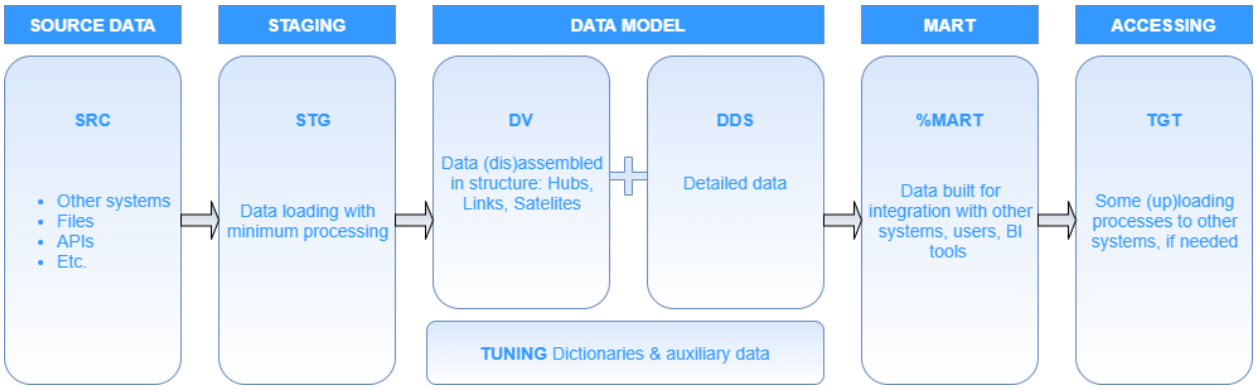
ETL pipelines directions from source data to users (applied to the Vinted data on the next page):



Shortly about the concept:

1. **Source data (SRC)**
   Just the data outside the Data warehouse, which is loaded into it.
2. **Staging data (STG)**
   Data uploaded into the DWH with minimum processing. If the DWH is wanted to be the unified reliable data source, all the incoming data should be accessible before the transformations.
3. **Data model (DV + DDS)**

3.1. **DV**: Building the data model, here according to some standards from the Data Vault methodology: H - hubs – store business keys, L - links - store connections between hubs, S - satellites - store keys attributes/descriptive data. Why this methodology:

- relatively easy in modeling
- data is structured
- getting rid of excessive data
- really scalable
- easy to load new data and add new data sources
- easy to build marts for users

But there could be some cons:

- many joins (which would increase building time e.g. in Hadoop)
- necessary to build marts for users/integration/bi tools etc. (but it's probably the purpose of the job so...)

3.2. **DDS**: Detailed data storage. Purposes:

- Can be used when there is no real purpose to disassemble incoming data into DV structure (e.g. provider prices). Also, it can be an intermediate layer between staging data and marts, like provider prices. I don't think that staging data should be used in data marts building, so it should be "approved" in data model like dds_provider_prices – see in the picture below.
- DDS tables can serve as a source for various reporting/integration marts. Shipment data can be used for some reports for stakeholders (REPMART), it also can be used for integration with some other systems (IMART). Data like dds_shipment can be used as a source for both and there would not be a need to build them twice.

3.3. **TUNING**: small dictionaries and some auxiliary data (in this case suitable for storaging unique product package descriptions).

4. **Data marts (%type_of_mart%MART)**
   Is built on modelled data. Ready-to-consume data for users (USERMART), integration (IMART), reporting tools (REPMART) etc.
5. **Access for the data (TGT)**

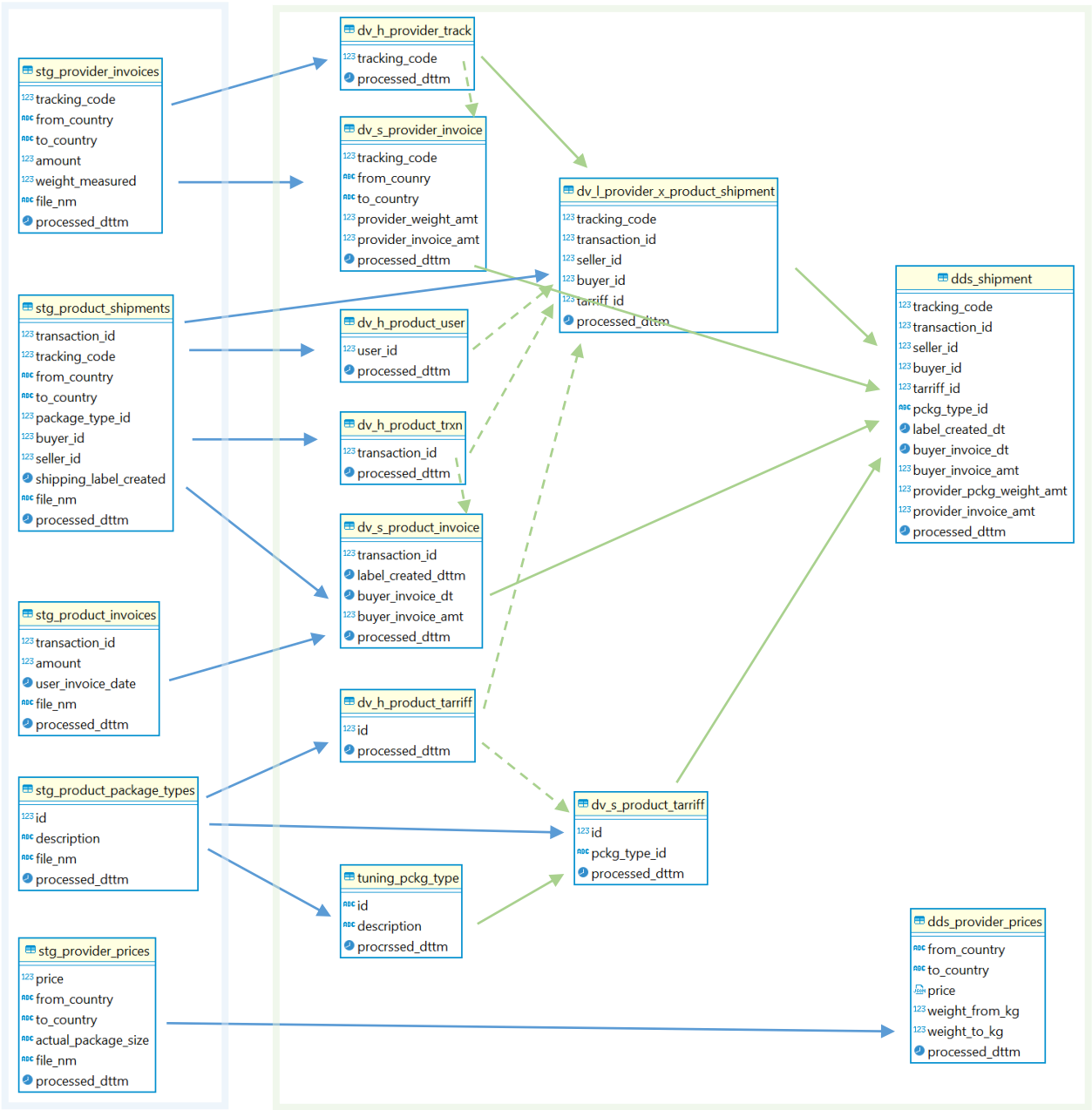
Data flows only move forward, not backward, from earlier layers to the latest ones (from STG to DV to DDS, not from DDS to DV to STG), to keep everything in order and avoid confusions.

Data flows for shipping prices:

Data uploaded from files directly to STG is transforming into Data model according to the approach descripted above. Here are not shown SRC files (it's easy to imagine them in the beginning). Also here are not presented Marts, but it's easy to build them on the data from green (data model) zone. Blue lines are for data which is coming from raw data, green lines – for the data coming from modelled data. Raw data is becoming a part of data model.

Notes: I assumed that product_package_types are more tariffs than actual package descriptions, so I decided to give them a personal hub. Descriptions are represented in a tuning table.

Naming: Layer + (if applicable) Area or source + Entity



(A kind of ER diagram from my postgres but for data flows)

The picture can look complicated for 5 data sources, but keeping in mind company's rapid growth (and data amounts increase) this structure would fit scalability requirement. Hubs would allow to store keys from various sources (providers, tariffs), only src_system should be added. Link stores all the keys. Satellites store keys attributes. Tuning would help to order some characteristics.

To prevent storing yet non-existing in the hubs keys in links & satellites (1) hubs can be also a source for building them or (2) retain keys system could be implemented: creating and writing new retain keys would be allowed while uploading new keys to the hubs, and using only existing retain keys while uploading to the links and satellites would be allowed.

If we look at the picture of tables and their dependencies above, it's easy to see there are many jobs would build them and they need to execute gradually. Such **scheduling could use topological sort graph algorithm to execute tasks in a proper order**. It's commonly used in scheduling tasks with many dependencies.

Incremental load

If keeping rapid growth of data in mind, incremental load is a necessity. It's valid here for the tables like Provider&Product invoices, Transactions. The target tables would be loaded with Update/Insert or Delete/Insert methods to load fresh updates.

However, for some tables incremental load could take more time than loading without using one (e.g. tables like dds_provider_prices - it's easier just to replace the data (Truncate + Insert) if it's not stored under SCD2).

Here I would suggest using date or timestamp fields for incrementing - processed_dttm or some business dates if needed. How a simple table UTILS_CUT_TABLE for incremental loading mechanism could look like:

field_name	description
job_name	Name of the job that uses incremental loading
source_name	Name of a source table that is incremented (not all source tables can be)
field_name	Field in table that would be used for selecting an increment
cut_value	The date/timestamp value. This value would be applied in a "where field_name > %this value%" clause for the source_table when the job runs next time Null, if the source has not been incremented yet
processed_dttm	date of adding a record

In each job mentioned in UTILS_CUT_TABLE increment would be applied to the souce_tables table like `select * from source_table where processed_dttm > cut_value --latest by processed_dttm cut_value date` and only this piece of data would be used during the running job.

In case when it would be needed to reload the whole source table: a new record with a very early cut_value date (e.g. 01/01/2000) could be added to the UTILS_CUT_TABLE table, so next time when a job is launched the increment mechenism would select all the records from source_table where processeed_dttm is greater than 01/01/2000. There is no need to clean cut history (can be used for occasional errors when data was lost somewhere), bc incremental mechanism would always choose the latest by processed_dttm cut_value.

Obviously this tool can be adjusted for using not only with tables, but also with source files.

Also more complicated versions of incremental mechanism could be implemented.

For example, if we would use incremental load in dv_s_product_invoice, we'd take a fresh small part from stg_product_shipments, but also we need to use in the job stg_product_invoices with the same key for joining. I'd call such a mechanism a Single Key Cut, when an increment is selected from one table (stg_product_shipments), and the keys from that increment are selected from a related table (stg_product_invoices).