School of Engineering and Computer Science

SWEN 432
Advanced Database Design and Implementation

Assignment 1

Due date: Wednesday 05 April at 23:59

The objective of this assignment is to test your understanding of Cassandra Cloud Database Management System and your ability to apply this knowledge. The Assignment is worth 5.0% of your final grade. The Assignment is marked out of 100.

You will need to use Cassandra to answer a number of assignment questions. Cassandra has been already installed on our school system. There is an Instruction for using Cassandra on our lab workstations given at the end of the Assignment.

1. Overview

Assume Wellington Tranz Metro has acquired an iPhone application that works as a data recorder for railway vehicles (cars, engines). The application uses mobile data connections to send information to servers in real time. The information includes measurements such as the location and speed of the vehicle. They have selected Cassandra as the CDBMS to use for this project. Your job is to design a database, and advise on its use. You are also required to implement a test database with a substantial part of sample data provided in the assignment, as a proof of concept. Note, the grouping of test data in sample data tables does not necessarily imply their belonging to the same or different Cassandra CQL tables.

2. Application Requirements

The application needs a database containing data about drivers, railway vehicles, time table, availability of drivers and vehicles, and current position and speed of vehicles.

Drivers

A database administrator creates an account for each railway vehicle driver through the iPhone application. Each account has the following information:

1. driver_name,

2. password,
3. mobile,
4. current_position (e.g. 'Wellington', or 'FA4567', or 'not_available'),
5. skill (a set of train types the driver is qualified to drive).

All driver names have to be unique.

Drivers can change their password through the iPhone application. A driver must provide their current password and the new one. The application must only update the password if the correct current password is provided. Assume the iPhone sends a hashed password, so you do not worry about encryption on the server side.

When a driver comes to work at a station, she/he updates her/his `current_position` by the name of a station (e.g. `'Wellington'`). When the application assigns a driver to a service it updates her/his `current_position` to a `vehicle_id` value (e.g. `'FA4567'`) of a vehicle that is also assigned to the service. When a driver deregisters from work, she/he sets her/his `current_position` to '`not_available`'.

The application also needs the number of days per month a driver registered at work for payroll calculation at the end of a month. The payroll calculation algorithm is not of our concern, but we need to provide the appropriate data.

When a driver qualifies to drive a new vehicle, her/his `skill` gets updated.

Railway Vehicle

A database administrator registers railway vehicles. Each railway vehicle registration has the following information:

1. `vehicle_id`
2. `status (e.g.'Upper Hutt', or 'in_use', or 'maintenance', or 'out_of_order'),`
3. `type (e.g. 'Gulliver', or 'Ganz Mavag', or 'Matangi', or 'Kiwi Rail)`

The attribute `vehicle_id` has to be unique.

If the `status` property of a train vehicle has a station name as the value, it means the vehicle is operational and available and its current location is the station with the given name.

Time Table

Wellington Metro Time Table contains information about lines, services, and stations. Wellington Metro contains several lines. Each line contains several services.

Each line has a `line_name` that has to be unique.

Each service of a line has:

1. service_no (the ordinal number of a service within a line), and
2. A sequence of (station_name, time, distance) triples.

In the sequence of (station_name, time, distance) triples, the first station_name is the name of the departure station, and the last is the name of the destination station. For all stations in the sequence, except for the destination station, the attribute time is the departure time. The attribute time for the destination station is the arrival time. The attribute time is of the type int (e.g. 1005 for 10:05, or 1947 for 19:47). The sequence is ordered according to the rising values of the attribute time.

Different services between the same two departing and destination station may have different numbers of stations. A station may be used by different lines, and their services.

Beside the name, a station also has:

1. latitude (of the type double), and
2. longitude (of the type double).

The time table data have a number of uses. One is to publish a time table for passengers. (Note, passengers are not interested for data like service_no, distance, latitude, longitude.) The other uses are discussed in the next sections.

Allocation of Vehicles and Drivers to Services

The application extracts a list of departure stations with all services departing from a station, and orders that data according to descending values of the service departure time.

For each departure station, the application extracts from the list of departure stations a service due to be dispatched next, finds an available vehicle at the departure station, stores the identification data of the service in the vehicle's iPhone (to allow pairing data points and services), and updates the status of the vehicle. The application also finds an available driver having the needed skill at the departure station, sends her/him an allocation message on her/his mobile, and updates her/his current_position. When the driver enters the vehicle, she/he will be allowed to start the engine only after a successful authentication. She/he performs authentication by connecting to her/his database record via iPhone.

When the service reaches the destination station, the driver turns off the engine, and the application updates the driver's and vehicle's records to reflect their availability at the destination station. Also, the application records the information that the vehicle travelled an additional distance. The application keeps record

about a daily and total distance travelled for each vehicle, and uses the data for planning the maintenance.

Data Points

The application automatically records information about the time, speed, and the position of an operational vehicle from the iPhone in the vehicle.

Each sampling of the position and other information is considered a Data Point. When the vehicle's engine has been started, the application starts sending data points every 10 seconds.

Each Data Point contains the identification data of the service and the following information:

1. `day` (of the type `int`, e.g. `20170326`),
2. `sequence` (of the type `timestamp`),
3. `latitude` (of the type `double`),
4. `longitude` (of the type `double`),
5. `speed` (of the type `double`),

The application uses data points to calculate:

- Estimated departure time of a service to be displayed on station screens for passengers' convenience, and
- Time delays at the destination station to be used to improve planning, time tabling, track maintenance, and other.

The details of the calculations above are out of the scope of the assignment, but to make these calculations possible, the database should allow retrieving:

- The last data point for a service on a day,
- Data points for a service on a day in a time interval, and
- For a given data point find: `time, distance, latitude`, and `station_name` of the closest (regarding latitiude) stations in the north and direction. Call these stations `north_neighbour` and `south_neighbour`.

Note: The neighbouring stations strongly depend on data points. Data points are generated by an iPhone for a service during its travel. Data points are not known in advance, accordingly neighbouring stations can't be stored in the database in advance. Neighbouring stations have to be computed upon receiving a data point for a service. So, you need to define an appropriate query that will take some data from a data point and use a table containing station data for the same service.

3. Consistency Requirements

The product team has agreed to the following consistency requirements:

1. Reading driver and vehicle data must be strongly consistent.
2. Reading Data Point and other data may be eventually consistent.

4. Infrastructure Requirements

The deployment of the application and the test database should involve:

1. One cluster with a total of 6 physical nodes.
2. Using the Cassandra 3.10 release.

5. Availability Requirements

The infrastructure team has agreed to the following Availability requirements:

- The keyspace must provide Strong Consistency for 100% of the data when one node is down.
- Assume, each physical node has just 1 virtual node.

Note: The whole text above is completely fictional. It is made for the purpose of a students' exercise, only.

Assignment Questions

Question 1. [10 marks] List the database write and update requests the application requires using plain English.

Question 2. [12 marks] List the read requests the application requires using plain English.

Question 3. [9 marks] Consider Cassandra data model design guidelines we discussed in lectures and list names of database tables the application requires using plain English. Recall, Cassandra tables strongly depend on requested queries. If there is no queries needing a table, the table should not exist. (Don't invent queries to justify the existence of any tables.) After each table name, list those queries you identified in your answer to question 2 that use the table.

Question 4. **[20 marks]** Create data model using CQL 3 statements that support the requirements. To answer questions, use Cassandra CCM. In your answers, copy your CCM and CQL commands.

a. **[5 marks]** Create a cluster and a keyspace that will satisfy infrastructure and availability requirements above.

b. **[15 marks]** Define tables listed in your answer to question 3 above. For the table definitions include any non default property settings. Optimize your database solution just for iPhone application queries you identified in question 2 above.

Question 5. **[20 marks]** Provide CQL3 statements to support each of the application write and update requests you specified in Question 1 above. Show the consistency level before each write and update statement.

Question 6. [29 marks] Provide CQL3 statements to support each of the application read requests you specified in Question 2 above. Show the consistency level before each read statement. In your answer copy your CQL statement and the result produced by Cassandra from the screen.

References

* CQL3 formal language definition
`https://github.com/apache/cassandra/blob/cassandra-2.0/doc/cql3/CQL.textile`

* CQL3 language reference from Data Stax
`http://www.datastax.com/documentation/cql/3.1/cql/cql_intro_c.html`

* CCM (already installed on campus) `https://github.com/pcmanus/ccm`

Sample Data

The following sample data can be used for testing:

drivers:

drv_name	cur_pos	mobile	pwd	skill
milan	Upper Hutt	211111	mm77	{Matangi}
pavle	Upper Hutt	213344	pm33	{Ganz Mavag, Guliver}
pondy	Wellington	216677	pondy	{Matangi, Kiwi Rail}
fred	Taita	210031	f5566f	{Gulliver, Ganz Mavag}
jane	Waikanae	213141	jjjj	{Matangi}

vehicles:

vehicle_id	status	type
FA1122	Upper Hutt	Matangi
FP8899	maintenance	Ganz Mavag
FA4864	Wellington	Matangi
KW3300	Wellington	KiwiRail

time table

Hutt Vale Line (north bound)

station_name	service_no						distance
	1	3	5	7	9	11	[km]
Wellington	0605					1935	0
Petone	0617					1947	8.3
Woburn						1950	11.0
Waterloo	0625					1955	13.3
Naenae						2001	16.9
Taita	0634					2010	19.0
Silverstream	0642					2019	26.5
Upper Hutt	0650					2025	34.3

Hutt Vale Line (south bound)

station_name	service_no						Distance [km]
	2	4	6	8	10	12	
Upper Hutt	0700					1900	0
Silverstream	0708					1907	7.8
Taita	0716					1918	15.3
Naenae						1927	17.4
Waterloo						2028	21.0
Woburn	0725					2030	23.3
Petone						2035	26.0
Wellington	0745					2050	34.3

Waikanae Line (north bound)

station_name	service_no						distance [km]
	1	3	5	7	9	11	
Wellington			1025				0
Paekakariki			1059				33.1
Paraparaumu			1118				51.3
Waikanae			1139				62.8

Note: In tables above, time is represented as integer. So, you may interpret say 1025 as 10:25.

station_name	longitude	latitude
Wellington	174.7762	-41.2865
Petone	174.8851	-41.227
Waterloo	174.9081	-41.2092
Taita	174.9608	-41.1798
Upper Hutt	175.0708	-41.1244
Paekakariki	174.951	-40.9881
Paraparaumu	175.0084	-40.9142
Waikanae	175.0668	-40.8755

Data Points:

sequence	latitude	longitude	speed [km/h]
2017-03-22 10:37:50+1300	174.77	-41.2262	29.1
2017-03-26 10:07:40+1300	175	-41.2012	70.1
2017-03-26 10:02:10+1300	175.07	-41.1255	40.5
2017-03-26 10:49:40+1300	174.8	-41.968	30.8
2017-03-26 10:48:40+1300	176.06	-41.3	38
2017-03-26 10:48:10+1300	175.89	-41.523	67.6
2017-03-26 10:47:40+1300	175.44	-40.081	54
2017-03-26 10:47:10+1300	174.8	-40.478	36

What to hand in:

- All answers both electronically and as a hard copy.
- A statement of any assumptions you have made.
- A file under the name `submit_file_17.cql` that contains all of your table and index declarations along with 3 `insert` statements per table and 3 `update` statements per each column you were expected to update. Note, Pavle is going to run the file using `source cqlsh` command.
- A `.cql` (e.g. `q6a.cql`) file for each valid `select` statement you issued against the database as an answer. Again, note: Pavle is going to run your `select` statements against your database using `source cqlsh` command.
- Please do not submit any .odt, .zip, or similar files. Also, do not submit your files in toll directory trees. All files in the same directory is just fine.

Using Cassandra ccm on a Workstation

`ccm` stands for Cassandra Cluster Manager. This is a tool that creates Cassandra clusters on a local server and thus it simulates a Cassandra network.

At the command line you need to type:

```
[~] % need ccm
```

to set up the environment. You may want to insert `need ccm` into your `.cshrc` file and thus to avoid typing it repetitively whenever you log on.

The `ccm` tool supports a great number of commands. In the Assignment 1, you will need only a few of them. To see the available `ccm` commands, type

```
% ccm
```

Many `ccm` commands have options. To see available options of a command, type

```
% ccm <command> -h
```

When running a `ccm` command, do not use a `-v` or `--cassandra-version` option. The proper version of Cassandra is already installed on our school network.

To create a Cassandra cluster, use `ccm create -n <no_of_nodes> <clster_name>`.

To see available clusters and which one is the current (designated by *), use `ccm list`.

To switch to another cluster, use `ccm switch <cluster_name>`.

To see the status of the current cluster, use `ccm status`.

To start the current cluster, use `ccm start`.

To stop the current cluster, use `ccm stop`.

To open a CQL session, use `ccm nodei cqlsh`.

To exit, from `cqlsh`, type `exit`.

Note: `ccm` commands will not work on any `netbsd` computers but that should not be a problem as almost all computers that students have access to nowadays are `Linux` boxes.

Warning:

- In all deployments the same ports are assigned to server nodes. After finishing a session you have to do **ccm stop** to stop all servers of your deployment and release ports for other uses. Failing to do so, you will make trouble to other people (potentially including yourself) wanting to use the same workstation. Later, if you want to use the same deployment again, you just do `ccm start` and your deployment will resume functioning reliably.

- **You are strongly advised to use Cassandra from school lab workstations.** The school does not undertake any guarantees for using Cassandra from school servers. You may install and use Cassandra on your laptop, but the school does not undertake any responsibilities for the results you obtain.