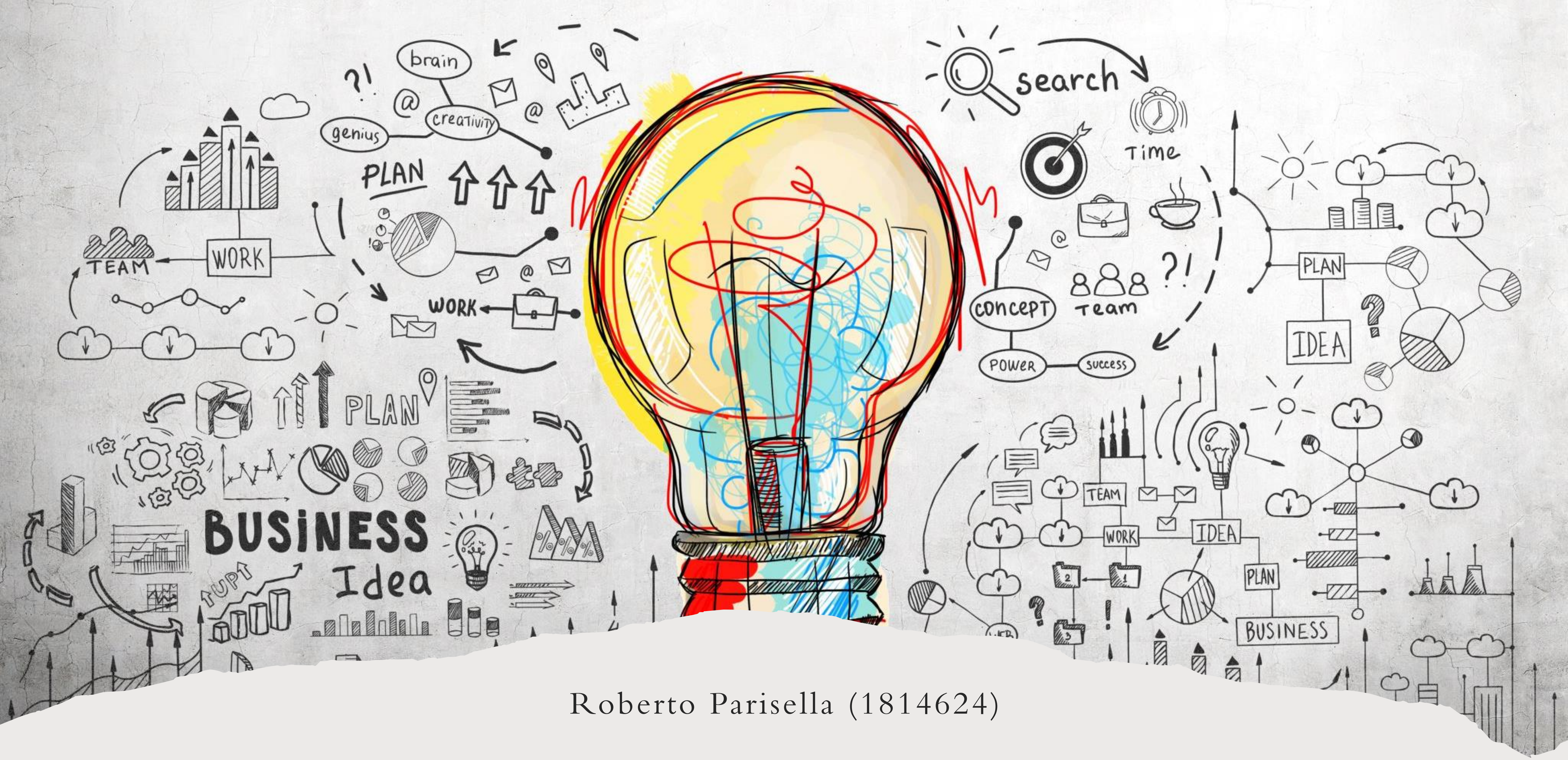


Roberto Parisella (1814624)

BIG DATA MANAGEMENT PROJECT

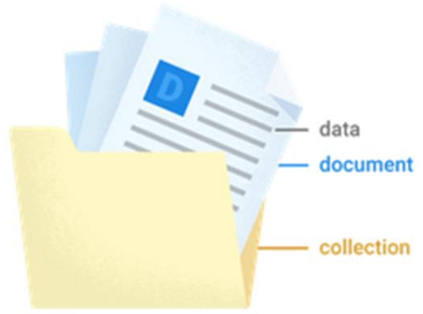
FIRESTORE: A GENERAL VIEW - KEY FEATURES

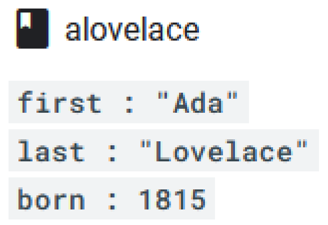
- Cloud Firestore is a cloud-hosted **NoSQL** database.
- Following Cloud Firestore's NoSQL **data model**, we store data in **documents** that contain fields mapping to values. These documents are stored in **collections**, which are containers for our documents that we can use to organize our data and build queries. Additionally, **querying** in Cloud Firestore is expressive, efficient, and flexible.
- Cloud Firestore allows to run sophisticated **ACID** transactions against our document data. This gives more flexibility in the way we structure our data.

* In this project we are not discussing Firestore from a cloud computing perspective

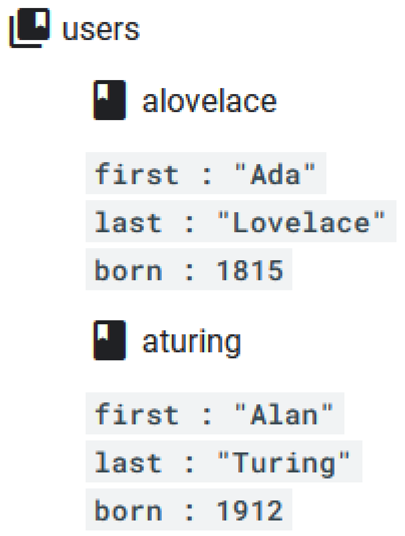
FIRESTORE: DATA MODEL - BASIC STRUCTURE

- In Cloud Firestore, the unit of storage is the **document**. A document is a lightweight record that contains fields, which map to values. Each document is identified by a name. We can treat documents as lightweight JSON records.
- Documents live in **collections**, which are simply containers for documents.
- Cloud Firestore is **schemaless**, so you have complete freedom over what fields you put in each document and what data types you store in those fields.





```
first : "Ada"
last : "Lovelace"
born : 1815
```

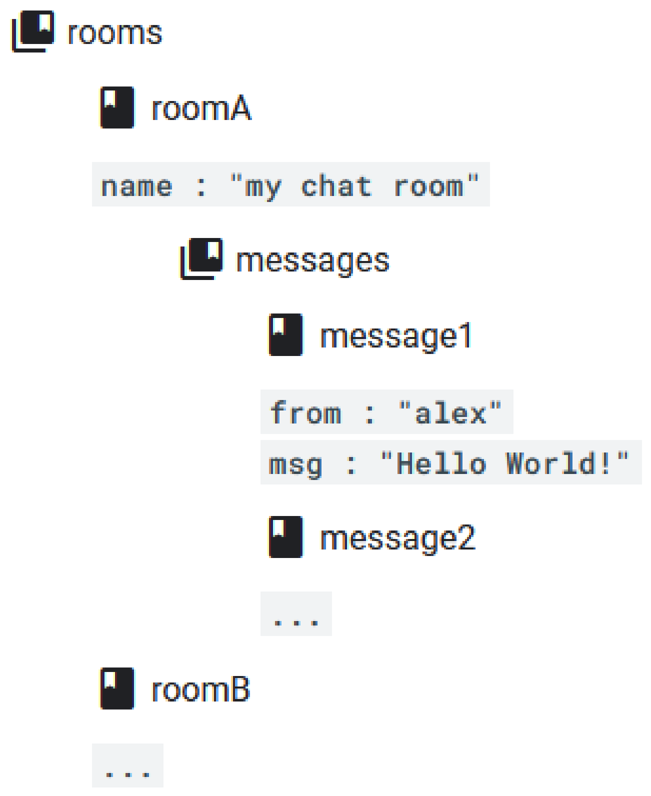


```
first : "Ada"
last : "Lovelace"
born : 1815
```

```
first : "Alan"
last : "Turing"
born : 1912
```

FIRESTORE: DATA MODEL - HIERARCHICAL DATA

- Cloud Firestore supports **hierarchical** organization.
- Consider an example chat app with messages and chat rooms: the best way to store messages in this scenario is by using **subcollections**. A subcollection is a collection associated with a specific document. It allow you to structure data hierarchically, making data easier to access.

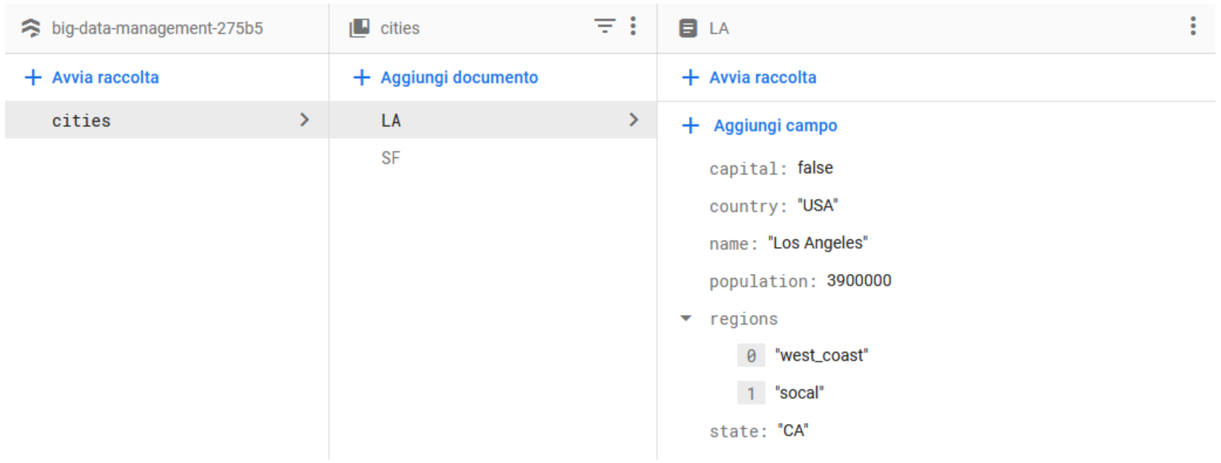


```
rooms
    roomA
    name : "my chat room"
        messages
            message1
            from : "alex"
            msg : "Hello World!"
            message2
            ...
    roomB
    ...
```

FIRESTORE: QUERY EXAMPLE - ADD DATA

```kotlin
val cities = db.collection("cities")

val data1 = hashMapOf(
        "name" to "San Francisco",
        "state" to "CA",
        "country" to "USA",
        "capital" to false,
        "population" to 860000,
        "regions" to listOf("west_coast", "norcal")
)
cities.document("SF").set(data1)

val data2 = hashMapOf(
        "name" to "Los Angeles",
        "state" to "CA",
        "country" to "USA",
        "capital" to false,
        "population" to 3900000,
        "regions" to listOf("west_coast", "socal")
)
cities.document("LA").set(data2)
```

* Code in Kotlin



```
capital: false
country: "USA"
name: "Los Angeles"
population: 3900000
regions
    0 "west_coast"
    1 "socal"
state: "CA"
```

* Firebase Firestore console

FIRESTORE: QUERY EXAMPLE - RETRIEVE DATA

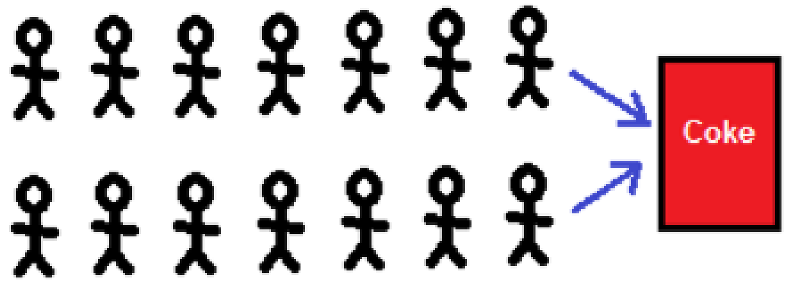
```
db.collection("cities")
        .whereEqualTo("capital", true)
        .get()
        .addOnSuccessListener { documents ->
            for (document in documents) {
                Log.d(TAG, "${document.id} => ${document.data}")
            }
        }
        .addOnFailureListener { exception ->
            Log.w(TAG, "Error getting documents: ", exception)
        }
```

- After creating a query object, we can easily retrieve the results:
- This query returns every city document where the regions field is an array that contains “west_coast” or “east_coast”

```
val citiesRef = db.collection("cities")

citiesRef.whereArrayContainsAny("regions", listOf("west_coast", "east_coast"))
```

FIRESTORE: ABOUT CONSISTENCY - ORGANIZATION

- Cloud Firestore allows to run sophisticated **ACID** transactions. There are atomic operations for reading and writing data. These operations can be:
  - **Transactions**: a transaction is a set of read and write operations on one or more documents.
  - **Batched Writes**: a batched write is a set of (only) write operations on one or more documents.



Concurrent: 2 queues, 1 vending machine

FIRESTORE: ABOUT CONSISTENCY - TRANSACTIONS

- A **transaction** consists of any number of get() (i.e. a read) operations followed by any number of write operations such as set(), update(), or delete(). In the case of a concurrent edit, Cloud Firestore runs the entire transaction again. For example, if a transaction reads documents and another client modifies any of those documents, Cloud Firestore retries the transaction. This feature ensures that the transaction runs on up-to-date and consistent data.
- Transactions **never partially** apply **writes**. All writes execute at the end of a successful transaction.
- A transaction can fail for the following reasons:
  - × The transaction contains read operations after write operations.
  - × The transaction read a document that was modified outside of the transaction. In this case, the transaction automatically runs again.

FIRESTORE: ABOUT CONSISTENCY - BATCHED WRITES

- **Batched writes** allow to execute multiple write operations as a single batch that contains any combination of set(), update(), or delete() operations. A batch of writes completes atomically and can write to multiple documents.
- Like transactions, batched writes are **atomic**. Unlike transactions, batched writes do not need to ensure that read documents remain un-modified which leads to fewer failure cases. They are not subject to retries or to failures from too many retries

FIRESTORE: ABOUT CONSISTENCY - DATA CONTENTION

- The **mobile/web SDKs** (Apple platforms, Android, Web, C++) use **optimistic concurrency controls** to resolve data contention. In the Mobile/Web SDKs, a transaction keeps track of all the documents you read inside the transaction. The transaction completes its write operations **only if** none of those documents changed during the transaction's execution.

* Mobile/web SDKs use optimistic concurrency controls, because they can operate in environments with high latency and an unreliable network connection. Locking documents in a high latency environment would cause too many data contention failures.

- The **server client libraries** (C#, Go, Java, Node.js, PHP, Python, Ruby) use **pessimistic concurrency controls** resolve data contention. In the server client libraries, transactions place **locks** on the documents they read. A transaction's lock on a document blocks other transactions, batched writes, and non-transactional writes from changing that document. A transaction releases its document locks at commit time.

* Server client libraries use pessimistic concurrency controls, because they assume low latency and a reliable connection to the database.

FIRESTORE: A PRACTICAL EXAMPLE - SCENARIO

- Suppose we want to develop a simple **social network** in which registered users can create posts.

1. For each **user** we want to register: first name, last name, birthdate, email, a profile image and the total number of posts.
2. For each **post** we want to register: the author of the post, title and content.

Note: this is the first idea, but if the social network becomes very popular, we can decide to add like reaction and the possibility to comment on any post.

FIRESTORE: A PRACTICAL EXAMPLE - POSSIBLE DATABASE

- If we decide to use a **relational database**, we could organize the database in two tables as the following:

```
1. users = [id, first_name,
   last_name, birthdate,
   email, image_url, n_posts]

2. posts = [id, user_id,
   title, content]
```

* "user_id" is a foreign key to "id" of user

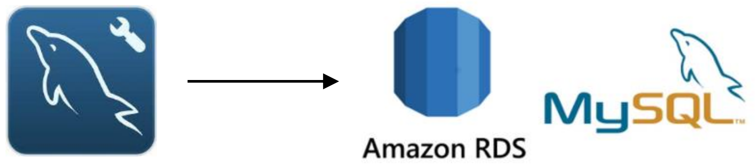
- If we decide to use a **document-based database**, we could organize the database in two collections as the following:

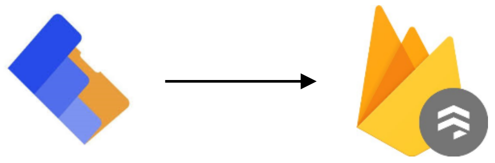
```
1. users -> user_id_1 -> values
         -> user_id_2 -> values
         -> ...

2. posts -> post_id_1 -> values
         -> post_id_2 -> values
         -> ...
```

FIRESTORE: A PRACTICAL EXAMPLE - TOOLS USED

- For the relational database I used an instance of **MySQL** configured through Amazon Relational Database Service (**Amazon RDS**, it is a collection of managed services that makes it easy to set up, operate, and scale databases in the cloud). I interacted with it through the GUI client **MySQL Workbench**.

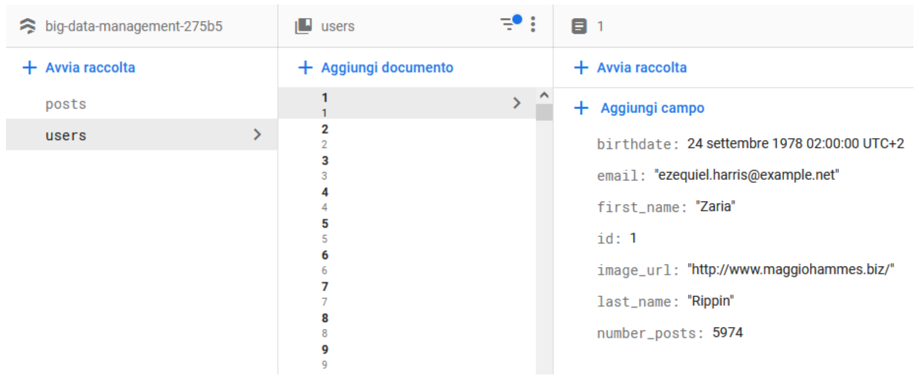
- For the document-based database I used **Cloud Firestore** and **Firefoo**, a powerful GUI client that facilitates interaction with the database and other Firebase services.





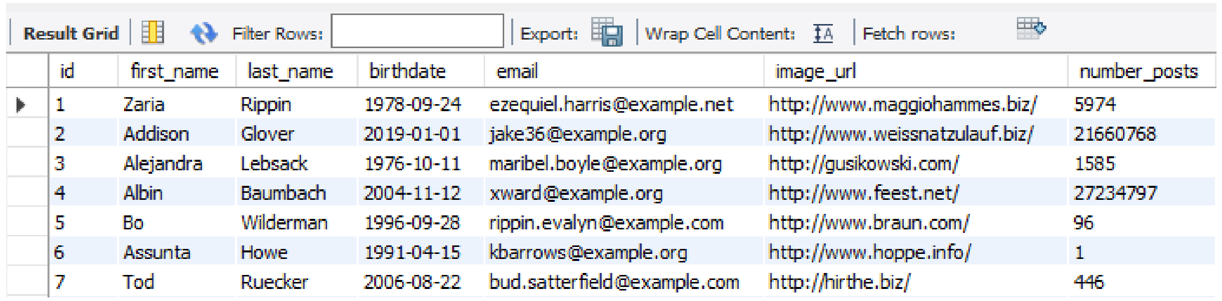
FIRESTORE: A PRACTICAL EXAMPLE - DATASET

- Starting from the schemas (“users” and “posts”) illustrated previously, I used an online tool to generate two different .csv files with **500 different dummy data** (to simulate a likely fully operational situation). Finally, I used the import functionality in MySQL Workbench and Firefoo to populate their respective databases.



```
birthdate: 24 settembre 1978 02:00:00 UTC+2
email: "ezequiel.harris@example.net"
first_name: "Zaria"
id: 1
image_url: "http://www.maggiohammes.biz/"
last_name: "Rippin"
number_posts: 5974
```

* Screen of Firestore database from Firebase console

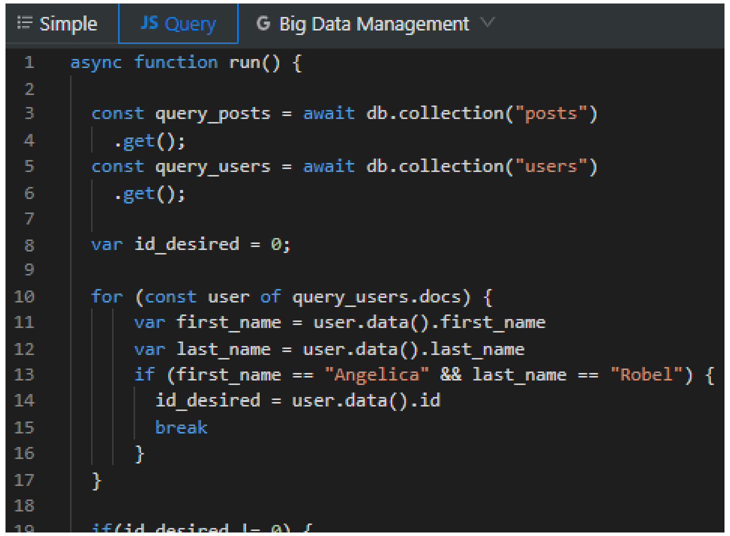


id	first_name	last_name	birthdate	email	image_url	number_posts
1	Zaria	Rippin	1978-09-24	ezequiel.harris@example.net	http://www.maggiohammes.biz/	5974
2	Addison	Glover	2019-01-01	jake36@example.org	http://www.weissnatzulauf.biz/	21660768
3	Alejandra	Lebsack	1976-10-11	maribel.boyle@example.org	http://gusikowski.com/	1585
4	Albin	Baumbach	2004-11-12	xward@example.org	http://www.feest.net/	27234797
5	Bo	Wilderman	1996-09-28	rippin.evalyn@example.com	http://www.braun.com/	96
6	Assunta	Howe	1991-04-15	kbarrows@example.org	http://www.hoppe.info/	1
7	Tod	Ruecker	2006-08-22	bud.satterfield@example.com	http://hirthe.biz/	446

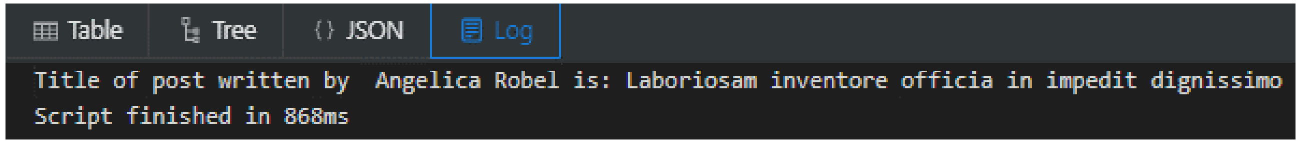
* Screen of MySQL “user” table taken from MySQL Workbench

FIRESTORE: A PRACTICAL EXAMPLE - SIMPLE JOIN IN FIRESTORE

- Suppose retrieving the title of the post written by the user Angelica Robel in Firestore.
- Cloud Firestore **does not support joining** collections in the backend. The client needs to retrieve the required data and performs the join locally. Whit large amount of data, obviously this operation is not very efficient.



```
async function run() {

  const query_posts = await db.collection("posts")
    .get();
  const query_users = await db.collection("users")
    .get();

  var id_desired = 0;

  for (const user of query_users.docs) {
      var first_name = user.data().first_name
      var last_name = user.data().last_name
      if (first_name == "Angelica" && last_name == "Robel") {
        id_desired = user.data().id
        break
      }
  }

  if(id_desired != 0) {
```



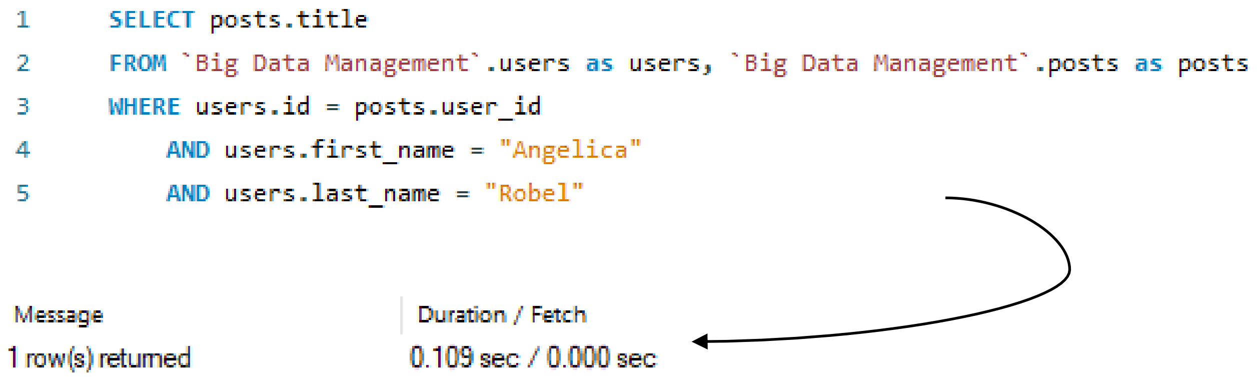
```
Title of post written by  Angelica Robel is: Laboriosam inventore officia in impedit dignissimo
Script finished in 868ms
```

× A simple join operation over a “small” dataset has finished in 868 ms, it’s not a very good result …

* Screenshot of Firefoo console; code in JavaScript using the Firebase Admin SDK

FIRESTORE: A PRACTICAL EXAMPLE - SIMPLE JOIN IN MYSQL

- Suppose retrieving the title of the post written by the user Angelica Robel in MySQL.
- It is well known that relational database **supports join** operation server side. So, the client can retrieve the data already elaborated.



```
SELECT posts.title
FROM `Big Data Management`.users as users, `Big Data Management`.posts as posts
WHERE users.id = posts.user_id
    AND users.first_name = "Angelica"
    AND users.last_name = "Robel"
```

```
Message                 Duration / Fetch
1 row(s) returned       0.109 sec / 0.000 sec
```

✓ MySQL has finished in 109 ms, it is much better than the result obtained with Firestore

* Screenshot of MySQL Workbench

FIRESTORE: A PRACTICAL EXAMPLE - NEW RELEASE

- Now suppose that the social network is having success, so we want to introduce **new features**: in the new release users can add a “like” reaction to other posts and comment them (with the possibility to reply to other comments).

× In **MySQL** we need to re-design the schema of the database: we could add another table “comments”, establish the new foreign key and put in “posts” other fields to keeping track the number of reactions and comments.

✓ **Cloud Firestore** has a dynamic schema that is much better for this purpose: we could add a new collection “comments” and put in “posts” other fields without re-design the schema of the database

FIRESTORE: A PRACTICAL EXAMPLE - CONCLUSION

- **Schema Design**: a relational SQL database is a better option for applications that require multi-row transactions such as an accounting system or for legacy systems that were built for a relational structure.
  However, NoSQL databases are much better suited for big data as flexibility is an important requirement which is fulfilled by their dynamic schema
- **Big Data Context**: NoSQL databases have a dynamic schema that is much better suited for big data as flexibility is an important requirement. An example of this is data from various social media sites such as Instagram, Twitter, Facebook, etc. NoSQL databases are horizontally scalable and can ultimately become larger and more powerful if required
- **Speed**: write operations in any NoSQL are much faster than any SQL database because it does not validate for schema, but query execution is faster in SQL databases because of structure. However, we can observe these differences only with GBs of data at once