Nikhil Mungel  
Vaishak Suresh (008646222)  
Junaid Abbasi (008154692)

Relational and ORM analysis

# Design Approach

## Technology stack used

* Python
* Django
* PostgreSQL during evaluation and MySQL during final implementation
* Psycopg2 and MysqlDB libraries for python

## De-Normalized schema

The initial table design was to not normalize the data. The entire observation data was inserted into one table.

### Advantages

* Inserting data is relatively easy
* All data, except station name is available in a single table.
* Queries are simple and usually no join is involved

### Disadvantages

* Handling duplicates is difficult. Same observations repeat because of the rolling window. This means batch inserts are not easy.
* Even ignoring already existing data does not help in batch inserts.
* The table is not normalized hence there is redundancy of data. The co-ordinates of the station is part of each observation. This repeats and takes up significant memory over time.

## Normalized schema

The next approach was to normalize the data.

### Advantages

* Data is logically split into tables
* Overhead of saving redundant data (station location) is avoided
* Not all tables will grow or get populated equally, this gives us a chance to partition or move the data to a separate disk with higher performance.

### Disadvantages

* More tables = more joins
* Inserts and updates are harder. There are not many updates, but inserting data requires checking if the data is already present.
* Some data like location do not naturally have primary key, so we had to use a surrogate key (Auto Incremented Integer)

The data was split into 3 tables. Below are the tables and the reasoning behind the table’s existence.

### Station Table

|  |
| --- |
| **Station** |
| Station\_id |
| Station\_name |

This table contains the station ID and the name of the station. Data is inserted into the table in the following two scenarios

* When the station data is received (2 hour interval)  
  In this process, both station\_id and station\_name are available. This is inserted and duplicates are updated.
* When the observation is received.  
  The observation data only has station\_id. We first check if the station is already in the station table. Inserts happen only if they do not exist. The comparison is done in the application by first loading all the existing station data and then inserts in batch.

### Location table

|  |
| --- |
| **Location** |
| Location\_id (Auto Increment) |
| Latitude |
| Longitude |
| Elevation |
| Station\_id (Foreign Key) |

This table holds all the location information for all the stations.

* A station can have multiple locations. Buoys have a fuzzy location, storing that in the station table means we’ll lose precision for the observations.
* Locations repeat in the observation, so maintaining a separate table with an id associated with the location helps reduce the amount of data in the observation table.
* This table makes it easier to query all the locations a station was at.
* It is also easier to get a range of locations based on the co-ordinates.

### Observation Table

|  |
| --- |
| **Observation** |
| Station\_id (FK)(pk) |
| Observation\_time (pk) |
| Temperature |
| Sknt |
| Wind\_direction |
| Gust |
| Pmsl |
| Altimeter |
| Dew\_point |
| Relative\_humidity |
| Weather |
| P24i |
| Location\_id |

The observation table is the table that grows with time.

* Each new observation received in the 15 minute window is inserted into this table.
* When an observation is received, the following process happens
  + Extract the station IDs and check if they are present in the station table. Insert new ones with blank station\_name for now.
  + Extract location name, check if the combination of lat, long and elev exist in the location table, if it does, get the location\_id, else insert and generate location\_id
  + Finally, with the location ID, insert into the observation table.

## Insert Strategy Evaluation

Data flows into the system at an interval of 15 minutes for the observation and 2 hours for the station information. This means that there is a 15 minute window for the inserts to happen. The inserts need not be as tuned as selects have to be.

Some of the insert strategies we examined are as below:

* Insert everything into the tables, ignore the integrity constraints.
  + This approach works well if the data was clean all the time and had no updates or missing values.
  + This also would mean that batch inserts would not happen.
* Insert only new values
  + With this approach, every insert would require atleast one complete table select and parse. This is inefficient.
  + Comparing each new row with each existing row is not efficient.
  + However, after finding out the new rows, it is easier to batch them and insert.
* Insert into a temporary table to clean data first
  + New observations can be inserted into a temporary table
  + A background process then scans the table and does a table to table copy of the new rows.
  + The actual application only uses the clean table.
  + The advantage of this approach is that the application always has the correct data.
  + The disadvantage is that it may be outdated at times by a 15 minute window

## Batch Insert VS Single Inserts

We first started out with PostgreSQL which has a method *copy\_from*() that takes a CSV file as input and dumps the data into tables. Below are the average performance difference between individual inserts and batch inserts.

Data Size: ~40000 records  
Iteration: 100  
Process: Clean table before each insert

|  |  |
| --- | --- |
| Single Inserts | ~ 4 minutes = 6ms per record |
| Batch Insert | 0.68 sec = 0.02ms per record |

There is a ~99% decrease using this strategy.

Inserts for MySQL took the following time.

|  |  |
| --- | --- |
| Single Inserts | 63.6 sec = 4 ms per record |
| Batch Inserts | 25.6 sec = 0.1 ms per record |

The below table indicates the comparison between PostgeSQL and MySQL

One conclusion to draw from this is that the copy\_from() method in PostgreSQL is more efficient than batching SQL queries. In general though, inserting in batch is much more efficient than single inserts.

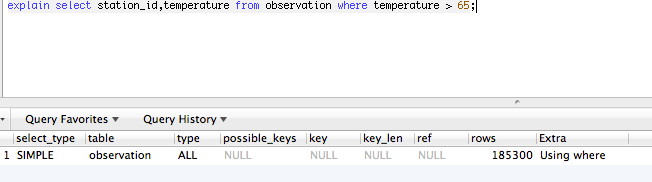
## Select query efficiency

Below are the queries we measured performance for.

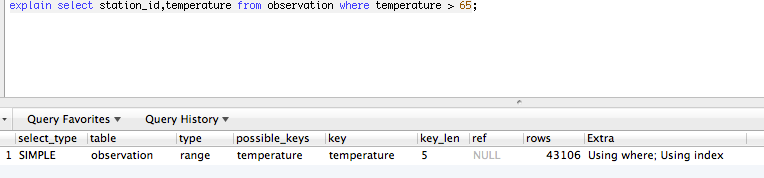
1. Select all observations for a temperature (0.19 vs 0.15)

**Query***: select station\_id, temperature from observation where temperature > 65;*

Without index on temperature: 175 ms



With index on temperature: 140 ms

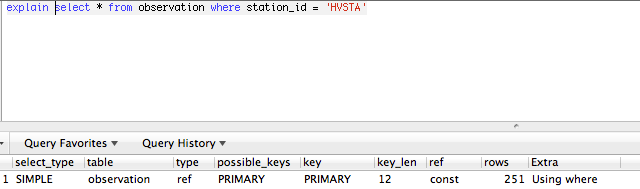


1. Select all observations for a station.

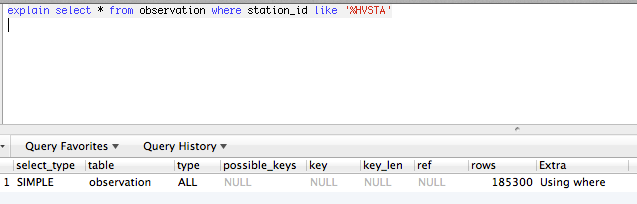
**Query:** *select \* from observation where station\_id = 'HVSTA'*

The observation are stored in the table with the station\_id as a part of the composite key, hence the query makes use of the implicit index.

Time taken: 1.9 ms



This is optimal if we know the station\_id. For any string comparison, the index isn’t used.



Time taken for a like query: 130.8 ms

1. Select all observations for a time

**Query:** *select \* from observation where observation\_time > '2013-09-01 00:15:00' and observation\_time < '2013-09-03 21:02:00'*

Range queries are expensive since each row has to be examined.

Time Taken: 1.134 or 1134 ms

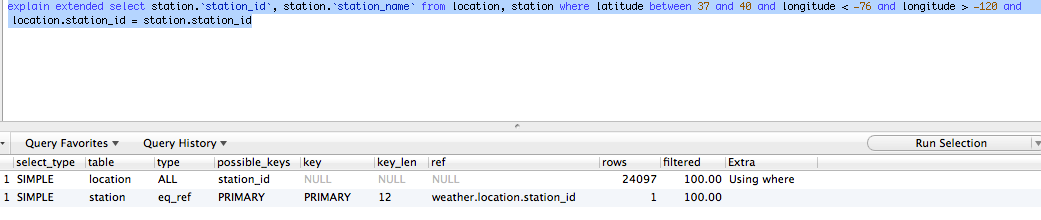
Since \* is used, none of the indexes are used. The same query, with the difference of selected column being ‘station\_id’ returns results in 137ms

1. Select all stations in an area

**Query:** select station.`station\_id`, station.`station\_name` from location, station where latitude between 37 and 40 and longitude < -76 and longitude > -120 and location.station\_id = station.station\_id

Time Taken: 17 ms

The query joins two tables based on the key, station\_id hence the selects are fast.



## Conclusion

Following are the observations about the non-ORM solution.

* Moving location and station information into two separate tables makes tables compact.
* Because of this normalization, range queries like all locations, all stations etc. require joins.
* PostgreSQL’s *Copy* method is very efficient for bulk inserts. MySQL requires batching queries in the cursor and then committing together, this isn’t very intuitive.
* MySQL has ‘*INSERT IGNORE*’ and ‘*INSERT ON DUPLICATE KEY … UPDATE*’methods that help avoid the integrity constraint problems while inserting in bulk.
* Indexing on columns that are frequently queried results in significant performance increase.
* Text comparison is VERY costly. Avoid using ‘like’ in select queries to large tables.
* Partitioning growing tables horizontally can be useful if the database is going to be split across machines.

Example: We decided to split the observation based on year like indicated below:

## 

The table to query can then be targeted to the specific db/table in the application. A background job/trigger takes care of partitioning the table and cleaning up the primary table.

* In case the partitioned table is split across multiple databases, the location and station tables will have to be replicated to avoid queries to two different machines.