Table of Contents

Table of Figures

Introduction

This project aims to demonstrate creating a data warehouse that will reflect the water quality data recorded from several sensors throughout the years between 2000 and 2016. Building the data warehouse includes preparing the data. In addition to the data warehouse design, the project also aims to demonstrate creating a machine learning solution that would be able to predict the future values of the sensors based on certain components.

Data Warehouse Design

Designing the data warehouse can be done by a systematic approach that starts with the schema design and concludes with the implemented data warehouse.

ETL

a. Creating the Star Schema

After several discussions about the schema design and the relevant columns needed for the problem at hand, the most suitable star schema would be as shown in figure 1 below:

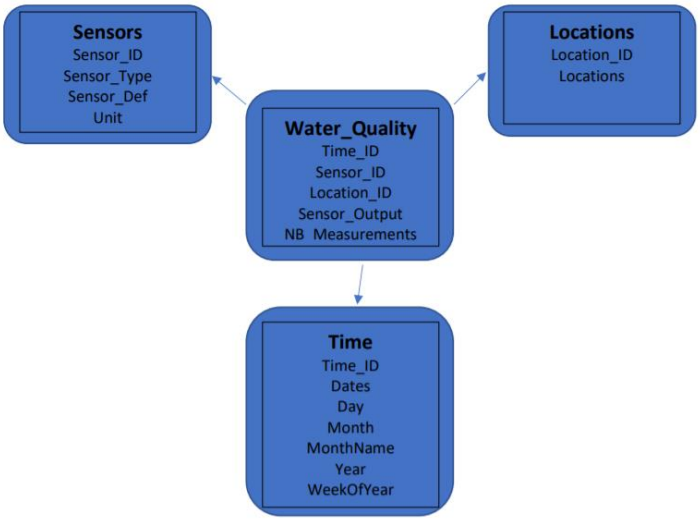

Figure 1 Star Schema

The dimension tables (Time, Sensors, and Locations) would include the main and unique values of the corresponding data related to them; Time has unique dates, Sensors has unique sensors, Locations has unique locations. Where the fact table would include the primary keys of each dimension table and two measurements (Sensor_Output and NB_Measurements).

Sensor_Output represents the results from the original data but the change of the data type from "double binary" to float would remove the wrong data type issue occurring in the staging area which relieves the necessity to clean the column in the data cleaning process, and NB_Measurements represents the number of measurements taken on the same day, using the same sensor, and on the same location.

The columns chosen would remove the need to clean the data for the staging area as these columns would be the only important part of the model and data cleaning would be done on such columns. The data types for some columns also clean the data, as in the "Sensor_id" column that was chosen to be a number instead of a binary double, fixing the wrong data type for the column in the process.

b. Exporting the data from MS ACCESS and creating the staging area

Transferring the data from MS ACCESS to Oracle was done by creating an ODBC Data Source from Windows that connect to SQL Developer, the connection to the OBIWAN server was done by creating an Oracle Data Source named OBIWAN, opening the dataset provided in MS ACCESS, and right clicking each table from 2000 to 2016 and exporting the tables through to SQL Developer by clicking on Export and choosing ODBC Database and entering the credentials for the SQL User.

After all the data was transferred to SQL Developer, there was 17 tables in the user's schema, combining all these tables can be done in several ways, I personally chose to create a copy of the 2000 table and including the data in the process, the new table was named All_Data. Afterwards, the following code lines were executed to copy all the other tables from 2001 to 2016 into the All_Data table created as a temporary staging area table, all tables were then dropped except the staging area table:

```
insert into all_data select * from "2001";
insert into all_data select * from "2002";
insert into all_data select * from "2003";
insert into all_data select * from "2004";
insert into all_data select * from "2005";
insert into all_data select * from "2006";
insert into all_data select * from "2007";
insert into all_data select * from "2008";
insert into all_data select * from "2009";
insert into all_data select * from "2010";
insert into all_data select * from "2011";
insert into all_data select * from "2012";
insert into all_data select * from "2013";
insert into all_data select * from "2014";
insert into all_data select * from "2015";
insert into all_data select * from "2016";
```

This way, the table All_Data contains all the data provided in one place without any changes.

Data Cleaning

The staging area has around 25000 rows, checking for NULL values in the columns chosen for the star schema was done by executing each code for checking the corresponding columns:

```
select * from all_data where "samplesamplingPointnotation" is null;
select * from all_data where "samplesamplingPointlabel" is null;
select * from all_data where "samplesampleDateTime" is null;
select * from all_data where "determinandlabel" is null;
select * from all_data where "determinanddefinition" is null;
select * from all_data where "determinandnotation" is null;
select * from all_data where "result" is null;
select * from all_data where "determinandunitlabel" is null;
```

The columns do not include any NULL values. however, the sensor_type column which corresponds to the "determinandlabel" has many values as "NO FLOW/SAMP" which could cause inconsistency in the data as the sensor is unknown in this row and it has no result. Hence, such values were deleted by executing the following code to remove all 164 "NO FLOW/SAMP" records:

```
delete from all_data where "determinandlabel" = 'NO FLOW/SAMP';
```

There were no obvious string mismatches in the Locations, Sensors, and Dates columns which can be seen by selecting the column and counting the results column grouped by the column selected.

Building the Warehouse

a. Creating the Dimension tables

The Time, Locations, and Sensors dimensions were all created by executing the following code:

```
create table locations (
location_id varchar(15) primary key,
Locations varchar(40));

create table time (
Time_id varchar(30) primary key,
dates varchar(20),
day varchar(3) not null,
month varchar(3) not null,
monthname varchar2(10),
year varchar(5) not null,
weekofyear number(3) not null);

create table Sensors(
sensor_id number(6) primary key,
sensor_type varchar(25) not null,
sensor_def varchar(75),
unit varchar(10));
```

The Locations table include the location_id as a primary key column which represents the column "samplesamplingPointnotation" in the original data and staging area, the other column is the locations column which is the name of the location corresponding to the location_id and is represented in the staging area by the column "samplesamplingPointlabel".

The Time table include the Time_id as a primary key column which represents the column "samplesampleDateTime" in the staging area, it also includes a dates column that represents the date only, day column that represents the day number, month column that represents the month number, monthname column that represents the month name, year column that represents the year number, and a weekofyear column that represent the week number in that year.

The Sensors table include the sensor_id as a primary key which represent “determinandnotation” in the staging area, it also includes a sensor_type column representing the “determinandlabel” in the staging area, a sensor_def column representing the “determinanddefinition” in the staging area, and a unit column representing the “determinandunitlabel” in the staging area.

b. Creating the fact table

The fact table of the schema was created by executing the following code:

```
create table water_quality(
Time_id varchar(30),
sensor_id number(6),
location_id varchar(15),
sensor_output float(10) not null,
NB_Measurements number(10),
constraint WQ_FKM foreign key (sensor_id) references sensors(sensor_id),
constraint WQ_FKL foreign key (location_id) references locations(location_id),
constraint WQ_FKT foreign key (Time_id) references time(Time_id),
constraint WQ_PK primary key (Time_id, sensor_id, location_id))
tablespace student2;
```

The fact table consists of the 5 columns mentioned before, the constraints in the code are what make this table the fact table, having a composite primary key consisting of the primary keys of the dimensions and having each column in the primary key as a foreign key referencing each dimension. The table was created in the student2 tablespace to avoid storage limitations.

c. Populating the dimension tables

The Locations and Sensors dimensions were populated using a simple select distinct query from the staging area and into the dimension, the following code lines were executed to populate the tables:

```
insert into locations select distinct "samplesamplingPointnotation", "samplesamplingPointlabel" from all_data;

insert into sensors select distinct "determinandnotation", "determinandlabel", "determinanddefinition",
"determinandunitlabel" from all_data;
```

The time table however was populated using a cursor, using the cursor was mainly to convert the data while populating the table simultaneously. The process can be also done with the fact table population, but I chose to do it separately to remove confusion and to ensure the data converted was correct. The following code was executed to perform the time dimension population process:

```
declare cursor times is
select distinct "samplesampleDateTime" from all_data;
days varchar(3);
dates varchar(20);
months varchar(3);
monthnames varchar2(10);
years varchar(5);
weekofyears number(3);
begin
   for i in times loop
   days:= to_char(to_date(substr(i."samplesampleDateTime",0,10), 'YY-MM-DD'),'DD');
   dates:= to_char(to_date(substr(i."samplesampleDateTime",0,10), 'YY-MM-DD'),'DD-MM-YYYY');
   months:= to_char(to_date(substr(i."samplesampleDateTime",0,10), 'YY-MM-DD'),'MM');
   monthnames:= to_char(to_date(substr(i."samplesampleDateTime",0,10), 'YY-MM-DD'),'Month');
   years:= to_char(to_date(substr(i."samplesampleDateTime",0,10), 'YY-MM-DD'),'YYYY');
   weekofyears:= to_number(to_char(to_date(substr(i."samplesampleDateTime",0,10), 'YY-MM-DD'),'iw')) ;
   insert into time values (i."samplesampleDateTime" , dates,days, months, monthnames, years, weekofyears);
end loop;
end;
/
```

The cursor takes distinct time_id values, meaning there could be duplicated dates but the time in the string makes it unique, this was done to ensure that the NB_measurements column is accurate and is not influenced by the absence of time in the counting process. The data was converted accordingly with the columns in the time dimension as explained previously.

The substring usage in the conversion process was used to extract the date only from the time_id and was then converted to the corresponding column usage.

d. Populating the fact table

Before populating the fact table, a temporary table “temp” was created to hold the NB_measurements values to be inserted into the fact table, this was done to simplify the cursor while populating the fact table, making it easier to debug any issues as well. Alternatively, a “select into” query can be used in the cursor that includes the query to populate the temporary table with slight adjustments. The temp table was created and populated by executing the following code:

```
create table temp(
sensorid number(10),
dateid varchar(50),
locationid varchar(50),
measurement number(10))
tablespace student2;

insert into temp
select f."determinandnotation",  substr(f."samplesampleDateTime",0,10), f."samplesamplingPointnotation",
count(f."result") as measurement from all_data f, sensors s, time t, locations l
where s.sensor_id = f."determinandnotation" and f."samplesampleDateTime" = t.time_id and
f."samplesamplingPointnotation" = l.location_id
group by  f."determinandnotation" , substr(f."samplesampleDateTime",0,10),
f."samplesamplingPointnotation";
```

The query takes the dates only using the substring function because the NB_measurements count depends only on the date and not the time, counting the “result” column produces the number of total measurements done in the staging area which corresponds to 24767 measurements in total, and counts only the ones with similar conditions and inserts the number of measurements done for the specific sensor on the same exact date and at the same location only.

The fact table was populated using a cursor as well, it includes joining the dimensions created and populated to ensure that the data warehouse design is accurate and useful, the dimension tables were joined in the query based on their primary keys and the corresponding column in the staging area. The following code was executed to populate the fact table:

```
declare cursor WQ is
select t.time_id, s.sensor_id, l.location_id, a."result" from all_data a
join sensors s on (a."determinandnotation" = s.sensor_id)
join locations l on (a."samplesamplingPointnotation" = l.location_id)
join time t on  (a."samplesampleDateTime" = t.time_id);
nb number(10);
begin
for i in WQ loop
select distinct measurement into nb from temp f, all_data a
where sensorid = i.sensor_id
and dateid = substr(i.time_id,0,10)
and locationid = i.location_id;
insert into water_quality values (i.time_id, i.sensor_id, i.location_id, i."result", nb);
end loop;
end;
/
```

After the process was done, the data warehouse design was completed. Therefore, all temporary tables were dropped along with the staging area table. The data warehouse design model can be viewed in SQL Developer as shown in figure 2 below:

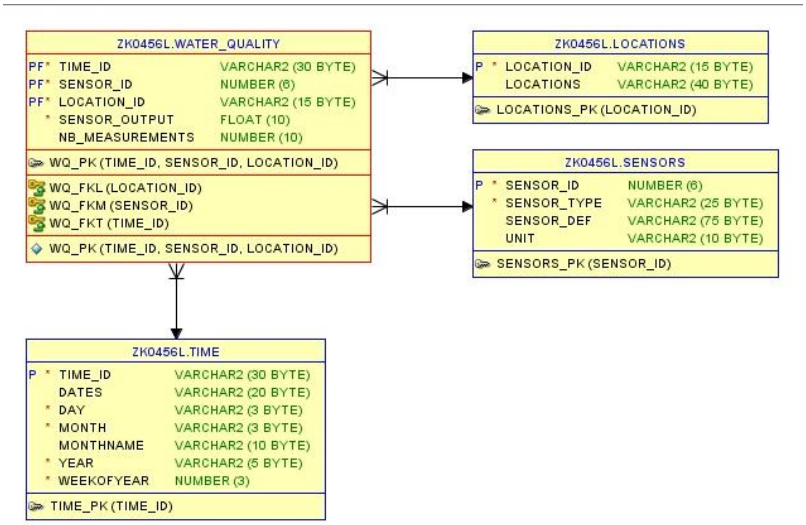

Figure 2 Data Warehouse Star Model

e. Statistical Queries

Each required statistic can be acquired by executing the code under each requirement:

1. The list of water sensors measured by type of it by month

```
select distinct s.sensor_id, s.sensor_type, t.monthname
from sensors s, time t, water_quality a
where a.sensor_id = s.sensor_id and t.time_id=a.time_id;
```

2. The number of sensor measurements collected by type of sensor by week

```
select s.sensor_type, t.weekofyear, count(w.sensor_output) as number_of_measurements
from water_quality w, sensors s, time t
where s.sensor_id = w.sensor_id and t.time_id = w.time_id
group by s.sensor_type, t.weekofyear;
```

3. The number of measurements made by location by month

```
select t.monthname, l.locations, count(w.sensor_output) as number_of_measurements
from water_quality w, time t, locations l
where w.time_id = t.time_id and w.location_id = l.location_id
group by t.monthname, l.locations;
```

4. The average number of measurements covered for PH per year

```
select round(avg(number_of_measurements),2) as average_number_of_pH_measurements_per_year
from ( select t.year,count(w.sensor_output) as number_of_measurements
        from water_quality w, time t, sensors s
        where t.time_id = w.time_id and s.sensor_id = w.sensor_id and s.sensor_type like 'pH%'
        group by t.year);
```

The requirement states number of measurements by year, but as an average per year it would result in the count as the count by year is fixed, therefore an average per year for all years was provided.

5. The average value of Nitrate measurements by locations by year

```
select t.year, l.locations, round(avg(w.sensor_output),2) as average_number_of_measurements
from water_quality w, time t, sensors s, locations l
where t.time_id = w.time_id and s.sensor_id = w.sensor_id and l.location_id = w.location_id and
s.sensor_type like 'Nitrate%'
group by t.year, l.locations;
```

An alternative to the wildcard of ‘Nitrate%’ would be the sensor_id of the sensor which is 117.

Python Machine Learning (ML) Solution

This section aims to produce an ML solution using Python in the Jupyter environment to predict the future values of the “Nitrate-N” sensor under the sensor_id 117. The main technique used in this project is the K-Nearest Neighbors (KNN) technique and it was adapted to the dataset we have accordingly.

Developing the ML Solution in Python

Establishing a connection between Python and Oracle

The first step to establish a connection between Jupyter and Oracle is to install the cx_Oracle files into the directory using the “!pip install cx_Oracle” line. Afterwards, importing the cx_Oracle files and the OS files and specifying the directory path, the following code establishes the connection between the Oracle database and the Python Jupyter Notebook environment:

Establishing a cursor connection allows the user to import data based on a SQL query, and before deciding which columns are most relevant to the Nitrate sensor, all columns from all tables where selected and the best correlation between the columns and the results column was found in the “location_id” or “location” both after encoding, and the “weekofyear” column, other columns have less correlation due to repetitive and fixed values of the columns.

The query used selects the “location_id”, “dates”, “weekofyear” and “sensor_output” columns from their corresponding tables in the data warehouse, and then the output of the query is stored in a Python Dataframe object. The following code was executed to obtain the Dataframe object under the variable named “data”:

The “data” variable now is a Dataframe object containing 1121 records of data related to the Nitrate Sensor. The “dates” column was selected for data preparation purposes that will be explained later, as for the rest of the columns selected; they had higher correlation with the column containing the values to be predicted and therefore were chosen.

Data Preparation

The first step in data preprocessing would be a label encoding process to eliminate all the string variables as the ML algorithm would only accept numbers into it and produces numbers in return. The label encoding process was done using SKLEARN’s preprocessing feature that performs label encoding and stores the encoding features in a variable, the encoding process was done for all three columns except the “dates” column which is used for other reasons and will not be used in the ML algorithm.

Label, and Locations_encoded represent the encoded values of the Results, and Locations columns respectively, another column was added using the “dates” column value which is the season of the measurement, which was produced depending on the astronomical season of the date the measurement was taken at.

The values provided by the for loop above was then encoded using the same method to convert the string into an encoded value these variables were then stored in a different Dataframe variable named “MLDF”.

The MLDF dataframe now contains all columns with encoded values as well as the “weekofyear” column that does not need encoding.

Cross-Validation

After the first few trials of the KNN algorithm, a cross-validation evaluation process was done to perform a 25-iteration trial on the number of K-Neighbors that produces the best solution, this is the reason this section comes in this report before the actual implementation of the ML algorithm.

The cross-validation process merely estimates the best number of neighbors needed to adjust the ML algorithm with the most suitable settings, this was done by executing the code below using SKLEARN's GridSearchCV feature.

The output of this cross-validation process indicates that the best possible optimization of the algorithm would be to set the number of K-Neighbors at 19 neighbors, which was used in the ML algorithm for this project and provided the best accuracy after some manual testing, proving the usefulness of the cross-validation process.

Adapting ML techniques

The first thing to do when adopting an ML technique is to split the data between training and testing data for validation, throughout the data preparation procedures, the data was split into inputs and outputs; where the input would include the encoded values of the "locations" and "season" columns as well as the "weekofyear" column, and the output would be the encoded values of the results column, this was done by storing the data in the X and Y variables as follows:

The data was then split into training data and testing data using SKLEARN's train_test_split feature.

Now, the algorithm would be trained on the X_train and the y_train variables and tested on the X_test and y_test variables, keep in mind that the algorithm would never see the testing variables throughout the training process. The following code adapts the KNN algorithm and sets the number of K-Neighbors to 19 as indicated in the cross-validation process:

Other SQL Queries

This section contains SQL queries that were not executed and are not in the database except for the extended time table requirement:

An extended TIME table should be created and populated.

```
create table Time_Extended (
Time_id varchar(30) primary key,
dates varchar(20),
day varchar(10) not null,
dayofweek varchar(20),
dayofyear varchar(5),
timeofday varchar(20),
month varchar(10) not null,
monthname varchar2(10),
year varchar(10) not null,
month_and_year varchar(40),
weekofmonth varchar(10),
weekofyear number(10) not null,
Quarter varchar(10));

declare cursor timess is
select distinct time_id from time;
days varchar(10);
dates varchar(25);
dayofweek varchar(20);
timeofday varchar(20);
```

```
months varchar(10);
monthnames varchar2(10);
years varchar(10);
weekofyears number(10);
month_and_year varchar(40);
weekofmonth varchar(10);
Quarter varchar(10);
dayofyear varchar(10);

begin
   for i in timess loop
   days:= to_char(to_date(substr(i.time_id,0,10), 'YY-MM-DD'),'DD');
   dates:= to_char(to_date(substr(i.time_id,0,10), 'YY-MM-DD'),'DD-MM-YYYY');
   months:= to_char(to_date(substr(i.time_id,0,10), 'YY-MM-DD'),'MM');
   monthnames:= to_char(to_date(substr(i.time_id,0,10), 'YY-MM-DD'),'Month');
   years:= to_char(to_date(substr(i.time_id,0,10), 'YY-MM-DD'),'YYYY');
   weekofyears:= to_number(to_char(to_date(substr(i.time_id,0,10), 'YY-MM-DD'),'ww'));
   dayofweek:= to_char(to_date(substr(i.time_id,0,10), 'YY-MM-DD'),'Day');
   timeofday:= substr(i.time_id,12,5);
   dayofyear:= to_char(to_date(substr(i.time_id,0,10), 'YY-MM-DD'),'DDD');
   month_and_year:= to_char(to_date(substr(i.time_id,0,10), 'YY-MM-DD'),'Month YYYY');
   weekofmonth:= to_char(to_date(substr(i.time_id,0,10), 'YY-MM-DD'),'w');
   Quarter:=to_char(to_date(substr(i.time_id,0,10), 'YY-MM-DD'),'Q');

   insert into time_extended values (i.time_id, dates, days, dayofweek, dayofyear, timeofday, months, monthnames,
years, month_and_year, weekofmonth, weekofyears, quarter);
end loop;
end;
/
```

The Average of the Nitrate-N sensor value by every two years materialized view

```
Create materialized view Avg_Nitrate_every_2years as
select '2000-2001' as period, avg(avg_2years) from (
select (year || '-' || (year+1) ) as years, avg(avg_year) as avg_2years from (
select t.year, round(avg(w.sensor_output),2) as avg_year from time t, water_quality w where t.time_id=w.time_id
and w.sensor_id=117  group by t.year)
group by (year || '-' || (year+1)) having (year || '-' || (year+1) ) like '%2000%')
Union
select '2003-2004' as period,avg(avg_2years) from (
select (year || '-' || (year+1) ) as years, avg(avg_year) as avg_2years from (
select t.year, round(avg(w.sensor_output),2) as avg_year from time t, water_quality w where t.time_id=w.time_id
and w.sensor_id=117  group by t.year)
group by (year || '-' || (year+1)) having (year || '-' || (year+1) ) like '%2004%')
Union
select '2005-2006' as period,avg(avg_2years) from (
select (year || '-' || (year+1) ) as years, avg(avg_year) as avg_2years from (
select t.year, round(avg(w.sensor_output),2) as avg_year from time t, water_quality w where t.time_id=w.time_id
and w.sensor_id=117  group by t.year)
group by (year || '-' || (year+1)) having (year || '-' || (year+1) ) like '%2006%')
Union
select '2007-2008' as period,avg(avg_2years) from (
select (year || '-' || (year+1) ) as years, avg(avg_year) as avg_2years from (
select t.year, round(avg(w.sensor_output),2) as avg_year from time t, water_quality w where t.time_id=w.time_id
and w.sensor_id=117  group by t.year)
group by (year || '-' || (year+1)) having (year || '-' || (year+1) ) like '%2008%')
Union
select '2009-2010' as period,avg(avg_2years) from (
select (year || '-' || (year+1) ) as years, avg(avg_year) as avg_2years from (
select t.year, round(avg(w.sensor_output),2) as avg_year from time t, water_quality w where t.time_id=w.time_id
and w.sensor_id=117  group by t.year)
```

```
group by (year || '-' || (year+1)) having (year || '-' || (year+1) ) like '%2010%')
Union
select '2011-2012' as period,avg(avg_2years) from (
select (year || '-' || (year+1) ) as years, avg(avg_year) as avg_2years from (
select t.year, round(avg(w.sensor_output),2) as avg_year from time t, water_quality w where t.time_id=w.time_id
and w.sensor_id=117  group by t.year)
group by (year || '-' || (year+1)) having (year || '-' || (year+1) ) like '%2012%')
Union
select '2013-2014' as period,avg(avg_2years) from (
select (year || '-' || (year+1) ) as years, avg(avg_year) as avg_2years from (
select t.year, round(avg(w.sensor_output),2) as avg_year from time t, water_quality w where t.time_id=w.time_id
and w.sensor_id=117  group by t.year)
group by (year || '-' || (year+1)) having (year || '-' || (year+1) ) like '%2014%')
Union
select '2015-2016' as period,avg(avg_2years) from (
select (year || '-' || (year+1) ) as years, avg(avg_year) as avg_2years from (
select t.year, round(avg(w.sensor_output),2) as avg_year from time t, water_quality w where t.time_id=w.time_id
and w.sensor_id=117  group by t.year)
group by (year || '-' || (year+1)) having (year || '-' || (year+1) ) like '%2016%');
```

The Create table statement that will Partition the Fact table by year.

```
create table water_quality2 (
Time_id varchar(30),
sensor_id number(6),
location_id varchar(15),
sensor_output float(10) not null,
NB_Measurements number(10),
year varchar(10),
constraint WQ_FKM foreign key (sensor_id) references sensors(sensor_id),
constraint WQ_FKL foreign key (location_id) references locations(location_id),
constraint WQ_FKT foreign key (Time_id) references time(Time_id),
constraint WQ_PK primary key (Time_id, sensor_id, location_id))
PARTITION BY HASH(year)
PARTITIONS 17;
```