Analyzing Airbnb listings

Arjan Langerak 4211235 | **Daan Vermunt** 4319494 | **Neha Sree Thuraka** 4743598 | **Nivedita Prasad** 4712099 | **Sharad Shriram** 4671082

One of the major issues that a person faces while travelling to an unknown city is finding a good lodging, especially when the duration of stay is just a few days. Airbnb is a platform that enables people to lease or rent short-term lodging. Analyzing this data can produce insights into travel and renting behaviour. In this project, the pros and cons of MySQL, Apache Cassandra, Apache Spark and Apache Flink in performing the tasks defined in Section 2 on Airbnb data are presented.

1 Data description

The Airbnb data contains 16 different cities in the United States from 2008 to 2018 which is collected from `insideairbnb.com`. Each city consists out of three different files: listings, calendar and reviews. The listing file contains a detailed description of each listing, the calendar file describes of the availability and price of each listing and the reviews file consists of reviews written by users about each listing. Each of these files is in a csv format. For the tasks defined in Section 2, the calendar, listings and reviews data is used. The data has been cleaned and only the columns that are needed for the tasks have been extracted. Detailed description about this is given in the following sub-sections.

1.1 Reviews

Table 1 shows the attributes used from *reviews* data and their corresponding descriptions. The data in the *comments* attribute has been cleaned to remove special characters, extra white spaces and non-english words.

Table 1: reviews

Attribute	Description
listing_id	Unique identifier for listing
id	Unique identifier for review
date	Date of the review
comments	Comment/review

Table 2: listings

Attribute	Description
id	Unique identifier for listing
state	State of listing
city	City of listing
room_type	Room_type of listing
price	Price of the listing

Table 3: calendar

Attribute	Description
listing_id	Unique identifier for listing
date	The day of the calendar
available	A Boolean whether or not the listing is available
price	The price for the listing this day

1.2 Listings

Table 2 shows the attributes used from *listings* data and their corresponding descriptions. The data in the *price* attribute has been processed to remove $ and has been converted into integer. Also, some entries from the data had to be removed that had values in the wrong place, for example, a city name instead of a price.

1.3 Calendar

Table 3 shows the attributes used from *calendar* data and their corresponding descriptions. The *price* attribute is cleaned up in the same way as in 1.2.

2 Queries

To test our systems, 3 tasks are defined to run on every system. For all stream tasks, the listings table is in memory and either the Calendar or Review data will be available in a stream.

1. What 5 words, except irrelevant words, are used most in reviews of each listing?
2. Finding the most popular city/state in each month of the year
3. Analyzing the trend of different types of listings over a period of time

For the first query, only need the Review data is needed, as this contain a listing ID and the text of every review. 'Irrelevant words' are defined as the list of stop words defined by the NLTK natural language processing framework.

For the second query, a precise popularity can't be made because the *available* attribute from calendar is also false in case the renter decided to have the listing not available. To get an estimate, the popularity is defined as the number of reviews a listings in a location gets in a given period of time.

For the last query the types of listings are defined by the options of room types. The trend of one options is defined as the amount of available days in that room type per month.

3 MySQL

MySQL is a domain specific programming language to manage data in relational database management system. The advantage of SQL is that it handles structured data where the relation between different entities/tables can be established with the help of joins. The data is organized in the form of tables consisting of columns and rows for efficient access.

3.1 Setup

To store data in MySQL we need tables and the dataset of listings and reviews are stored as tables. As a first step, the columns have to be manually created after analyzing the data-type and values present in each column. The queries were executed in MySQL workbench and later connected to python to perform further analysis.

3.2 ETL Process

The data has to be pre-processed in any Extract-Transform-Load (ETL) process. For this project, python is used for this.

Initially the data was extracted from the data csv files and read into panda dataframes. Then, the date columns are checked if these are in the correct format (yyyy-mm-dd). Any wrong entries are removed from the dataframes.

Similarly, the '$' and ',' symbol from the price columns are removed. This takes a lot of time as the pandas has to process more than 10,00,000 rows. Furthermore, the columns had to be changed to float and the Nan values had to filled with 0.

The SQL tables were created for listings and reviews and then the data was loaded from the pre-processed csv files using import wizard. This task took almost 3 hours and more to load data into the tables. This already shows that MYSQL wouldnt be suited for handling large amounts of data fast.

For all the queries we imported to pymysql package of python to connect to the mysql database at the backend.

3.3 Query 1

```
//To execute queries in the tables present in the wdm
  ↪  database
use wdm;
//Query 1
SELECT listing_id, comments FROM reviews;

SET group_concat_max_len = CAST(
    (SELECT SUM(LENGTH(comments)) + COUNT(*) * LENGTH('
      ↪  ')
    FROM reviews )
    AS UNSIGNED);

SELECT listing_id,
GROUP_CONCAT( r.comments SEPARATOR ' ') as
  ↪  concat_comments
FROM reviews r
GROUP BY listing_id;
```

The query shown above extracts the listing id and the comments that has to be grouped based on the listing id. The comments are concatenated with each other in order to find the 5 most frequent used words other than the stop words as mentioned by the nltk package in python. Packages like FreqDist, Counter, word_tokenize, stopwords were additionally imported for determining the frequent words in the concatenated comments. The result were common words like "place", "comfortable", "clean", "home", "like" for listing id "10034077" and for listing id "10034183" it was "apartment", "beautiful","helpful" for example.

3.3.1 Pro and Cons

+ Since no join was involved, the retrieval was just 0.016 seconds.
− Loading of the data into the table review requires more than 4 hours and the connection timed out after 2 hours passed. So, only half of the data was loaded into the tables.

3.4 Query 2

```
//Query 2
select state, city, month1 from reviews_reduced r,
  ↪  listings_reduced l where r.listing_id = l.id group
  ↪  by l.state;
```

The query stated above uses an inner join between the reviews and the listings tables to combine the different tables. This result data is due to the grouping of the different states and city selected. The Most popular State was TX and the most popular city was Austin.

3.4.1 Pro and Cons

+ The inner join can be used to combine data which have a common identifier.
− The results is shown for only few cities and states keeping the time involved in mind for retrieving the query. The time involved exceeds the 6000 seconds we could edit from 600 seconds in the MySQl application. Hence connection time out occurs even before the query has been successfully executed.

3.5 Query 3

```
//Query 3
select r.year1 ,l.room_type from reviews_redu_room r,
  ↪  listings_redu_room l where r.listing_id = l.id
  ↪  group by l.room_type;
```

Here, an inner join is also made between the reviews and listings table with listing id as the primary key and foreign key used for join and the results were that the trends were Entire home/apt for 2016 and Private room for 2017.

3.5.1 Pro and Cons

For this query, the pro's and cons are the same as for the previous query. However, to be able to test the queries, a fragment of the different tables are made which are suffixed with '_reduced'.

4 Apache Cassandra

Apache Cassandra is a distributed NoSQL database management system.

4.1 Setup

To store the tables in Cassandra, a keyspace is needed. Cassandra makes use of two kinds of replication strategies to store the replicas

SimpleStrategy: It contains only data center
NetworkTopologyStrategy: It contains multiple data centers

The replication factor can help in assigning the number of replicas we wish to retain. A keyspace *'wdm'* has been created with *SimpleStrategy* and with a replication factor 1. Further, using *cassandra.cluster* package in python, the data has been analyzed.

4.2 Query 1

Listing 1 shows the data model in Cassandra for Query 1. The review_id (partition key) has been used to determine the node the row should be saved in. The listing_id (clustering key) helps in deciding where in node it should be stored. In other words, it can be used for sorting.

Listing 1: Data model for Query 1

```
CREATE TABLE review (
listing_id int,id int,comments text, PRIMARY KEY (id,
  ↪  listing_id)
);
```

Simple python code has been used to join the reviews that have same listing_id. A set of stopwords from the package *nltk* have been used to filter out the words.

4.2.1 Pros and Cons

+ The data which actually consists of around 1.1 million rows is loaded in approximately 3 minutes.

4.3 Query 2

This query uses two datasets: reviews and listings from Table 1 and 2. To use it in Cassandra, the data has been transformed into a single dataset. As the data in Cassandra cannot be grouped by attributes that are not partition key/ the clustering key, to perform this query, two different data models had to be done. One with state as the partition key in Listing 2 and the other with city and state in Listing 3. Month has been used as clustering key in both of them.

Listing 2: Data model for finding popular state

```
CREATE TABLE popularstate(day int, month int, year int,
  ↪  state text, PRIMARY KEY (state,month));
```

Listing 3: Data model for finding popular city

```
CREATE TABLE popularcity(
day int, month int, year int, state text, city text,
  ↪  PRIMARY KEY ((city,state),month)
);
```

4.3.1 Pros and Cons

− Cannot do joins: so a new dataset had to be created
− Need to store data multiple times

4.4 Query 3

Similar to that of Query 2, this query also uses two datasets: reviews and listings from Table 1 and 2. Therefore, the two datasets had to be combined into a single dataset for this query as well. Data model for this query shown in listing 4 has room_type as the partition key and year as the clustering key.

Listing 4: Data model for finding trend of room type over years

```
CREATE TABLE roomtrend(year int, room_type text, PRIMARY
  ↪  KEY (room_type,year));
```

4.4.1 Pros and Cons

This query has also the same cons as the previous query.

4.5 Summary of Pros and Cons

+ It can handle huge volumes of data and can load them really fast
+ It has high writing speed. It took less than 1 minute to write more than 1 million rows
+ As it saves many replicas of the data, there may not be loss of data: fault tolerant
− The data is to be modelled depending upon the queries. So rather than working on structure of query, it is important to model the data depending upon the query
− May have to restructure data depending upon the query as it is not possible to join two different tables: For example, creating a single table from two datasets
− May have to store the same data in different ways depending upon the query as it is not possible to group by attributes other than the partition keys

5 Apache Kafka & Nifi

The upcoming two frameworks require streams of data. In order to create streams from the data files which can be run in a distributed fashion, Apache Kafka and Apache Nifi have been used.

Kafka is a distributed publish-subscribe message system which can provide streams of data that are available at specified topics.

Nifi is a system for creating a dataflow system. It allows to use and create nodes, which are called processors that can be connected together. These nodes will fulfill various roles such as routing data to other nodes, acts as an input or an output to the system. In our setup, the input of the system are the data files and the output are nodes that publish the data to a Kafka topic as JSON objects.

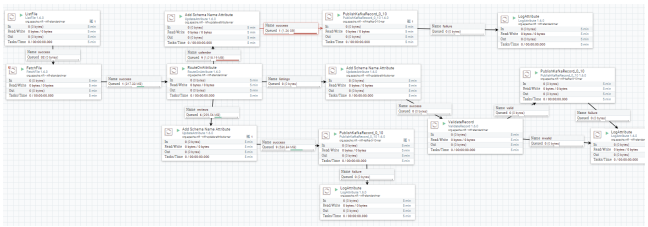

Fig. 1: Nifi model for streaming data files to Kafka

6 Apache Flink

Apache Flink is a stream processing framework which can also be used for batch processing of data.

6.0.1 Running a Flink job

In our test setup, a single Flink cluster was started on a local machine by using the supplied scripts that comes with Flink. There is also a Maven setup which allows for a quick start into developing Flink jobs that starts a minicluster from the Maven project.

6.0.2 Connecting to a stream

To connect to a Kafka stream, special classes exists that can consume data from a Kafka stream and create a Flink datastream object. To be able to this succesfully, the incoming stream needs to be deserialized. For this, an inbuild deserializer is used that creates a string object. This is then passed into a JSON parses which will map the strings to simple data objects (case classes).

6.1 Queries

Because Flink handles data as a stream our approach to the queries should allow for that. This means that every data item can only be viewed once. Below, the different queries are listed.

```
val counts = reviews
.flatMap { r => {
    r.review.get.split("\\s")
    .map(s => s.replaceAll("\"", "")
        .replaceAll("\\s", "")
        .toLowerCase())
    .filter(s => s != "")
    .map(s => ListingWord(r.listing_id, s))
    }} // (1)
.filter( s => !stopWords.contains(s.word)) // (2)
.map { w => ListingWordCounts(
   w.listing_id, Array(WordCount(w.word, 1))) } // (3)
.groupBy("listing_id")
.reduce( (el1, el2) =>
    wordCountPerListingReducer(el1, el2)) // (4)
.map( w => getHighestFive(w)) // (5)
```

For the first query the process takes 5 steps, in step 1 a flat map converts the entire review into a set of tuples with the listing id and the word used. Next, a filter filters out all words that were defined as irrelevant (2) and map the rest to a tuple of a listing id and an array containing one word with a count of 1 (3). After this, it is reduces to a single tuple per listing id and an array of each unique word with the count of that word in all reviews of that listing (4). Lastly the top 5 words are used as the result of the query (5).

```
val popularLocation = reviewStream
      .filter(obj => listings.contains(obj.listing_id))
      .map(r => DateLoc(r.date,
        ↪  listings(r.listing_id).city))
      .map(dl => DateLocCount(
        dateToMonth(dl.month),
        DateLoc(dateToMonth(dl.month), dl.loc),
        1)) // (1)
      .keyBy("dateLoc") // (2)
      .sum("count")
      .keyBy("month") // (3)
      .max("count")
```

For the second query, the review stream is also used, but this stream does not contain all the location of a listing. So for every item that comes in, a lookup is needed in the listings table. The listings table is small enough to fit in memory so it is loaded to a HashTable, but in many use cases this might not be the case which can makes this lookup a bottleneck in the system. Also, there are listing id's that there are not available in the listings table but do exists in the review data. To deal with that, the data is first filtered to remove entries with id's that do not exist in the listings table.
The query starts by mapping the review to a date and location and a count of 1, this is where the lookup takes place (1). Next, a sum will count the date location pairs (2). Next the result is obtained by taking the maximum per month (3).

```
val listingsTypes = calendar
      .filter(obj => listings.contains(obj.listing_id))
      .filter(c => c.available == "t") // (1)
      .map (c => CalItemTypeCount(
      CalItemType(
        dateToMonth(c.date),
        listings(c.listingId).listType),
      1)) // (2)
      .keyBy("calItemType") // (3)
      .sum("count")
```

The last query uses a stream of the calendar data, and like the previous query it also requires a lookup from the listings table and filtering out id's that are not available.
First, all unavailable days are filtered(1) and then map to a count per month - room type tuple (2). Now, a sum per month - item type tuple is performed to get the result (3).

6.1.1 Flink batch

All queries where also implemented using the batch processing of Flink. Flink allows to read the files directly from CSV and process them using a MapReduce schema. It was a painless task to convert the streaming jobs to batch jobs.

6.2 Pro's and cons

+ It can handle stream of data which means it can process real time data.
+ After the learning curve, handling data feels just like regular programming with functions calls, objects etc.
– Creating a streaming system for the csv files adds a lot of complexity to the system.

7 Apache Spark

Apache Spark was developed to overcome the limitation of MapReduce and uses the resilient distributed dataset (RDD) as the architectural framework over HDFS. RDD offers a good deal of processing custom data which means we create the rules to process data that are to be tested. RDD APIs of Spark do not support Python and has the least optimization which makes RDD the slowest in terms of performance. Spark DataFrames and Spark-SQL offer faster, better performances as they are well-optimized for code generation and performs very well in Scala as shown from the query execution times. Spark offers a lot of useful data processing packages which make it suitable for data analysis and given that it is fault-tolerant, distributed data analysis can be done in ease.

7.1 Spark-SQL

SQL is used for analytics but it is a pain to connect data processing pipelines like Spark or Hadoop to SQL database. Spark SQL not only contains all the advance database optimization, but also seamlessly intermixes SQL queries with Scala. This yields high performance, achieved by using techniques from the database and facilitates relation data processing. Spark-SQL supports modern data sources like CSV which makes it easier to get-started but is not as flexible as RDD to customize how data is to be read.

7.2 Implementation

Spark was setup in a Linux VM with 1 processor, 6GB of Memory and 30 GB of Storage on a host system which has Core i5 processor, 12GB Memory and 500GB storage. The datasets are loaded as Spark Dataframes and takes the most time. For the implementation we use a single cluster as there was not sufficient time to study, implement and compare the performance of a single-cluster vs. multi-cluster set up of Apache Spark. The design-rationale is supported with the available local data which may not benefit with a distributed

processing setup. The testing of the queries was done on the interactive `spark-shell` and is later combined as a single file which can be run in `spark-shell`.

Data Preparation: Before Implementing the queries we load the data to the spark environment as a dataframe and then do some initial preprocessing which include cleaning the data, applying filters for fundamental data operations and then creating an abstraction (views) of the data to use Spark-SQL for querying. The following code snippet illustrates this process for the calendar dataset:

```
//calendar dataset
val calendar = sqlContext.read
  ↪  .format("com.databricks.spark.csv")
  ↪  .option("header","true")
  ↪  .option("inferSchema","true")
  ↪  .load("../calendar.csv")

//data cleaning #1 - remove rows where id is not numeric
val cleanCalendar =
  ↪  calendar.select("*").filter($"listing_id" rlike
  ↪  "^([0-9])+$")

//register df as temp sql view
cleanCalendar.createOrReplaceTempView("calendarView")
```

7.3 Discussion on the Queries

Query:1 - To look at the commonly used words to describe a Airbnb location we do a word count on the reviews. For this query, we first process the reviews dataset and do a count of the frequency of word occurrences.

```
spark.time((session.sql("select count(distinct
  ↪  listing_id)  from calendarView where available =
  ↪  'f'" )).show())
```

A methodological implementation will use a tokenizer, and then remove stop words. When we tried the same using built-in Spark packages, we encountered YARN run time errors.

Query:2- For the second query we look at popular listings, cities, places, etc. The implementation of this query was done by creating a join on the calendar and listings dataset based on the listing id. We then make a query which does a sum on the availability of the listings, if the listing was not available, we assume it is popular. From a query to check the availability of a listing we found some which were unavailable throughout the year. This made us change the query to get more insight of the listing and the query is as follows:

```
spark.time((session.sql("select id, name, room_type,
  ↪  property_type, neighborhood_overview, city,
  ↪  sum(case when available='f' then 1 END) from
  ↪  combinedData group by listing_id order by sum(case
  ↪  when available='f' then 1 END) desc)" )).show())
```

However, this query gave an SQL Parser error which we were unable to resolve.

Query:3- For the third query, where we query the calendar dataset, to see availability and retrieve the listings that are not available most of the time. The execution time for this query was 6 minutes 35 seconds. This query looks at the number of times a particular listing is unavailable and since the dataset is basically a time-series per listing which is bounded we assume that a high sum on the listing means it is more popular.

```
spark.time((session.sql("select(select listing_id,
  ↪  sum(case when available='f' then 1 END) from
  ↪  calendarView group by listing_id order by sum(case
  ↪  when available='f' then 1 END) desc)" )).show())
```

The same query can have a DataFrame version which is given as:

```
calendar.groupBy("listing_id").count().filter(calendar("available")
  ↪  === "f")
```

7.4 Observations and Inferences

Spark took anywhere between 20 to 40 minutes to load the data which amounts to close to 2.5GB. For SQL-like queries which involved aggregation/ counting, etc. the query execution took 5-6 minutes on the calendar dataset which was 1.4 GB in size. Simpler queries like displaying select attributes took the least time for execution in the range of 21-40 seconds for tables. From the query execution time we observe and infer that the Apache Spark is able to process a large data dump quickly and without a spike in hardware resource consumption.

From a platform perspective, Spark with Scala has a steep learning curve and with a plethora of different usages, methods and APIs finding the right tools for this dataset was cumbersome. The experience with Spark-SQL was good, it is straight-forward for getting started. The limitation while implementing queries 2 and 3 was that sub-queries was not allowed to be added as a condition which made filtering results difficult. We believe this might be addressed in future releases.

Code

All the implementation aspects, data and source can be accessed on the Github Repository: https://github.com/DaanVermunt/WDM-group-5.