***Data Loading Strategy and Problem Resolution***

Data Loading Strategy

The general strategy that we used to load the source data into the database was to first target the tables that did not rely on any foreign keys and establish them. This made it easier when creating the tables which rely on primary keys from other tables as foreign keys.

We did not put any constraints into the database until all of the data was loaded and formatted correctly into the correct data types. The primary reason for this was because a fair portion of the data had to be further validated and formatted after it was imported into the database. An example of this would be to take the date, start time, and end time of each event. All of these entries are meant to be dates however they have to be imported into the database first as normal strings then later formatted into a date data type within the database.

One of the biggest problems that we faced when devising our strategy for loading the source data was using the primary keys of tables as foreign keys. Finding a way to effectively populate those tables with the correct foreign data proved to be a fairly large challenge. The way that we overcame the task was to use an update query on the foreign key column, which matches up the correct key from the foreign table with the appropriate entry in the current table. The most notable example of this in action would be with the LOCATIONID from the LOCATION table, due to how often it is used as a foreign key. The common steps that we took when utilising this column as a foreign key were:

1. Load all data relevant to the table that we are currently loading, leaving the entire LOCATIONID foreign key column ***null*** for the time being.
2. Add a number of extra ***temporary columns*** to the current table which could be used in the update query to avoid ambiguity when matching rows (for the LOCATIONID foreign key, these temporary columns were usually Building, RoomNo, and RoomName).
3. Execute an update query on the foreign key column, adding the corresponding values from the primary key table.
4. Run queries and manually check that the values entered as foreign keys match correctly to the data that is contained in the source excel document.
5. Remove the temporary columns from the table, and add constraints and primary keys.

When creating the tables, one of the things that we made sure to do on creation was to make the triggers and sequences for the appropriate surrogate keys when needed.

The first setback that we encountered was that we realised that we first had to organise the source. There was no conceivable way that we would have been able to import the data as it was in the initial document so we decided to split it up into a series of documents that correspond to the different entities that were going to be included in our database.

The strategy that we used to find out what data we needed for each of the tables was to first analyse the data and then look back at our final ERD for the system and see how compatible it was with the data we were presented with. What we found was that some aspects of the ERD not sync very well with the source data. To combat this we had to go through our tables and make certain changes ranging from adding, removing or renaming attributes of a given table, to removing a table entirely.

An example of us invoking this strategy would be with the CONFIGURATION table currently in the database; it differs from that of the table in the ERD. We found that the data that is associated with the configuration of each room is not compatible with the table layout of that in the ERD.

-----

Issues were encountered when loading the data specifically the way in which the unique identifiers for the PCs were generated inside excel document. The fact that the numbers were randomly generated within a range meant that there was a possibility that not all of the IDs were unique. This meant that we had to validate the data before importing it into the database. To do this our primary method was to employ a built-in function inside of excel that checks for duplicates within a selected amount of cells (Function: Duplicate Values, Path: Home > Conditional Formatting > Highlight Cells Rules > Duplicate Values).

-----

One of the most common but less threatening issues that we ran into during the data loading stage was several different inconsistencies in the excel source document.

***Problem***: Misspelt data in the database, causing entries to remain null when attempted to update the foreign key column.

**Solution**: The amount of cases we had of this was relatively few, so our solution to this issue was to just manually correct the spelling errors.

***Problem:***There were cases from time to time where there were entries in the data that were inconsistent in the rest of the data. A prime example of this would be that there were entries in the equipment allocation data containing times that were not even listed as times in the equipment data.

**Solution**: In most cases of this situation we thought it was most prudent to just delete the line which had the non-existent time, as it didn’t make sense for the database to claim that an item was there that wasn’t even recorded.

***Problem*:** In several different fields of the excel source document, the actual data types of the values stored were incorrect for the context that they were in. For example, the numbers stored for the capacity values for each different configuration for each location were stored as text in the spreadsheet, which translated into strings for the database, which caused issues when trying to import them into a column that is of the number data type.

**Solution**: In excel you are able to change the type of data that is stored inside the cells, so it was a fairly simple matter to select all of the incorrectly typed cells and change them to the correct format.

***Problem***: In practically every section of the excel source document, there were empty rows scattered throughout the data, which Oracle reads in as an empty line, creating many unnecessary rows inside of the database when imported.

**Solution**: There were a couple of different ways that we tackled this problem, finding out the more efficient way later on. The first solution that we came up with was to go through the excel document and manually delete the blank rows for each section of data, which proved to be a fairly time consuming and arduous task. The second, and much more time efficient solution that we came up with, was to import all of the empty rows with the data, and use a delete script on the table to remove all of the rows that have any given column as null, as that means the rest of the entry will be null too.

**Detailed Description**

For this section of the Data Loading explanation, there will be more in depth explanations as to how we achieved certain specific aspects of the data loading.

The first exercise to be discussed is how we managed to successfully merge the time and date columns from the source data into a single field inside the database, which is of data type “Date”.

First of all, the reasoning that we applied when making this decision was that, due to the fact that Oracle databases do not have a field data type that is specifically for time, it would make more sense to combine the time with the date as the time would have to be stored in a field with the date type “Date” anyway. The only major change that this would make in terms of exporting the database or importing new data into the database is the formatting that would have to be applied to the data when extracting it from the database.

There was an issue that was found with formatting the time however, when extracting the database; it does not take the time into account the time when making the insert statements. The only way that we found to counter this issue was to later go and manually change the insert scripts to include the date data and change the formatting to take the time into account. This kind of fix is viable in a case such as this with such a small amount of data, but in a much larger situation, an alternate method would need to be used.

The first action that we had to perform was to make two temporary fields inside of the BOOKING entity that would hold the separated time and date of each booking. Both of these fields we of data type “Varchar2” as it will be necessary to treat them as strings at a later point.

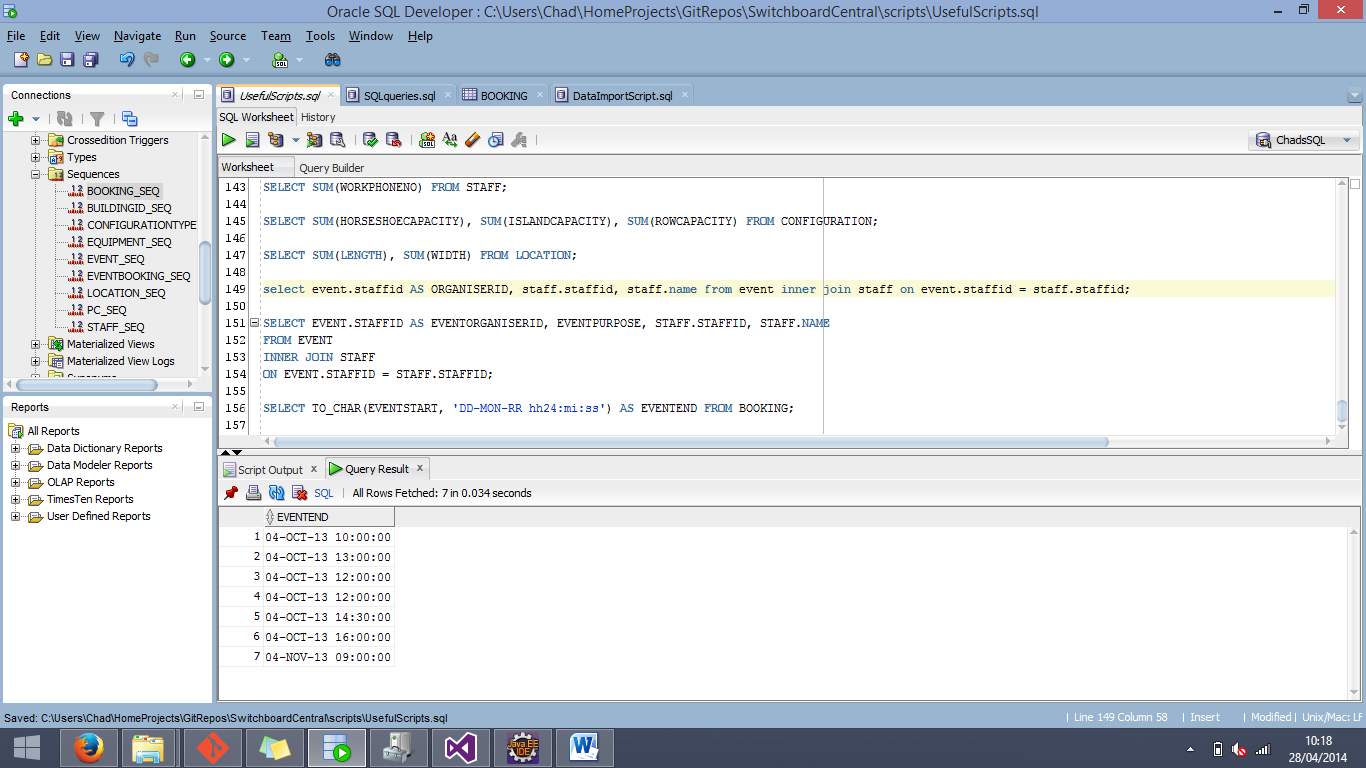
The next step was to then create the field that would hold the final data in the date format. This column had to be made with the Default Null constraint and then be filled in later with an update statement.

Before attempting the update, we had to test that we would successfully be able to update the Date column with something that is able to be formatted into a date. For this we used a specialised select statement that concatenated the time and date when extracting it from the database, which is shown below:

SQL Script:

SELECT TO\_CHAR(TO\_DATE(TO\_CHAR(BOOKINGDATE || ' ' || STARTTIME), 'dd-mm-yy hh24:mi:ss'), 'yyyy-mm-dd hh24:mi:ss') as NEWDATE FROM BOOKING;

The results of the above query are shown below, indicating that it is possible update the permanent column in this fashion.



The SQL Script used for the update is shown below:

SQL Script:

UPDATE EVENTEND BookingNew

SET ENDDATE = (SELECT TO\_DATE(TO\_CHAR(BOOKINGDATE || ' ' || ENDTIME), 'DD-MON-RR hh24:mi:ss')

FROM BOOKING BookingOld

WHERE BookingOld.BOOKINGID = BookingNew.BOOKINGID);

UPDATE EVENTSTART BookingNew

SET ENDDATE = (SELECT TO\_DATE(TO\_CHAR(BOOKINGDATE || ' ' || STARTTIME), 'DD-MON-RR hh24:mi:ss')

FROM BOOKING BookingOld

WHERE BookingOld.BOOKINGID = BookingNew.BOOKINGID);