILY** is still supported for “historical” reasons.

Cassandra Query Language - CREATING an INDEX

Indexes are one of the most important elements of a table in Cassandra. They allow to query the column efficiently. It is kind of hard to notice such an advantage on a small set of data, while it is essential in big datasets.

Secondary Indexes are created with the following command.

```
cqlsh:keyspace> CREATE INDEX <identifier>
                ON <table_name> (<column_name>);
```

Let’s create an index on the column name of the table person.

```
cqlsh:population> CREATE INDEX person_name
                  ON person (name);
```

Cassandra Query Language - DELETE an INDEX

Indexes can also be deleted through the **DROP** command.

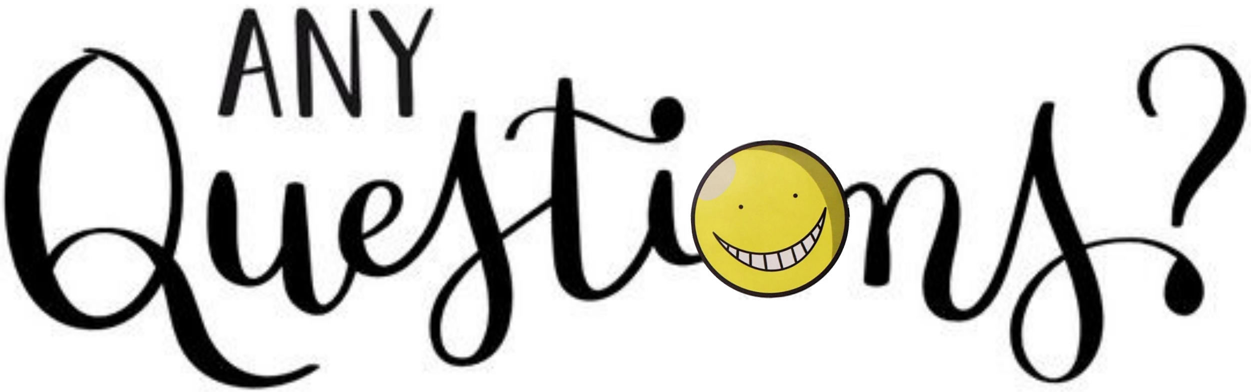
```
cqlsh:keyspace> DROP INDEX index_name
```

Let’s add a new index on the address column of the table person…

```
cqlsh:Population> CREATE INDEX person_address
                  ON person (address)
```

… then remove it.

```
cqlsh:Population> DROP INDEX person_address
```


ANY
Questions?

Cassandra Query Language - INSERT Data

Let’s see how to **INSERT** data within our tables.

```
cqlsh:keyspace> INSERT INTO <tablename>(<column_name1>,
                                          <column_name2>, ...)
                 VALUES (<column_value1>, <column_value2>....)
                 USING <option>;
```

Let’s try and insert a new person in our table

```
cqlsh:population> INSERT INTO person(personal_id, address, age,
                                     birth_date, gender, name)
                  VALUES (‘FRNTRZ95E12F675T’, ‘Via Milano 12’,
                  26, ‘12-05-1995’, ‘Male’, ‘Francesco Terzani’);
```

Cassandra Query Language - SELECT Data

Let’s see how to **SELECT** the data within our tables.

```
cqlsh:keyspace> SELECT <field_list>
                FROM <table_name>
                WHERE <conditions>
```

Let’s select the person we just inserted within or database using their personal_id.

```
cqlsh:population> SELECT *
                  FROM person
                  WHERE personal_id = ‘FRNTRZ95E12F675T’
```

Cassandra Query Language - SELECT Data

Let's retrieve the person we inserted within our database through their age.

```
cqlsh:population> SELECT *
                  FROM person
                  WHERE age = 26
```

An **Invalid Request Error** is shown as the age column has no associated primary or secondary index! Indeed, being Cassandra a column-oriented database, all the operations are optimized to extract data from columns. To solve this issue, it's necessary to query with respect to the attributes included in the **primary key** or to create a **secondary index**. Be careful that not all the operations are supported (e.g., most comparison operators needs the additional statement **ALLOW FILTERING**).

Cassandra Query Language - UPDATE Data

Let's see how to **UPDATE** tuples within our database.

```
cqlsh:keyspace> UPDATE <table_name>
                SET <column_name> = <new_value>, ...
                WHERE <condition>;
```

Let's update Francesco's address to 'Via Milani 13'

```
cqlsh:population> UPDATE person
                  SET address = 'Via Milani 13'
                  WHERE personal_id = 'FRNTRZ95E12F675T';
```

Cassandra Query Language - DELETE Data

Let’s see how to **DELETE** the data from our tables.

```
cqlsh:keyspace> DELETE
                FROM <table_name>
                WHERE <condition>;
```

Let’s try deleting Francesco using their address

```
cqlsh:population> DELETE
                  FROM person
                  WHERE address = ‘Via Milani 13’;
```

Cassandra Query Language - DELETE Data

An **Invalid Query Error** is displayed as we are not performing a **DELETE** operation using a primary key, which is against Cassandra's standard operations pattern.

Let's perform a proper **DELETE** operation using the primary key.

```
cqlsh:population> DELETE
                  FROM person
                  WHERE personal_id = 'FRNTRZ95E12F675T';
```

Let's check whether our operation was successfully performed.

```
cqlsh:population> SELECT * FROM person
```

Cassandra Query Language - BATCH

A set of **INSERT**, **UPDATE** and **DELETE** operations can be organized in **BATCH**. In that way, they are executed one after another with a single command.

```
cqlsh:keyspace> BEGIN BATCH
                 <insert_statement>;
                 <update_statement>;
                 <delete_statement>;
                APPLY BATCH;
```

Cassandra Query Language - CAPTURE

When the amount of data within a database grows, it can be really tough to visualize it within a terminal. Fortunately, Cassandra provides us with a few commands to overcome this problem.

The **CAPTURE** command followed by the path of the folder in which store the results and the name of the file.

```
cqlsh> CAPTURE D:/Program Files/Cassandra/Outputs/output.txt;
```

To interrupt the **CAPTURE** you can run the following command.

```
cqlsh> CAPTURE off;
```

Cassandra Query Language - EXPAND

The **EXPAND** command provides extended outputs within the console when performing queries. It must be executed before the query to enable it.

```
cqlsh> EXPAND on;
```

To interrupt the **EXPAND** you can run the following command.

```
cqlsh> EXPAND off;
```

Cassandra Query Language - SOURCE

The **SOURCE** command allows you to run queries from textual files. The command accepts the path to the file with the query.

```
cqlsh> SOURCE D:/Program Files/Cassandra/Queries/query_1.txt;
```

Cassandra Query Language - FOREIGN KEY (?)

Many of you may be asking themselves, how about **FOREIGN KEYS** and relationships between tables?

The answer to this question is pretty simple.

In Cassandra there is no concept of **FOREIGN KEYS** and/or relationships, if you want any cross-table check to be performed, you have to manage it by yourself.

N.B. As mentioned before, Cassandra wasn't created to perform such operations, but to be able to query a lot of data quickly and efficiently. If such operations are needed, you'd better reconsider your DB choice.

Cassandra Query Language - Data Types

Cassandra supports many different data types, like text, varint, float, double, Boolean, etc.

In particular, it supports two particular data types

- Collections
- User-defined data types

Collections are pretty easy to define and update

```
cqlsh:keyspace> CREATE TABLE test(email list<text>, ...)
```

```
cqlsh:keyspace> UPDATE test SET email = email + [...] WHERE ...
```

Cassandra Query Language - User-defined Data Types

When it comes to user-defined data types the complexity increases, as it is necessary to define the data type before using it.

```
cqlsh:keyspace> CREATE TYPE <type_name> (
                              <column_definition>
                              ...
                            )
```

To check that the new type has been properly created, you can use the **DESCRIBE** operator. User-defined data types support the **ALTER** and **DROP** operations.

```
cqlsh:keyspace> DESCRIBE TYPE <type_name>
```

ANY Questions?

POLITECNICO
MILANO 1863

Exercise Session

Cassandra Query Language - Exercise Session

Create a keyspace named “**car_dealer**”.

```
cqlsh> CREATE KEYSPACE car_dealer
       WITH replication = {‘class’: ‘SimpleStrategy’,
                           ‘replication_factor’: 3};
```

Check the existence of the keyspace.

```
cqlsh> DESCRIBE keyspaces;
```

```
cqlsh> DESCRIBE car_dealer;
```

Cassandra Query Language - Exercise Session

Create a table named “**car**” within the keyspace with the following attributes **car_id** (uuid, primary key), **brand** (textual), **max_speed** (integer), **price** (float), **consumption_lt_per_km** (float) and sorted by **max_speed** in **ascending** order.

```
cqlsh:car_dealer> CREATE TABLE Car (
                     car_id uuid,
                     brand text,
                     max_speed varint,
                     price float,
                     consumption_lt_per_km float,
                     PRIMARY KEY (car_id, max_speed) )
                  WITH CLUSTERING ORDER BY (max_speed ASC);
```

Cassandra Query Language - Exercise Session

Add a new column to the table, named “features” that contains the set/list of features of the cars (e.g., air conditioning, etc.). Each “feature” is made of name and description.

```
cqlsh:car_dealer> CREATE TYPE feature (
                          name text,
                          description text );
```

```
cqlsh:car_dealer> ALTER TABLE car
                  ADD features list<frozen<feature>>;
```

When using a user-defined data type, it is necessary to use the **frozen** keywords.
A frozen data type can only be overwritten, it can’t edited anymore.

Cassandra Query Language - Exercise Session

As we do not want to deal with complex fields, let's remove the "**features**" field.

```
cqlsh:car_dealer> ALTER TABLE car
                  DROP features;
```

Insert a new value in the table (use the function **uuid()** to get a unique identifier).

```
cqlsh:car_dealer> INSERT INTO car (car_id, brand, max_speed,
                  price, consumption_lt_per_km)
                  VALUES (uuid(), 'Ferrari', 320, 200000.00,
                  30.00);
```

Cassandra Query Language - Exercise Session

Run the data creation operations from the car_data.txt file.

```
cqlsh:car_dealer> SOURCE '<path_to_your_folder>/car_data.txt';
```

Check that all the data have been properly uploaded.

```
cqlsh:car_dealer> SELECT * FROM car;
```

Extract all the cars that cost more than 100’000.

```
cqlsh:car_dealer> SELECT * FROM car WHERE price > 100000
                  ALLOW FILTERING;
```

Cassandra Query Language - Exercise Session

Extract all the cars that cost exactly 35’000 (without using **ALLOW FILTERING**).

```
cqlsh:car_dealer> CREATE INDEX car_price ON car (price);
```

```
cqlsh:car_dealer> SELECT * FROM car WHERE price = 35000;
```

Extract the sum of all the prices of the cars in the DB.

```
cqlsh:car_dealer> SELECT SUM(price) FROM car;
```

Cassandra Query Language - Exercise Session

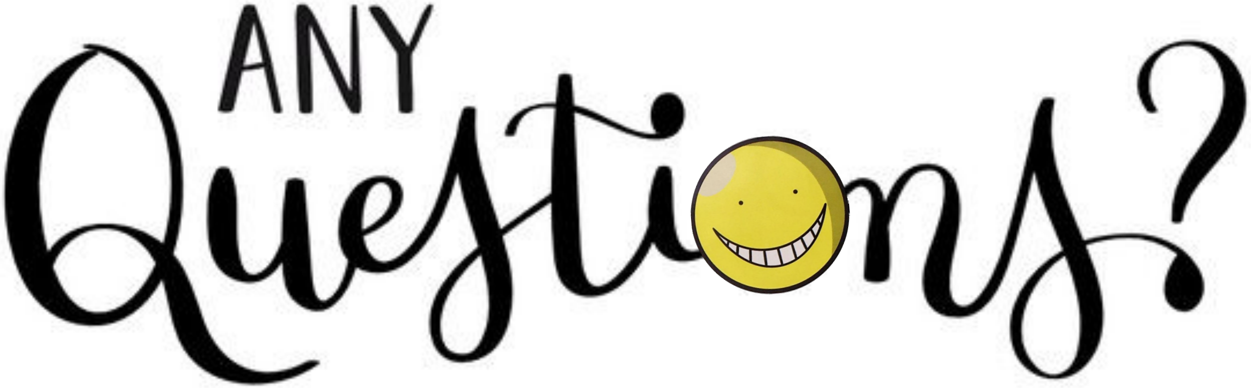
Count the number of Ferrari cars in the DB and store it in a file named “ferrari_count.txt”.

```
cqlsh:car_dealer> CAPTURE <path_to_folder>/ferrari_count.txt;
```

```
cqlsh:car_dealer> CREATE INDEX car_brand ON car (brand);
```

```
cqlsh:car_dealer> SELECT COUNT(*) FROM car
                  WHERE brand = ‘Ferrari’;
```

```
cqlsh:car_dealer> CAPTURE off;
```


ANY
Questions?

POLITECNICO
MILANO 1863

Redis

Redis - Introduction

Redis is a **Key-Value** database whose values can be accessed through the name of the key.
The stored values can be simple types (e.g., numbers) or complex structures (e.g., lists).

Redis - Storing and Collecting Data

Storing a single data instance on redis is very easy and can be achieved using the **SET** operator.

```
> SET key_name key_value
```

Collecting a value is as easy as storing it. It's enough to use the **GET** operator. The only requirements is that we need to know the **key_name**.

```
> GET key_name
```

Redis - Storing and Collecting Data

Sometimes, before collecting data it's necessary to check the existence of the **key_name** to be sure that the **GET** operator is correctly executed.

The **EXISTS** operator checks whether a key-value pair with a given key_name exists. It returns a value of **1** if the field exists, **0** otherwise.

```
> EXISTS key_name
```

Redis - Deleting Data

Redis also supports the deletion of key-value pairs and through the **DEL** operator.

The only requirements is that we need to know the **key_name**.

```
> DEL key_name
```

Redis - Updating Data

Numerical key-value pairs can be updated using a series of operators

- **INCR** – Increases the value by 1.
- **DECR** – Decreases the value by 1.
- **INCRBY** – Increases the value by the amount set.
- **DECRBY** – Decreases the value by the amount set.

```
> INCR key_name
```

```
> INCRBY key_name value
```

Redis - Atomic Operations

Redis supports (**a sort of**) scripting. For example, the operators we just explained can be executed through the following code.

```
> x = GET key_name
  x = x + 1
  SET key_name x
```

The proposed script is a general way of executing the increasing or decreasing operations. Then, why do we need them?

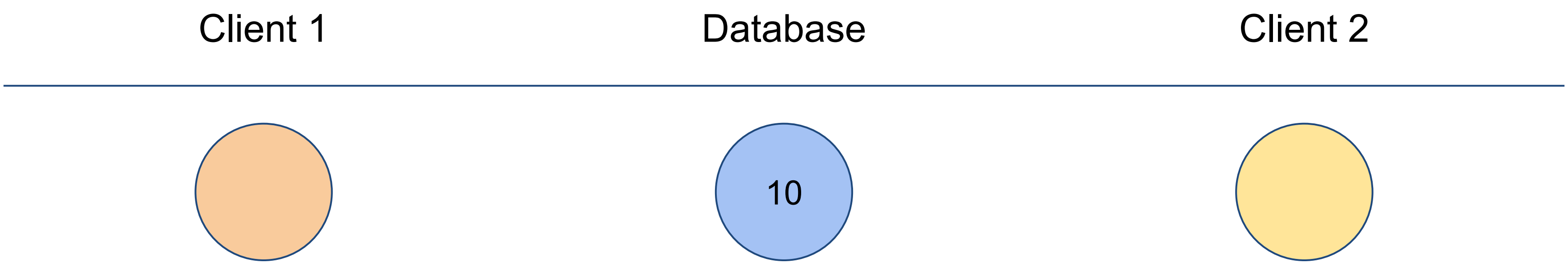
Redis - Atomic Operations

The reason is that these operations are **ATOMIC OPERATIONS**.

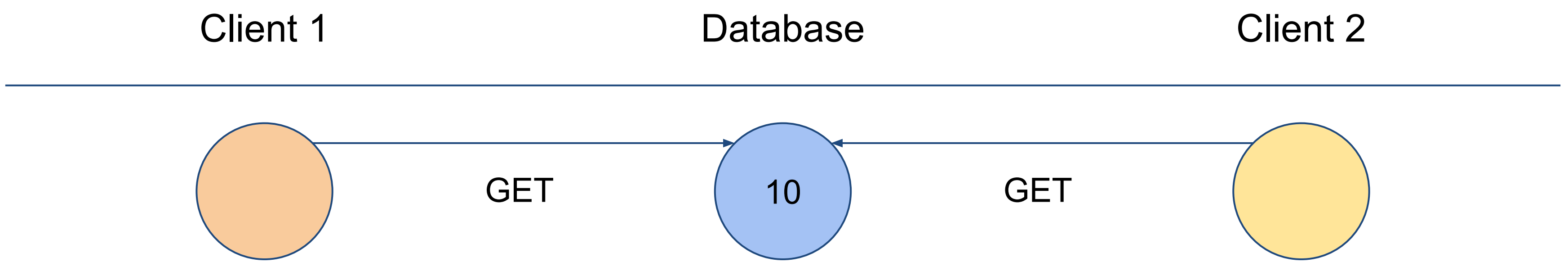
ATOMIC OPERATIONS are not affected by problems in case of **concurrent access**. Let's see what could happen if two different clients try to update the same variable without using the script.

```
> x = GET key_name
  x = x + 1
  SET key_name x
```

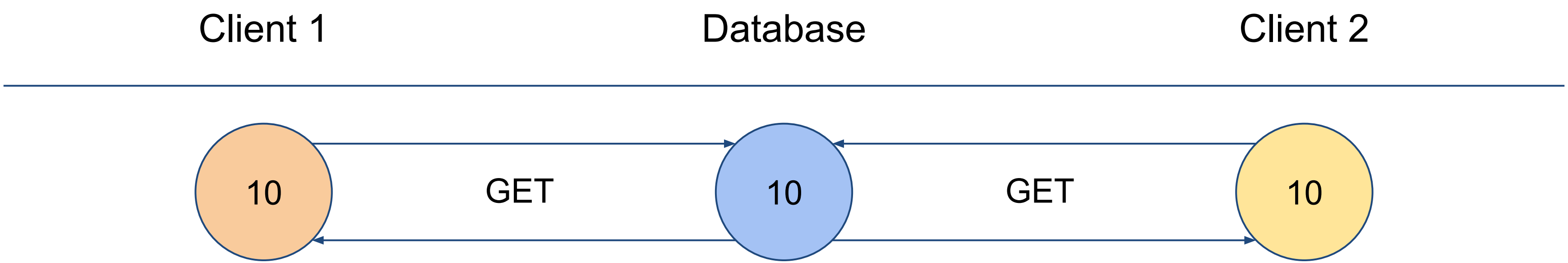
Redis - Atomic Operations



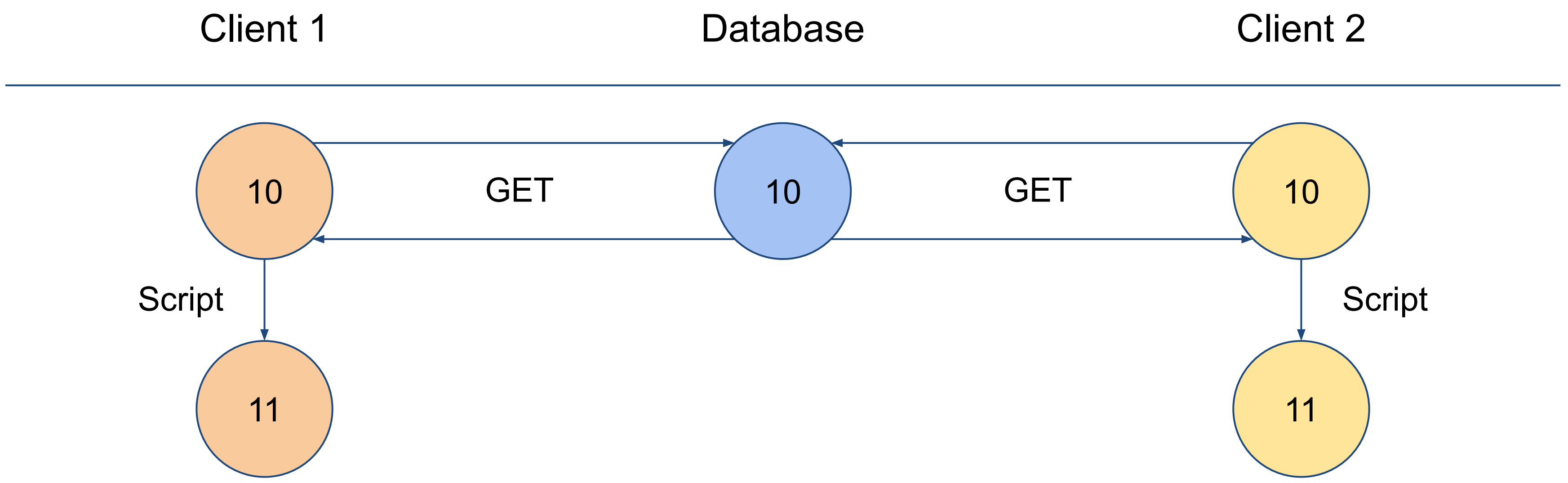
Redis - Atomic Operations



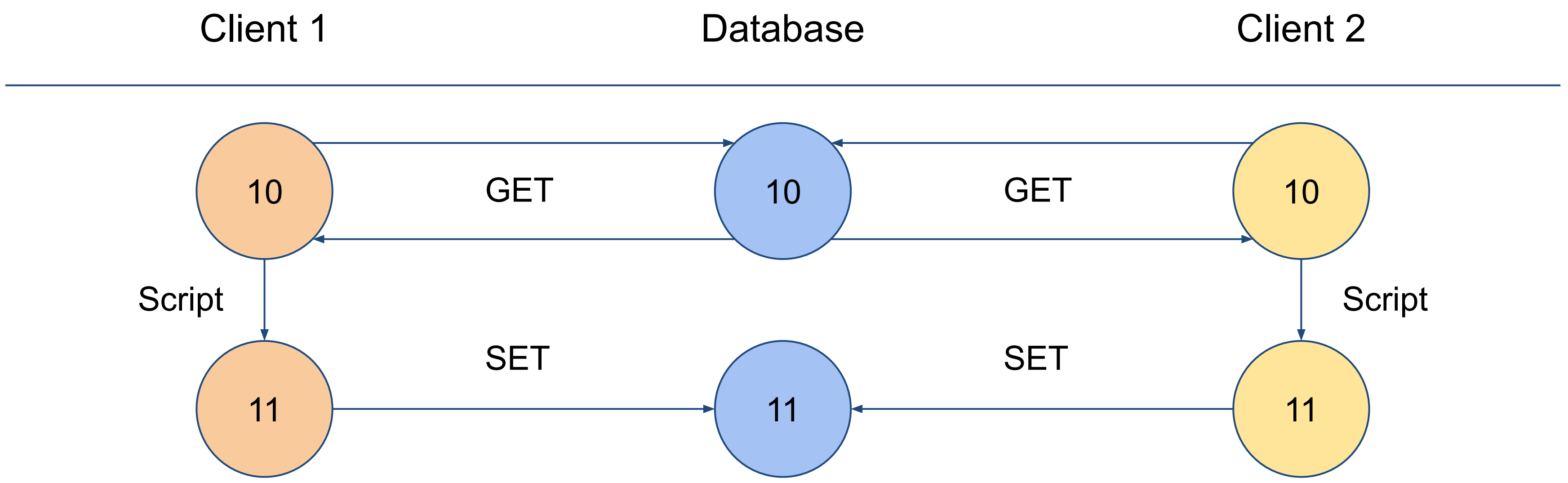
Redis - Atomic Operations



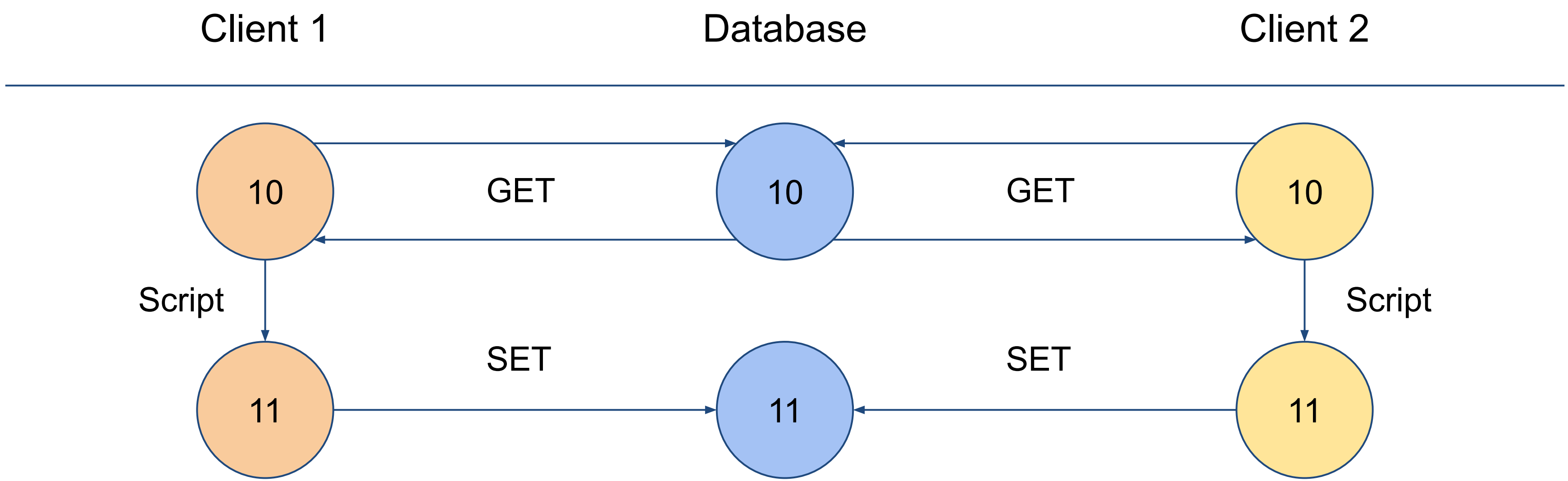
Redis - Atomic Operations



Redis - Atomic Operations



Redis - Atomic Operations



Using the script will cause these two increases to be applied incorrectly!

Redis - Temporary Values

Redis supports the creation of **temporary key-value pairs**. This can be achieved using the following operators.

- **EXPIRE** – Set the lifespan of a variable (in seconds).
- **TTL** – Returns the remaining lifespan of a variable (in seconds).

```
> SET key_name key_value
> EXPIRE time_to_live
```

```
> TTL key_name
```

PEXPIRE and **PTTL** achieves the same operations **in milliseconds**.

Redis - Temporary Values

The **TTL** operator returns the following values

- **remaining_lifespan** – The remaining time the key will be maintained.
- **-1** – The key **will never** expire.
- **-2** – The key **has already** expired.

Furthermore, not only it is possible to set the **TTL** when creating the key, but whenever the key is **SET**, the **TTL** is reset.

```
> SET key_name key_value EX time_to_live
```

Redis - Permanently Storing Temporary Variables

What if we realized we need to permanently store a temporary variable?

The **PERSIST** operator can be used to solve such a problem. It permanently stores a temporary variable on the database.

```
> PERSIST key_name
```

Redis - Complex Data Structure - Lists

Redis allows to store and manage complex data structures, like **LISTS**.
Creating a list is very simple, it's enough to use the following operators.

- **RPUSH** – Puts one or more new elements at the end of the list.
- **LPUSH** – Puts one or more new elements at the beginning of the list.

N.B. If the key does not exist, it will be created. A key with an **empty** list will be automatically **deleted**.

```
> LPUSH key_name key_value
```

Redis - Complex Data Structure - Lists

The **LRANGE** operator can be used to retrieve a subset of values from a list.

```
> LRANGE key_name first_index last_index
```

The values of **first_index** are positive numbers (e.g., 0, 1, etc.).

The values of **last_index** can be positive or negative numbers (e.g., -1, etc.)

Negative numbers (e.g., **-N**) means that the operator will return all values besides the last **N-1** values (e.g., -2 means that we are collecting all the elements, besides the last one).

Redis - Complex Data Structure - Lists

The **LPOP** and **RPOP** operators are used to remove items from lists.

- **LPOP** – Removes the first item of the list and returns it.
- **RPOP** – Removes the last item of the list and returns it.

```
> LPOP key_name
```

It is also possible to obtain the length of the list using the **LLEN** operator.

```
> LLEN key_name
```

Redis - Complex Data Structure - Sets

SETS are similar to lists but they don’t have a specific value order and each value can only appear once per set. **SETS** are useful as it is very quickly to test the existence of a value, which is not as easy when it comes to lists.

SETS can be managed using the following operators.

- **SADD** – Adds one or more values to a list. Returns 1 if the element is correctly added, 0 otherwise.
- **SREM** – Removes one or more values from a list. Returns 1 if the element is correctly removed, 0 otherwise.

```
> SADD key_name key_value_1 key_value_2 etc.
```

Redis - Complex Data Structure - Sets

Other very useful set operations are

- **SISMEMBER** – Returns 1 if the value is part of the set, 0 otherwise.
- **SMEMBER** – Returns all the members in the set.
- **SUNION** – Combines two or more sets into one set.

```
> SISMEMBER key_name key_value
```

```
> SMEMBER key_name
```

```
> SUNION key_name_1 key_name_2
```

Redis - Complex Data Structure - Sorted Sets

Since sets are not ordered, Redis introduced the concept of **ORDERED SET**.

In an **ORDERED SET**, each element of the set is assigned to a score that is used to define the order in the set.

An ordered set is created using the **ZADD** operator and its elements are retrieved using the **ZRANGE** operator.

```
> ZADD key_name key_score key_value_1 etc.
```

```
> ZRANGE key_name first_index last_index
```

Redis - Complex Data Structure - Hashes

HASHES are the best way to store objects. They are **mappings** between string fields and string values.

HASHES are managed using the following operators.

- **HSET** – Creates a hash and assigns one or more string fields with their values to the hash. Numbers are managed in the same way.
- **HGETALL** – Returns a hash with its fields.
- **HGET** – Returns a single field from a hash.

```
> HSET key_name field_1 value_1 field_2 value_2 etc.
```

ANY Questions?