

**CmpE226 – Database Design**

**Project 1: Database Schema Design and Comparison of JDBC & ORM approaches for Large Datasets.**

**Submitted To**

Prof. John Gash

**Date of Submission**

Oct 20, 2013

**Submitted By**

Krishna Nambiar [008639605]

Rahul Mehta

Gayathiri Krishnan [007075601]

Manish Belsare [008297146]

Contents

Contents 2

1. Introduction 4

2. Data Source and Analysis of Data 4

2.1 Observations about downloaded data 4

3. Our Goal 5

4. Implementation Approaches 5

4.1 Approach 1 (Status – Rejected) 5

4.1.1 Rationale 5

4.1.2 Execution 5

4.1.3 Result 6

4.2 Approach 2 (Status – Implemented) 6

4.2.1 Assumptions 6

4.2.2.Execution 6

4.2.3 Results 6

5. Execution 7

5.1 System Specification 7

5.2 Database Design 7

5.2.1 Partitioning 10

5.2.2 Data Loading 12

5.2.3 JDBC vs. Hibernate 13

5.2.4 Test Cases 16

6. Testing 19

6.1 Benchmark Results 19

6.3.1 Creation of new indexes 21

6.3.2 Creation of foreign keys 22

6.3.3 Query Optimization 22

7. Lessons Learned 23

8. Current Limitations and Future Plans 24

9. How to load and execute the demo 24

10. Individual Contribution 25

11. Bibliography 25

1. Introduction

Handling large sets of data is a very big challenge for companies since while designing a database schema it is important to understand the effects of the design approach on Schema and applicability of Object Relational Mapping tools. One good approach to study this is to work with data that comes naturally from variety of sources and is live i.e data is constantly changing every few hours or minutes. Such kind of data has hidden patterns that give a wide understanding of effects on the storage, performance and scalability of the database those are occurring constantly. These effects can be studied effectively by collecting real time data and then categorizing it into stages for understanding the performance overload, response time for a large dataset and optimization of queries and stored procedures for better performance.

# 2. Data Source and Analysis of Data

We are using the weather data provided by MesoWest for this project. MesoWest was developed to supplement the Automated Surface Observation System (ASOS), which includes the weather stations maintained by the National Weather Service (NWS), Federal Aviation Administration (FAA) and the Department of Defense. MesoWest was developed through a collaborative effort between University of Utah and the forecasters at the Salt Lake City WFO. Currently MesoWest collects information from 47 public and 23 commercial sources (Horel, Splitt, & Dunn, 2002). Due to the multitude of sources from which MesoWest gathers data, it can provide an enhanced spatial coverage with an average of 9.3 miles between stations. The stations are identified using the NWS assigned identifier, if it is assigned; otherwise a unique 3-5 character alphanumeric identifier is generated. As part of the MesoWest processing cycle, the data is disseminated through various methods every 15 minutes.

## 2.1 Observations about downloaded data

1. Frequency: 15 minutes
2. The observation file contains overlapping time periods or a sliding window per station.
3. The combination of Station id, date and time (S-D-T) can be used as the primary identifier.
4. S-D-T entries may repeat within a file or in future files. An S-D-T entry in a newer file may contain updated recordings for that particular S-D-T combination.
5. Some stations report observations with a delay. For example, a file that came in at 2013/09/10 01:00 may contain data from some stations for 2013/09/09 22:00 or a file that comes in towards midnight may contain observations from some stations with the next days date.
6. Some files may be empty, and a few files contain corrupted format for certain fields. A value of (-9999) is used to represent null readings of attributes.
7. The station information file may be considered a slowly changing dimension, in that most of the files contain the same information, with a few having new stations, or having updated statuses /locations of stations.
8. For the station information file, the station id (primary\_id column in the file) can be taken as the unique identifier, because it is maintained as such, as per the MesoWest system documentation.

# 3. Our Goal

We are planning to use this data and build a database for weather analytics executing climatology queries with support for GIS and spatial querying. Ideally our database design should allow for efficient querying of a large dataset, and storage of data covering a large time period – spanning multiple years. It should also be designed for easy archival over a designated time window. We would also be giving a higher weightage for select queries over DML queries. The reason being that inserts/updates/deletes happen only in 15-minute intervals (currently) while the select queries would be more frequent and running over a larger dataset.

# 4. Implementation Approaches

We decided to take up a bottom-up approach by designing the database schema first and then building the application layer on top of it. The rationale behind this approach was that we could decouple the database design from the application, thereby allowing us to use any language or paradigm to build up the front end without affecting the database design, if the need arises to upgrade the front end on the longer run since we were designing the schema for analytics, long term storage and archival. We also decided to do a lot of heavy lifting at the database end, by using stored procedures to move data around in tables. In this way, the performance differences would come only from the way different languages or frameworks handled the select queries.

We also looked at various existing weather data systems, like ClimaDB (Jr & Dascalu, 2011), research papers on weather data warehousing (Joga, 2000), and MesoWest architecture, to come up with our final approach.

## 4.1 Approach 1 (Status – Rejected)

Partitioning the data based on station names. One table per station.

### 4.1.1 Rationale

We assumed that this approach would work since the granularity of the most of the queries would be based on a station or a list of stations. Therefore we could directly hit the station tables for the required stations and fetch all required information.

### 4.1.2 Execution

A master observation table was created from which all the station tables inherited. Before-Insert triggers were put in the master observation table to create station table, if it did not exist, and redirect the insert into the appropriate table.

Data was loaded in an unlogged staging table, which did not have any constraints or indexes. Using writable CTEs, the data was cleansed – duplicates deleted, observation values updated – and then inserted into the master observation table. The triggers on the master observation table redirected the inserts into the correct station table.

### 4.1.3 Result

We found that this approach was not very efficient. The partitioning of tables according to station names created more than 20,000 tables. Even though the tables were comparatively small in size, the main bottleneck was with the query planner. Due to the large number of tables the pg\_catalogs and pg\_inheritance table grew enormously. This lead to an unacceptable bottleneck in the query planning stages since Postgres had to go through the entire catalog table to get the statistics, and the inheritance tree to find out the appropriate tables to hit based on the constraint exclusions. Simple select queries were consistently taking more than 2 minutes to complete. If constraint exclusion was disabled and then a query on the master table was issued, then it would have to scan through all of the 20,000+ inherited tables. Therefore this approach was dropped.

## 4.2 Approach 2 (Status – Implemented)

Columnar tables for each of the weather attribute (Temperature, Wind, Humidity, Pressure and Precipitation) which is further partitioned based on a time window (week). GIS and spatial querying enabled through PostGIS extension.

### 4.2.1 Assumptions

Since we are targeting the weather analytics domain, creating thin fact tables for each weather attribute should help us run analytical queries faster because now more pages can be loaded onto the memory. Further, we decided to split every week’s data into a new table. The time window of 7 days was chosen after analyzing the average number of observations coming in. A key table and 5 attribute tables would mean that we would have 312 fact tables per year. As per Postgres documentation, the database engine can easily handle up to a few thousand inherited tables easily. This would mean that we could run gather data in the active database for up to 5 years before starting the archival process.

Even for archival, since each partition is maintained as a separate table, it can be easily moved / dropped, without affecting the gathered statistics.

### 4.2.2.Execution

PostGIS was installed and a new database created using the PostGIS template. Master tables were created for the Key and each of the attributes. The weekly partitioned tables inherited from these master tables. Constraint based on date and unique ID was put on the Key table and unique id constraint on the attribute tables. Geometry column was put in the station table to run spatial queries. Stored procedures were used to load data from the staging table into the fact tables. A detailed explanation of the process used is given in [Section 4.](#_5._Execution)

### 4.2.3 Results

A week of data averages around 12.5 million unique observations with an average table size of 400 MB. The Key table averaged around 700MB and the observation tables averaged 300 MB. Almost all out select and insert queries clocked under a minute.

# 5. Execution

## 5.1 System Specification

Operating System : Mac OS X Mountain Lion, 64bit

RAM : 8GB DDR3

Processor : 2.4 Ghz Intel Core i5

HDD : Toshiba 500 GB, 5200RPM

Postgres : v9.3

PostGIS : v2.1

**Initial DB Tuning**

Postgres was upgraded to v9.3 because it uses a much smaller memory footprint and does not require any changes to the kernel level memory parameters (shmax, shmin etc).

shared\_buffers : 1GB

work\_mem : 8MB

temp\_buffers : 8 MB

checkpoint\_segments : 10MB – because default value was making checkpoints in less than 15sec during bulk insert operations

constraint\_exclusion : partition

Logging was reduced to a minimum for the final tests.

## 5.2 Database Design

Figure 2 shows the hig-level design for the master fact tables and the dimension tables. The “location” dimension (“\_dim\_location”) stores the station information. The “geom” column in that table stores the geometry column generated from the longitude and latitude of the station using:

SET geom = ST\_transform(ST\_PointFromText('POINT(' || longitude || ' ' || latitude || ')',4269),32661) ;

Latitude and Longitude values use a Spherical Reference ID (SRID) of 4269, which is for a point in a flat surface. We convert that to an SRID of 32661 for calculating surface distances using a spherical reference.

The tables \_*dim*\_date and \_*dim*\_time hold the date and time dimensions. More attributes may be added to these dimensions based on user requirements. Currently it allows us to generate queries based on conditions like *where time=”Morning” or date in “Quarter 1”* etc.

The \_*dim*\_key table holds the keys or the lookup information for the observations in the normalized fact tables – humidity, precip, pressure, wind and temperature. The relationships between these tables are shown in Figure 2. Please note that dim\_key, \_humidity, \_temp, \_wind, \_pressure, and \_precip are master tables and therefore do not have any data in them. Then actual date is stored in the tables that inherit these tables. Further, **no physical foreign key constrains** are maintained at the database level. Referential integrity is maintained through the code in stored procedures used to populate these tables.

The uniqueid is of bingint datatype, which allows us to have approximately 9.2e18 unique values. This would mean that we could store values for more than 14billion years (assuming 12million records per week) before unique ids run out.

All other observations are stored in “real” / “double precision” datatypes. And the stn\_id is stored as a string.

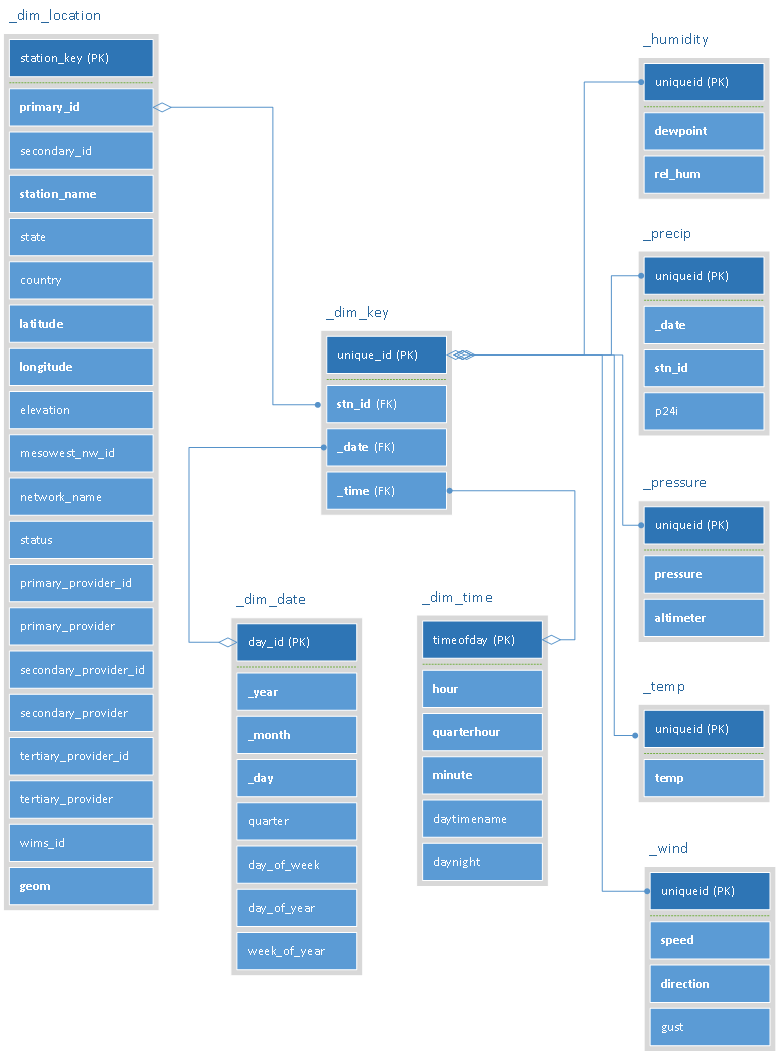


Figure :DB schema overview

### 5.2.1 Partitioning

We decided to partition the observation data on a weekly basis. A static partitioning is used, in the sense that, the current weeks data is always in the table named <xx>\_curr (xx could be key, temp, precip, pressure, wind or humidity) (Figure 2). Every Sunday (since the week runs from Monday to Sunday) after the 2245hrs file is loaded, the data is moved into a table called <xx>\_<year>\_<week> and a new xx\_curr table set is created for the next week. Constrain exclusion in enabled by bounding the date range on the key table and by bounding the uniqueid column on the observation tables. Therefore a set of xx\_year\_week tables will have the same date and unique id range.

Advantages of this approach:

1. No costly triggers are required on the master tables to move data into the correct tables on insert.
2. Simplifies our load procedures since the data is always inserted only in xx\_curr tables.
3. We can maintain the xx\_curr tables optimized for inserts – no foreign key constraints, only the minimum required indexes.
4. The xx\_year\_week tables can be optimized for selects - more indexes, if required. A vacuum freeze can be run on these tables as they would not be modified any further.
5. Analyze needs to be run only once on the xx\_week\_year tables since they have static data.
6. Any write locks or slightly slower reads will happen only on the xx\_curr tables.
7. The xx\_year\_week tables can be moved out very easily for archival.
8. Users can easily identify the table they need to query based on the week number for which they need data.

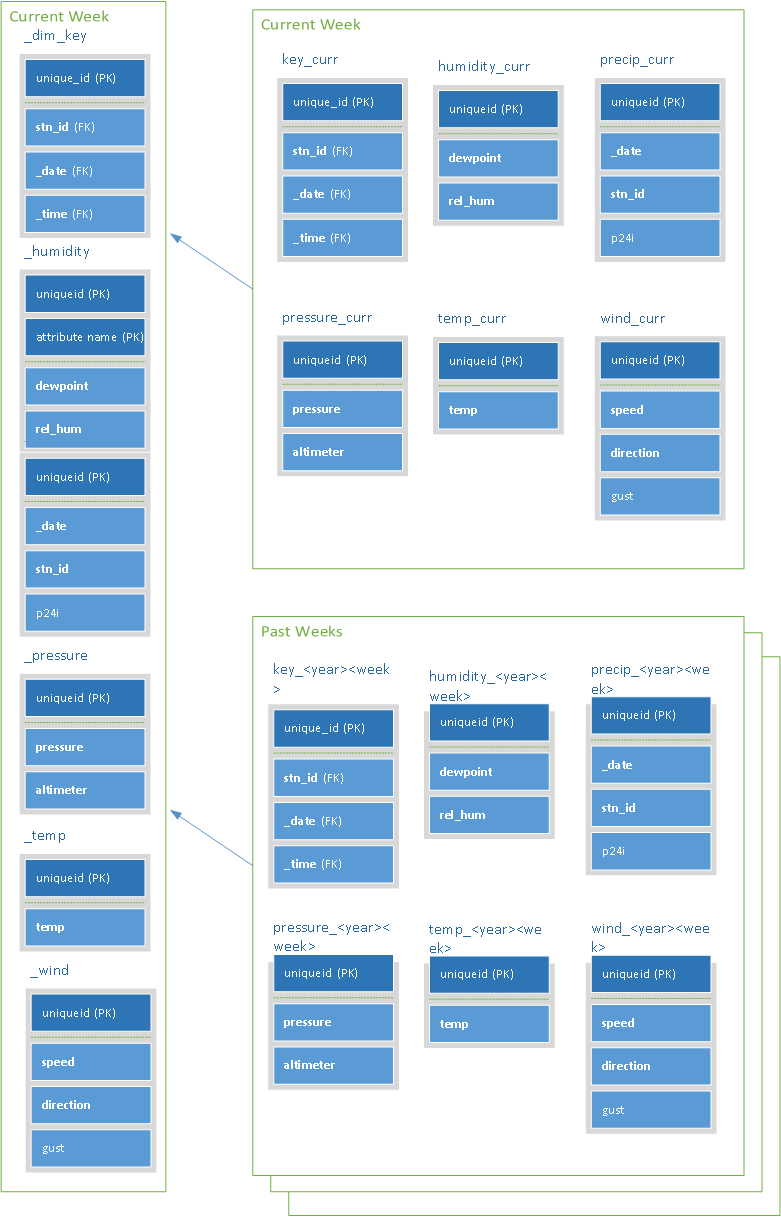


Figure : Partitioning in Postgres

### 5.2.2 Data Loading

We decided to do the majority of the loading efforts via stored procedures. The front-end application just dumps the data into the staging table and calls the stored procedures to load the table. The java program did minimal data cleansing like replacing -9999 with NULL value, transforming dates to SQL format etc. This process would enable us to easily move the loading application from one language to another, since only the CSV read and insert into staging has to be done. The ideal batch size was found to be 500 for both JDBC and Hibernate applications. (batch sizes of 1, 100, 500 and 1000 were tested).

The process flow chart is shown in figure 4.

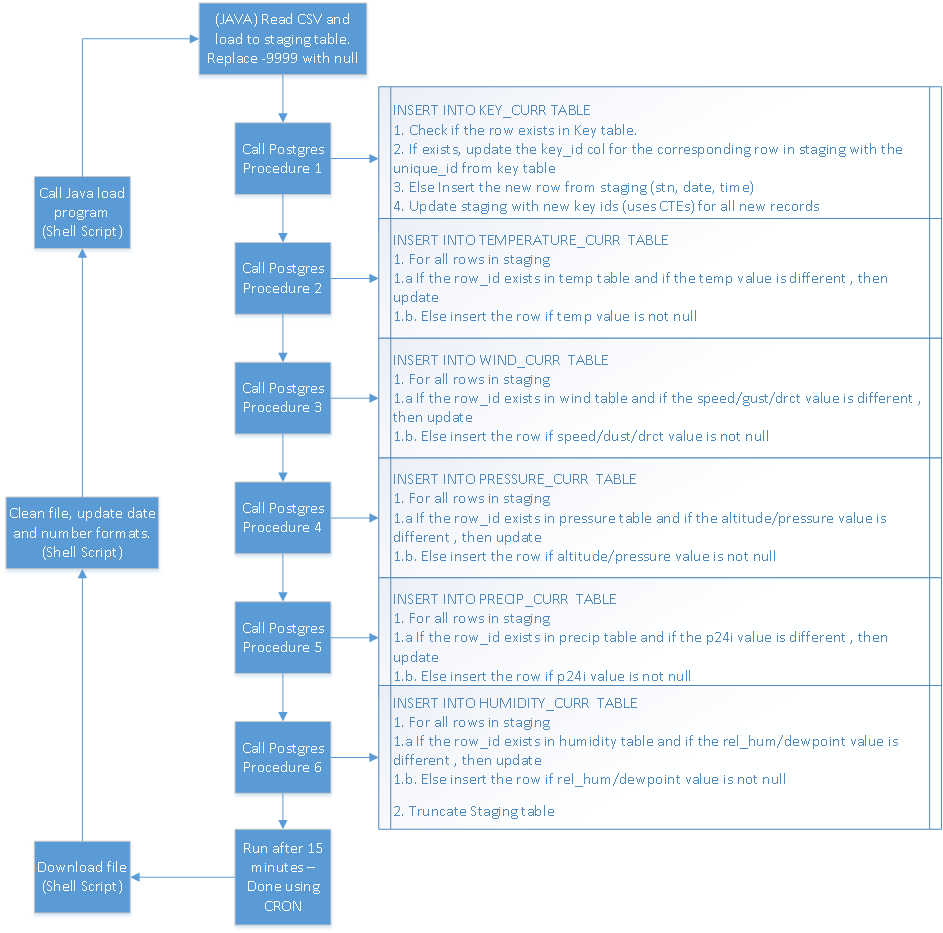


Figure : Flowchart for data loading

Shell script running on CRON was used to download the data. This script did the initial cleanse, by deleting the first couple of blank lines, changing the date format from YYYYMMDD/HHMM into two columns as yyyy-DD-mm, HH:mm:ss.

When moving data from the staging to the key table, logic for UPSERT (update or insert) by comparing the S-D-T combination in the key table and assigning the corresponding key\_id back to the staging table record was used to avoid having duplicate entries. This key\_id was used to insert data into the observation tables. Records were inserted in the observation tables only if it was not null and the new value was different from the one already present if the key\_id already existed.

(Note: The SQL stored procedures are included with the submission)

The use of stored procedures to move data among tables, ensured that the average load time for both JDBC and Hibernate approaches was maintained around 40 seconds.

In order to increase the load performance, no indices other than the primary key was maintained in any of the observation xx\_curr table. For the key table, it was found that having an index on the station+date+time combination apart from the primary id on uniqueid reduced the load time for that table from the 200sec range to 15-30 sec range for the same data files. Even though this may be thought of as counter intuitive, the new index helps the process because a lot of selects are run against the S-D-T combination inside the stored procedures.

Figure : File load times for JDBC & Hibernate

For prior week tables (xx\_year\_week), multiple indexes were built based on test query plans and statistics. The creation of selected indexes was automated into the weekly script that generates these tables. This ensured that the older tables provided good performance on select and at the same time does not affect inserts.

### 5.2.3 JDBC vs. Hibernate

Since we are following the bottom-up approach, no schema changes were made for JDBC or Hibernate implementations. Batch loading was used in both applications with a batch size of 500.

In-class annotations were used to describe the relations for Hibernate. One to one and one to many relations were created between Location, Key and observation tables, as shown in Figure 5. The relation between Location and Key was based on station names (location.primary\_id=key.stn\_id) and that from Key to observations were based on the uniqueid value (key.unique\_id=<observation>.uniqueid).

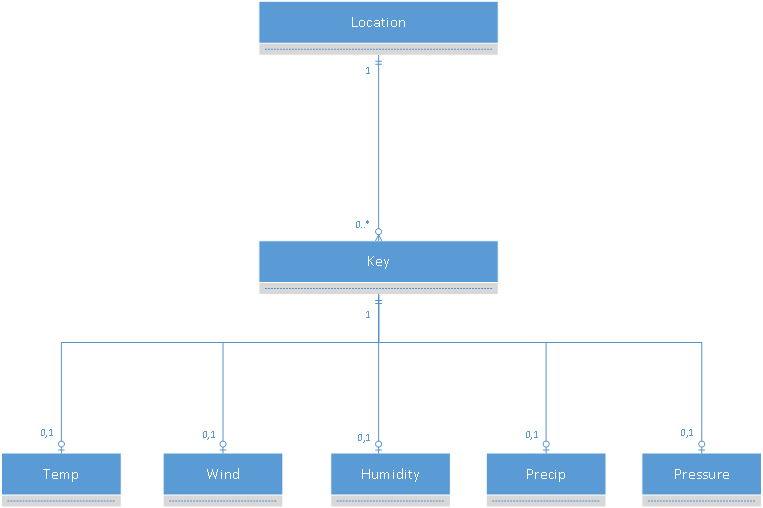


Figure : Hibernate class diagram in Crow’s feet notation.

@Entity

@Table(name = "\_dim\_key")

public class DimKey implements Serializable{

@Id

@Column(name = "unique\_id")

private long uniqueid;

@Column(name = "stn\_id")

private String stn\_id;

@Column(name = "\_date")

private Date date;

@Column(name = "\_time")

private Time time;

@OneToOne(fetch=FetchType.LAZY)

@PrimaryKeyJoinColumn

private Temperature temperature;

@OneToOne(fetch=FetchType.LAZY)

@PrimaryKeyJoinColumn

private Humidity humidity;

@OneToOne(fetch=FetchType.LAZY)

@PrimaryKeyJoinColumn

private Precipitation precipitation;

@OneToOne(fetch=FetchType.LAZY)

@PrimaryKeyJoinColumn

private Pressure pressure;

@OneToOne(fetch=FetchType.LAZY)

@PrimaryKeyJoinColumn

private Wind wind;

@ManyToOne(fetch=FetchType.LAZY)

@JoinColumn (name="\_time" ,referencedColumnName="timeOfday" , insertable=false,updatable=false)

private TimeDimension timeDim;

@ManyToOne(fetch=FetchType.LAZY)

@JoinColumn (name="\_date" ,referencedColumnName="day\_id" , insertable=false,updatable=false)

private DateDimension dateDim;

@ManyToOne(fetch=FetchType.LAZY)

@JoinColumn (name="stn\_id" ,referencedColumnName="primary\_id" , insertable=false,updatable=false)

private Location location;

Code Segment : Key class snippet

@Entity

@Table(name = "humidity\_curr")

public class Humidity implements Serializable{

@Id

@GeneratedValue

@Column(name = "uniqueid")

private long uniqueid;

@Column(name = "dewpoint")

private Float dewpoint;

@Column(name = "rel\_hum")

private Float rel\_hum;

Code Segment : Observation class snippet

@Entity

@Table(name = "\_dim\_location")

public class Location implements Serializable{

@GeneratedValue

@Column(name = "station\_key")

private long station\_key;

…. Other variable declarations …

@OneToMany(fetch=FetchType.LAZY)

@JoinColumn (name="stn\_id",referencedColumnName="primary\_id",insertable=false,updatable=false)

private List<DimKey> key;

Code Segment : Location class snippet

### 

### 5.2.4 Test Cases

The following test case queries were run against JDBC and Hibernate applications. All queries were done on the master tables, to check if the correct child tables are hit according to constraint exclusion criterion.

#### Test Case 1: Min, Max and Avg temperature for all stations in a state for a day.

Select DL.station\_name, DK.\_date, max(TC.temp) as "Max Temp", min(TC.temp) as "Min Temp", avg(TC.temp) as "Avg Temp"

from \_temp TC, \_dim\_location DL, \_dim\_key DK

WHERE

DL.state='CA' and

DL.primary\_id=DK.stn\_id and

DK.unique\_id=TC.uniqueid and

DK.\_date = '2013-09-08'

Group by 1, 2

ORDER BY 1, 2

#### Test Case 2: Min, Max and Avg temperature for all stations in a ten mile radius from a latitude and longitude

SELECT

K.\_date,avg(T."temp"),min(T."temp"),max(T."temp")

FROM

\_dim\_key K, \_temp T,\_dim\_location L

WHERE

K.stn\_id=L.primary\_id AND

K.unique\_id=T.uniqueid AND

K.\_date between '2013-09-09'::date AND '2013-09-16'::date AND

L.geom && st\_expand(st\_transform(st\_PointFromText('POINT(-121.92417 37.35917)', 4269),32661), 16093) AND

st\_distance(st\_transform(st\_PointFromText('POINT(-121.92417 37.35917)', 4269),32661),L.geom) < 16093

GROUP BY 1

**Comments**

* This query was run only on JDBC, as Hibernate does not support Geometry types and operations.
* Using just st\_distance condition to calculate distances from our concerned zip (92230) to all other zipcodes is not an optimal solution. We would rather have a bounding box of 10 miles around our interested point, filtering out the points that interact with this box then calculate the exact distances to the center. This is done my adding && operator and st\_expand function. Use of the second function reduced the execution time from 3 minutes to less than 40 seconds.
* 10 miles = 16093 meters.
* ST\_Contains function can be used to find out all stations within an area bounded by a polygon.

#### Test Case 3: Find all stations in a state where it rained in the morning

SELECT L.station\_name, K.stn\_id, K.\_date, AVG(P.p24i)AS "Average Morning Rain"

FROM

\_dim\_key K, \_dim\_time T, \_precip P, \_dim\_location L

WHERE

T.daytimename='Morning' AND

K.\_time=T.timeofday AND

K.\_date between '2013-09-10'::date AND '2013-09-16'::date AND

K.stn\_id=L.primary\_id AND

P.uniqueid=K.unique\_id AND

L.state='CA' AND

P.p24i>0.1

GROUP BY 1, 2,3

#### Test Case 4: Find stations where there were gusts > 33 knots and the direction of wind at that speed.

SELECT d.stn\_id ,W.speed, W.gust, \_mesowest\_drct(W.direction)

FROM

\_dim\_key d, \_wind w

WHERE

d.unique\_id=w.uniqueid AND

w.gust>33 AND

d.\_date between '2013-09-09' ::date AND '2013-09-16'::date

#### Test Case 5: Find mean of 9AM temperature and relative humidity

SELECT K.stn\_id, L.station\_name, K.\_date, AVG(H.rel\_hum), AVG(T.temp)

FROM

\_temp T, \_dim\_key K, \_humidity H, \_dim\_location L

WHERE

T.uniqueid=K.unique\_id AND

H.uniqueid=K.unique\_id AND

L.primary\_id=K.stn\_id AND

K.\_time='09:00:00'::time without time zone AND

L.state='CA' AND

K.\_date between '2013-09-09' ::date AND '2013-09-16'::date

GROUP BY 1, 2,3

#### Test Case 6: Find all weekends when it rained

SELECT K.\_date, K.stn\_id

FROM

\_precip P, \_dim\_key K ,\_dim\_date D, \_dim\_time T

WHERE

P.uniqueid=K.unique\_id AND

K.\_date=D.day\_id AND

K.\_time=T.timeofday AND

T.daytimename='Morning' AND

D.day\_of\_week IN (0,6) AND

P.p24i>0.1

#### Test Case 7: Find average morning, afternoon, evening and night temperatures.

SELECT L.station\_name, K.stn\_id, K.\_date, TM.daytimename, AVG(T.temp)AS "Average Temp"

FROM

\_dim\_key K, \_dim\_time TM,\_dim\_location L, \_temp T

WHERE

K.\_date between '2013-09-10'::date AND '2013-09-16'::date AND

TM.timeofday=K.\_time AND

K.stn\_id=L.primary\_id AND

T.uniqueid=K.unique\_id AND

L.state='CA'

GROUP BY 1, 2,3,4

#### Test Case 8: Find average morning, afternoon, evening and night Wind Direction.

SELECT

L.station\_name, K.\_date, T.daytimename,

round(SQRT(POWER(avg(W.speed\*sin(RADIANS(W.direction))),2)+ POWER(avg(W.speed\*cos(RADIANS(W.direction))),2))::numeric,2)

as "Avg Speed", \_mesowest\_drct(round((DEGREES(ATAN(avg(W.speed\*sin(RADIANS(W.direction))) / avg(W.speed\*(cos(RADIANS(W.direction))+0.0001))))+360)::numeric,2))

as "Avg Direction"

FROM

\_dim\_location L, \_dim\_key K, \_dim\_time T, \_wind W

WHERE

K.unique\_id=W.uniqueid AND

K.stn\_id=L.primary\_id AND

K.\_time=T.timeofday AND

L.state='CA' AND

W.direction>0 AND W.speed>0 AND

K.\_date between '2013-09-10'::date AND '2013-09-15'::date

GROUP BY L.station\_name, K.\_date, T.daytimename

ORDER BY L.station\_name, K.\_date

**Comments**

* Wind direction is given in degrees and therefore we cannot use the normal AVG function to find the average.
* Vector Averaging for Wind Speed and Direction:

From the sequence of N observations of Ai (instantaneous horizontal wind direction) and Ui, (instantaneous horizontal wind speed) the mean east-west, Ve, and north-south, Vn, components of the wind are (U.S. ENVIRONMENTAL PROTECTION AGENCY, 2000) ,

The resultant mean wind speed and direction are,

* \_mesowest\_drct() is a custom function that converts from direction in degrees to compass directions.

# 6. Testing

## 6.1 Benchmark Results

|  |  |  |
| --- | --- | --- |
| Test Case # | With 1 day’s data | With 1 week’s data (in sec) |
| 1 | 1.67 | 18.71 |
| 2 | .23 | 4.5 |
| 3 | 1.14 | 12 |
| 4 | .98 | 10.4 |
| 5 | .97 | 10 |
| 6 | .45 | 9.6 |
| 7 | 3.23 | 38 |
| 8 | 3.54 | 38.94 |

**6.2 Test Execution Results**

**Database Statistics before final test execution**

Total raw row records processed : 98,834,796

Total unique records inserted : 65,602,230

|  |  |  |  |
| --- | --- | --- | --- |
| Table Name | Total Rows | Table Size | Index Size |
| \_dim\_date | 4,019 | 208 KB | 104 KB |
| \_dim\_time | 1,440 | 120 KB | 75 KB |
| \_dim\_location | 40,360 | 18 MB | 11 MB |
|  |  |  |  |
| dim\_key\_curr | 12,351,400 | 778 MB | 1182 MB |
| dim\_key\_2013\_37 | 14,348,751 | 846 MB | 1517 MB |
| dim\_key\_2013\_38 | 13,729,888 | 788 MB | 1199 MB |
| dim\_key\_2013\_39 | 12,464,975 | 715 MB | 1017 MB |
| dim\_key\_2013\_40 | 12,707,216 | 729 MB | 1104 MB |
|  |  |  |  |
| \_wind\_curr | 9,559,971 | 464 MB | 348 MB |
| \_wind\_2013\_37 | 10,431,376 | 506 MB | 599 MB |
| \_wind\_2013\_38 | 9,709,454 | 471 MB | 351 MB |
| \_wind\_2013\_39 | 8,846,734 | 429 MB | 319 MB |
| \_wind\_2013\_40 | 8,986,082 | 436 MB | 322 MB |
|  |  |  |  |
| \_temp\_curr | 10,561,682 | 448 MB | 684 MB |
| \_temp\_2013\_37 | 11,273,505 | 490 MB | 741 MB |
| \_temp\_2013\_38 | 10,741,778 | 455 MB | 691 MB |
| \_temp\_2013\_39 | 9,776,799 | 415 MB | 630 MB |
| \_temp\_2013\_40 | 9,940,895 | 422 MB | 640 MB |
|  |  |  |  |
| \_precip\_curr | 4,927,905 | 245 MB | 107 MB |
| \_precip\_2013\_37 | 5,181,877 | 259 MB | 116 MB |
| \_precip\_2013\_38 | 4,962,459 | 247 MB | 108 MB |
| \_precip\_2013\_39 | 4,511,724 | 225 MB | 98 MB |
| \_precip\_2013\_40 | 4,612,270 | 230 MB | 100 MB |
|  |  |  |  |
| \_pressure\_curr | 7,013,671 | 296 MB | 270 MB |
| \_pressure\_2013\_37 | 7,624,119 | 322 MB | 290 MB |
| \_pressure\_2013\_38 | 7,063,562 | 298 MB | 270 MB |
| \_pressure\_2013\_39 | 6,420,412 | 271 MB | 245 MB |
| \_pressure\_2013\_40 | 6,533,220 | 276 MB | 248 MB |
|  |  |  |  |
| humidity\_curr | 9,111,920 | 386 MB | 333 MB |
| humidity\_2013\_37 | 9,966,199 | 422 MB | 785 MB |
| humidity\_2013\_38 | 9,262,435 | 392 MB | 332 MB |
| humidity\_2013\_39 | 8,432,532 | 357 MB | 304 MB |
| humidity\_2013\_40 | 8,573,211 | 363 MB | 309 MB |

Figure : Test Results before optimization

Observations after analyzing the query plan for the test cases were:

* + - 1. Indexes on the key table are hit properly.
      2. Selection of tables based on constraint exclusion was being done only for dim\_key table. The reason it is not applied for observation tables was that limits on the constraint column (uniqueid) is not specified.
      3. Sequential scans were being done on observation tables for most of the test scenarios. Even though it not inherently bad since the tables have a very small width, it could cause problem when larger numbers of tables have to be scanned.
      4. The performance of JDBC was almost on par with time taken by native PSQL runs, while hibernate almost always took more time.

**6.3 Optimizations**

Our target is to try and reduce the number of observation tables hit per query and attempt for index scans wherever possible.

### 6.3.1 Creation of new indexes

A new index on observation tables based on uniqueid+ the value column was tried. After running ANALYZE and rechecking the query plan, it was found that there was no change. The query was still executing sequential scans on observation tables.

### 6.3.2 Creation of foreign keys

Foreign keys were created on observation tables connecting its uniqueid column with the unique\_id column in the corresponding dim\_key table. . After running ANALYZE and rechecking the query plan, it was found that there was no change. The query planner was not utilizing any of the new foreign keys.

### 6.3.3 Query Optimization

A good way to limit the number of tables being scanned is to reduce the numbers of tables being used in the query. For this we have to force the planner to use constraint exclusions on the observation tables as well.

For this, instead of running the query in a single step, we could break it down into two steps. Step 1 would find the min and max unique ids from the dim\_key table for the scenario. Then we could plug in these values as well in the main query to constrain the number of observation tables being hit.

As an example, for test case 4, the query can be split as follows:

Query 1:

SELECT min(d.unique\_id), max(d.unique\_id)

FROM

\_dim\_key d

WHERE

d.\_date between '2013-09-09' ::date AND '2013-09-16'::date

Query 1 gives the result as 14755460 and 39020307.

Query 2:

SELECT

d.stn\_id ,W.speed, W.gust, \_mesowest\_drct(W.direction)

FROM

\_dim\_key d, \_wind w

WHERE

d.unique\_id=w.uniqueid AND w.gust>33 AND

d.\_date between '2013-09-09' ::date AND '2013-09-16'::date AND

w.uniqueid between 14755460 AND 39020307

In this manner, due to the explicit mention of the constraints for the wind table, constraint exclusion was used by planner to limit the number of tables being scanned.

Even after including the approx. 20 seconds for the first query to run, we had a drastic improvement in performance as shown in figure 6. These new queries completed in a fraction of time taken by the original queries.

In case of JDBC, we could even query specific tables that contain the data for the dates we need instead of querying on the master tables. But since this cannot be done with Hibernate, this approach was not used for testing.

Figure : Test Results after query optimization

# 7. Lessons Learned

1. Real life data sources need to be cleaned thoroughly before it can be used.
2. The design and performance of the schema depends a lot on how well we understand the data and the use case we are trying to solve.
3. A design that avoids or reduces the use of triggers and rules may provide better performance.
4. Having an index doesn’t always ensure better performance.
5. JDBC gives us a lot of flexibility in writing the queries, at the cost of increased lines of code and deep understanding of the underlying schema.
6. Hibernate requires much less lines of code, but reduces the flexibility in tuning the query to our needs.
7. Hibernate also demands a learning curve for the uninitiated.
8. The performance of the database depends a lot on its architecture and the working of its components. Care must be taken when designing a schema to adhere to the innate limitations of the database selected. In our case, even though there was no limitation on the number of tables we could have in a database, as the number of tables increased, the planner had to spend more time to come up with an efficient plan. This was the reason we dropped Approach 1.
9. It is difficult to debug query level issues in hibernate.
10. The performance of an hibernate based application depends on how the classes and the relation (1-to-1, Many-to-1, 1-to-Many) among them are defined.

# 8. Current Limitations and Future Plans

**Limitations**

1. We are not handling the issue of overlapping dates in a file. Assumption is that if a file comes at time *t* all the observations in it are for time>=t.
2. Currently we are not processing the station data file. It is simply copied into the *\_dim\_location* table using the psql “copy” command.

**Future Plans**

1. Handling above mentioned limitations.
2. Automate creation of daily, weekly and monthly aggregates tables, so that a big bulk of use cases can be fetched directly from those tables.
3. Expand the architecture in include sharding. It could be done in a manner to localize the queries. Since inserts are happening only every 15 minutes, we could easily replicate data into remote shards.

# 9. How to load and execute the demo

**9.1 Loading the database**

1. Load the database backup from file Mesowest.backup. Make sure that PostGis is installed, and the database to which you are restoring has a postgis template.
2. Ensure that all the mesowest-stored procedures have been loaded. Code for stored procedure is also provided in the zip file.
3. Ensure that the table \_*processed\_count* has only one row with values 0,0,1.

**9.2 Executing JDBC Code**

1. Update the database name and credentials in the ConnectionUtil.java file.
2. Place all the sample files in a folder, and update that path in CSVLoader.java.
3. Run ant load.
4. For testing, uncomment the required test case function from JDBCTestUtil.java.
5. Run ant test.

**9.3 Executing Hibernate code**

1. Update the database name and credentials in the hibernate.cfg file.
2. Update the location of hibernate path in build.xml
3. Place all the sample files in a folder, and update that path in CSVLoader.java.
4. Run ant load.
5. For testing, uncomment the required test case function from TestCase.java.
6. Run ant test.

# 10. Individual Contribution

Figure : Individual Contribution

# 11. Bibliography

Horel, J., Splitt, M., & Dunn, L. (2002). Mesowest: Cooperative mesonets in the western United States. *Bulletin of the …*, (FEBRUARY), 211–225. Retrieved from http://journals.ametsoc.org/doi/abs/10.1175/1520-0477(2002)083%3C0211%3AMCMITW%3E2.3.CO%3B2

Joga, M. C. (2000). *Data Warehouse Techniques to support global on-demand weather forecast metrics*. *IRE Transactions on Education*. Air Force Institute of Technology. Retrieved from http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4322266

Jr, M. M., & Dascalu, S. (2011). Architecting Climate Change Data Infrastructure for Nevada. *Advanced Information …*. Retrieved from http://link.springer.com/chapter/10.1007/978-3-642-22056-2\_38

U.S. ENVIRONMENTAL PROTECTION AGENCY. (2000). EPA Meteorological Monitoring Guidance for Regulatory Modeling Applications Meteorological Monitoring Guidance (pp. 6.5–6.6).