Dylan Parker

D191 Advanced Data Management

VDM TASK 1 – AUTOMATING DATA INTEGRATION

In today’s business landscape, the power of business intelligence derived from efficient usage of big data cannot be understated. It is extremely critical that companies utilize well thought out ETL techniques to organize and access their data for analysts to do their job effectively. We are going to look at a simple example of such an ETL pipeline using postgresql to access a database, with the ultimate goal of informing us about a common business problem.

The database we are going to look at is the DVD Database on Labs on Demand, a virtual environment that allows us to use pg Admin4 as a postgresql GUI frontend to access this database. The database is a publicly available example database that contains information regarding rental transactions from a fictional DVD rental company. This database contains a snowflake schema involving a centra fact table that describes the rental transactions by rental id,. It expands to many levels of dimension tables describing the different components that fuel those transactions, such as customers, stores, films and employees.

For data analysis to be valuable, it needs to have real world implications, and thus we are going to look at a business problem that we would realistically want to inform ourselves about. I have decided to look at the genres of the movies that this DVD rental company has rented, and aggregate this data to determine which genres are the most rented genres, and also which genres are the most profitable? This is a realistic business question that could be used to inform decisions about which DVDs to purchase and offer to their customers, or how many copies of different genres of DVDs should they keep in stock.

To accomplish this, we will need to produce a summary table that contains a row for each unique DVD genre, and an aggregate count of the rental ids, as well as an aggregate sum of the rental amounts. This will require us to take the payment fact table, and join it with the category dimension table, going through a lot of other tables on the way to get there. We will also create a detailed table that contains the non-aggregated collection of fields that we will need to populate the summary table on demand. This will optimize our ETL operations for analysis and allow us to easily refresh our data in as close to real-time as we desire.

**As noted previously, the summary table will contain the following aggregate fields :**

* Genre
* Total number of rentals
* Total amount of sales

**The detailed table will contain the information we will need to accomplish this task**

* Rental ID – will be used to compute aggregate total number of rentals
* Customer ID – could be useful for different approaches to informing our analysis
* Film ID – needed to access the genre
* Amount – the rental dollar amount that will be used to compute the aggregate total sales
* Rental rate – could be useful for different approaches to informing our analysis
* Name – the genre name that will be using to group our aggregations

In the detailed table, both the rental ID and amount fields will require custom transformations to give us the information we need in the summary table. We will need to take a count of rental IDs using the count() function, and we will need the sum of all the rentals amounts using the sum() function. We will write a custom function to be able to trigger this process of populating the summary table automatically to streamline our ETL pipeline.

The business use of this particular summary report will be to inform business decisions regarding what genres of DVDs they should focus on, which ones are not as important, and potentially how many copies of certain DVDs they should be keeping in inventory. The fields for the detailed version of the table are constructed to allow for further exploration along this axis. For example, you could look at the rental rates to determine if some genres that have lower amounts of total rentals might be cost efficient because they results in a disproportionately high amount of revenue due to their rental cost. The detailed table in essence will allow for deriving new or updated summary tables that could inform different business needs or decisions.

From an ETL standpoint the detailed table provides a redundant access point for the data needed to efficiently conduct a targeted analysis of the data. To streamline this process as much as possible, I have demonstrated how one can fully automate this process, refreshing the data in both the summary and detailed tables to keep the information used for analysis current and relevant. As you will see in my code, I have defined a custom function that deletes all the information in the summary table, then re populates it from the detailed table. Then I define a custom trigger that happens whenever the detailed table is updated and calls the function that updates the summary table. Then there is a custom procdure that updates the detailed table that is designed to be scheduled or automated. One call to the procedure will clear the detailed table and repopulate it, and this operation will trigger the function that then populates the summary table.

This process can easily be automated by using a task scheduler to make calls to the procedure on some predetermined set schedule. For this type of analysis, depending on how often the company’s database is updated, it might be beneficial to generate this summary report every night, so that whenever someone accesses it, it remains relevant. At a minimum I believe you would want to update this report every week to allow for trend analysis, especially in an industry where movie releases are happening all the time and understanding the demand for these is critical when you have to decide how many copies of new releases the company should stock in their stores.

In summary, we took a look at a simple example of such an ETL pipeline using postgresql to access a database, with the ultimate goal of informing us about a common business problem.

Attached are the SQL statements that I used to build the ETL pipeline described in this paper

--First lets make sure the detailed table doesn’t exist before we create it

DROP TABLE IF EXISTS detail;

--Next lets create our empty detailed table, and verify it

CREATE TABLE detail (

rental\_id INTEGER,

customer\_id INTEGER,

film\_id INTEGER,

amount NUMERIC,

rental\_rate NUMERIC,

name VARCHAR

);

SELECT \* FROM detail;

--next lets make sure that the summary table doesn’t exist before our CREATE statement

DROP TABLE IF EXISTS summary;

--Lets create our empty summary table that will hold our summary report data, and verify

CREATE TABLE summary (

genre VARCHAR,

num\_rentals INTEGER,

total\_revenue NUMERIC,

);

SELECT \* FROM summary;

--This is what our statement to populate the detailed table will look like

INSERT INTO detail (rental\_id, customer\_id, film\_id,

amount, rental\_rate, name

)

SELECT p.rental\_id, p.customer\_id, f.film\_id, p.amount, film.rental\_rate, c.name

FROM payment AS p

INNER JOIN rental as r ON p.rental\_id = r.rental\_id

INNER JOIN inventory as i ON r.inventory\_id = i.inventory\_id

INNER JOIN film as f ON i.film\_id = f.film\_id

INNER JOIN film\_category as fc ON f.film\_id = fc.film\_id

INNER JOIN category as c ON fc.category\_id = c.category\_id;

--lets verify that worked properly

SELECT \* FROM detail;

--This function will clear the summary table, and populate it with transformed

--data from the detailed table.

CREATE FUNCTION update\_sum()

RETURNS TRIGGER

LANGUAGE plpgsql

AS

$$

BEGIN

DELETE FROM summary;

INSERT INTO summary (

SELECT name AS genre, COUNT(rental\_id)

AS num\_rentals,

SUM(amount) AS total\_revenue

FROM detail

);

RETURN NEW;

END;

$$;

--This trigger will call our function every time an insert operation is performed on the

--detailed table

CREATE TRIGGER sum\_update

AFTER INSERT ON detail

FOR EACH STATEMENT

EXECUTE PROCEDURE update\_sum();

--this stored procedure will clear the detailed table and then update it, which will in turn trigger

--the function that updates the summary table **This should be**

**--automated to run on a set schedule using task scheduler every Monday morning. This will**

**--capture all of the business from the previous week and allow our report to inform**

**--the weekly business strategy meeting that takes place at 10:00am every Monday.**

CREATE PROCEDURE update\_detail()

LANGUAGE plpgsql

AS

$$

BEGIN

DELETE FROM detail;

INSERT INTO detail (rental\_id, customer\_id, film\_id,

amount, rental\_rate, name

)

SELECT p.rental\_id, p.customer\_id, f.film\_id, p.amount, film.rental\_rate, c.name

FROM payment AS p

INNER JOIN rental as r ON p.rental\_id = r.rental\_id

INNER JOIN inventory as i ON r.inventory\_id = i.inventory\_id

INNER JOIN film as f ON i.film\_id = f.film\_id

INNER JOIN film\_category as fc ON f.film\_id = fc.film\_id

INNER JOIN category as c ON fc.category\_id = c.category\_id;

END;

$$;

--This call to the stored procedure is what causes our ETL pipeline to run. **This is what needs to**

**--be automated using task scheduler.**

CALL update\_detail();

SELECT \* FROM detail;

SELECT \* FROM summary;

References

<https://stackoverflow.com/questions/12343984/insert-trigger-to-update-another-table-using-postgresql>

<https://www.postgresqltutorial.com/postgresql-plpgsql/postgresql-create-function/>