Data Warehousing Guide - The ETL Processing - 4/4

qimia.io/en/blog/Data-Warehousing-Guide-The-ETL-Processing

Introduction

Welcome back to the last part of our tutorial on data warehousing. In the previous articles, we presented techniques and principles for creating a professional data warehouse (DWH). Furthermore, we showed how to construct a data warehouse based on an artificial OLTP schema by identifying the underlying business process. Finally, we gave a practical example of how to define dimensions and facts in that Data Warehousing Schema.

In this last article, we will show you how to create an ETL pipeline for filling the tables in an appropriate way.

We will be using a **PostgreSQL** database and create procedures and functions for the data processing. You can find all of the code in our Github.

The article is structured in three sections. The first section is on general aspects of the processing such as the correct order. The second one is on the processing of the Dimension tables and the third one is on the processing of the Fact tables.

1. General Aspect on Data Warehouse Processing

1.1. The ETL Processing Order

As some of the tables in our data warehouse depend on each other, we need to follow a certain order in the processing. As a first step, the dimension tables are filled.

Here we create the surrogate key from a hash function and set it as a primary key for the dimension. This key is then used as a foreign key in the fact tables. In our scenario, we also have a dependency of all dimensions to the date dimension, which is why we need to populate that dateDim first. The salesDailyFact table is an aggregate of the salesFact, which is why you have to fill the atomic salesFact before the salesDailyFact.

Summing it up, we need to process the date dimension first, then all other dimensions, then the atomic fact tables, and the aggregated facts in the end. However, that processing order is only mandatory as we choose to have the foreign key dependency. It delivers a guarantee that there is no fact that references a not existing dimension, but it lacks parallelization.

Nowadays cloud data warehouse implementations such as Azure's Synapse or AWS's Redshift do not enforce foreign key constraints, due to their distributed nature. In such an environment the dimensions and facts might be loaded in parallel. However, this bonus of

parallelization comes with the lack of implicit control on the completeness of fact and dimension keys.

1.2. Other General Aspects

There are way more aspects that have to be taken into account when building a professional ETL pipeline for a data warehouse. Some of them will be covered in the following like backfilling vs. daily batching scenarios. However, mentioning all of them is out of the scope of this blog. Instead, we list some of the more important aspects that data warehouse architects and developers have to consider:

- How to guarantee data quality and how to test data?
- Data latency - how fast do business users need the data in the DWH?
- Data archiving - how much data should be kept in the DWH?
- Which skills do my ETL developers have?
- What is my budget?

All these questions have to be answered when setting up the ETL pipeline. Some of them will influence the architecture of the pipelines themself. Others, such as the budget or the capability of the staff, influence the choice of the ETL tool.

2. Dimension Processing

The dimension processing is basically split into two parts. We first process the dateDim, which is a type 0 slowly changing **Outrigger** dimension. Then we process the type 2 slowly changing dimensions storeDim, productDim, employeeDim. However, we will not go into details of all dimensions, but just focus on the dateDim and storeDim exemplarily.

2.1. The dateDim

The date dimension is a very stable and predictable dimension. The only question that the DWH-architect has to think about is how many dates should be in there. In our case, we wrote a statement to get all date columns in our OLTP data.

```
SELECT table_schema,table_name,
column_namefrom information_schema.columns
where 1=1and table_schema = 'qimia_oltp'
and data_type = 'date';
```

This can then be used to select all minimum and maximum values from these columns, which should give an appropriate range for the dateDim. To be on the safe side we added ten years of future data. Then we use ***SELECT...INTO*** to store these min and max values in a temporary table ***dwh_dates***.

```
DROP TABLE IF EXISTS dwh_dates;

SELECT MIN(minmax.min) min_date, MAX(minmax.max) max_date
         INTO TEMP dwh_dates
         from (
                  SELECT MIN(employed_since) min, MAX(employed_since) max
                  from qimia_oltp.employees
                  UNION ALL
                  SELECT MIN(opened_since) min, MAX(last_redesigned) max
                  from qimia_oltp.stores
                  UNION ALL
                  SELECT MIN(selling_date) min, MAX(selling_date) max
                  from qimia_oltp.sales
                  UNION ALL
                  SELECT MIN(start_date) min, MAX(end_date) max
                  from qimia_oltp.discounts
                  UNION ALL
                  SELECT CURRENT_DATE, (CURRENT_DATE + INTERVAL '10 year')::date
              ) minmax
;
```

However, one could also use a vast fixed time range. In this case, you would loose some flexibility with regards to changes in the OLTP schema, but you no longer have to process the dataDim every time you run the pipeline. In either case, storage capacity should not be a problem here. In order to populate the dateDim we wrote a function, that basically loops over the defined time range.

```
CREATE OR REPLACE FUNCTION qimia_olap.f_dateDim(start_date date, end_date date)
RETURNS VARCHAR
as $proc$
declare
    date_var date := start_date ;
    date_diff int := (end_date - start_date) ;
begin
FOR i IN 0 .. date_diff LOOP
        INSERT INTO qimia_olap.datedim (
             dateKey
            , date
            , dayOfWeekName
            , dayOfWeek
            , dayOfMonth
            , dayOfYear
            , calendarWeek
            , calendarMonthName
            , calendarMonth
            , calendarYear
            , lastDayInMonth
        )
    VALUES ( to_char(date_var, 'YYYYMMDD')::integer
           , date_var
           , to_char(date_var, 'Day')
           , date_part('dow', date_var)
           , date_part('day', date_var)
           , date_part('doy', date_var)
           , date_part('week', date_var)
           , to_char(date_var, 'Month')
           , date_part('month', date_var)
           , date_part('year', date_var)
           , to_char((date_trunc('MONTH', date_var) + INTERVAL '1 MONTH' -
interval '1 day'), 'day')
           )
    ON CONFLICT (datekey) DO NOTHING;
    date_var := date_var + INTERVAL '1 day' ;
    end loop;
    RETURN 'DateDimFilled';
END; $proc$
LANGUAGE 'plpgsql'
```

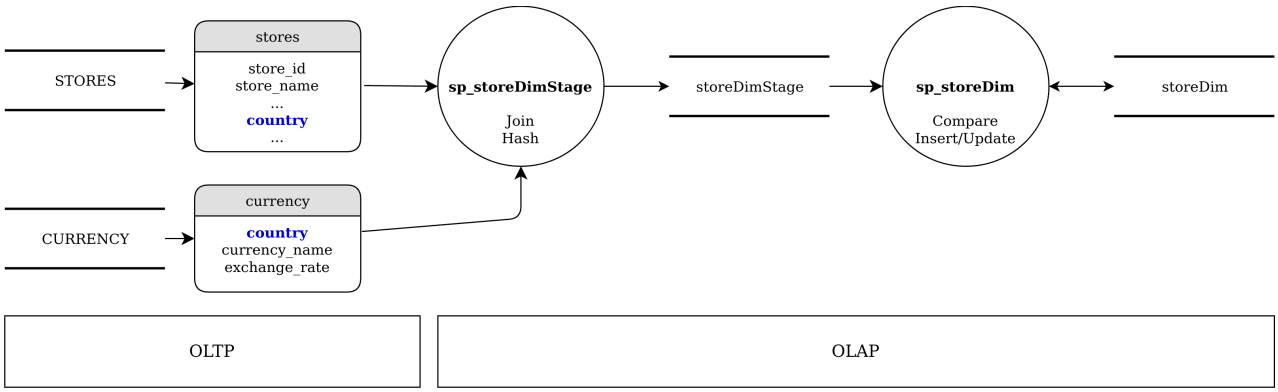
Be aware that the **ON CONFLICT (datekey) DO NOTHING;** allows that the procedure can be called several times with an overlapping date range. After defining that function, the ETL processing is done by calling that function.

```
SELECT qimia_olap.f_datedim((SELECT min_date from dwh_dates), (SELECT max_date
from dwh_dates));
SELECT qimia_olap.f_datedim(CAST('2999-12-31' as date), CAST('2999-12-31' as
date));
```

The date '2999-12-31' is used as a default value for the validUntil columns in the dimensions. Therefore, we need to add this date as an entry in the date dimension as well.

2.2. Other Dimension Tables

For the other dimensions storeDim, productDim, employeeDim, we set up a staging table to simplify the ETL process. We explain that process for the storeDim exemplarily.



Dim Processing

The basic idea is to join all tables needed for the dimension from the OLTP schema, create a hash on that gathered information, and write it to a staging table. From there the data is compared to what is already in the dimension table. If all entries in staging and dimension table are the same, nothing happens. If there are entries in the dimension, which are no longer in the staging, it means, that they were deleted in the OLTP schema, meaning they are no longer valid. Therefore, these entries have to be updated in the data warehouse such that they are also invalid. On the other hand, if there are entries in the staging table which are not yet present in the dimension, they will be inserted. The whole logic of that is done in two stored procedures **sp_storeDimStage** and **sp_storeDim**.

```
CREATE OR REPLACE PROCEDURE qimia_olap.sp_storeDimStage()
language sql
as
$$
TRUNCATE TABLE qimia_olap.employeeDimStage;
INSERT INTO qimia_olap.storeDimStage
( storeKey
, storeID
, storeName
, storeZipCode
, storeCity
, storeCountry
, sellingSquareFootage
, totalSquareFootage
, storeOpenedSince
, storeLastRedesigned
, storeCurrency
, storeCurrencyExchangeRate)
SELECT CAST
    (
        h_bigint(
                CONCAT(
                        st.store_id
                    , st.store_name
                    , st.zip_code
                    , st.city
                    , st.country
                    , st.selling_square_footage
                    , st.total_square_footage
                    , st.opened_since
                    , st.last_redesigned
                    , cu.currency_name
                    , cu.exchange_rate
                    )) as bigint
    ) as storeKey
     , st.store_id
     , st.store_name
     , st.zip_code
     , st.city
     , st.country
     , st.selling_square_footage
     , st.total_square_footage
     , to_char(st.opened_since, 'YYYYMMDD')::integer
     , to_char(st.last_redesigned, 'YYYYMMDD')::integer
     , cu.currency_name
     , cu.exchange_rate
FROM qimia_oltp.stores st
         LEFT JOIN qimia_oltp.currency cu
                   on st.country = cu.country
    ;
$$;
```

In the sp_storeDimStage procedure, we have two commands. First we **TRUNCATE** the staging dimension. Then we **INSERT INTO stagingTable (column1,...) SELECT val1,... from oltpTables**. We do not need any UPSERT logic here, because we truncate all

information in the staging table before. In that procedure, data is selected from two OLTP tables, that are joined together. The values are mostly selected and inserted as they are. However, few of them need some conversion, such as casting from date to integer. The most interesting column is the Key-column. Here we use a custom function **h_bigint** for creating an integer hash value from the concatenated string of the other columns. The function itself is defined from the MD5 hash function.

```
CREATE OR REPLACE FUNCTION  h_bigint(text) RETURNS bigint as
$$
SELECT ('x' || substr(md5($1), 1, 16))::bit(64)::bigint;
$$ language sql;
```

After calling that **sp_storeDimStage** procedure, the staging table will be populated with the most recent data from the OLTP tables.

```
CALL qimia_olap.sp_storeDimStage();
```

The **sp_storeDim** procedure is slightly more complex.

```
CREATE OR REPLACE PROCEDURE qimia_olap.sp_storeDim()
    language sql
as
$$
INSERT INTO qimia_olap.storeDim
( storeKey
, storeID
, storeName
, storeZipCode
, storeCity
, storeCountry
, sellingSquareFootage
, totalSquareFootage
, storeOpenedSince
, storeLastRedesigned
, storeCurrency
, storeCurrencyExchangeRate)
SELECT storeStage.*
from qimia_olap.storeDim storeDim
         right join qimia_olap.storeDimStage storeStage
                    on storeDim.storeKey = storeStage.storeKey
where 1 = 1
    and storeDim.storeKey IS NULL
   or storeDim.storeValid = 'Expired'
ON CONFLICT (storeKey)
    DO UPDATE SET storeValid      = 'Valid'
                , storeValidUntil = 29991231
    ;

UPDATE qimia_olap.storeDim
SET storeValidUntil = to_char(CURRENT_DATE, 'YYYYMMDD')::integer,
    storeValid      = 'Expired'
where storeDim.storeKey in
      (
          SELECT storeDim.storeKey
          from qimia_olap.storeDim storeDim
                   left join qimia_olap.storeDimStage storeStage
                             on storeDim.storeKey = storeStage.storeKey
          where 1 = 1
            and storeStage.storeKey IS NULL
            and storeDim.storeValid = 'Valid'
      )
    ;
$$;
```

Again, we have two commands in there. One for inserting potentially new values (new or modified stores)and one for updating the no longer valid entries. Let's consider the **INSERT INTO ... SELECT** argument first. Here we select everything from the staging table and join it with a **right join** to the dimension. Key column NULLs in the dimension table ('**where storeDim.storeKey IS NULL**') mean that these values are new to the dimension table. Therefore, we have to insert them. Furthermore, we also need to treat the special case, that data can be changed back to a previous value.

The storeDim might not be the best example for that, but in the product dimension, such an event is more likely. Probably the supplier for a product changes and a few months later the supplier is again the old one. However, in the ETL process, we cover that by querying also the non-valid stores **storeDim.storeValid = 'Expired'** and defining a conflict handling:

```
ON CONFLICT (storeKey)
DO UPDATE SET
storeValid      = 'Valid'
, storeValidUntil = 29991231
```

However, such a scenario will cause that there are overlapping time ranges. The original (and new) value are valid from the beginning until '2999-12-31', whereas the intermediate value was valid in a certain range within that range. Depending on the business case one would probably need a more elaborate procedure, that maybe even differentiates between data correction (intermediate value is deleted) and valid data changes (intermediate values are kept). Now, let's go into details on the updating. Here, we select the valid stores from the dimension and left join the stores from the staging table. Filtering that subset on the NULL Key Values gives us the now no longer valid store entries. Those ones are updated, by setting the validUntil column to the CURRENT_DATE value and the valid column to FALSE. We can call that procedure similarly to the storeDimStage procedure:

```
CALL qimia_olap.sp_storeDim();
```

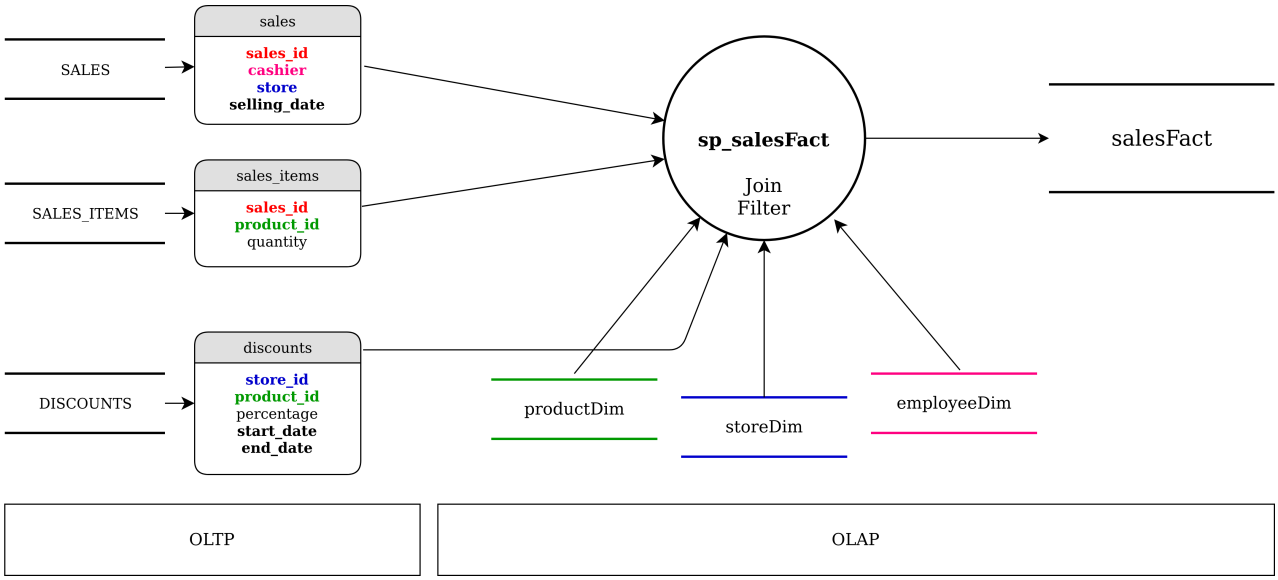
The processing of the other dimensions is quite similar. The only real difference comes when mapping the OLTP data to the OLAP staging table. Here you might experience some other type casts, joins or specific calculation, such as converting the employees salary to a "normed" currency. This completes the dimension processing. In the next chapter we will deal with the two fact tables.

3. Fact Processing

In our DWH Schema, we have two fact tables. One of them is the most atomic fact table, that contains each product that is sold with its respective quantity. The other one is an aggregated fact table, that contains daily assembled data on the product and store level independently of the specific sale to which they belong. Let's consider the way bigger but probably easier atomic salesFact table first.

3.1. Sales Fact

Similar to the dimension processing, we created a stored procedure **sp_salesFact** to extract, transform and load data to the salesFact table.



Fact Processing

The main difference here is that we do not need to hash any sales key, because we do not need to reference a specific sale in any other table. Furthermore, we do not need a staging table as we do not need to compare our previous INSERTSs to the actual one, as they do not have a validity time span. Let's go into the details of that procedure.

```
CREATE OR REPLACE PROCEDURE qimia_olap.sp_salesFact(start_process_date date,
end_process_date date)
    language sql
as
$$
INSERT INTO qimia_olap.salesFact
( salesID
, productKey
, dateKey
, employeeKey
, storeKey
, quantity
, regularUnitPrice
, discountUnitPrice
, purchaseUnitPrice
, totalRegularPrice
, totalDiscountPrice
, totalPurchasePrice
, totalRevenue
, revenueMargin)
SELECT factAgg.sales_id
     , factAgg.productKey
     , factAgg.dateKey
     , factAgg.employeeKey
     , factAgg.storeKey
     , factAgg.quantity
     , factAgg.regularUnitPrice
     , factAgg.discountUnitPrice
     , factAgg.purchaseUnitPrice
     , factAgg.quantity * factAgg.regularUnitPrice
     , factAgg.quantity * factAgg.discountUnitPrice
     , factAgg.quantity * factAgg.purchaseUnitPrice
     , factAgg.quantity * (factAgg.discountUnitPrice - factAgg.purchaseUnitPrice)
     , factAgg.discountUnitPrice / factAgg.purchaseUnitPrice
from (SELECT sa.sales_id                                                    as
sales_id
           , prDIM.productKey                                               as
productKey
           , to_char(sa.selling_date, 'YYYYMMDD')::integer                  as
dateKey
           , empDIM.employeeKey                                             as
employeeKey
           , stDIM.storeKey                                                 as
storeKey
           , si.quantity                                                    as
quantity
           , prDIM.productSellingPrice                                      as
regularUnitPrice
           , prDIM.productSellingPrice * (1 - COALESCE(di.percentage, 0.0)) as
discountUnitPrice
           , prDIM.productPurchasePrice                                     as
purchaseUnitPrice
      from qimia_oltp.sales sa
               left join (SELECT sales_id
                               , product_id
                               , quantity
```

```
                        from qimia_oltp.sales_items
      ) si
                       on sa.sales_id = si.sales_id
              left join (SELECT employeeID
                             , employeeKey
                         from qimia_olap.employeeDim
                         where 1 = 1
                           and employeeValid = 'Valid') empDIM
                       on sa.cashier = empDIM.employeeID
              left join (SELECT storeID
                             , storeKey
                         from qimia_olap.storeDim
                         where 1 = 1
                           and storeValid = 'Valid') stDIM
                       on sa.store = stDIM.storeID
              left join (
          SELECT productKey
               , productSellingPrice
               , productPurchasePrice
               , productID
          from qimia_olap.productDim
          where 1 = 1
            and productValid = 'Valid') prDIM
                       on si.product_id = prDIM.productID
              left join (
          SELECT product_id, store_id, percentage, start_date, end_date
          from qimia_oltp.discounts
          where 1 = 1
            and discounts.start_date <= end_process_date
            and discounts.end_date >= start_process_date
      ) di
                       on prDIM.productID = di.product_id
                           and sa.store = di.store_id
                           and di.start_date <= sa.selling_date
                           and di.end_date >= sa.selling_date
      where 1 = 1
        and sa.selling_date BETWEEN start_process_date AND end_process_date
     ) factAgg
ON CONFLICT (salesID, productKey)
    DO NOTHING
    ;
$$;
```

First of all, the procedure has two parameters, which are **start_** and **end_process_date**. These parameters can be used for flexible scheduling. For example, the procedure could be used in a backfilling scenario. Then the start date would reach back to the oldest data in the OLTP sales table. However, the typical use case would be to have that procedure running hourly, daily, or weekly depending on the business case. Next, the procedure selects the sales from the OLTP schema. They are filtered by their **selling_date**.

```
where sa.selling_date BETWEEN start_process_date AND end_process_date
```

After that, the oltp.sales_items, oltp.discounts and olap.productDim, olap.storeDim and olap.employeeDim are joined to the sales. The discounts are prefiltered to only include the ones that are valid on the processed dates. Similarly the dimension tables are filtered to include the ones with a **'Valid'** validity flag.

```
SELECT ... FROM qimia_olap.productDim
where productValid = 'Valid'
```

When joining the dimension tables, we also load the surrogate keys storeKey, employeeKey, productKey into the fact table. As the last step, we need to calculate a few values from the selected data, such as the **discountUnitPrice** or the **revenueMargin**. However, there are no aggregates. All the calculations are done on the most atomic level. All needed values are then inserted into the salesFact table. Again, we use the INSERT INTO ... SELECT statement. Here, we do not expect any duplicate entries on the most atomic salesID, productKey combination. This is why we state **ON CONFLICT (...) DO NOTHING**. However, if we would expect in our OLTP business process to see multiple entries of the same product-sale pair, we would have to adjust that.

For example there might be some difficult discount logic, such as "buy three, pay two". In this case we would probably find the same product in the same sale, but once with a 0% discount (and quantity = 2) and once with a 100% discount (and quantity = 1). However, such a scenario is not foreseen in our artificial OLTP data.

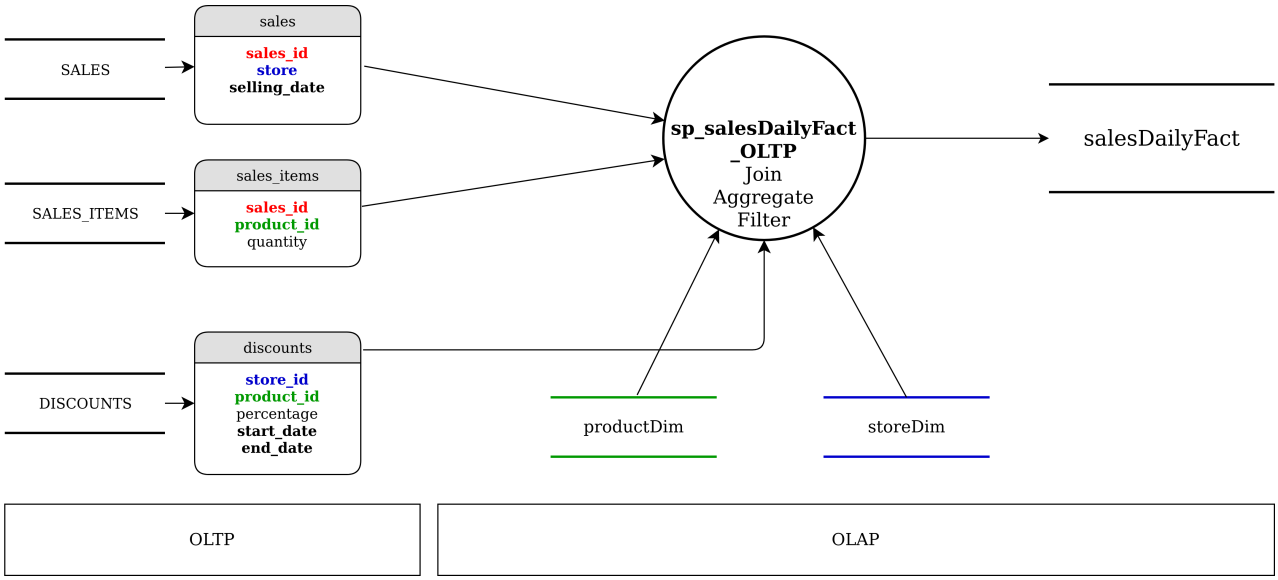
3.2. Sales Daily Fact

For loading data to the **salesDailyFact** we have two options. Either we load the data, similarly to the **salesFact** from the OLTP data or we just aggregate on the **salesFact** from the OLAP schema. The second one is the more stable and easier option. Assuming, an error occurs in the OLTP database or the business process changes, we would have to adopt the ETL process for the atomic data only, whereas the daily aggregated Fact table stays congruent to the atomic one. However, that option is only possible if the table just includes the same values and attributes that the atomic fact provides.

For example an aggregation on the supplier could not be withdrawn from the salesFact solely. Here we present both possibilities. The reader will recognize that the processing from OLTP is almost a copy of the one for the **salesFact**, whereas the processing within the OLAP schema stands out in its bare simplicity. Lets consider the load from the OLTP data first.

3.2.1. Daily Aggregated Fact from OLTP

The following graph visualizes the ETL process that populates the **salesDailyFact** with data loaded from the OLTP schema.



Daily Fact Processing OLTP

Again, we wrapped a stored procedure around the INSERT INTO ... SELECT statement with one parameter being the **process_day** in order to have an ETL pipeline that can be scheduled.

```
CREATE OR REPLACE PROCEDURE qimia_olap.sp_salesDailyFact_OLTP(process_day date)
    language sql
as
$$
INSERT INTO qimia_olap.salesDailyFact
( dateKey
, storeKey
, productKey
, transactions
, quantity
, regularUnitPrice
, discountUnitPrice
, purchaseUnitPrice
, totalRegularPrice
, totalDiscountPrice
, totalPurchasePrice
, totalRevenue
, revenueMargin)
SELECT dailyFactAgg.dateKey
     , dailyFactAgg.storeKey
     , dailyFactAgg.productKey
     , dailyFactAgg.transactions
     , dailyFactAgg.quantity
     , dailyFactAgg.regularUnitPrice
     , dailyFactAgg.discountUnitPrice
     , dailyFactAgg.purchaseUnitPrice
     , dailyFactAgg.regularUnitPrice * dailyFactAgg.quantity
     , dailyFactAgg.discountUnitPrice * dailyFactAgg.quantity
     , dailyFactAgg.purchaseUnitPrice * dailyFactAgg.quantity
     , (dailyFactAgg.discountUnitPrice - dailyFactAgg.purchaseUnitPrice) *
dailyFactAgg.quantity
     , dailyFactAgg.discountUnitPrice / dailyFactAgg.purchaseUnitPrice
from (
         SELECT to_char(process_day, 'YYYYMMDD')::integer
as dateKey
              , stDIM.storeKey
as storeKey
              , prDIM.productKey
as productKey
              , SUM(DISTINCT sa.sales_id)
as transactions
              , SUM(si.quantity)
as quantity
              , MIN(prDIM.productSellingPrice)
as regularUnitPrice
              , MIN(prDIM.productSellingPrice) * (1 - COALESCE(MIN(di.percentage),
0.0)) as discountUnitPrice
              , MIN(prDIM.productPurchasePrice)
as purchaseUnitPrice
         from qimia_oltp.sales sa
                  left join (SELECT sales_id
                                  , product_id
                                  , quantity
                             from qimia_oltp.sales_items
         ) si
                            on sa.sales_id = si.sales_id
```

```
                left join (SELECT storeID
                                , storeKey
                           from qimia_olap.storeDim
                           where 1 = 1
                             and storeValid = 'Valid') stDIM
                          on sa.store = stDIM.storeID
                left join (SELECT productKey
                                , productID
                                , productSellingPrice
                                , productPurchasePrice
                           from qimia_olap.productDim
                           where 1 = 1
                             and productValid = 'Valid') prDIM
                          on si.product_id = prDIM.productID
                left join (
            SELECT product_id, store_id, percentage, start_date, end_date
            from qimia_oltp.discounts
            where 1 = 1
              and discounts.start_date <= process_day
              and discounts.end_date >= process_day
        ) di
                          on prDIM.productID = di.product_id
                              and sa.store = di.store_id
                              and di.start_date <= process_day
                              and di.end_date >= process_day
        where 1 = 1
          and sa.selling_date = process_day
        group by dateKey, storeKey, productKey
     ) as dailyFactAgg
ON CONFLICT (dateKey, storeKey, productKey)
    DO UPDATE SET transactions       = EXCLUDED.transactions
                , quantity           = EXCLUDED.quantity
                , regularUnitPrice   = EXCLUDED.regularunitprice
                , discountUnitPrice  = EXCLUDED.discountunitprice
                , purchaseUnitPrice  = EXCLUDED.purchaseunitprice
                , totalRegularPrice  = EXCLUDED.totalregularprice
                , totalDiscountPrice = EXCLUDED.totaldiscountprice
                , totalPurchasePrice = EXCLUDED.totalpurchaseprice
                , totalRevenue       = EXCLUDED.totalrevenue
                , revenueMargin      = EXCLUDED.revenuemargin
    ;
$$;
```

We observe a few differences to the atomic salesFact. First of all, we do not need to select the employeesDim, as we chose the granularity such, that is does not resolve which item was sold by which employee. Furthermore, we obviously select data of one date and not a time range.

```
where sa.selling_date = process_day
```

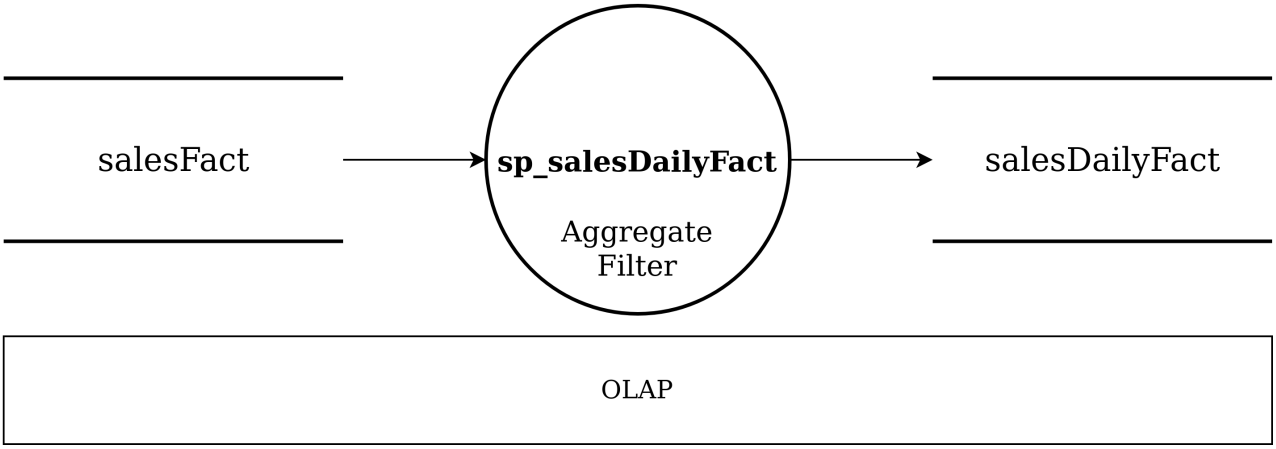
The most crucial difference, however, is that we need to group the data.

```
group by dateKey, storeKey, productKey
```

That allows us to include the summed product quantity, or the summed sales. However, grouping makes it necessary to define aggregate functions for all columns that are not aggregated on. For example the **productSellingPrice**, although unique for a store, date, product combination, needs to be aggregated as well. Here we choose to take the MIN() value as there is no aggregate function like FIRST() or LAST() in PostgreSQL. As a last step, we have to take care of handling PRIMARY KEY conflicts when inserting. Here we choose to use an **DO UPDATE SET ...** statement. The idea behind that is to be able to reprocess data. For example, it could happen that some sales were not processed to the OLTP schema correctly. Therefore, these sales reach the OLTP schema a few days later. Then the **salesFact** has to be reloaded including the new unique sales with unique salesID. However, the **salesDailyFact** is not unique down to that sales level. Therefore, we would need to aggregate the new, complete data and replace the previous.

3.2.2. Daily Aggregated Fact from OLAP

The following graph visualizes the ETL process that populates the **salesDailyFact** with data loaded from the OLAP **salesFact**.



Daily Fact Processing

Accordingly to the simplicity of the graph, the stored procedure that conducts that processing is also quite short and easily readable.

```
CREATE OR REPLACE PROCEDURE qimia_olap.sp_salesDailyFact(process_day date)
    language sql
as
$$
INSERT INTO qimia_olap.salesDailyFact
( dateKey
, storeKey
, productKey
, transactions
, quantity
, regularUnitPrice
, discountUnitPrice
, purchaseUnitPrice
, totalRegularPrice
, totalDiscountPrice
, totalPurchasePrice
, totalRevenue
, revenueMargin)
SELECT sa.datekey                 as dateKey
     , sa.storeKey                as storeKey
     , sa.productKey              as productKey
     , SUM(DISTINCT sa.salesID)   as transactions
     , SUM(sa.quantity)           as quantity
     , MIN(sa.regularunitprice)   as regularUnitPrice
     , MIN(sa.discountUnitPrice)  as discountUnitPrice
     , MIN(sa.purchaseUnitPrice)  as purchaseUnitPrice
     , SUM(sa.totalRegularPrice)  as totalRegularPrice
     , SUM(sa.totalDiscountPrice) as totalDiscountPrice
     , SUM(sa.totalpurchaseprice) as totalPurchasePrice
     , SUM(sa.totalrevenue)       as totalRevenue
     , AVG(revenuemargin)         as revenueMargin
from qimia_olap.salesfact sa
where 1 = 1
  and sa.datekey = to_char(process_day, 'YYYYMMDD')::integer
group by dateKey, storeKey, productKey
ON CONFLICT (dateKey, storeKey, productKey)
    DO UPDATE SET transactions       = EXCLUDED.transactions
                , quantity           = EXCLUDED.quantity
                , regularUnitPrice   = EXCLUDED.regularunitprice
                , discountUnitPrice  = EXCLUDED.discountunitprice
                , purchaseUnitPrice  = EXCLUDED.purchaseunitprice
                , totalRegularPrice  = EXCLUDED.totalregularprice
                , totalDiscountPrice = EXCLUDED.totaldiscountprice
                , totalPurchasePrice = EXCLUDED.totalpurchaseprice
                , totalRevenue       = EXCLUDED.totalrevenue
                , revenueMargin      = EXCLUDED.revenuemargin
    ;
$$;
```

Here, we select the data solely from the **olap.salesFact** and filter it on the **datekey** to fit to the **process_day**. Again, we group by dateKey, storeKey, productKey and aggregate values with the MIN(), SUM(), or AVG() function, where applicable. Also, the **DO UPDATE SET** is an exact copy of the one in the previously described procedure.

All in all the ETL pipeline that reads from the OLAP salesFact should be preferred. One could even create a materialized view if the data is just another representation of a certain table. This completes the ETL processing article and therefore our blog on data warehousing.

Wrapping It Up

Putting it all together, we gathered data warehouse principles, such as what is a star schema, how to define the grain of a fact table, and how to identify dimensions and facts from the business bus matrix.

In the next step we saw how to create a proper DWH schema from a given artificial, but realistic OLTP dataset.

Furthermore, we presented details on the schema, meaning how to create dimension and fact tables and how to choose relations or update procedures.

In this last article, we saw how to populate the star schema with an ETL pipeline based on PostgreSQL stored procedures.

With this, we would like to thank you for your interest! See you next time and stay tuned for further articles!

Sources

- PostgreSQL - Aggregate Functions
- PostgreSQL - INSERT with conflict handling