Università
Ca'Foscari
Venezia

Master's Degree programme — Second Cycle
(D.M. 270/2004)
in Informatica — Computer Science

Final Thesis

—

Ca' Foscari
Dorsoduro 3246
30123 Venezia

Boosting meteorological statistics and analysis with big data assimilation: clusterization of a large amount of weather data

SUPERVISOR
Ch. Prof. Renzo Orsini

CANDIDATE
Fabio Rachello
Matriculation number 832795

ACADEMIC YEAR
2016/2017

BOOSTING METEOROLOGICAL STATISTICS AND ANALYSIS WITH BIG DATA ASSIMILATION: CLUSTERIZATION OF A LARGE AMOUNT OF WEATHER DATA

FABIO RACHELLO

Dept. of Environmental Sciences, Informatics and Statistics
Ca' Foscari University
Venice, Italy
October 2017

Stay hungry, stay foolish.

— Steve Jobs

ABSTRACT

For a meterological monitoring center, it's very important the presence of an archive with meteorological data of the past that is however continuously updated with new data added daily and hourly. This data is fundamentally used to make analysis and statistics about old events and not simply for meteorological forecasting. Storing a huge quantity of data with an usual relational database is obviously possible, but more data is stored more time is necessary in order that the query, used to retrieve requested information, has been elaborated.

We firstly present an analysis of the database used nowadays by a meterological monitoring center then the same database with more stored data in order to make a prediction of the future situation about the response time, secondly we consider a Big Data structure and we propose a solution to improve the response time, comparing the previous situation based on a relational database with the system based on NOSQL.

ACKNOWLEDGMENTS

All the time I spent for this graduation had been characterized by several difficultes but despite that every thing had been an oppurtunity to learn something useful for me and my life. During the time I worked on this thesis I received help and support from several people. I would like to thank my supervisor, prof. Renzo Orsini, for his availability to my questions and his prompt asnwers.

I have to thank Marco Gaion that gave me the opportunity to study and learn a lot about Big Data and how working inside a company is, and I have to thank also Alvise Spanò because without him I would not have met Marco and I would not have written this thesis.

I would also like to acknowledge all my friends. In particular huge thanks to Andrea that always supported me, but also Marco, Claudio, Mattia and Elena for the support and the great laughs.

Last but not least all my gratitude goes to Erika that always bore me and to my parents that supported and encouraged me in every moment of my life.

Fabio

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

CSV	Comma Separated Values
DB	Data Base
DBMS	Data Base Managament System
PK	Primary Key
FK	Foreign Key
JSON	JavaScript Object Notation
RDBMS	Relational Data Base Managament System
SQL	Structured Query Language
XML	eXtensible Markup Language

1 INTRODUCTION

Meteorological data is not simply equal to "meteorological forecasting", that is what most the people would think. In fact most of meteorological data is about past events, and this is very important, for example to study a specific phenomenon along a period of time or about a possible risk in a specific place, for example an avalanche. Actually all meteorological data is divided between several regional authorities, in this way it's easy to understand why often it's difficult to know which authority own the needed data and also if it's necessary to request this data, for example through a PEC mail.

For these reasons we want to make a database that contains all the Italian meteorological data, a system already adopted in other countries. In this way this system could let to save much time avoiding to search data that is so disorganized at the source. This could be very useful to supervise the realtime meteorological situation which a snowfall on a motorway could be a clear example. This tool could let to have a dense coverage of measurements that make easier to do all the operations previously described.

A meteorological system that has some similarities with the system previously described (a unique system that collect data from different sources) already exists, but it has only 100-200 stations that collect measurements. I collaborated with a company that offers many services like consultancy, project management, functional analysis, code review, and many others. This company manages system and data of a meteorological service society that not only provides meteorological forecasts but also performs climatological and statistical reports and analysis of past severe meteorological events such as storms (heavy rains, hails, strong winds), drought, heatwaves. Its work is to provide fast and suitable answers to the different and specific needs from several users like road operators, airports, supervisors of civil protection and supervisors of insurance's risk for agriculture. It offers professional meteorological services also for specific situations like concerts or other temporary events, providing real-time support (nowcasting) to notify the customer about the weather conditions. Obviously this work requires to store a lot of data, and when the amount of data recorded is too big, the time to get a response from the database could be too long.

The aim of this project is to have into the same system data collected by several thousands of stations, stations that are already collected and all respecting the OMM standards (Mondial Meteorological Organization).

1.1 CONTRIBUTIONS

The contributions of this thesis can be summarized as follows:

- We considered the actual situation (retrieving some response times from the Data Base (DB) actually in use by the meteorological society).
- We prepared a testing environment (based on PostgreSQL) divided into 3 different databases (all with the same structure of the one in use) to analyze the response time among different situations.
- We filled up the 3 Postgres's DB with fake data and iterate some queries to provide an average response time for each one.
- We prepared a testing environment (based on MongoDB NOSQL) divided into 3 different databases to analyze the response time among different situations.
- We filled up the 3 MongoDB's DB with fake data and iterate some queries to provide an average response time for each one.
- We make a comparison between the results deriving from the 2 testing environments.

1.2 THESIS STRUCTURE

The thesis is organized as follows:

- In Chapter 2 we provide the problem specification and the motivation to migrate to other systems.
- In Chapter 3 we summarize the basic concepts related to SQL and NOSQL systems.
- In Chapter 4 we describe the structure of the SQL Database and we analyse the problem, we describe the testing enviroment, the scripts to insert and retrieve data and we provide the results.
- In Chapter 5 we describe the structures of the NOSQL Database adopted providing 2 alternatives. Then we describe the testing enviroment, the scripts to insert and retrieve data and in the end we provide the results.
- In Chapter 6 we end summarizing all the results comparing them and the different strategies adopted and concluding with suggestions for future work.

2 PROBLEM DESCRIPTION

In this chapter we discuss the basic concepts related to Relational Data Base Managament System (RDBMS), and the spcification of the problem of large amount of data with relation to the response time. In Section 2.1 we provide a formal problem specification, while in Section 2.2 we explain why is more convenient to migrate from a Structured Query Language (SQL) system to a NOSQL one.

2.1 PROBLEM SPECIFICATION

Web during last years grew very fast and exponentially, both as technology as well as capability in many areas, from personals to the working ones. A very simple example that everyone knows is given by social networks, first among all is Facebook. At the beginning it was born as little platform inside a University and it grew so much that now it's difficult to find a person that doesn't know it or doesn't have a Facebook account. It's difficult to realize how much data is stored nowadays into this service and how much it is constantly added. With this growth one of the system used to store information it began to be every day more incompatible, this because with the growth of system slowly grew also the time needed to find and provide the data requested from the DB. Another example of this phenomenon is the storage of meteorological data: if we consider our country there are something like 6 000 stations that provide information regularly, and that data is constantly added to a database that consequently has more or less 350 000 new insertions every day. With a daily flow of data so big within 2 years the amount of data would be really huge, in the specific it could have more than 250 millions of records stored into the database. Search data into a table with more than 200 milions of records to retrieve the requested data could required a lot of time, and this is not acceptable. Usually this amount of data is not so difficult to be managed by SQL systems, but the kind of aggregate queries necessary for the specific application domain described in thesis could require much time.

For this reason, collaborating with a meterological monitoring center, we studied the actual situation of their DB, and provide a prediction of the future situation also considering the possibility to add a set of data of past years.

2.2 MOTIVATION TO MIGRATE

If we consider 350 000 new records every day, it means that there are more than 10 millions of new records every month and more than 127 millions of new records every year, with an amount of data so big in few years that a DB could have almost 500 millions of records. If then we consider the possibility to add also the data from some decades ago to nowadays the amount of data would be much more. The main problem in these cases is that bigger is the amount of data higher is the time necessary to get the result of a query from the DB. With this big quantity of data there could be an average waiting till 2-3 seconds or more to get the result of query and this waiting is not "user-friendly". Indeed if we see the Google's support information we could see that the 60% of mobile users expects that mobile websites load within 3 seconds, while 75% of people usually abandon the website if the loading takes more than 5 seconds. It's clear that in the case of a huge amount of data it's necessary to find another solution to storage and retrieve information from the DB.

3 RDBMS AND NOSQL SYSTEM

Usually Data Base Managament System (DBMS) are used to storage data. There are different types but the most common and diffused are relational ones that are SQL, System Query Language, based. There are also variations inside them, always based on SQL: MySQL, PostgreSQL, Microsoft SQL Server, SQLite, Oracle (Database) and many others. NoSQL systems, originally referring to "non SQL" or "non relational", in fact they are modeled in a way that differs from the relational systems, althought they may support SQL-like query languages. The large diffusion of NoSQL system is motivated from the needs of the so called Web 2.0 and the most famous companies that characterized it, e.g. Facebook, Google, Amazon and others.

3.1 RDBMS

A common way to organize and collect data is using a SQL system, in the specific we considered Postgres. SQL is a domain-specific language used to manage data inside a RDBMS. In this kind of systems data is structured in tables, each table is divided in columns that are a set of data of a particular type (one for each row), and rows where each one represent a single item. There is always a column marked as Primary Key (PK), Primary Key, a unique value that let to distinguish one item from another, there could be also one or more columns marked as Foreign Key (FK), Foreign Key, this value usually is not unique. A fundamental concept is the relation between two tables, this is made possibile by the link between a PK of one table and a FK of another table. Another important concept is the JOIN clause that combine columns from one or more tables, usually using the relations between these tables. But what combine all these concepts and make possible to use and retrieve the data is the QUERY, a statement that using specific clauses, predicates and expressions let to retrieve data from the DB.

A functionality frequently adopted is the capability of *stored procedures* that is the possibility to save a complex query as a function easier to call back when necessary, especially if that query is usually used.

Another important concept is the indexing, indeed an index is used to speed up the search in the DB, it can be used to efficiently find all row matching some column in the query and then walking through only that subset of the table to find exact matches. PostgreSQL pro-

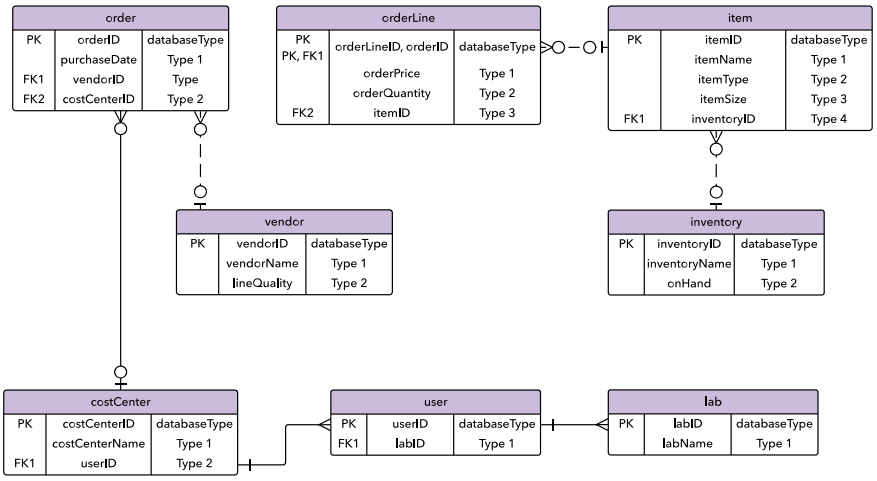

Figure 3.1: Example of SQL's scheme

vides the index methods B-tree, hash, GiST, and GIN, it would be possible to define a own index but it could be fairly complicated [2].

An example of a RDBMS's scheme is shown in Figure 3.1.

3.2 NOSQL SYSTEM

Another way to organize and collect data, adopted in the last years usually with big Databases, is NoSQL. The power of NoSQL is its scalability, that let the system to handle a a growing amount of work, distributed system capability and eventually consinstency, that is to achieve high availability informally guarantees that, if there are not new updates to a specific item, eventually all accesses to that item will return the last updated value [4][6].

There are different NoSQL frameworks: key-value stores, document stores, tabular stores, object data stores or graph databases. Key-value store is a model similar to hash tables, in fact there is a connection between a key and one or more values. Document store is similar to key-value model because stored objects are associated via characters string keys, but this model provide some structure and econding of the data, and there are different encodings like eXtensible Markup Language (XML), JavaScript Object Notation (JSON) or BSON [3][7]. HBase is an example of tabular store, composed by a column-oriented data layout that has fault tolerant methods to store and manage data. This model is also compression capable allowing to increase the amount of data that can be represented [7][4]. Graph databases

are common too, their idea is simple, they are composed by a collection of vertices, that are the modelled entities, these vertices could be connected by edges, that indicate how these vertices are related. A big advantage of Graph Databases is their flexibility, in fact they are naturally additive, that means that add new relationships does not disturb existing queries and functionalities [10][6].

NoSQL systems are characterized by horizontal scalability that is that system load could be shared to many servers [9].

Schemas of Document based databases are flexible, in fact if SQL's tables have a specific structure, NoSQL does not have any imposition on Documents' one. This let documents to match data fields of any entity even if data is varied [3].

In the specific we consider MongoDB where, instead of SQL's tables, there are Collections, where there are fields (equivalent to SQL's columns) and Documents (equivalent to SQL's rows). Data records are stored as BSON documents, BSON is a binary representation of JSON documents, though it contains more data types than JSON [1].

Here there are the Primary and Foreign keys too, but there are mainly two ways to make relations between items: the first is through *references* the second uses *embedded documents* [3], an example is shown in Figure 3.2. MongoDB's references are equal to the SQL's ones, they works in the same way, an embedded document means that we insert an item of a Collection inside the item of another. There are many ways to retrieve data and the most important and used are Collection's methods like MAPREDUCE, AGGREGATE and FIND.

The first method *db.collection.mapReduce()* is a method with two important parameters: map, a function that associates or "maps" a value with a key and emits the key and value pair, and reduce, function that "reduces" to a single object all the values associated with a particular key [1].

db.collection.aggregate(pipeline, options) has many pipeline operators: *$project*, works as the SQL's PROJECT, *$lookup*, that is the equivalent of a LEFT JOIN, *$sort*, *$unwind*, that 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, it outputs "n" documents where "n" is the number of array elements and can be zero for an empty array [1].

db.collection.find(query, projection) selects documents in a collection or view. It has two optional parameters: the first where a selection filter can be applied using query operators likefor exapmple *$gt* and *$lt* that match values that are respectively greater and lower of the specified parameter, the second where can be specified the fields to display or not in the result [1].

There is something similar to the PostgreSQL's *stored procedures* and this is *stored Javascript*, indeed there is a special system collection named system.js that can store JavaScript functions for reuse.

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
            },
  access: {
             level: 5,
             group: "dev"
           }
}
```

Embedded sub-document

Embedded sub-document

Figure 3.2: Example of NOSQL's embedded documents

Indexing is present too, indexes in MongoDB are similar to indexes in other database systems indeed that store a small portion of the collection's data set in an easy to traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field. The ordering of the index entries supports efficient equality matches and range-based query operations [1].

4 POSTGRESQL ANALYSIS

Usually insert and retrieve data from a DB are operations simple and fast, but when the amount of data stored is huge both the insert as well as the query could take more time. Indeed, if there is a lot of data, indexes should be used to reduce the response time of queries but this affects the inseriments because for each new record added, the system must check all the indexes related and this could take a lot of time with respect to the number of records that already exist in the table. In this Section 4.1 we explain the structure of the DB, in Section 4.2 we describe the testing enviroment used, in Sections 4.3 and 4.4 we present briefly the scripts used to insert and retrieve data respectively and in the Section 4.5 we illustrate the response's times obtained.

4.1 DB STRUCTURE DESCRIPTION

The structure of the DB is shown in 4.1 except for few tables omitted because not relevant for this analysis. In the specific we have 7 tables, in particular district and city have a FK to the PK in the table "region", table "station" have 2 FK to the PK in the tables "city" and "station_model" respectively, in the end the tables "measurement_daily" and "measurement_realtime" have each one a FK to to the table "station". This structure shows that:

- each district is linked to one region.
- each city is linked to one region that will be the same of the district that it is linked to.
- each station is linked to one city and to one station_model.
- each measurement (both daily both realtime) is linked to one station.

Moreover in the tables there are many indexes, all implemented with B-tree method, to improve the search of the queries, as we alredy explained in Section 3.1. We will list the indexes grouped by table:

District:

- fki_district_region_id_fkey: index on region_id's column.

City:

- fki_city_district_id_fkey: index on district_id's column.

%

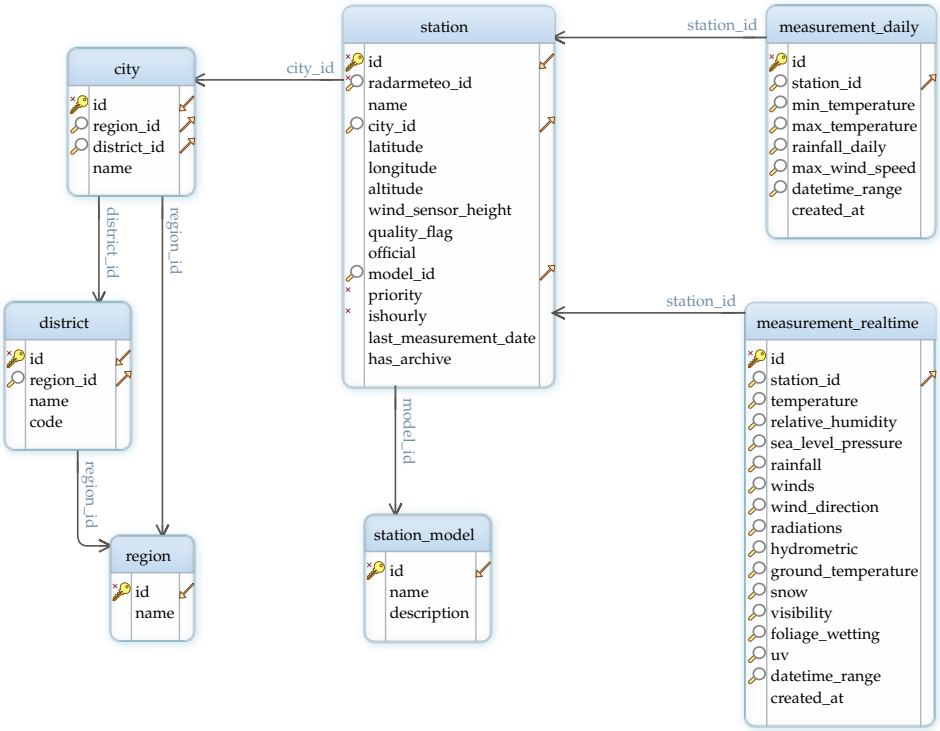

Figure 4.1: Database's structure

- fki_city_region_id_fkey: index on region_id's column.

Station:

- fki_station_city_id_fkey: index on city_id's column.
- fki_station_model_id_fkey: index on model_id's column.
- model_idx: index on model_id's column.
- radarmeteo_id_idx: index on radarmeteo_id's column.

Measurement_daily:

- datetime_range_idx: index on datetime_range's column.
- fki_misuration_daily_station_id_fkey: index on station_id's column.
- station_max_temp_daily_idx: index on columns of station_id and max_temperature.
- station_max_wind_daily_idx: index on columns of station_id and max_wind_speed.
- station_min_temp_daily_idx: index on columns of station_id and min_temperature.
- station_rainfall_daily_idx: index on columns of station_id and rainfall_daily.

Measurement_realtime:

- fki_misuration_realtime_station_id_fkey: index on station_id's column.
- station_datetime_realtime_unique_idx: unique index on columns of station_id and datetime_range.
- station_foliage_realtime_idx: index on columns of station_id and foliage_wetting.
- station_ground_temp_realtime_idx: index on columns of station_id and ground_temperature.
- station_hydro_realtime_idx: index on columns of station_id and hydrometric.
- station_radiations_realtime_idx: index on columns of station_id and radiations.
- station_rainfall_realtime_idx: index on columns of station_id and rainfall.

- station_rel_hum_realtime_idx: index on columns of station_id and relative_humidity.

- station_sea_realtime_idx: index on columns of station_id and sea_level_pressure.

- station_snow_realtime_idx: index on columns of station_id and snow.

- station_temperature_realtime_idx: index on columns of station_id and temperature.

- station_uv_realtime_idx: index on columns of station_id and uv.

- station_visibility_realtime_idx: index on columns of station_id and visibility.

- station_wind_dir_realtime_idx: index on columns of station_id and wind_direction.

- station_winds_realtime_idx: index on columns of station_id and winds.

There are also many stored procedures, that we briefly defined in Section 3.1, here we show only the most relevant in relation to the problem we are dealing with:

- insert_measurement_daily(radarmeteo_id character varying, min_temperature smallint, max_temperature smallint, rainfall_daily smallint, max_wind_speed smallint, datetime_range timestamp with time zone):

```
DECLARE
    station_sel_id integer;
    measurement_id integer;
    now timestamp with time zone;
    archive boolean;
BEGIN
    SELECT id FROM station WHERE upper(trim(station.
        radarmeteo_id)) = upper(trim(insert_measurement_daily.
        radarmeteo_id)) INTO station_sel_id;
    IF FOUND THEN
        SELECT id FROM measurement_daily WHERE station_id =
            station_sel_id AND measurement_daily.
            datetime_range = insert_measurement_daily.
            datetime_range INTO measurement_id;
        IF FOUND THEN
            IF min_temperature IS NOT NULL THEN
                UPDATE measurement_daily SET min_temperature =
                    insert_measurement_daily.min_temperature
                    WHERE id = measurement_id;
            END IF;
            IF max_temperature IS NOT NULL THEN
                UPDATE measurement_daily SET max_temperature =
                    insert_measurement_daily.max_temperature
                    WHERE id = measurement_id;
            END IF;
            IF rainfall_daily IS NOT NULL THEN
                UPDATE measurement_daily SET rainfall_daily =
                    insert_measurement_daily.rainfall_daily
                    WHERE id = measurement_id;
            END IF;
            IF max_wind_speed IS NOT NULL THEN
                UPDATE measurement_daily SET max_wind_speed =
                    insert_measurement_daily.max_wind_speed
                    WHERE id = measurement_id;
            END IF;
        ELSE
            SELECT current_timestamp INTO now;
            INSERT INTO measurement_daily VALUES (default,
                station_sel_id, min_temperature,
                max_temperature, rainfall_daily,
                max_wind_speed, datetime_range, now) RETURNING
                 id INTO measurement_id;
            SELECT has_archive FROM station WHERE id =
                station_sel_id INTO archive;
            IF NOT archive THEN
                PERFORM station_id FROM measurement_realtime
                    WHERE station_id = station_sel_id AND
```

```
                        measurement_realtime.datetime_range < NOW
                        () - '1 hour'::INTERVAL LIMIT 1;
                    IF FOUND THEN
                        UPDATE station SET has_archive = TRUE
                            WHERE id = station_sel_id;
                    ELSE
                        PERFORM station_id FROM measurement_daily
                            WHERE station_id = station_sel_id AND
                            measurement_daily.datetime_range < NOW
                            () - '1 hour'::INTERVAL LIMIT 1;
                        IF FOUND THEN
                            UPDATE station SET has_archive = TRUE
                                WHERE id = station_sel_id;
                        END IF;
                    END IF;
                END IF;
            END IF;
        RETURN measurement_id;
        ELSE
            RETURN -1;
        END IF;
END;
```

- insert_measurement_realtime(radarmeteo_id character varying, temperature smallint, relative_humidity smallint, sea_level_pressure smallint, rainfall smallint, winds smallint, wind_direction smallint, radiations smallint, hydrometric smallint, ground_temperature smallint, snow smallint, visibility integer, foliage_wetting boolean, uv smallint, datetime_range timestamp with time zone):

```
DECLARE
    station_sel_id integer;
    measurement_id integer;
    now timestamp with time zone;
    archive boolean;
BEGIN
    SELECT id FROM station WHERE upper(trim(station.
        radarmeteo_id)) = upper(trim(
        insert_measurement_realtime.radarmeteo_id)) INTO
        station_sel_id;
    IF FOUND THEN
        SELECT id FROM measurement_realtime WHERE station_id =
             station_sel_id AND measurement_realtime.
            datetime_range = insert_measurement_realtime.
            datetime_range INTO measurement_id;
        IF FOUND THEN
            IF temperature IS NOT NULL THEN
                UPDATE measurement_realtime SET temperature =
                    insert_measurement_realtime.temperature
                    WHERE id = measurement_id;
            END IF;
            IF relative_humidity IS NOT NULL THEN
```

```
15                  UPDATE measurement_realtime SET
                        relative_humidity =
                        insert_measurement_realtime.
                        relative_humidity WHERE id =
                        measurement_id;
16              END IF;
17              IF sea_level_pressure IS NOT NULL THEN
18                  UPDATE measurement_realtime SET
                        sea_level_pressure =
                        insert_measurement_realtime.
                        sea_level_pressure WHERE id =
                        measurement_id;
19              END IF;
20              IF rainfall IS NOT NULL THEN
21                  UPDATE measurement_realtime SET rainfall =
                        insert_measurement_realtime.rainfall WHERE
                         id = measurement_id;
22              END IF;
23              IF winds IS NOT NULL THEN
24                  UPDATE measurement_realtime SET winds =
                        insert_measurement_realtime.winds WHERE id
                         = measurement_id;
25              END IF;
26              IF wind_direction IS NOT NULL THEN
27                  UPDATE measurement_realtime SET wind_direction
                         = insert_measurement_realtime.
                        wind_direction WHERE id = measurement_id;
28              END IF;
29              IF radiations IS NOT NULL THEN
30                  UPDATE measurement_realtime SET radiations =
                        insert_measurement_realtime.radiations
                        WHERE id = measurement_id;
31              END IF;
32              IF hydrometric IS NOT NULL THEN
33                  UPDATE measurement_realtime SET hydrometric =
                        insert_measurement_realtime.hydrometric
                        WHERE id = measurement_id;
34              END IF;
35              IF ground_temperature IS NOT NULL THEN
36                  UPDATE measurement_realtime SET
                        ground_temperature =
                        insert_measurement_realtime.
                        ground_temperature WHERE id =
                        measurement_id;
37              END IF;
38              IF snow IS NOT NULL THEN
39                  UPDATE measurement_realtime SET snow =
                        insert_measurement_realtime.snow WHERE id
                        = measurement_id;
40              END IF;
41              IF visibility IS NOT NULL THEN
42                  UPDATE measurement_realtime SET visibility =
                        insert_measurement_realtime.visibility
                        WHERE id = measurement_id;
43              END IF;
44              IF foliage_wetting IS NOT NULL THEN
```

```
                UPDATE measurement_realtime SET foliage_wetting = insert_measurement_realtime.foliage_wetting WHERE id = measurement_id;
            END IF;
            IF uv IS NOT NULL THEN
                UPDATE measurement_realtime SET uv = insert_measurement_realtime.uv WHERE id = measurement_id;
            END IF;
        ELSE
            SELECT current_timestamp INTO now;
            INSERT INTO measurement_realtime VALUES (default, station_sel_id, temperature, relative_humidity, sea_level_pressure, rainfall, winds, wind_direction, radiations, hydrometric, ground_temperature, snow, visibility, foliage_wetting, uv, datetime_range, now)
                RETURNING id INTO measurement_id;
            INSERT INTO measurement_last VALUES (station_sel_id, temperature, relative_humidity, sea_level_pressure, rainfall, winds, wind_direction, radiations, hydrometric, ground_temperature, snow, visibility, foliage_wetting, uv, datetime_range, now)
                ON CONFLICT ON CONSTRAINT measurement_last_pk DO
                UPDATE SET
                    temperature = insert_measurement_realtime.temperature
                    ,relative_humidity = insert_measurement_realtime.relative_humidity
                    ,sea_level_pressure = insert_measurement_realtime.sea_level_pressure
                    ,rainfall = insert_measurement_realtime.rainfall
                    ,winds = insert_measurement_realtime.winds
                    ,wind_direction = insert_measurement_realtime.wind_direction
                    ,radiations = insert_measurement_realtime.radiations
                    ,hydrometric = insert_measurement_realtime.hydrometric
                    ,ground_temperature = insert_measurement_realtime.ground_temperature
                    ,snow = insert_measurement_realtime.snow
                    ,visibility = insert_measurement_realtime.visibility
                    ,foliage_wetting = insert_measurement_realtime.foliage_wetting
                    ,uv = insert_measurement_realtime.uv
```

```
                    ,datetime_range =
                        insert_measurement_realtime.
                        datetime_range
                    ,created_at = now
                WHERE measurement_last.station_id =
                    station_sel_id;
            SELECT has_archive FROM station WHERE id =
                station_sel_id INTO archive;
            IF NOT archive THEN
                PERFORM station_id FROM measurement_realtime
                    WHERE station_id = station_sel_id AND
                    measurement_realtime.datetime_range < NOW
                    () - '1 hour'::INTERVAL LIMIT 1;
                IF FOUND THEN
                    UPDATE station SET has_archive = TRUE
                        WHERE id = station_sel_id;
                ELSE
                    PERFORM station_id FROM measurement_daily
                        WHERE station_id = station_sel_id AND
                        measurement_daily.datetime_range < NOW
                        () - '1 hour'::INTERVAL LIMIT 1;
                    IF FOUND THEN
                        UPDATE station SET has_archive = TRUE
                            WHERE id = station_sel_id;
                    END IF;
                END IF;
            END IF;
        END IF;
    RETURN measurement_id;
    ELSE
        RETURN -1;
    END IF;
END;
```

- fe_get_measurements_archive_daily(stations integer[], min_datetime_range timestamp with time zone>, max_datetime_range timestamp with time zone, filtervalue text, min_min_temperature_daily smallint, max_min_temperature_daily smallint, min_max_temperature_daily smallint, max_max_temperature_daily smallint, min_rainfall_daily smallint, max_rainfall_daily smallint, min_max_wind_speed_daily smallint, max_max_wind_speed_daily smallint):

```
DECLARE
    stmt varchar = '';
    selectStmt varchar = 'SELECT station.radarmeteo_id,
      date(datetime_range) AS datetime_range,
      district.name AS district,
      city.name AS city,
      min_temperature::real / 10::real,
      max_temperature::real / 10::real,
      rainfall_daily::real / 10::real,
      max_wind_speed::real / 10::real,
      extract(epoch FROM datetime_range) as unix_datetime ';
```

```

    fromStmt varchar = ' FROM measurement_daily
      INNER JOIN station ON measurement_daily.station_id = station.id
      INNER JOIN city ON station.city_id = city.id
      INNER JOIN district ON city.district_id = district.id ';

    whereStmt varchar = ' WHERE measurement_daily.station_id = ANY($1) ';
    orderStmt varchar = ' ORDER BY station.radarmeteo_id ';

    stationIds integer[] = NULL;
BEGIN
    IF filtervalue = 'min_temperature_daily' THEN
        whereStmt = whereStmt || ' AND (min_temperature BETWEEN '|| min_min_temperature_daily ||' *10 AND '|| max_min_temperature_daily ||' *10) AND min_temperature IS NOT NULL ';
    END IF;
    IF filtervalue = 'max_temperature_daily' THEN
        whereStmt = whereStmt || 'AND (max_temperature BETWEEN '|| min_max_temperature_daily ||' *10 AND '|| max_max_temperature_daily ||' *10) AND max_temperature IS NOT NULL ';
    END IF;
    IF filtervalue = 'rainfall_daily' THEN
        whereStmt = whereStmt || 'AND (rainfall_daily BETWEEN '|| min_rainfall_daily ||' *10 AND '|| max_rainfall_daily ||' *10) AND rainfall_daily IS NOT NULL ';
    END IF;
        IF filtervalue = 'max_wind_speed_daily' THEN
        whereStmt = whereStmt || 'AND (max_wind_speed BETWEEN '|| min_max_wind_speed_daily ||' *10 AND '|| max_max_wind_speed_daily ||' *10) AND max_wind_speed IS NOT NULL ';
    END IF;
    IF min_datetime_range IS NOT NULL AND max_datetime_range IS NOT NULL THEN
        whereStmt = whereStmt || 'AND (datetime_range BETWEEN '''|| min_datetime_range ||''' AND '''|| max_datetime_range ||''') AND datetime_range IS NOT NULL ';
    END IF;

    stmt = stmt || selectStmt || fromStmt || whereStmt || orderStmt || ';';
    RETURN QUERY EXECUTE stmt USING (stations);
    Return;
END;
```

- fe_get_measurements_archive_hourly(stations integer[], filtervalue text, min_datetime_range timestamp with time zone, max_datetime_range timestamp with time zone, min_temperature smallint, max_temperature smallint, min_relative_humidity smallint, max_relative_humidity

smallint, min_sea_level_pressure smallint, max_sea_level_pressure smallint, min_rainfall smallint, max_rainfall smallint, min_winds smallint, max_winds smallint, min_wind_direction smallint, max_wind_direction smallint, min_radiations smallint, max_radiations smallint, min_hydrometric smallint, max_hydrometric smallint, min_ground_temperature smallint, max_ground_temperature smallint, min_snow smallint, max_snow smallint, min_visibility smallint, max_visibility smallint, foliage_wetting boolean, min_uv smallint, max_uv smallint):

```
DECLARE
    station_id integer;
    comma boolean = false;
    stmt varchar = '';
    selectStmt varchar = 'SELECT station.radarmeteo_id,
      timezone(''UTC'', datetime_range) AS datetime_range,
      extract(epoch FROM datetime_range) as unix_datetime,
      district.name AS district,
      city.name AS city,
      temperature::real / 10::real,
      rainfall::real / 10::real,
      winds::real / 10::real,
      wind_direction ';

    valueStmt varchar = '';
    asStmt varchar = ' AS filtered_value ';
    fromStmt varchar = ' FROM measurement_realtime
      INNER JOIN station ON measurement_realtime.station_id =
          station.id
      INNER JOIN city ON station.city_id = city.id
      INNER JOIN district ON city.district_id = district.id ';

    whereStmt varchar = ' WHERE measurement_realtime.
        station_id = ANY($1) ';
    orderStmt varchar = ' ORDER BY datetime_range, station.
        radarmeteo_id ';
BEGIN
    filterValue = lower(trim(filterValue));

    IF filterValue = 'temperature' THEN
        whereStmt = whereStmt || ' AND (temperature BETWEEN '
            || min_temperature ||' *10 AND '|| max_temperature
             ||' *10)
        AND temperature IS NOT NULL ';
    END IF;
    IF filterValue = 'relative_humidity' THEN
        valueStmt = ', (relative_humidity::real)::text AS
            filtered_value ';
        whereStmt = whereStmt || ' AND (relative_humidity
            BETWEEN '|| min_relative_humidity ||' AND '||
            max_relative_humidity ||')
        AND relative_humidity IS NOT NULL ';
    END IF;
    IF filterValue = 'sea_level_pressure' THEN
```

```
        valueStmt = ', (sea_level_pressure::real / 10::real)::text AS filtered_value ';
        whereStmt = whereStmt || ' AND (sea_level_pressure BETWEEN '|| min_sea_level_pressure ||' *10 AND '|| max_sea_level_pressure ||' *10)
        AND sea_level_pressure IS NOT NULL ';
    END IF;
    IF filterValue = 'rainfall' THEN
        whereStmt = whereStmt || ' AND (rainfall BETWEEN '|| min_rainfall ||' *10 AND '|| max_rainfall ||' *10)
        AND rainfall IS NOT NULL ';
    END IF;
    IF filterValue = 'winds' THEN
        whereStmt = whereStmt || ' AND (winds BETWEEN '|| min_winds ||' *10 AND '|| max_winds ||' *10)
        AND winds IS NOT NULL ';
    END IF;
    IF filterValue = 'wind_direction' THEN
        whereStmt = whereStmt || ' AND (wind_direction BETWEEN '|| min_wind_direction ||' AND '|| max_wind_direction ||')
        AND wind_direction IS NOT NULL ';
    END IF;
    IF filterValue = 'radiations' THEN
        valueStmt = ', (radiations::real)::text AS filtered_value ';
        whereStmt = whereStmt || ' AND (radiations BETWEEN '|| min_radiations ||' AND '|| max_radiations ||')
        AND radiations IS NOT NULL ';
    END IF;
    IF filterValue = 'hydrometric' THEN
        valueStmt = ', (hydrometric::real)::text AS filtered_value ';
        whereStmt = whereStmt || ' AND (hydrometric BETWEEN '|| min_hydrometric ||' AND '|| max_hydrometric ||')
        AND hydrometric IS NOT NULL ';
    END IF;
    IF filterValue = 'ground_temperature' THEN
        valueStmt = ', (ground_temperature::real / 10::real)::text AS filtered_value ';
        whereStmt = whereStmt || ' AND (ground_temperature BETWEEN '|| min_ground_temperature ||' *10 AND '|| max_ground_temperature ||' *10)
        AND ground_temperature IS NOT NULL ';
    END IF;
    IF filterValue = 'snow' THEN
        valueStmt = ', (snow::real)::text AS filtered_value ';
        whereStmt = whereStmt || ' AND (snow BETWEEN '|| min_snow ||' AND '|| max_snow ||')
        AND snow IS NOT NULL ';
    END IF;
    IF filterValue = 'visibility' THEN
        valueStmt = ', (visibility::real)::text AS filtered_value ';
        whereStmt = whereStmt || ' AND (visibility BETWEEN '|| min_visibility ||' AND '|| max_visibility ||')
```

```
        AND visibility IS NOT NULL ';
    END IF;
    IF filterValue = 'foliage_wetting' THEN
        valueStmt = ', foliage_wetting::text AS filtered_value ';
        whereStmt = whereStmt || ' AND foliage_wetting = ' || foliage_wetting;
    END IF;
    IF filterValue = 'uv' THEN
        valueStmt = ', (uv::real)::text AS filtered_value ';
        whereStmt = whereStmt || ' AND (uv BETWEEN '|| min_uv ||' AND '|| max_uv ||')
        AND uv IS NOT NULL ';
    END IF;
    IF min_datetime_range IS NOT NULL AND max_datetime_range IS NOT NULL THEN
        whereStmt = whereStmt || ' AND (datetime_range BETWEEN '''|| min_datetime_range ||''' AND '''|| max_datetime_range ||''') AND datetime_range IS NOT NULL ';
    END IF;

    IF valueStmt = '' THEN
        valueStmt = valueStmt || ', NULL::text AS filtered_value ';
    END IF;

    stmt = stmt || selectStmt || valueStmt || fromStmt || whereStmt || orderStmt || ';';
    RETURN QUERY EXECUTE stmt USING (stations);
    Return;
END;
```

- fe_get_measurements_realtime(stations integer[], filtervalue text):

```
DECLARE
    station_id integer;
    comma boolean = false;
    stmt varchar = '';
    selectStmt varchar = 'SELECT station.radarmeteo_id,
      datetime_range,
      extract(epoch FROM datetime_range) as unix_datetime,
      district.name AS district,
      city.name AS city,
      temperature::real / 10::real,
      rainfall::real / 10::real,
      winds::real / 10::real,
      wind_direction ';

    asStmt varchar = ' AS filtered_value ';
    filterStmt varchar = '';
    fromStmt varchar = ' FROM measurement_last
      INNER JOIN station ON measurement_last.station_id = station.id
```

```
        INNER JOIN city ON station.city_id = city.id
        INNER JOIN district ON city.district_id = district.id ';

    whereStmt varchar = ' WHERE measurement_last.station_id =
        ANY($1) ';
    orderStmt varchar = ' ORDER BY station.radarmeteo_id ';
BEGIN
    filterValue = lower(trim(filterValue));

    IF filterValue != 'temperature' AND filterValue != '
        rainfall' AND filterValue != 'winds' AND filterValue
        != 'wind_direction' THEN
        IF filterValue = 'foliage_wetting' THEN
            filterStmt = ', (' || filterValue;
        ELSE
            filterStmt = ', (' || filterValue || '::real';
        END IF;
        IF filterValue = 'sea_level_pressure' THEN
            filterStmt = filterStmt || ' / 10::real';
        END IF;
        filterStmt = filterStmt || ')::text' || asStmt;
    END IF;

    IF filterStmt = '' THEN
        filterStmt = filterStmt || ', NULL::text' || asStmt;
    END IF;

    stmt = stmt || selectStmt || filterStmt || fromStmt ||
        whereStmt || orderStmt || ';';
    RETURN QUERY EXECUTE stmt USING (stations);
    Return;
END;
```

From this *stored procedure* we can notice that it exploits another table, "measurement_last", as cache, so this could be considered an optimization with relation to the results.

4.2 TESTING OVERVIEW

Now that we described how the system is structured, we need to do many queries to understand how much is the response time of the system and this not only considering the actual size of the system but also the size of the system in a future context where would be more and more data than now. Obviously we can't do that on the system actually in use because we would overload the system as well as because if an error would occur the system should be backuped, in both the situations it would be unusable for the users for a period of time.

So we prepared 3 DB on a server, all with the same number of regions (20), districts (100) and cities (8000) that are more or less the

	DB_SMALL	DB_REAL	DB_BIG
station	1 000	25 000	200 000
measurement_daily	6 400	3 200 000	35 000 000
measurement_realtime	28 000	18 000 000	200 000 000

Table 4.1: Tables' composition (number of records)

quantity in our State, station_model includes the types of meteorological stations that are used. In the DB actually in use there are 10 models, we considered the same number of station's models for all the databases, the other tables are composed as shown in Table 4.1.

4.3 INSERTION SCRIPT

Once created the 3 databases we used a SQL's dump file, obatined from the DB actually in use, to reproduce the same structure on the 3 databases, then we filled them up. To do this we created a script in Python with many functions, the script need also the module *psycopg2* to be installed and imported to establish connections with PostgreSQL's server. We don't describe all the functions used, but only the ones more interesting for the aim of this test, other functions create the other necessary data (like regions, districts and cities) and save them into local variables to make easier the next inseriments. All the main functions have been thought to be used also with multithreading, that is because we noticed some improvements in distributing the inseriments among more threads. In fact every of these functions have 3 inputs, the DB where insert the data (DB_SMALL, DB_REAL or DB_BIG), *beg* and *end* that indicate the range of records that every single thread has to insert. All the function have the first lines (3 to 5) equal, in these lines the function create the connection and set the relative cursor.

The first one used is the fuction necessary to fill up the records inside the table "station":

```
1 # function that inserts stations
2 def insert_stations(beg, end, db):
3         conn=psycopg2.connect(database=db, user="user",
              password="password", host="127.0.0.1", port="5434"
              )
4         print("Opened database successfully")
5         cur = conn.cursor()
6         for count in range(beg, end):
7                 radar_id="IT"
8                 name="station_"
```

```
                city=""
                lat=0.0
                lon=0.0
                alt=0
                wind_height=0
                quality=False
                off=False
                model=""
                priority=0
                ishourly=False
                if (count<10):
                        radar_id+="0000"
                        name+="0000"
                else:
                        if (count<100):
                                radar_id+="000"
                                name+="000"
                        else:
                                if (count<1000):
                                        radar_id+="00"
                                        name+="00"
                                else:
                                        if (count<10000):
                                                radar_id+="0"
                                                name+="0"
                radar_id+=str(count)
                name+=str(count)
                city=cities[random.randint(0, 7999)]
                lat=random.uniform(37, 47)
                lon=random.uniform(7, 18)
                alt=random.randint(0, 3000)
                wind_height=random.randint(0, 15)
                if (random.randint(0,1)):
                        quality=True
                if (random.randint(0,1)):
                        off=True
                model=stations_model[random.randint(0, 9)]
                priority=random.randint(1,10)
                if (random.randint(0,1)):
                        ishourly=True
                cur.execute("SELECT public.insert_station(CAST
                    ('"+radar_id+"' AS VARCHAR), CAST('"+name+
                    "' AS VARCHAR), CAST('"+city+"' AS VARCHAR
                    ), CAST("+str(lat)+" AS REAL), CAST("+str(
                    lon)+" AS REAL), CAST("+str(alt)+" AS
                    SMALLINT), CAST("+str(wind_height)+" AS
                    SMALLINT), "+str(quality)+", "+str(off)+",
                     CAST('"+model[0]+"' AS VARCHAR), CAST("+
                    str(priority)+" AS SMALLINT), "+str(
                    ishourly)+")")
                conn.commit()
        conn.close()
        print "Records STATIONS inserted successfully";
```

The function, after having established the connection, has a cycle that iterates for *end-beg* times and for each iteration:

- sets the variables to default values (lines 7 to 33).
- sets the variables, randomly or not (lines 34 to 48).
- executes the query, saves the change and closes the connection (lines 49-51).

The second function used is the fuction necessary to fill up the records inside the table "measurement_daily", this function has one more input n_stations that adapts the inseriment with relation to the DB where it has been used:

```
1  # function that inserts measurements daily
2  def insert_measurement_daily(beg, end, db, n_stations):
3          conn=psycopg2.connect(database=db, user="user",
               password="password", host="127.0.0.1", port="5434"
               )
4          print("Opened database successfully")
5          cur = conn.cursor()
6          delay=0
7          for c in range(beg, end):
8                  station=random.randint(1,n_stations)
9                  radar_id="IT"
10                 temp_min=0
11                 temp_max=0
12                 rainfall=0
13                 winds=0
14                 if (station<10):
15                         radar_id+="0000"
16                 else:
17                         if (station<100):
18                                 radar_id+="000"
19                         else:
20                                 if (station<1000):
21                                         radar_id+="00"
22                                 else:
23                                         if (station<10000):
24                                                 radar_id+="0"
25                 radar_id+=str(station)
26                 temp_min=random.randint(-50,10)
27                 temp_max=random.randint(20,60)
28                 rainfall=random.randint(0,200)
29                 winds=random.randint(0,300)
30                 delay=c/n_stations*100
31                 time=datetime.utcnow() - timedelta(days=delay)
32                 time = time.isoformat()
33                 cur.execute("SELECT public.
                       insert_measurement_daily(CAST('"+radar_id+
                       "' AS VARCHAR), CAST('"+str(temp_min)+"'
                       AS SMALLINT), CAST('"+str(temp_max)+"' AS
                       SMALLINT), CAST('"+str(rainfall)+"' AS
                       SMALLINT), CAST('"+str(winds)+"' AS
                       SMALLINT), CAST('"+str(time)+"' AS
                       TIMESTAMP WITH TIME ZONE))")
34                 conn.commit()
35         conn.close()
```

```
36          print "Records MEASUREMENTS_DAILY inserted
                successfully";
```

The function, after having established the connection, has a cycle that iterates for *end-beg* times and for each iteration:

- a delay variable is setted to the default value, this variable is used to distribute the datetime_range values of the records (line 6).
- a random value for station variable is setted (line 8).
- sets the variables to default values (lines 9 to 24).
- sets the variables, randomly or not (lines 25 to 30).
- sets the time variable with relation to the delay and convert it using isoformat standard (lines 31-32).
- executes the query, saves the change and closes the connection (lines 33-35).

The third function used is the fuction necessary to fill up the records inside the table "measurement_realtime". This function has the same inputs of the last one:

```
1  # function that inserts measurements realtime
2  def insert_measurement_real(beg, end, db, n_stations):
3          conn=psycopg2.connect(database=db, user="user",
               password="password", host="127.0.0.1", port="5434"
               )
4          print("Opened database successfully")
5          cur = conn.cursor()
6          delay=0
7          for c in range(beg, end):
8                  station=random.randint(1,n_stations)
9                  radar_id="IT"
10                 temp=0
11                 rel_hum=0
12                 sea_press=0
13                 rainfall=0
14                 winds=0
15                 wind_dir=0
16                 rad=0
17                 hydro=0
18                 ground_temp=0
19                 snow=0
20                 visibility=0
21                 foliage_wet=False
22                 uv=0
23                 if (station<10):
24                         radar_id+="0000"
25                 else:
26                         if (station<100):
```

```
27                                  radar_id+="000"
28                          else:
29                                  if (station<1000):
30                                          radar_id+="00"
31                                  else:
32                                          if (station<10000):
33                                                  radar_id+="0"
34                  radar_id+=str(station)
35                  temp=random.randint(-50,60)
36                  rel_hum=random.randint(0,100)
37                  sea_press=random.randint(960,1060)
38                  rainfall=random.randint(0,200)
39                  winds=random.randint(0,300)
40                  wind_dir=random.randint(0,360)
41                  rad=random.randint(0,2000)
42                  hydro=random.randint(0,3000)
43                  ground_temp=random.randint(-50,60)
44                  snow=random.randint(0,100)
45                  visibility=random.randint(0,20000)
46                  if (random.randint(0,1)):
47                          foliage_wet=True
48                  uv=random.randint(0,20)
49                  delay=c/n_stations
50                  time=datetime.utcnow() - timedelta(hours=delay
                        )
51                  time=time.isoformat()
52                  cur.execute("SELECT public.
                        insert_measurement_realtime(CAST('"+
                        radar_id+"' AS VARCHAR), CAST('"+str(temp)
                        +"' AS SMALLINT), CAST('"+str(rel_hum)+"'
                        AS SMALLINT), CAST('"+str(sea_press)+"' AS
                         SMALLINT), CAST('"+str(rainfall)+"' AS
                        SMALLINT), CAST('"+str(winds)+"' AS
                        SMALLINT), CAST('"+str(wind_dir)+"' AS
                        SMALLINT), CAST('"+str(rad)+"' AS SMALLINT
                        ), CAST('"+str(hydro)+"' AS SMALLINT),
                        CAST('"+str(ground_temp)+"' AS SMALLINT),
                        CAST('"+str(snow)+"' AS SMALLINT), CAST('"
                        +str(visibility)+"' AS INTEGER), '"+str(
                        foliage_wet)+"', CAST('"+str(uv)+"' AS
                        SMALLINT), CAST('"+str(time)+"' AS
                        TIMESTAMP WITH TIME ZONE))")
53                  conn.commit()
54          conn.close()
55          print "Records MEASUREMENTS_REALTIME inserted
                successfully";
```

The function, after having established the connection, has a cycle that iterates for *end-beg* times and for each iteration:

- a delay variable is setted to the default value, this variable is used to distribute the datetime_range values of the records (line 6).
- a random value for station variable is setted (line 8).

- sets the variables to default values (lines 9 to 33).
- sets the variables, randomly or not (lines 34 to 49).
- sets the time variable with relation to the delay and convert it using isoformat standard (lines 50-51).
- executes the query, saves the change and closes the connection (lines 52-54).

We have to report that inseriments' step hasn't been so easy as expected, beacuse into DB_BIG at the beginning, after some millions of data inserted we estimated to end within 1 week or 2 at least, but after 30 millions of records inserted we realised that the estimate was wrong. This happened because inseriments became slower, indeed, at the beginning the time required to insert 10 000 000 of records was one day, instead, with 30 000 000 - 40 000 000 records already stored, the time required to insert the same amount of data was several days. This probably was caused by indexing updating at every inseriment, so we removed every constraint and filled up the tables with psql's COPY command and in input a Comma Separated Values (CSV) file with the remaining data and restoring the constraints once finished.

4.4 QUERY SCRIPT

Once filled up all the databases the next step is to understand how much time is needed to a query to terminate and provide the result. To do this we used the *stored procedures* already inside the DB that were explained in Section 4.1: fe_get_measurements_archive_daily and fe_get_measurements_realtime.

The first stored procedure has as first input an array of stations, then 2 *datetime_range* values that we chose randomly, a *filtervalue* that we decided to be always "min_temperature", and the 2 values, minimum and maximum, for the *filtervalue* chosen; the second stored procedure has as first input an array of stations, then a *filtervalue* that we decided to be always "temperature".

To consider a more realistic situation as well as the worst case feasible for each query we implemented 2 versions, the first one with a little amount of stations the second one with all the stations. The amount of stations is around the 10-20% of the total number of stations of the DB considered and for queries inside measurement_daily and inside measurement_realtime, as shown in Table4.2.

As it was said in Section 4.2, we need to understand how much is the response time of queries, to do that we should consider any possible influence of external factors so each query should be iterated for several times. Greater is that number of iterations, better is the result

	DB_SMALL	DB_REAL	DB_BIG
all_stations	1 000	25 000	200 000
partial_stations	200	2 000	20 000

Table 4.2: Stations' variables

but, obviously, also the time to wait. Therefore after several attempts, we found a good compromise with 200 iterations. With relation to the *datetime_range* values of the *stored procedure* fe_get_measurements_archive_daily, because we want analyse the worst case possible during the tests, we always used the minimum and maximum values (of the table) as inputs.

Following we reported the main functions of the script that worked on the table "measurement_daily" (we omitted the first part of the script where arrays with the stations were created, 2 array for each DB with random values for the partial one, as shown in Table 4.2):

```
# connection to the DB
conn1=psycopg2.connect(database="db_small", user="user",
    password="password", host="127.0.0.1", port="5432")
print("Opened database successfully")
cur1 = conn1.cursor()
conn2=psycopg2.connect(database="db_real", user="user",
    password="password", host="127.0.0.1", port="5432")
print("Opened database successfully")
cur2 = conn2.cursor()
conn3=psycopg2.connect(database="db_big", user="user",
    password="password", host="127.0.0.1", port="5432")
print("Opened database successfully")
cur3 = conn3.cursor()

for q in range(1, number_of_queries+1):
min_min_temp=random.randint(-50, -25)
max_min_temp=random.randint(-15, 10)

        # DB_SMALL queries
        start_time = datetime.utcnow()
        for i in range(1,n_cycles+1):
                cur1.execute("SELECT public.
                    fe_get_measurements_archive_daily(ARRAY["+
                    str(partial_stations_small)+"], CAST('"+
                    str(s_time)+"' AS TIMESTAMP WITH TIME ZONE
                    ), CAST('"+str(e_time)+"' AS TIMESTAMP
                    WITH TIME ZONE), 'min_temperature', '"+str
                    (min_min_temp)+"', '"+str(max_min_temp)+"
                    ')")
                row=cur1.fetchall()
        times_daily_few=(datetime.utcnow() - start_time).
            total_seconds()
        # times_daily_few written to file

```

```
        start_time = datetime.utcnow()
        for i in range(1,n_cycles+1):
                cur1.execute("SELECT public.fe_get_measurements_archive_daily(ARRAY["+str(all_stations_small)+"], CAST('"+str(s_time)+"' AS TIMESTAMP WITH TIME ZONE), CAST('"+str(e_time)+"' AS TIMESTAMP WITH TIME ZONE), 'min_temperature', '"+str(min_min_temp)+"', '"+str(max_min_temp)+"')")
                row=cur1.fetchall()
        times_daily_all=(datetime.utcnow() - start_time).total_seconds()
        # times_daily_all written to file

        # DB_REAL queries
        start_time = datetime.utcnow()
        for i in range(1,n_cycles+1):
                cur2.execute("SELECT public.fe_get_measurements_archive_daily(ARRAY["+str(partial_stations_med)+"], CAST('"+str(s_time)+"' AS TIMESTAMP WITH TIME ZONE), CAST('"+str(e_time)+"' AS TIMESTAMP WITH TIME ZONE), 'min_temperature', '"+str(min_min_temp)+"', '"+str(max_min_temp)+"')")
                row=cur2.fetchall()
        times_daily_few=(datetime.utcnow() - start_time).total_seconds()
        # times_daily_few written to file

        start_time = datetime.utcnow()
        for i in range(1,n_cycles+1):
                cur2.execute("SELECT public.fe_get_measurements_archive_daily(ARRAY["+str(all_stations_med)+"], CAST('"+str(s_time)+"' AS TIMESTAMP WITH TIME ZONE), CAST('"+str(e_time)+"' AS TIMESTAMP WITH TIME ZONE), 'min_temperature', '"+str(min_min_temp)+"', '"+str(max_min_temp)+"')")
                row=cur2.fetchall()
        times_daily_all=(datetime.utcnow() - start_time).total_seconds()
        # times_daily_all written to file

        # DB_BIG queries
        start_time = datetime.utcnow()
        for i in range(1,n_cycles+1):
                cur3.execute("SELECT public.fe_get_measurements_archive_daily(ARRAY["+str(partial_stations_big)+"], CAST('"+str(s_time)+"' AS TIMESTAMP WITH TIME ZONE), CAST('"+str(e_time)+"' AS TIMESTAMP WITH TIME ZONE), 'min_temperature', '"+str(min_min_temp)+"', '"+str(max_min_temp)+"')")
```

```
50                  row=cur3.fetchall()
51          times_daily_few=(datetime.utcnow() - start_time).
                total_seconds()
52          # times_daily_few written to file
53
54          start_time = datetime.utcnow()
55          for i in range(1,n_cycles+1):
56                  cur3.execute("SELECT public.
                        fe_get_measurements_archive_daily(ARRAY["+
                        str(all_stations_big)+"], CAST('"+str(
                        s_time)+"' AS TIMESTAMP WITH TIME ZONE),
                        CAST('"+str(e_time)+"' AS TIMESTAMP WITH
                        TIME ZONE), 'min_temperature', '"+str(
                        min_min_temp)+"', '"+str(max_min_temp)+"')
                        ")
57                  row=cur3.fetchall()
58          times_daily_all=(datetime.utcnow() - start_time).
                total_seconds()
59          # times_daily_all written to file
```

The script is so divided :

- firstly create the connections to the 3 databases with relative cursors (lines 2-10).
- for each iteration in the cycle:
  - random values are setted fot variables min_min_temperature and max_min_temperature (lines 13-14).
  - for each DB the same query with the same values of temperature is iterated with partial number of stations as well as with all of them, writing to file the time needed to the iteration of that query (lines 16-59).

The script that worked on the table "measurement_realtime" is very similar to the script above, the only difference is only about the inputs of the *stored procedure*:

```
1  # connection to the DB
2  conn1=psycopg2.connect(database="db_small", user="user",
       password="password", host="127.0.0.1", port="5432")
3  print("Opened database successfully")
4  cur1 = conn1.cursor()
5  conn2=psycopg2.connect(database="db_real", user="user",
       password="password", host="127.0.0.1", port="5432")
6  print("Opened database successfully")
7  cur2 = conn2.cursor()
8  conn3=psycopg2.connect(database="db_big", user="user",
       password="password", host="127.0.0.1", port="5432")
9  print("Opened database successfully")
10 cur3 = conn3.cursor()
11
12 for q in range(1, number_of_queries+1):
13
```

```
14          # DB_SMALL queries
15          start_time = datetime.utcnow()
16          for i in range(1,n_cycles+1):
17                  cur1.execute("SELECT public.
                        fe_get_measurements_realtime(ARRAY["+str(
                        partial_stations_small)+"], 'temperature')
                        ")
18                  row=cur1.fetchall()
19          times_daily_few=(datetime.utcnow() - start_time).
                total_seconds()
20          # times_daily_few written to file
21
22          start_time = datetime.utcnow()
23          for i in range(1,n_cycles+1):
24                  cur1.execute("SELECT public.
                        fe_get_measurements_realtime(ARRAY["+str(
                        all_stations_small)+"], 'temperature')")
25                  row=cur1.fetchall()
26          times_daily_all=(datetime.utcnow() - start_time).
                total_seconds()
27          # times_daily_all written to file
28
29          # DB_REAL queries
30          start_time = datetime.utcnow()
31          for i in range(1,n_cycles+1):
32                  cur2.execute("SELECT public.
                        fe_get_measurements_realtime(ARRAY["+str(
                        partial_stations_med)+"], 'temperature')")
33                  row=cur2.fetchall()
34          times_daily_few=(datetime.utcnow() - start_time).
                total_seconds()
35          # times_daily_few written to file
36
37          start_time = datetime.utcnow()
38          for i in range(1,n_cycles+1):
39                  cur2.execute("SELECT public.
                        fe_get_measurements_realtime(ARRAY["+str(
                        all_stations_med)+"], 'temperature')")
40                  row=cur2.fetchall()
41          times_daily_all=(datetime.utcnow() - start_time).
                total_seconds()
42          # times_daily_all written to file
43
44          # DB_BIG queries
45          start_time = datetime.utcnow()
46          for i in range(1,n_cycles+1):
47                  cur3.execute("SELECT public.
                        fe_get_measurements_realtime(ARRAY["+str(
                        partial_stations_big)+"], 'temperature')")
48                  row=cur3.fetchall()
49          times_daily_few=(datetime.utcnow() - start_time).
                total_seconds()
50          # times_daily_few written to file
51
52          start_time = datetime.utcnow()
53          for i in range(1,n_cycles+1):
```

```
                cur3.execute("SELECT public.fe_get_measurements_realtime(ARRAY["+str(all_stations_big)+"], 'temperature')")
                row=cur3.fetchall()
        times_daily_all=(datetime.utcnow() - start_time).total_seconds()
        # times_daily_all written to file
```

4.5 RESULTS

Results of the queries on the table "measurement_daily" are shown in Table 4.3. Considering the queries with a partial set of stations response's times are accetables, while considering queries with all the set of stations times are too high. A mean of 29 000 seconds indicates that each query needs 145-146 seconds to be completed on the DB_BIG, while on DB_REAL mean is 1 550 second sthat correspond to 7,75 seconds for each query and these times are too much in both cases.

Results of the queries with the *stored procedure* fe_get_ measurements_hourly are shown in Table 4.5, response's times are higher than fe_get_measurements_daily's ones and this is coherent with respect to the amount of data of these 2 tables. Here for the bigger database we did only 10 queries beacuse the times were always too high, that is more than 1 000 seconds for each single query. From Table 4.4 we could notice they are lower than the response's times of fe_get_measurements_hourly and of fe_get_measurements_daily too, this probably beacuse they use the table measurement_last as cache, this let to have better response's times.

In Figure 4.2 we could notice that using the *stored procedure* with a partial list of stations there is a big increasing of response's times, and also in Figure 4.3, using all the stations it is clear that the increment is even bigger.

Average response's times of the *stored procedures* fe_get_ measurements_archive_daily and fe_get_ measurements_archive_hourly, Figures 4.2, 4.3 and 4.4, show how the trend is the same: higher is the amount of data higher is the response's time.

We also tried to cluster the tables to get some improvement but we did not get significant results, the difference was very low.

	DB_SMALL		DB_REAL		DB_BIG	
	FEW	ALL	FEW	ALL	FEW	ALL
QUERY 1	1,14	3,65	18,00	1455,21	141,01	32415,94
QUERY 2	1,14	3,87	20,17	1618,38	142,39	29064,72
QUERY 3	1,12	3,87	20,83	1735,59	142,72	30090,95
QUERY 4	1,12	3,88	20,68	1597,14	142,58	26274,81
QUERY 5	1,19	3,70	17,86	1372,86	139,51	28934,93
QUERY 6	1,16	4,38	21,59	1644,58	143,25	28347,66
QUERY 7	1,10	3,88	20,02	1579,93	142,16	29136,82
QUERY 8	1,11	3,84	19,81	1534,87	142,56	28998,97
QUERY 9	1,13	3,69	18,66	1385,84	141,61	29532,28
QUERY 10	1,13	4,33	22,66	1689,19	141,39	30593,07
QUERY 11	1,11	3,85	20,32	1698,72	142,43	31758,34
QUERY 12	1,14	3,89	19,62	1498,43	140,79	32287,84
QUERY 13	1,10	3,71	18,40	1513,65	140,99	27484,81
QUERY 14	1,11	3,65	17,98	1445,55	141,17	27119,35
QUERY 15	1,12	4,23	17,87	1402,24	141,03	27312,71
QUERY 16	1,12	4,12	20,45	1583,83	141,33	28633,73
QUERY 17	1,16	3,89	21,35	1635,55	141,54	27553,36
QUERY 18	1,14	3,91	19,94	1584,67	141,83	28634,38
QUERY 19	1,13	4,04	20,74	1605,06	142,12	30655,24
QUERY 20	1,13	3,78	18,69	1478,74	140,90	28906,77
MEAN	1,13	3,91	19,78	1553,00	141,67	29186,84

Table 4.3: Results of fe_get_measurements_daily (seconds)

	DB_SMALL		DB_REAL		DB_BIG	
	FEW	ALL	FEW	ALL	FEW	ALL
QUERY 1	0,90	2,65	7,06	58,86	64,98	546,25
QUERY 2	0,88	2,48	6,60	56,84	61,10	543,38
QUERY 3	0,85	2,50	6,58	56,83	61,01	516,62
QUERY 4	0,85	2,48	6,55	56,00	59,83	534,89
QUERY 5	0,86	2,51	6,58	57,25	61,13	543,85
QUERY 6	0,86	2,51	6,55	56,70	61,14	543,56
QUERY 7	0,87	2,52	6,58	56,92	60,89	544,63
QUERY 8	0,86	2,47	6,57	56,66	60,84	543,46
QUERY 9	0,86	2,49	6,60	56,84	61,04	542,91
QUERY 10	0,85	2,47	6,55	56,71	61,55	546,87
QUERY 11	0,89	2,49	6,56	56,87	60,91	544,75
QUERY 12	0,86	2,51	6,56	56,81	61,05	545,37
QUERY 13	0,87	2,50	6,56	56,75	59,92	512,05
QUERY 14	0,88	2,49	6,94	56,01	59,96	511,70
QUERY 15	0,86	2,52	6,56	55,96	60,20	512,33
QUERY 16	0,87	2,50	6,59	56,55	61,02	535,66
QUERY 17	0,88	2,47	6,62	57,06	59,76	542,35
QUERY 18	0,85	2,48	6,73	56,45	59,95	543,26
QUERY 19	0,86	2,51	6,58	56,78	60,12	541,33
QUERY 20	0,88	2,51	6,60	56,85	60,04	540,72
MEAN	0,87	2,50	6,63	56,78	60,82	536,80

Table 4.4: Results of fe_get_measurements_realtime (seconds)

	DB_SMALL		DB_REAL		DB_BIG	
	FEW	ALL	FEW	ALL	FEW	ALL
QUERY 1	9,47	30,53	2492,25	30595,69	200000+	200000+
QUERY 2	8,70	31,91	2423,48	31846,99	200000+	200000+
QUERY 3	11,37	29,32	2067,99	28882,83	200000+	200000+
QUERY 4	12,34	31,46	2312,42	29221,96	200000+	200000+
QUERY 5	11,20	29,76	2096,06	27181,23	200000+	200000+
QUERY 6	9,00	29,36	2142,64	29751,44	200000+	200000+
QUERY 7	9,44	29,80	2211,36	29593,92	200000+	200000+
QUERY 8	12,08	29,40	2359,16	30317,80	200000+	200000+
QUERY 9	11,92	30,28	2417,56	30264,32	200000+	200000+
QUERY 10	11,08	31,40	2361,72	30537,00	200000+	200000+
QUERY 11	9,36	31,92	2280,96	28983,32	-	-
QUERY 12	9,72	31,40	2340,52	27993,36	-	-
QUERY 13	11,28	31,80	2184,08	28409,80	-	-
QUERY 14	11,96	31,44	2201,24	29597,64	-	-
QUERY 15	12,04	31,16	2341,92	30305,32	-	-
QUERY 16	10,00	29,88	2385,64	29743,08	-	-
QUERY 17	9,40	30,04	2395,12	30305,76	-	-
QUERY 18	9,76	29,60	2257,40	27855,56	-	-
QUERY 19	11,48	29,52	2209,80	28426,16	-	-
QUERY 20	10,72	31,84	2189,32	29401,00	-	-
MEAN	10,62	30,59	2283,53	29460,71	200000+	200000+

Table 4.5: Results of fe_get_measurements_hourly (seconds)

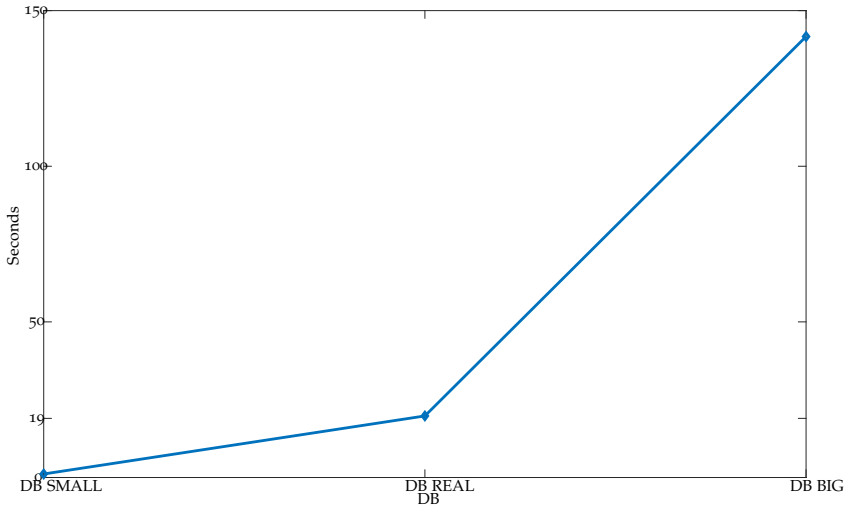

Figure 4.2: Plot of fe_get_measurements_archive_daily's results with few stations

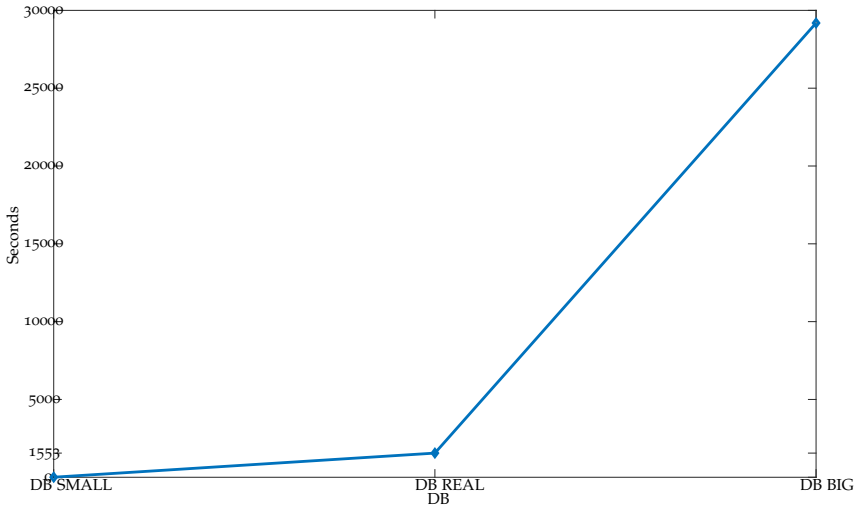

Figure 4.3: Plot of fe_get_measurements_archive_daily's results with all stations

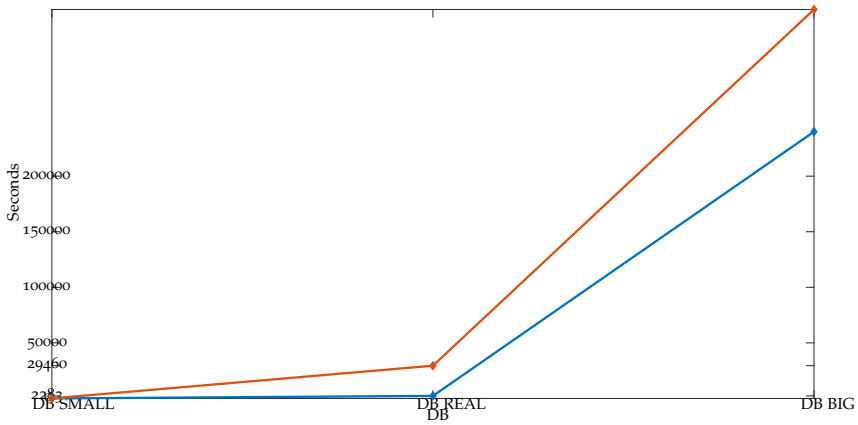

Figure 4.4: Plot of fe_get_measurements_archive_hourly's results (blue line with few stations, red line with all stations)

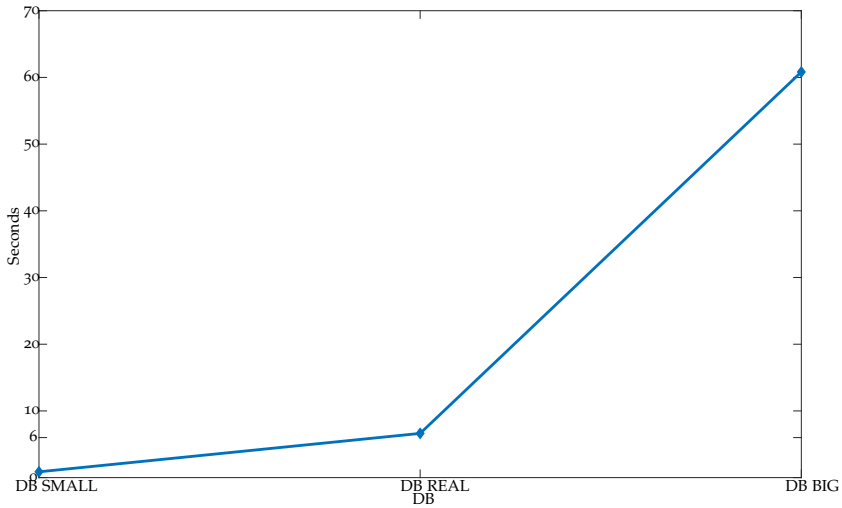

Figure 4.5: Plot of fe_get_measurements_archive_realtime's results with few stations

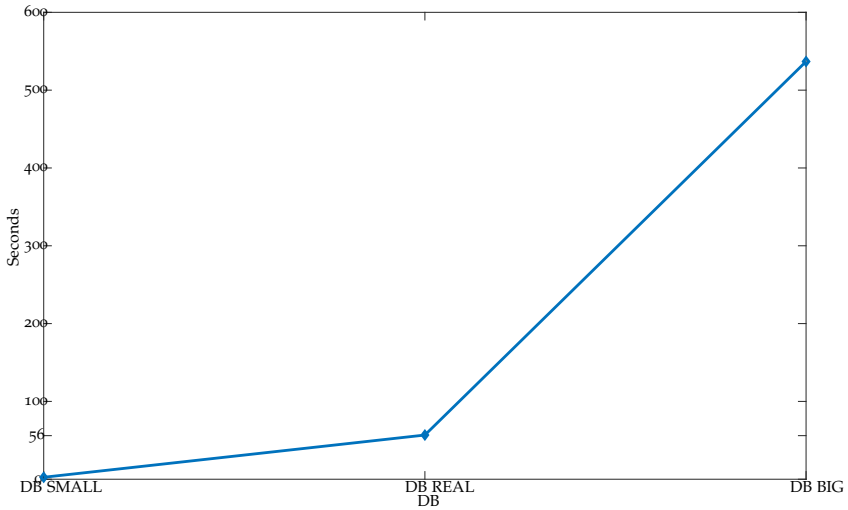

Figure 4.6: Plot of fe_get_measurements_archive_realtime's results with all stations

5 MONGODB ANALYSIS

As shown before, SQL systems could be in trouble when they are used with a huge quantity of data. In this chapter we study an alternative to SQL systems analyzing the performances using a NoSQL instead, in the specifc we used MongoDB as described in Section 3.2.

In Section 5.1 we describe in which ways we structured the DB. In Section 5.2 we describe the testing enviroment used. In Sections 5.3 and 5.4 we present the scripts used to insert and retrieve data respectively and in the Section 5.5 we illustrate the response's times obtained.

5.1 DB STRUCTURE DESCRIPTION

As alredy explained in Chapter 3 NoSQL systems have different structures and operate in a different way, but, despite this, these systems could be adapted to simulate a relational system. In fact for each table we constructed a Collection with the same fields and in particular with one reserved as FK field used for referencing. But considering that this strategy is always a "simulation" of a relational mechanism, because this system was not born to be used in this way, we considered also the only other way to link 2 different document e.g. embedding one inside the other, as alredy illustrated in Section 3.2. For this reason new collections that adopt embedding's modality are created and renamed "measurement_daily_new" and "measurement_realtime_new" respectively, as shown in Figure 5.1.

To embedding information we don't need all the other Collections as region, district, city and station_model anymore. Therefore, considering that every information inside these Collections was related somehow to the stations, we decided to include in each measurement an embedded document "station" where are included all the other data for which were necessary the JOINs used in the SQL system.

5.2 TESTING OVERVIEW

As we did in SQL system we prepared 3 databases with the same collections as described in Section 5.1, then we create one Collection for each table present in SQL system through the command *db.createCollection("collection_name") [1]* in each DB.

We kept the same number of cities, districts, regions and station_models for each DB, we considered also the same number of stations and measurements for each database like in PostgreSQL system as

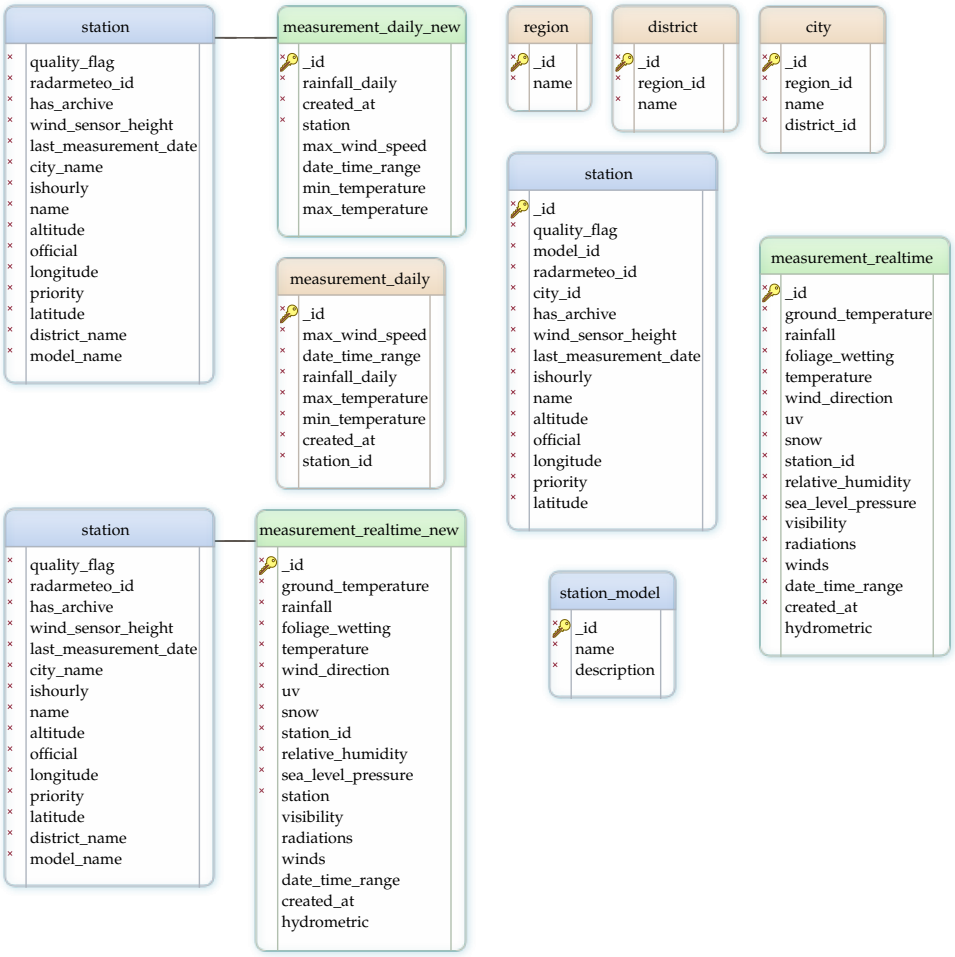

Figure 5.1: Database's structure

shown in Table 4.1 . We used also the same indexes, creating them through the command *db.collection_name.createIndex({field:1})* [1] and *db.collection_name.createIndex({field_1:1, field_2:1})* [1] for compound indexes.

Because our aim is to improve the performance of the system we focused on the worst cases, so all the queries used on NoSQL system consider always all the stations.

A package needed to be installed is *pymongo* so that Python's scripts could establish connections to MongoDB's databases [5]. In this case we don't implement a version that uses multithreading because the inseriments were faster enough.

5.3 INSERTION SCRIPT

At the beginning of the script there are 2 lines common for all the functions: they create the connection to the specified DB [5]:

```
# connection is established
client = MongoClient()
db = client.DB_NAME
```

Here we also setted global values for number of stations, and number of daily's and realtime's measurements.

As in Section 4.3 we omitted the inseriments for the collections of city, district, region, station_model. The first function presented inserts data into station's collection:

```
# function that insert stations into the DB
def insert_stations():
        for i in range(1, n_stations+1):
                radar_id="IT"
                quality=False
                off=False
                ishourly=False
                if (i<10):
                        radar_id+="0000"
                else:
                        if (i<100):
                                radar_id+="000"
                        else:
                                if (i<1000):
                                        radar_id+="00"
                                else:
                                        if (i<10000):
                                                radar_id+="0"
                radar_id+=str(i)
                if (random.randint(0,1)):
                        quality=True
                if (random.randint(0,1)):
                        off=True
```

```
24                  if (random.randint(0,1)):
25                          ishourly=True
26                  db.station.insert_one({"_id":i, "radarmeteo_id
                        ":str(radar_id), "name":"station_"+str(i),
                         "city_id": random.randint(1,n_cities+1),
                        "latitude": random.uniform(37, 47), "
                        longitude": random.uniform(7, 18), "
                        altitude": random.randint(0, 3000), "
                        wind_sensor_height": random.randint(0, 15)
                        , "quality_flag": quality, "official": off
                        , "model_id": random.randint(1, 11), "
                        priority":random.randint(0, 10), "ishourly
                        ": ishourly, "last_measurement_date": None
                        , "has_archive": True })
```

The function for each iteration:

- sets the variables to default values (lines 4 to 18).
- sets the variables, randomly or not (lines 19 to 25).
- executes the query (lines 26).

The second function inserts documents inside collection "measurement_daily":

```
1 # function that insert data into collection "measurement_daily
    "
2 def insert_measurement_daily():
3         for i in range(1, n_measurements_daily+1):
4                 delay=i/n_stations*100
5                 time=datetime.utcnow() - timedelta(days=delay)
6                 time = time.isoformat()
7                 db.measurement_daily.insert_one({"_id": i, "
                      station_id": random.randint(1,n_stations
                      +1), "min_temperature": random.randint
                      (-50,10), "max_temperature": random.
                      randint(20,60), "rainfall_daily": random.
                      randint(0,200), "max_wind_speed": random.
                      randint(0,300), "date_time_range": time, "
                      created_at": datetime.utcnow().isoformat()
                      })
```

Where for each iteation:

- variables are setted, randomly or not (lines 4 to 6).
- data is inserted in the collection, setting at the same time remaining variables on the fly (line 7).

The third function inserts documents inside collection "measurement_realtime":

```
1  # function that insert data into collection "measurement_realtime"
2  def insert_measurement_real():
3          for i in range(1, n_measurements_real+1):
4                  foliage_wet=False
5                  if (random.randint(0,1)):
6                          foliage_wet=True
7                  delay=i/n_stations*100
8                  time=datetime.utcnow() - timedelta(hours=delay)
9                  time = time.isoformat()
10                 db.measurement_realtime.insert_one({"_id": i, "station_id": random.randint(1,n_stations+1), "temperature": random.randint(-50,60), "relative_humidity": random.randint(0,100), "sea_level_pressure": random.randint(960,1060), "rainfall": random.randint(0,200), "winds": random.randint(0,300), "wind_direction": random.randint(0,360), "radiations": random.randint(0,2000), "hydrometric": random.randint(0,2000), "ground_temperature": random.randint(-50,60), "snow": random.randint(0,100), "visibility": random.randint(0,20000), "foliage_wetting": foliage_wet, "uv": random.randint(0,20), "date_time_range": time, "created_at": datetime.utcnow().isoformat()})
```

Where for each iteation:

- boolean variable is setted to default value (line 4).
- variables are setted, randomly or not (lines 5 to 9).
- data is inserted in the collection, setting at the same time remaining variables on the fly (line 10).

As explained in the end of Section 5.1 other 2 collections were created and to respect the emedding modality new functions are needed. The first one fills up collection "measurement_daily":

```
# new function that insert data into collection "measurement_daily_new"
def insert_measurement_daily_new():
        for i in range(1, n_measurements_daily+1):
                radar_id="IT"
                quality=False
                off=False
                ishourly=False
                r=random.randint(1, n_stations)
                if (r<10):
                        radar_id+="0000"
                else:
                        if (r<100):
```

```
13                                  radar_id+="000"
14                          else:
15                                  if (r<1000):
16                                          radar_id+="00"
17                                  else:
18                                          if (r<10000):
19                                                  radar_id+="0"
20                  radar_id+=str(r)
21                  if (random.randint(0,1)):
22                          quality=True
23                  if (random.randint(0,1)):
24                          off=True
25                  if (random.randint(0,1)):
26                          ishourly=True
27                  city="city_"+str(random.randint(1,n_cities+1))
28                  district="district_"+str(random.randint(1,
                        n_districts+1))
29                  model="model_"+str(random.randint(1, 11))
30                  delay=i/n_stations*100
31                  time=datetime.utcnow() - timedelta(days=delay)
32                  time = time.isoformat()
33                  db.measurement_daily_new.insert_one({"
                        min_temperature": random.randint(-50,10),
                        "max_temperature": random.randint(20,60),
                        "rainfall_daily": random.randint(0,200), "
                        max_wind_speed": random.randint(0,300), "
                        date_time_range": time, "created_at":
                        datetime.utcnow().isoformat(), "station":
                        {"radarmeteo_id":str(radar_id), "name":"
                        station_"+str(r), "city_name": city, "
                        district_name": district, "latitude":
                        random.uniform(37, 47), "longitude":
                        random.uniform(7, 18), "altitude": random.
                        randint(0, 3000), "wind_sensor_height":
                        random.randint(0, 15), "quality_flag":
                        quality, "official": off, "model_name":
                        model , "priority":random.randint(0, 10),
                        "ishourly": ishourly, "
                        last_measurement_date": None, "has_archive
                        ": True }})
```

Where for each iteation:

- variables are setted to default value (line 4 to 7).
- variables are setted, randomly or not (lines 8 to 32).
- data is inserted in the collection, setting at the same time remaining variables on the fly (line 33).

The second one fills up collection "measurement_realtime":

```
1 # new function that insert data into collection "
      measurement_realtime_new"
2 def insert_measurement_real_new():
3         for i in range(1, n_measurements_real+1):
```

```
4                   radar_id="IT"
5                   quality=False
6                   off=False
7                   ishourly=False
8                   r=random.randint(1, n_stations)
9                   if (r<10):
10                          radar_id+="0000"
11                  else:
12                          if (r<100):
13                                  radar_id+="000"
14                          else:
15                                  if (r<1000):
16                                          radar_id+="00"
17                                  else:
18                                          if (r<10000):
19                                                  radar_id+="0"
20                  radar_id+=str(r)
21                  if (random.randint(0,1)):
22                          quality=True
23                  if (random.randint(0,1)):
24                          off=True
25                  if (random.randint(0,1)):
26                          ishourly=True
27                  city="city_"+str(random.randint(1,n_cities+1))
28                  district="district_"+str(random.randint(1,
                        n_districts+1))
29                  model="model_"+str(random.randint(1, 11))
30                  foliage_wet=False
31                  if (random.randint(0,1)):
32                          foliage_wet=True
33                  delay=i/n_stations*100
34                  time=datetime.utcnow() - timedelta(hours=delay
                        )
35                  time = time.isoformat()
36                  db.measurement_realtime_new.insert_one({"_id":
                         i, "station_id": random.randint(1,
                        n_stations+1), "temperature": random.
                        randint(-50,60), "relative_humidity":
                        random.randint(0,100), "sea_level_pressure
                        ": random.randint(960,1060), "rainfall":
                        random.randint(0,200), "winds": random.
                        randint(0,300), "wind_direction": random.
                        randint(0,360), "radiations": random.
                        randint(0,2000), "hydrometric": random.
                        randint(0,2000), "ground_temperature":
                        random.randint(-50,60), "snow": random.
                        randint(0,100), "visibility": random.
                        randint(0,20000), "foliage_wetting":
                        foliage_wet, "uv": random.randint(0,20), "
                        date_time_range": time, "created_at":
                        datetime.utcnow().isoformat(), "station":
                        {"radarmeteo_id":str(radar_id), "name":"
                        station_"+str(r), "city_name": city, "
                        district_name": district, "latitude":
                        random.uniform(37, 47), "longitude":
                        random.uniform(7, 18), "altitude": random.
                        randint(0, 3000), "wind_sensor_height":
```

```
random.randint(0, 15), "quality_flag":
quality, "official": off, "model_name":
model , "priority":random.randint(0, 10),
"ishourly": ishourly, "
last_measurement_date": None, "has_archive
": True }})
```

Where for each iteation:

- variables are setted to default value (line 4 to 7).
- variables are setted, randomly or not (lines 8 to 35).
- data is inserted in the collection, setting at the same time remaining variables on the fly (line 36).

We have to report that inseriments into MongoDB's system were more faster than the inseriments into PostgreSQL system, in fact all the data in the 3 databases (both the collections used with reference's modality as well as the embedded ones) were inserted in about 1-2 weeks.

5.4 QUERY SCRIPT

In this Section we are going to illustrate the 4 functions used to query the collections (2 using references, 2 using embedded documents), obviously the amount of stations in the databases respects Table 4.2, and as done in Section 4.4we iterated each query 200 times. We have to specify that for these queries we assumed an optimization (considered the high cost of each aggregation), e.g. aggregation between the collections "station", "city" and "district" had been done only one time before the iteration's cycle because only if a new record is inserted in one of these collections this operation should be done again. For this reason this aggregation is not considered when calculating the response's times.

The first function query the collection "measurement_daily" and has 4 inputs (min_temperature, max_temperature, the number of iterations of the query and in the end the connection to the DB):

```
1 # function that query the table "measurement_daily" with
    references
2 def query_daily(min, max, n_cycles, dd):
3         dd.station.aggregate([{"$lookup":{"from":"city","
              localField":"city_id", "foreignField":"_id", "as":
               "cities"}},{"$unwind":"$cities"},{"$project":{"
              _id":1, "radarmeteo_id":1, "district_id":"$cities.
              district_id", "city_name":"$cities.name"}},{"$out"
              :"station_city"}])
4         dd.station_city.aggregate([{"$lookup":{"from":"
              district","localField":"district_id", "
              foreignField":"_id", "as": "districts"}},{"$unwind
              ":"$districts"},{"$project":{"_id":1, "
```

```
            radarmeteo_id":1, "city_name":1, "district_name":"
            $districts.name"}},{"$out":"station_city_district"
            }])
5       start_time = datetime.utcnow()
6       for i in range(1,n_cycles+1):
7               dd.measurement_daily.aggregate([{"$match":{"
                    min_temperature":{"$gt":random.randint
                    (-30, 14), "$lt":random.randint(18, 50)
                    }}},{"$lookup":{"from":"
                    station_city_district","localField":"
                    station_id", "foreignField":"_id", "as": "
                    scd"}},{"$unwind":"$scd"},{"$project":{"
                    _id":0,"min_temperature":1,"
                    max_temperature":1, "rainfall_daily":1, "
                    max_wind_speed":1, "radarmeteo_id":"$scd.
                    radarmeteo_id", "city_name":"$scd.
                    city_name", "district_name":"$scd.
                    district_name"}},{"$out":"result"}])
8       time=(datetime.utcnow() - start_time).total_seconds()
9       return(str(time))
```

Where:

- it aggregates the collections “station” and “city” creating a temporary collection named “station_city” (line 3).
- it aggregates the collections “station_city” (just created) and “district” creating a temporary collection named “station_city_district” (line 4).
- “start_time” variable saves the actual time (line 5).
- inside the cycle the aggregation between the collections “measurement_daily” and “station_city_district” is iterated (lines 6 to 7).
- “time” variable saves the duration of the cycle and then it is returned and then written to file (line 8 to 9)

The second function query the collection “measurement_realtime”, and has 2 inputs (the number of iterations of the query and in the end the connection to the DB where operate):

```
1 # function that query the table "measurement_realtime" with
    references
2 def query_realtime(n_cycles, dd):
3         dd.station.aggregate([{"$lookup":{"from":"city","
            localField":"city_id", "foreignField":"_id", "as":
             "cities"}},{"$unwind":"$cities"},{"$project":{"
            _id":1, "radarmeteo_id":1, "district_id":"$cities.
            district_id", "city_name":"$cities.name"}},{"$out"
            :"station_city"}])
```

```
4         dd.station_city.aggregate([{"$lookup":{"from":"
              district","localField":"district_id", "
              foreignField":"_id", "as": "districts"}},{"$unwind
              ":"$districts"},{"$project":{"_id":1, "
              radarmeteo_id":1, "city_name":1, "district_name":"
              $districts.name"}},{"$out":"station_city_district"
              }])
5         start_time = datetime.utcnow()
6         for i in range(1,n_cycles+1):
7                 dd.measurement_realtime.aggregate([{"$lookup"
                      :{"from":"station_city_district","
                      localField":"station_id", "foreignField":"
                      _id", "as": "scd"}},{"$unwind":"$scd"},{"
                      $project":{"_id":0, "temperature":1, "
                      relative_humidity":1, "sea_level_pressure"
                      :1, "rainfall":1, "winds":1, "
                      wind_direction":1, "radiations":1, "
                      hydrometric":1, "ground_temperature":1, "
                      snow":1, "visibility":1, "foliage_wetting"
                      :1, "uv":1, "radarmeteo_id":"$scd.
                      radarmeteo_id", "city_name":"$scd.
                      city_name", "district_name":"$scd.
                      district_name"}},{"$out":"result"}])
8         time=(datetime.utcnow() - start_time).total_seconds()
9         return(str(time))
```

This function is almost identical to the previous one, it differs for the last collection used for the aggregation, the absence of *$match* operator (because there is not filtering in the respective PostgreSQL's *stored procedure*) and on the name of the fields projected. Next function is the alternative version to the first one:

```
1 # function that query the table "measurement_daily_new" with
      embedded docs
2 def query_daily_new(min, max, n_cycles, dd):
3         start_time = datetime.utcnow()
4         for i in range(1,n_cycles+1):
5                 dd.measurement_daily_new.find({"
                      min_temperature":{"$gt":min, "$lt":max}},
                      {"_id":0, "min_temperature":1, "
                      max_temperature":1, "rainfall_daily":1, "
                      max_wind_speed":1, "station.radarmeteo_id"
                      :1, "station.city_name":1, "station.
                      district_name":1})
6         time=(datetime.utcnow() - start_time).total_seconds()
7         return(str(time))
```

In this function:

- "start_time" variable saves the actual time (line 3).
- inside the cycle the aggregation between the collections "measurement_daily" and "station_city_district" is iterated (lines 4 to 5).

- "time" variable saves the duration of the cycle and then it is returned and then written to file (lines 6 to 7).

Last function is the alternative version to the second one:

```
1 # function that query the table "measurement_realtime_new"
    with embedded docs
2 def query_realtime_new(n_cycles, dd):
3         start_time = datetime.utcnow()
4         for i in range(1,n_cycles+1):
5                 dd.measurement_realtime_new.find({"_id":0, "
                      temperature":1, "relative_humidity":1, "
                      sea_level_pressure":1, "rainfall":1, "
                      winds":1, "wind_direction":1, "radiations"
                      :1, "hydrometric":1, "ground_temperature"
                      :1, "snow":1, "visibility":1, "
                      foliage_wetting":1, "uv":1, "station.
                      radarmeteo_id":1, "station.city_name":1, "
                      station.district_name":1})
6         time=(datetime.utcnow() - start_time).total_seconds()
7         return(str(time))
```

5.5 RESULTS

The Tables 5.1 and 5.2 show the response's times of measurement_daily and measurement_realtime respectively using references and embedded documents on the 3 databases, analysing them we could notice the big increasing of response's times of queries that uses references while queries that used embedding documents had response's times extremely low in every DB. The Figures 5.2 and 5.3 represent the plot of the means of these response's times where is more evident the increasing using references and constant time using embedded documents.

	DB_SMALL		DB_REAL		DB_BIG	
	REF.	EMB.	REF.	EMB.	REF.	EMB.
QUERY 1	35,92	0,01	7768,66	0,01	154464,55	0,01
QUERY 2	35,29	0,01	12860,02	0,01	142572,63	0,01
QUERY 3	19,34	0,01	8069,04	0,01	93039,71	0,01
QUERY 4	45,28	0,01	8237,16	0,01	69096,29	0,01
QUERY 5	30,96	0,01	5282,25	0,01	109433,64	0,01
QUERY 6	37,64	0,01	11374,66	0,01	79055,08	0,01
QUERY 7	27,76	0,01	7551,88	0,01	99232,40	0,01
QUERY 8	26,10	0,01	9047,64	0,01	113985,80	0,01
QUERY 9	42,38	0,01	6820,46	0,01	119400,87	0,01
QUERY 10	42,22	0,01	9394,77	0,01	110518,33	0,01
QUERY 11	29,63	0,01	13599,16	0,01	114438,72	0,01
QUERY 12	20,55	0,01	8744,83	0,01	66653,73	0,01
QUERY 13	41,97	0,01	10498,80	0,01	71095,51	0,01
QUERY 14	31,09	0,01	11647,68	0,01	152098,21	0,01
QUERY 15	41,07	0,01	11383,38	0,01	98325,01	0,01
QUERY 16	29,45	0,01	8366,45	0,01	90483,23	0,01
QUERY 17	30,67	0,01	7937,22	0,01	95384,45	0,01
QUERY 18	30,80	0,01	10420,59	0,01	102448,72	0,01
QUERY 19	37,23	0,01	10844,03	0,01	106946,09	0,01
QUERY 20	38,85	0,01	9381,15	0,01	100683,47	0,01
MEAN	33,71	0,01	9461,49	0,01	104467,82	0,01

Table 5.1: Results of queries on collections "measurement_daily" and "measurement_daily_new" (seconds).

	DB_SMALL		DB_REAL		DB_BIG	
	REF.	EMB.	REF.	EMB.	REF.	EMB.
QUERY 1	12,25	0,01	5822,80	0,01	166334,00	0,02
QUERY 2	32,24	0,01	5558,12	0,01	156123,46	0,02
QUERY 3	29,41	0,01	5620,91	0,01	93366,61	0,01
QUERY 4	29,81	0,01	4846,48	0,01	75501,33	0,01
QUERY 5	32,67	0,01	4831,61	0,01	71065,21	0,01
QUERY 6	26,80	0,01	4595,17	0,01	68164,50	0,01
QUERY 7	30,60	0,01	4638,10	0,01	69718,30	0,01
QUERY 8	16,54	0,01	4797,75	0,01	80768,28	0,01
QUERY 9	15,22	0,01	4535,93	0,01	79738,49	0,01
QUERY 10	15,48	0,01	5522,32	0,01	71361,02	0,01
QUERY 11	14,54	0,01	5176,10	0,01	69682,66	0,01
QUERY 12	15,78	0,01	5193,51	0,01	68960,96	0,01
QUERY 13	13,49	0,01	5142,85	0,01	68114,03	0,01
QUERY 14	15,54	0,01	5247,11	0,01	67510,68	0,01
QUERY 15	13,50	0,01	5016,84	0,01	68052,12	0,01
QUERY 16	15,77	0,01	4854,08	0,01	75288,54	0,01
QUERY 17	13,45	0,01	4725,61	0,01	77512,87	0,01
QUERY 18	14,87	0,01	5101,77	0,01	85437,09	0,01
QUERY 19	12,34	0,01	5254,30	0,01	80112,35	0,01
QUERY 20	12,89	0,01	4655,96	0,01	79032,21	0,01
MEAN	19,16	0,01	5056,87	0,01	83592,24	0,01

Table 5.2: Results of queries on collections "measurement_realtime" and "measurement_realtime_new" (seconds).

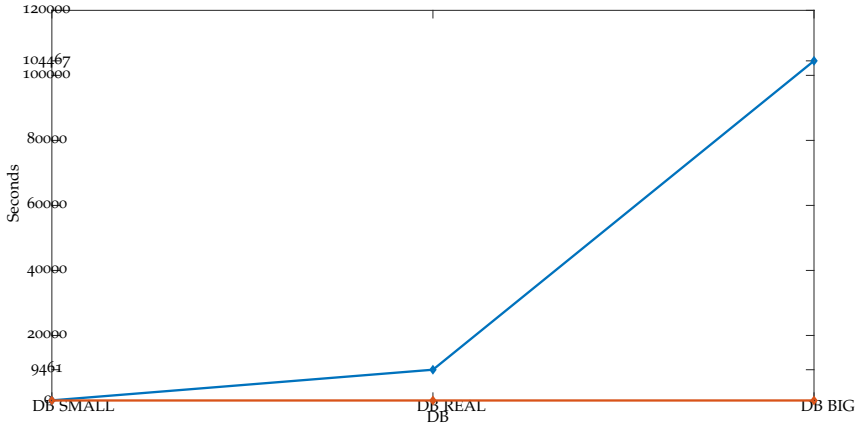

Figure 5.2: Plot of the results on measurement_daily (reference as blue line and embedded as red one).

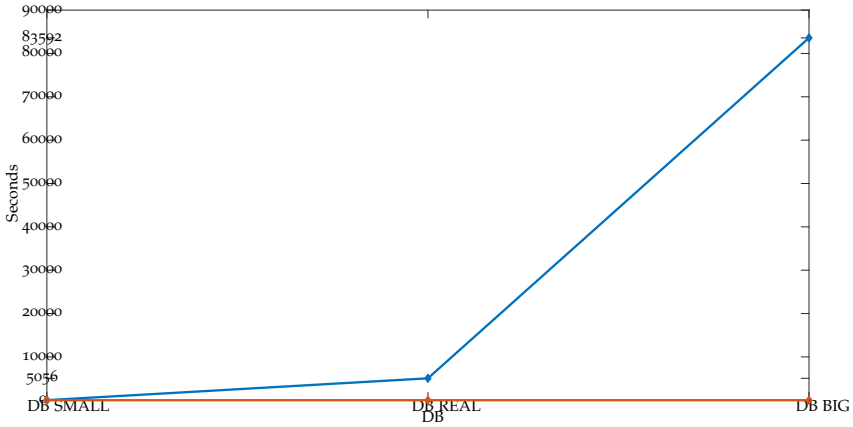

Figure 5.3: Plot of the results on measurement_realtime (reference as blue line and embedded as red one).

6 CONCLUSIONS

In Chapter 3 we described the main characteristics of RDBMS and NoSQL systems, now in Section 6.2 we provide a comparison between them and in Section 6.1 we compare the results of the 2 testing enviroments.

6.1 COMPARISON BETWEEN RESULTS

Considering Tables 4.3, 5.1 and the plots of their mean values in Figures 4.3, 5.2 we could say that *$aggregation* method of NoSQL system has response's times higher than the JOIN operator of SQL system. But if we consider also 4.5, 5.2 and their plots 4.4, 5.3 we could notice that with a huge quantity of data MongoDB's queries using references have better response' times than PostgreSQL's ones, although they are still too high (about 418 seconds for a single query).

This indicates that MongoDB, although it could work like a relational system, it has not been made for it, and if there are not tables with a huge amount of records SQL is more advisable. We also noticed that in MongoDB's system queries on collection "measurement_daily" took more time than queries on collection "measurement_realtime", this confirms that MongoDB works better with a greater amount of records.

In the end we could say that:

- using *$aggregation* method inside MongoDB's system should be avoided if not strictly necessary, because this operation is usually more expensive than a JOIN of a SQL system, despite optimizations on both the systems.
- if the use of a huge amount of data could be expected, it is highly reccomended using a NoSQL system adopting methods that avoid referencing.

6.2 COMPARISON BETWEEN SYSTEMS AND DIFFERENT STRUCTURES

In SQL systems usually data linked together are stored in different tables and there is the need to get them together, to do this JOIN operator is used. But this operator is highly expensive [8] and, although through several optimization this problem could be limited a lot, it cannot be eliminated when a huge amount of data is used. These optimizations include the use of indexes (singles and compound) and filtering data inside the tables.

	DB_SMALL			DB_REAL			DB_BIG		
	PSQL	MDB (ref.)	MDB (emb.)	PSQL	MDB (ref.)	MDB (emb.)	PSQL	MDB (ref.)	MDB (emb.)
station	0,61	0,24	-	6	4	-	58	48	-
daily	1,77	1,24	3,05	814	619	1533	8817	6775	16820
realtime	15	9,05	17,30	9567	5819	11098	109955	64659	123580
total	17,38	10,53	20,35	10387	6443	12631	118830	71482	140400

Table 6.1: Tables' sizes (Megabytes)

MongoDB's system is very different from SQL ones, although it let users to use references it is not made for that in fact at the contrary there is a performance's worsening. We also have to say that using embedded documents there is a lot of redundancy because the elements that before were alone in another collection in this case are replicated many times. This has the side effect that the dimension of the collection is bigger, as shown in Table 6.1 and more evident from Figure 6.1, Figure 6.2 and Figure 6.3.

Considering MongoDB's system using aggregation and then embedding documents, the space's increasing is between 90-150% (90% for table "measurement_daily", 147% for table "measurement_ realtime"), but if we consider the space needed into PostgreSQL and then the space required for the Collections that use embedding documents, the space's increasing would be around 12-90% (80% for table "measurement_daily", 12-16% for table "measurement_realtime"). At the contrary table station had a space's decreasing about 17-60% (obviously this table is not present in the system with embedding data).

If we analyze the total space of each system, not including MongoDB with references, we could evidence that the migration from PostgreSQL to MongoDB would require around 18-21% more space.

6.3 FUTURE WORK

This experience let me to study and analyse a system very different from a SQL one beacuse during my experience at my university I did not have the possibility to learn much about NoSQL systems and only sometimes they were mentioned without explaining in the detail how they work. This experience gave me also the possibility to work in company for several months letting me to understand how it is working for it.

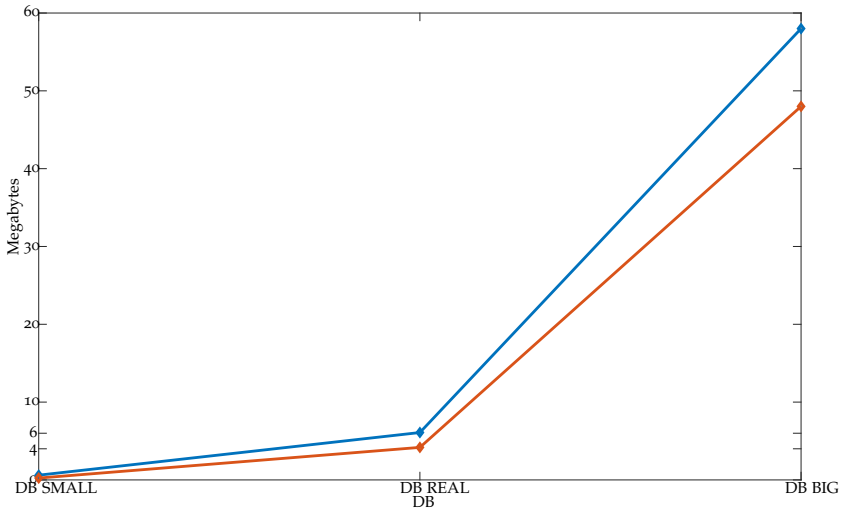

Figure 6.1: Plot of dimension of table/collection "station" (PostgreSQL as blue line and MongoDB as red one).

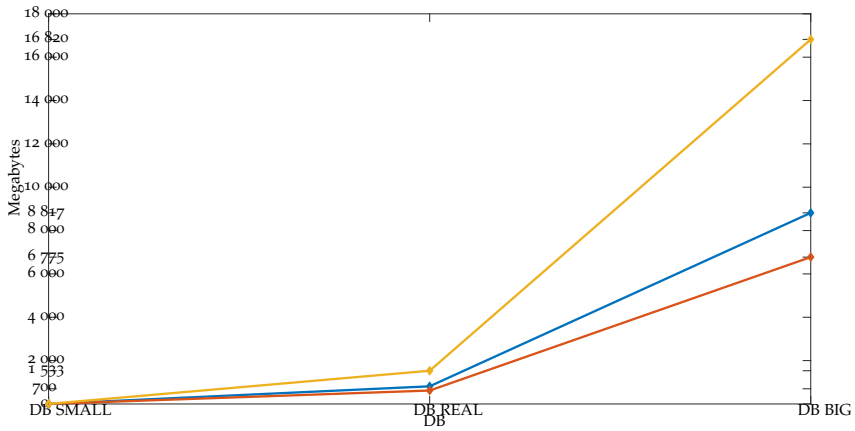

Figure 6.2: Plot of dimension of table/collection "measurement_daily" (PostgreSQL as blue line, MongoDB with references as red one and MongoDB with embedding data as yellow one).

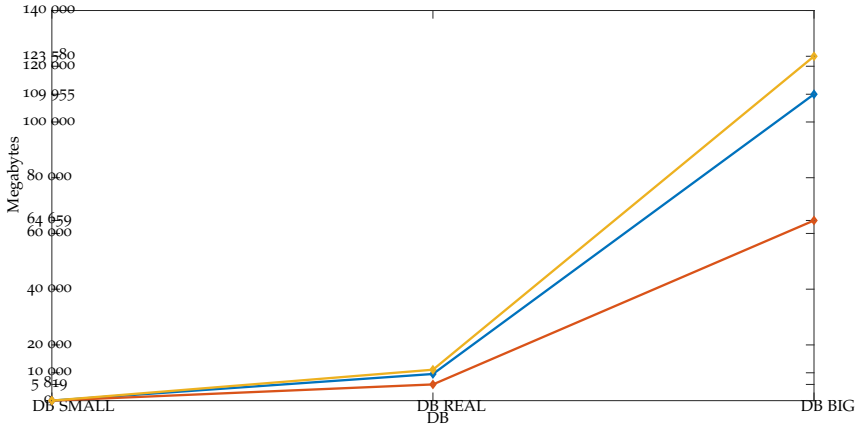

Figure 6.3: Plot of dimension of table/collection "measurement_realtime" (PostgreSQL as blue line, MongoDB with references as red one and MongoDB with embedding data as yellow one).

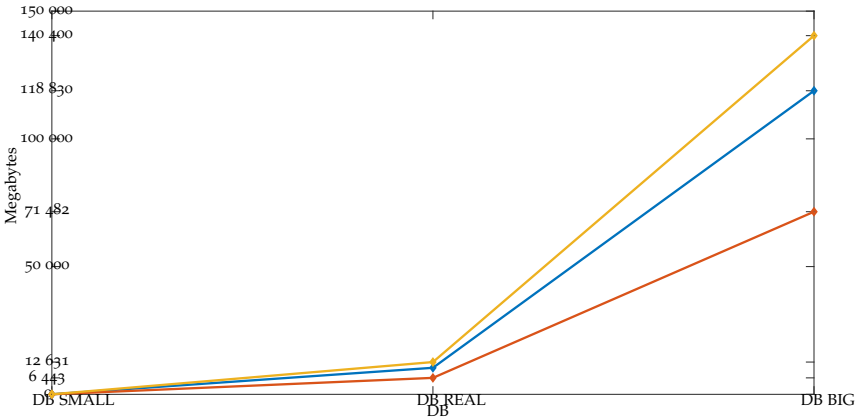

Figure 6.4: Plot of dimension of Databases (PostgreSQL as blue line, MongoDB with references as red one and MongoDB with embedding data as yellow one).

With our work we focused on improving the performance of a system with a huge amount of data. As first contribution we presented a possible future giving also a way to improve also the actual system. This allows to improve the table "measurement_daily" that also in DB_REAL could be problematic. As second contribution we presented a valid alternative to the actual system, considering also the effects caused by the redundancy and the relative increasing space of storage required.

Further work could consider different NoSQL platforms like Hadoop, Cassandra, CouchDB, Redis and others and analyse the perfomance and the redundancy letting a more accurate choice about the best one for the DB actually in use.

BIBLIOGRAPHY

[1] Mongodb documentation. URL https://docs.mongodb.com/manual/.

[2] Postgresql documentation. URL https://www.postgresql.org/docs/9.6/static/index.html.

[3] Jagdev Bhogal and Imran Choksi. Handling big data using nosql. *2015 29th International Conference on Advanced Information Networking and Applications Workshops*, pages 393–398, 2015. doi: 10.1109/CCCM.2008.236. URL http://ieeexplore.ieee.org/document/7096207/.

[4] Lars George. *HBase The Definitive Guide*. O'Reilly Media, 2011. ISBN 978-1-449-39610-7. URL http://shop.oreilly.com/product/0636920014348.do.

[5] David Hows, Peter Membrey, Eelco Plugge, and Tim Hawkins. *The definitive Guide to MongoDB*. Apress, 2015. ISBN 978-1-4842-1183-054999. URL http://www.apress.com/it/book/9781484211830.

[6] Krish Krishnan. *Data Warehousing in the Age of Big Data*. Elsevier Academic Press, 2013. ISBN 978-0-12-405891-0. URL https://www.elsevier.com/books/data-warehousing-in-the-age-of-big-data/krishnan/978-0-12-405891-0.

[7] David Loshin. *Big Data Analytics From Strategic Planning to Enterprise Integration with Tools, Techniques, NoSQL, and Graph*. Elsevier Academic Press, 2013. ISBN 978-0-12-417319-4. URL https://www.elsevier.com/books/big-data-analytics/loshin/978-0-12-417319-4.

[8] Renzo Orsini, Antonio Albano, and Giorgio Ghelli. *Fondamenti di Basi di Dati*. Zanichelli, 2005. ISBN 88-08-17003-9. URL https://www.zanichelli.it/ricerca/prodotti/fondamenti-di-basi-di-dati?hl=fondamenti%20basi%20dati.

[9] Sharvari Rautmare and Dr. D. M. Bhalerao. Mysql and nosql database comparison for iot application. *2016 IEEE International Conference on Advances in Computer Applications (ICACA)*, pages 235–238, 2016. doi: 10.1109/ICACA.2016.7887957. URL http://ieeexplore.ieee.org/document/7887957/.

[10] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, 2013. ISBN 978-1-449-35626-2. URL http://shop.oreilly.com/product/0636920028246.do.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*. `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured at:

http://postcards.miede.de/