University of Pisa
School of Engineering

Large Scale and Multi-Structured Databases

StockSim: Stock Portfolio Simulator

Supervisor
Prof. Pietro Ducange

Students
Marco Pinna
Rambod Rahmani
Yuri Mazzuoli

January 5, 2021

Contents

I Documentation **2**

1 Introduction **4**

2 Actors and requirements **6**
- 2.1 Actors 6
- 2.2 Requirements 7
  - 2.2.1 Functional requirements 7
  - 2.2.2 Non-functional requirements 7

3 UML diagrams **8**
- 3.1 Use Case diagram 9
- 3.2 Class diagram 10

4 Database **11**
- 4.1 Dataset 11
  - 4.1.1 NasdaqTrader 11
  - 4.1.2 Yahoo! Finance 12
- 4.2 MongoDB 12
  - 4.2.1 Aggregations 14
  - 4.2.2 Indexes 15
- 4.3 Apache Cassandra 16
  - 4.3.1 Aggregations 17
  - 4.3.2 Indexes 19
- 4.4 Sharding and Replicas 19
- 4.5 Apache Cassandra vs MongoDB 20

5 Software architecture **21**
- 5.1 Maven Multi-Module Structure 21
  - 5.1.1 Library 21
  - 5.1.2 Server 21
  - 5.1.3 Client 21
- 5.2 Apache Maven Assembly Plugin 21

6 Conclusions **22**

II User Manual **23**

7 StockSim Server Manual **25**

8 StockSim Client Manual **28**

8.1 StockSim Client User Mode 28

8.2 StockSim Client Admin Mode 28

Part I

Documentation

Chapter 1

Introduction

StockSim is a Java application which, as main feature, allows users to simulate stock market portfolios. The StockSim application is composed by two main programs:

- **StockSim Server**: supposed to be running 24/7 to ensure historical data is always up-to-date;
- **StockSim Client**: can be launched in either `admin` or `user` mode.

The StockSim Server is not thought to be distributed to end users, whereas the StockSim Client can be used by both administrators and normal users. The choice was made to provide the same program to both administrators and normal users with two different running modes. Administrators can add new ticker symbols, new administrator accounts, delete both administrator and normal user accounts. Normal users have access to stocks and ETFs historical data, day by day, starting from 2010. They can create their own stock portfolios, run simulations a visualize the resulting statistics.

Before continuing with what follows, the following terms should be clarified:

- the **stock market** is any exchange that allows people to buy and sell stocks and companies to issue stocks; a stock represents the company's equity, and shares are pieces of the company;
- a collection of investments owned by an investor makes up his or her **portfolio**; you can have as few as one stock in a portfolio, but you can also own an infinite amount of stocks or other securities;
- a **stock symbol** is a one- to four-character alphabetic root symbol that represents a publicly traded company on a stock exchange; Apple's stock symbol is AAPL, while Walmart's is WMT;
- the NYSE and Nasdaq are open from Monday through Friday 9:30 A.M. to 4:00 P.M. (eastern time);

- the NYSE and Nasdaq close at 4 P.M., with after-hours trading continuing until 8 P.M.; the close simply refers to the time at which a stock exchange closes to trading;
- trading stocks after normal market hours through an electronic market, typically between 4:05 and 8:00 P.M., is **after-hours trading**;
- the **high** is the highest price at which a stock traded during a period;
- the **low** is the lowest price of the period;
- **open** means the price at which a stock started trading when the opening bell rang; it can be the same as where the stock closed the night before, but not always; sometimes events such as company earnings reports that happen in after-hours trading can alter a stock's price overnight;
- **close** refers to the price of an individual stock when the stock exchange closed shop for the day; it represents the last buy-sell order executed between two traders; in many cases, this occurs in the final seconds of the trading day;
- the **adjusted closing price** amends a stock's closing price to reflect that stock's value after accounting for any corporate actions; a stock's price is typically affected by some corporate actions, such as stock splits, dividends, and rights offerings; adjustments allow investors to obtain an accurate record of the stock's performance;
- **volume** is the total number of shares traded in a security over a period; every time buyers and sellers exchange shares, the amount gets added to the period's total volume.

Chapter 2

Actors and requirements

The main actors and the functional requirements are defined based on the previous simple description of the application. There is also a particular actor which is in charge of the automatic update of the dataset; non-functional requirements are the characteristics that ensure satisfactory interaction with users and real world utility of the product.

2.1 Actors

Based on the application design, four actors can interact with the system:

- A **Guest user** is someone who is not registered on the application; this actor does not own private credentials for the login; in order to exploit the application main functionalities, it has to register a new account. The registration it is the only action allowed for a Guest;

- A **Registered user** is someone who's registered on the application; this actor own private credentials (username and password) for the login; this actor can login as a user into the client application and utilize its main features like watch stocks information, compose portfolios and simulate them.

- An **Admin user** is someone who is registered on the application with administration credentials; this actor can login as an admin into the client and do some maintenance operations; this operations includes check the integrity of the entire dataset and add new stocks to it;

- The **Data Updater** is a thread running on one server; this thread is suppose to run always, and it is in charge of update the dataset with the new information coming from the stock market daily sessions; it is also in charge to find and fix (at list report) integrity issues.

The full use case diagram is provided on chapter 3.

2.2 Requirements

2.2.1 Functional requirements

- The user, upon launching the application, should be able to sign-up and sign-in.
- The user should be able to search for a ticker and view info about it.
- The user should be able to view charts on the history of a ticker.
- The user should be able to create and delete portfolios.
- The user should be able to add/remove tickers to/from their portfolio.
- The user should be able to run simulations on a portfolio.
- The user should be able to visualize statistics about a simulation and possibly view charts on them.
- Only admins should be able to add new stocks to the database.
- Only admins should be able to add other admins.
- The software should always provide the most recent data about a ticker.

2.2.2 Non-functional requirements

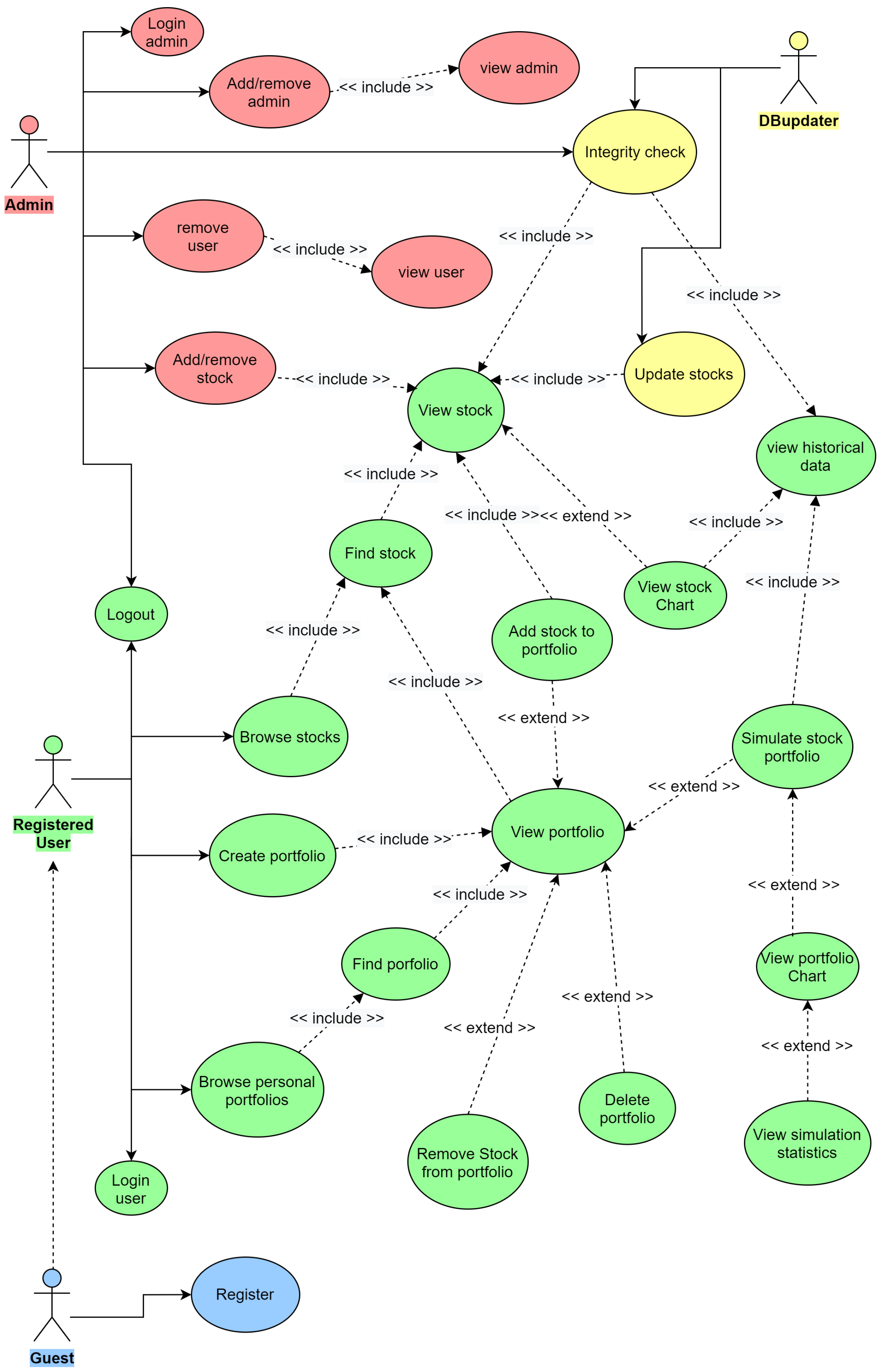
- The retrieval of data about tickers and portfolios should be fast.
- User password should be stored in a secure way (i.e. hashed).

Chapter 3

UML diagrams

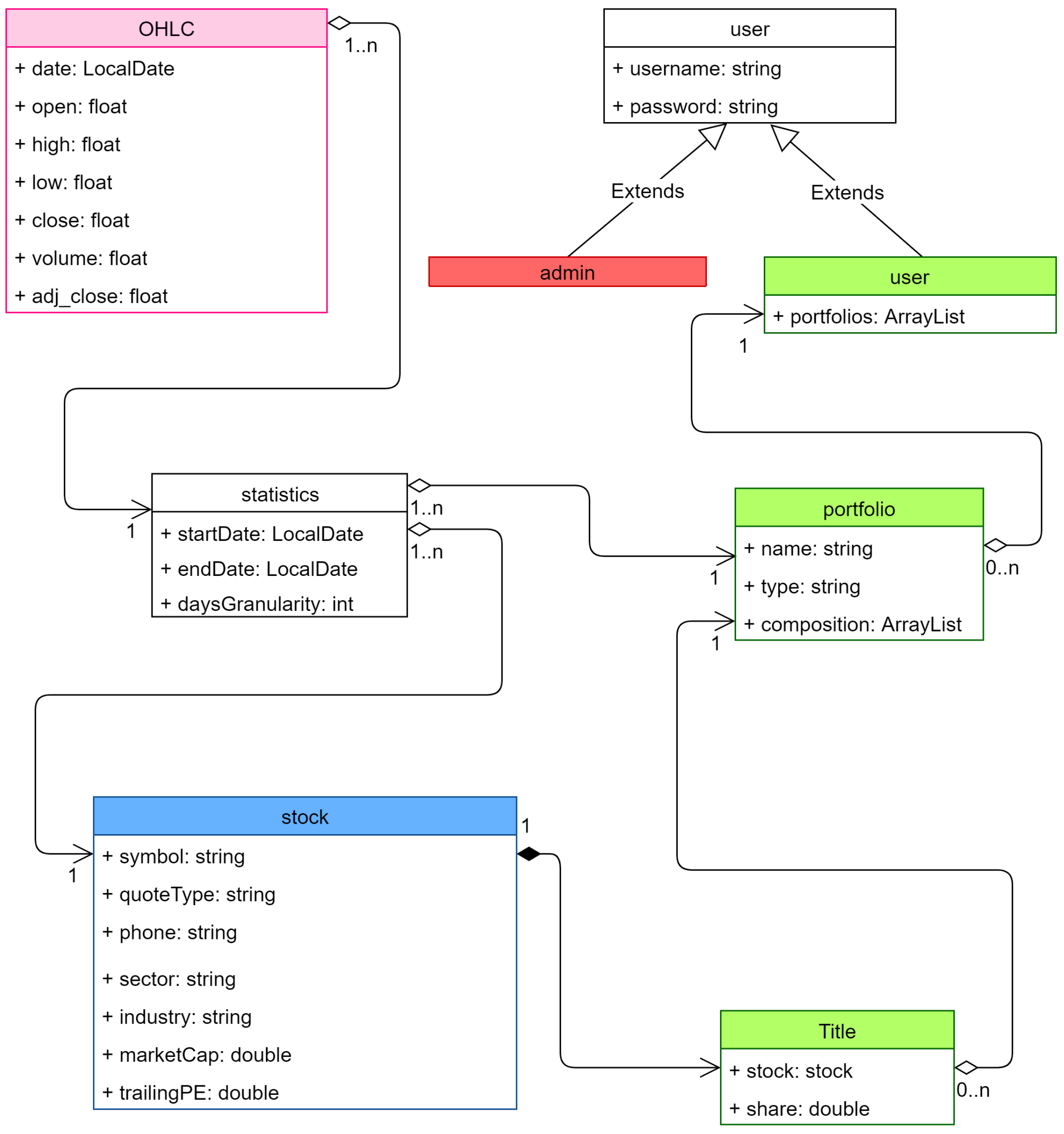
Blablabla.

3.1 Use Case diagram



3.2 Class diagram

Something about functional requirements.



Chapter 4

Database

The dataset is composd by 8236 stocks from the US stock market, along with their general informations and historical data; the application also need to store users' and admins' credential, personal informationof users, composition and details of each user's porfolio. We decided to use a column database for the storage of historical data; those informations represent around the 99% of our dataset and they are going to grow very fast during time; aggregation and financial analytics on these volumes of data will perform better in a colum database where data storage is design to optimize this type of operations; We decided to store every other information in a document database, in order to exploit the schemaless property for save memory; information frequently needed toghether will be stored in the same document and indexes are created to speedup linking beetween documents;

4.1 Dataset

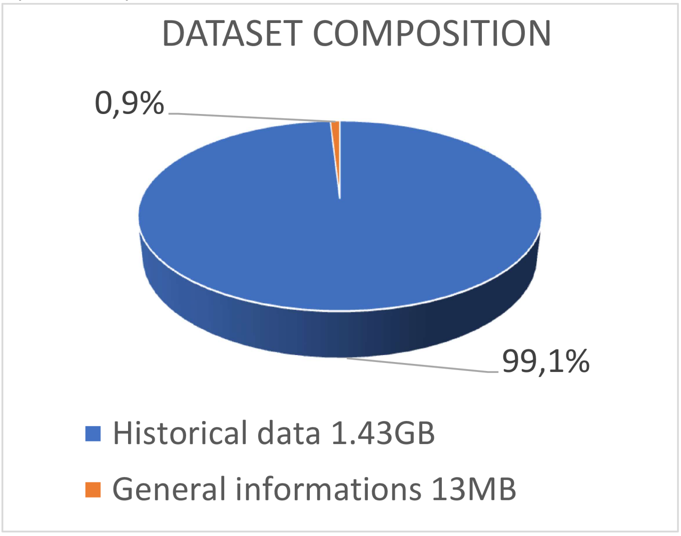
The initial set of data it's been taken from the web, thanks to www.nasdaqtrader.com and finance.yahoo.com, using python scripts with pandas, yfinmace and json as support libraries.

4.1.1 NasdaqTrader

The Nasdaq Stock Market (Nasdaq) is the largest U.S. equities exchange venue by volume. https://www.nasdaqtrader.com/
We choose to take our set of stocks from the Nasdaq index, because it's very popular and include a large number of stock, representative of differenteconomy sector. This will allow users to interact with big and famous comopanies stocks (like Google, Apple, Tesla...), but also to try smaller companies and/or minor sectors investemts. NasdaqTrader provides us a stocks'symbols' list of all the stocks entered in the nasdaq index from 1970 till now;

4.1.2 Yahoo! Finance

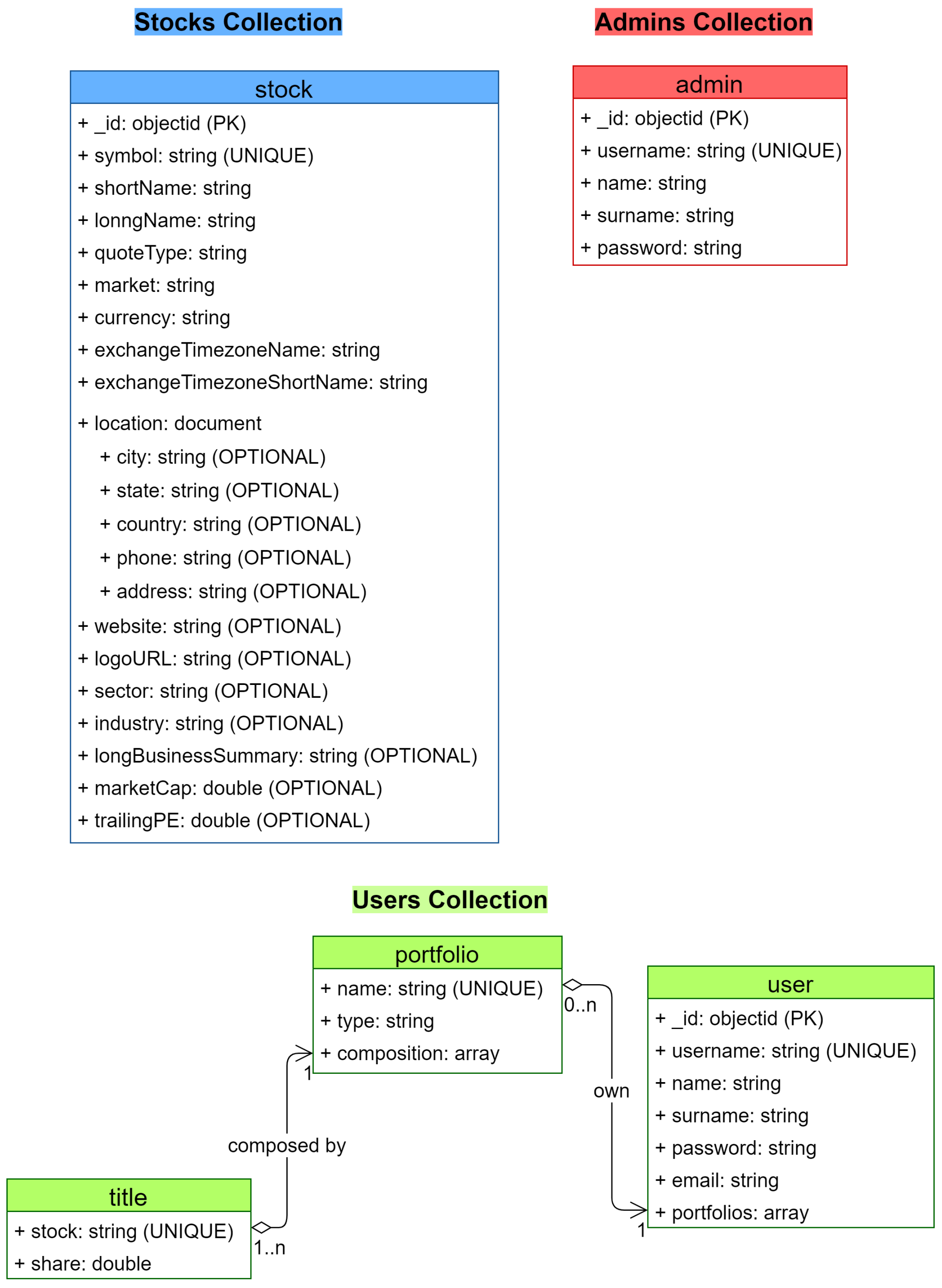
”Yahoo Finance provides free stock quotes, up-to-date news, portfolio management resources, international market data, social interaction and mortgage rates that help you manage your financial life.” https://finance.yahoo.com/
Yahoo Finance is a service, been part of the yahoo network, that provide a lot of information about stocks and companies; they are frequantly updated, relible and good organized.
We decided to use this service to retrive the starting dataset of stocks; we extract only the fields that we needed, and parse into a JSON file. In this way is possible to rebuild from scratch this dataset into mongoDB with few commands (including mongoimport).
With Yahoo Finance is also possible to retrive historical data of market values for every stocks. Using this service it's been possible to build a dataset of all the market values of each stocks coming from NasdaqTrader; values are collected daily, and we decided to take all the values from 2010 to 2020; this dataset (around 1.43 GB) it's been parsed to CVS files and than imported into a Cassandra Cluster. Thanks to the Yahoo Finance service, it's possible to update every day the databse with the last session results. It's also possible to add a new stock to the dataset, coming from every market exchange of every country.



4.2 MongoDB

”MongoDB is a general purpose, document-based, distributed database build for modern application developers and for the cloud era.” Taken from www.mongodb.com.
MongoDB is a very famous document database with a great support for cloud operations, witch will improve the avaliability of our application. It also suport a lot of analytics functions and the creation of custom indexes in order to speedup read operations. In order to orgnize data ina a meaningfull and memery-optimal way, we opted for this

structure:



This scheme is composed by 3 collections: **stocks**, **users**, **admins**;

- The stocks collection contains one document for each stock; inside this document are stored all the general information about the stock, wich is identified by the attribute SYMBOL; some basic information are always presents, while others are missing for some stocks; we decided to keep these last type of informations where possible, exploiting the schemaless property of the documentat databse;

- The users collection contains one docuemt for each user regitered on the application; for every user login credentials are stored, along with few personal information; for every user is also stored an array of documents named PORFOLIOS: this array contains the porfolios of the user. Each portfolio has a scheme, witch include an array of TITLEs, named COMPOSITION, witch represent the settlement of the portfolio itsef. This nested structure it's been preferred from splittin data in different collections, because all the information of a user, including his portfolios, are frequently needed toghether; on the other hand, there aren't such operations that involve porfolios owneds by different users.
- The admins collection contains the admins login credentials toghether with few personal informations about them; we decided to crerate a separated collection for administators to improve the security of the administration features: in this way is impossible to inject administration privileges throw the login command.

4.2.1 Aggregations

One of the main features of own application is the possibility to choose some stocks from the market and combine them in to a portfolio. When a user is looking for a stock, he want to know statistic about **industies** and **sectors**, along with classification by **level of capitalization** and **PE ratio**; in ordewr to do so, we will provide these aggregation pipelines:

- the total market capitalization of each sector
- the total market capitalization of each industry
- the total market capitalization of stocks coming from the same country
- the avarage PE ratio of stocks working in the same sectors
- the avarage PE ratio of stocks working in the same industry
- the avarage PE ratio of stocks coming from the same country
- the avarage PE ratio of stocks beeing in a specific range of market capitalization

We provide here an example of an aggregation mongo query:

```
    /**
    * Aggregates data with filtering and grouping by an attribute, can
       compute
    * sum, avg ecc.
    *
```

```java
     * @param collection the collection where to perform the operation;
     * @param filter filter to be used to find the documents.
     * @param groupField the filed used to group the aggregation
     * @param aggregator the aggregator function and field
     *
     * @return iterable object containing the result of the aggregation.
     */
    public AggregateIterable<Document> aggregate(final Bson filter, final
        String groupField, final BsonField aggregator, final
        MongoCollection<Document> collection) {
        Bson match = Aggregates.match(filter);
        return  collection.aggregate(
                Arrays.asList(match, Aggregates.group("$"+groupField,
                    aggregator)));
    }
```

```java
     MongoCollection<Document> collection1 =
         dbManager.getCollection(
             StocksimCollection.STOCKS.getCollectionName()
         );
     // aggregate examples
     final Bson equity= eq("quoteType", "EQUITY"); //filter(s)
     // name of the field projected, field to accumulate
     // type of accumulation (sum, avg...)
     final BsonField
         marketCapAccumulator=Accumulators.sum("totalCap","$marketCap");
     AggregateIterable<Document> aggregateList =
                                          // grouping attribute
             dbManager.aggregate(equity, "sector", marketCapAccumulator,
                 collection1);
     for (Document document : aggregateList) {
         System.out.println(document);
     }
     // avg example with nested attribute
     final BsonField PEAccumulator=Accumulators.avg("avgPE","$trailingPE");
     aggregateList =
             dbManager.aggregate(equity, "location.country",
                     PEAccumulator, collection1);
     for (Document document : aggregateList) {
         System.out.println(document);
     }
```

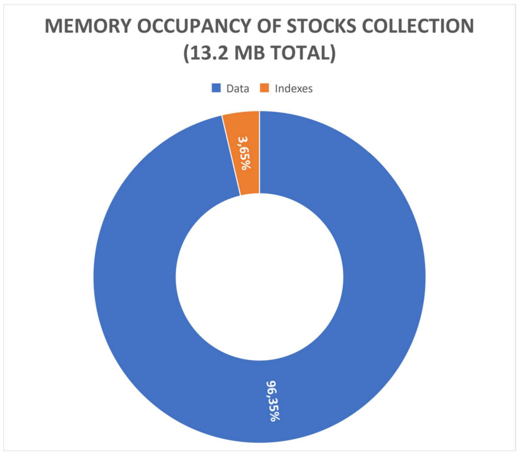
4.2.2 Indexes

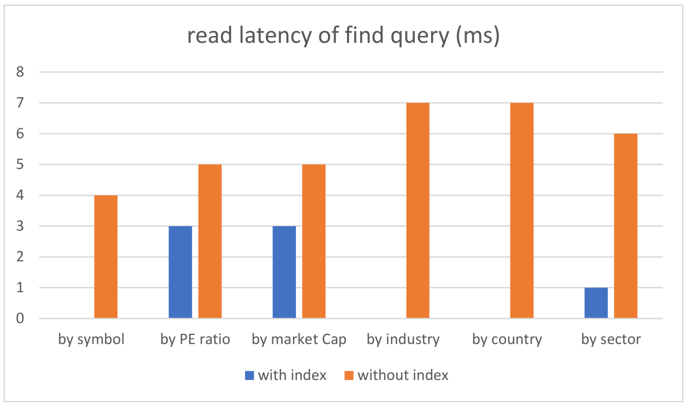
In order to speeup read operation in the document database, we decided to introduce some custom indexes:

- a REGULAR and UNIQUE index on the attribute **symbol** in the collection stocks;
- a REGULAR index on the attribute **marketCAP** in the collection stocks;
- a REGULAR index on the attribute **trailingPE** in the collection stocks;

- a REGULAR index on the attribute **sector** in the collection stocks;
- a REGULAR index on the attribute **industry** in the collection stocks;
- a REGULAR index on the attribute **country** in the collection stocks;
- a REGULAR and UNIQUE index on the attribute **username** in the collection users;

We provide some statistic that endorse our idexes choises





Analog results can be found about the username index in the users collection.

4.3 Apache Cassandra

"The Apache Cassandra database is the right choice when you need scalability and high availability without compromising performance. Linear scalability and proven fault-tolerance on commodity hardware or cloud infrastructure make it the perfect platform for mission-critical data. Cassandra's support for replicating across multiple datacenters is best-in-class, providing lower latency for your users and the peace of mind of knowing that you can survive regional outages." www.cassandra.apache.org
Apache Cassandra is a database designed for high scalability and availability; it's capable to handle a huge amount of data and manage it in a decentralized architecture across multiple nodes. It's build to be write optimized, but with right indexes choises also read latency can be improved; tables schemes and analytics functions can be customized. This is the scheme of our Cassandra databse:

Ticker		
PK	**PARTITION KEY**	Symbol: string
	CLUSTERING KEY	Date: LocalDate
		Open: float Close: float High: float Low: float Volume: float Asjusted_close: float

4.3.1 Aggregations

In order to provide snaphots and statistics of stocks and portfolios trends oevr time, we explit the customization functionalities of Cassandra; a custom aggaregation it's been created, specifically to provide aggegate values of more than one day, for a given perdiod of time; this allow us to obtain different granularity for stock market data, computed on server side; this will not over overwelm the server, because the aggregator will execute, for each row, beetween a memory access and the following. This will reduce by a lot the data to be transitted from the node to the client, saving bandwith and time.

```

/* State function to be executed for every row*/
CREATE OR REPLACE FUNCTION PeriodStateParam (

/* the state, containing the aggregation result till this row */
    state map<date,frozen<map<text, float>>>,
/*  the parameter ndays indicate the duration
*   of the period aggregation, in days          */
    ndays int, data date,
    open float,  close float , high float,  low float,
    volume float ,  adj_close float)
CALLED ON NULL INPUT
RETURNS map<date,frozen<map<text, float>>>
LANGUAGE java AS '
if (data != null) {
    int d=0;
    Map<String, Float> statemap=null;

    for(d=1; d<ndays;){
        if((statemap=state.get(
                data.add(Calendar.DAY_OF_MONTH,d)
            ))!=null)
```

```
                break;
            d++;
        }
        if(d==ndays){
            statemap=new HashMap<String, Float>();
            statemap.put("open", open);
            statemap.put("close", close);
            statemap.put("high", high);
            statemap.put("low", low);
            statemap.put("volume", volume);
            statemap.put("adj_close", adj_close);
            state.put(data,statemap);
        }
        else{
                if(high>statemap.get("high"))
                    statemap.replace("high", high);
                if(low<statemap.get("low"))
                    statemap.replace("low", low);
                statemap.replace("volume",statemap.get("volume")+ volume);
                statemap.replace("open",open);
                state.replace(data, statemap);
        }
    }
    return state;'
     ;

    /*  aggregate declartation
    *   this aggregation geenrate a map data structure (JSON like):
    *   the key is the end date of each period of nday days,
    *   and the value is another map containing the aggregate
    *   values of the period as:
    *        the open of the first day
    *        the close and adjusted close of the last day
    *        the maximun of the highs
    *        the minimum of the lows
    *        the sum of the volumes
    */
    CREATE OR REPLACE AGGREGATE PeriodParam
        ( int, date,float, float,float, float,float, float )
    SFUNC PeriodStateParam
    STYPE map<date,frozen<map<text, float>>>
    INITCOND {}; /* no initial condition is necessary */

    /* example of usage, it can be used also with grouping by symbol */
    select PeriodParam(
        20, date, open, close, high, low, volume, adj_close)
        as Period from tickers where
        date<'2020-12-1' and date>'2020-6-10'
        and symbol='TSLA';
```

4.3.2 Indexes

- The PARTITIO KEY index is part of the PRIMARY KEY and it's used to shard the dataset across the nodes. This index is build on the string symbol, unique for each stock;

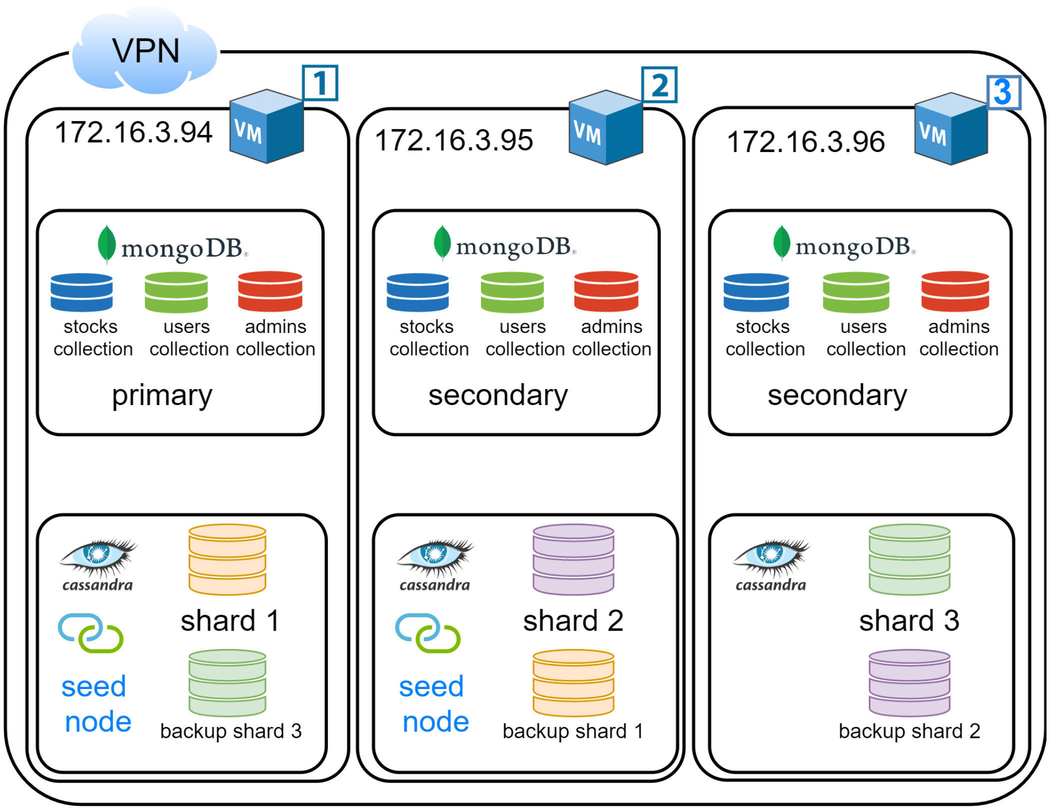
- the CLUSTERING KEY it's also part of the PRIMARY KEY, and it's used to mantain rows chronologic orde. This index is build on the attribute date;

4.4 Sharding and Replicas

The MongoDB cluster and the Cassandra cluster are deployed on 3 vitual machines provided by the University of Pisa; Our architecture is oriented to the availability of the service, and build for the maximum scalability and decentralization.

- The casandra cluster is build among all 3 nodes, and data are sharded by the ticker symbol; in this way every node store 1/3 of the main dataset, and aggregation functions are computed on records that stay in the same node; each node also store a backup of the data assigned to another node, give us a replicazion factor of 2. There are 2 seed nodes, wich are responsible for the cluster: the cluster is online as long as one of them keep working. This is indeed a minor issue, because in any case, with only ne node, the dataset would by incomplete. The decentralized behavior of Cassabdra ensure than even if all the node go offline, the cluster return available as soon as one seed server go back online.

- The mongoDB cluster is also build among all 3 virtual machines, and the service is replicated; an initial primary server is elected, then another one will take it's place if it goes down. The cluster is available as long as one server is working, and incoming traffic could be balanced with the ”nearest” preference on client connection; in this way the client would connect to the server with the lowest ping time.



4.5 Apache Cassandra vs MongoDB

Chapter 5

Software architecture

5.1 Maven Multi-Module Structure

5.1.1 Library

5.1.2 Server

5.1.3 Client

5.2 Apache Maven Assembly Plugin

The Assembly Plugin for Maven enables developers to combine project output into a single distributable archive that also contains dependencies, modules, site documentation, and other files.

Chapter 6

Conclusions

Content.

Part II

User Manual

Chapter 7

StockSim Server Manual

For an application dealing with the stock market, it is essential to always provide up-to-date, consistent and reliable data. This is the purpose of the **StockSim Server** program. It is not intended to be distributed to end users. It is thought to be running 24/7.

The StockSim Server has two different startup modes: the first one

```
$ java -jar Server.jar

Welcome to the StockSim Server.

DATA CONSISTENCY CHECK SUCCESS. PROCEEDING WITH UPDATE.

Updating historical data for: IRBT.
Last update date: 2020-12-30.
Days since last update: 2.
Historical data updated for IRBT. Moving on.

...

Updating historical data for: EWU.
Last update date: 2020-12-28.
Days since last update: 4.
Historical data updated for EWU. Moving on.

...


Updating historical data for: XRX.
Last update date: 2020-12-28.
Days since last update: 4.
Historical data updated for XRX. Moving on.

Historical data update terminated.
Elapsed time: 1 hrs, 37 mins, 3622 secs.
Exceptions during update process: 8.
```

```
Failed updates: [ARKG, HDS, IWFH, LDEM, WWW, XVV, GINN, AAAU].

Available Commands:

status        check databases status.
update        update databases historical data.
quit          quit Stocksim server.
>
[UPDATER THREAD] Current New York time: 2021-01-02T07:50-[America/New_York]
[UPDATER THREAD] Going to sleep for 13 hours before next update.
>
```

which executes the `historical data update` procedure right after startup. Whereas, the second startup mode can be triggered using the `--no-update` command line argument:

```
$ java -jar Server.jar --no-update

Welcome to the StockSim Server.

Available Commands:
status        check databases status.
update        update databases historical data.
quit          quit Stocksim server.
>
[UPDATER THREAD] Current New York time: 2021-01-02T06:29-[America/New_York]
[UPDATER THREAD] Going to sleep for 14 hours before next update.
> update
DATA CONSISTENCY CHECK SUCCESS. PROCEEDING WITH UPDATE.

Updating historical data for: AUPH.
Last update date: 2020-12-31.
Days since last update: 1.
Historical data for AUPH already up to date. Moving on.

...

Updating historical data for: EWU.
Last update date: 2020-12-31.
Days since last update: 1.
Historical data for EWU already up to date. Moving on.

...

Updating historical data for: XRX.
Last update date: 2020-12-31.
Days since last update: 1.
Historical data for XRX already up to date. Moving on.

Historical data update terminated.
Elapsed time: 0 hrs, 4 mins, 44 secs.
Exceptions during update process: 3.
Failed updates: [ARKG, XVV, DUAL].
```

```

Available Commands:
status        check databases status.
update        update databases historical data.
quit          quit Stocksim server.
>
```

in this mode, no update is executed right after startup, the main menu is shown and the user can decide the action to be performed.

In the previously shown examples, we should pay attention to the following

- `DATA CONSISTENCY CHECK SUCCESS. PROCEEDING WITH UPDATE.`
- `the sotcksim server automatically detects the last update date, the number of days since the last update and fetches the required data using Yahoo Finance for EACH and EVERY ticker symbol.`
- `Historical Data Update logs.`
- `[UPDATER THREAD]`

Chapter 8

StockSim Client Manual

The StockSim Client has two different running modes: the first one

```
Welcome to the StockSim Client.

*** [RUNNING IN USER MODE] ***

Available Commands:
login          login to your user account.
quit           quit StockSim client.
```

is the `user mode`. Whereas, the second running mode can be triggered using the `--admin` command line argument:

```
$ java -jar Client.jar --admin

Welcome to the StockSim Client.

*** [RUNNING IN ADMIN MODE] ***

Available Commands:
login          login to your admin account.
quit           quit StockSim client.
```

and is the `admin mode`. Although they might look like the same, the available menu actions differ once the user/admin login has been executed.

8.1 StockSim Client User Mode

8.2 StockSim Client Admin Mode