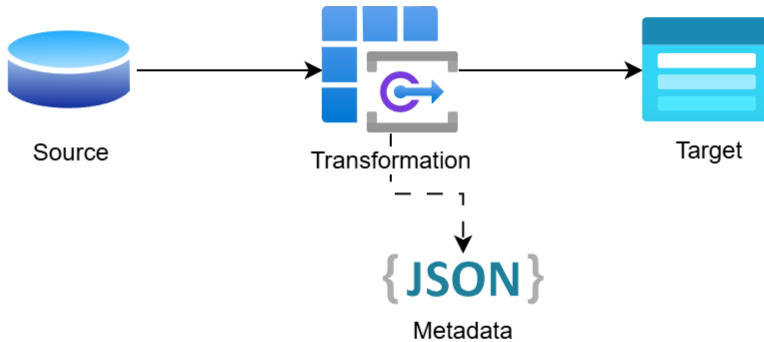
Implementation

Every data movement follows the same principle, source data is transformed and made available in the target and along the line some side effect data is generated also.



This applies to ETL flows, where a program is called at specific times reads the change data from the source, applies transformations and the result of the transformations are stored in the target.

But the same can be said about data federation (Fabric Shortcuts), where data is read from the target and the request is re-routed to the source dynamically.

Or realtime replication where the source pushes changes to a consumer, the transformation is a pure 1:1 mapping and the data physically copied into the target.

Note that "source" does not mean necessarily a remote source system. It could also mean that two tables in Fabric Lakehouse are joined and put as a fact table into Fabric Warehouse.

Having said all of that, one decision was that the target is always a physical persistence, a table, for query performance reasons.

When we now consider the different tools and methods, this leaves to some obvious and less obvious requirements. Each method must be able to ....

- [ ] connect to any kind of source system. For example, StarTek is an onPrem Rest API, Labware an onPrem Oracle database, SAP Datasphere a cloud SaaS offering, ... A method that does not allow for all current and future sources cannot be used.
- [ ] support any kind of transformation. SQL like transformation like joins, projections, filter, aggregation, window functions. But also transformations and functions outside of the SQL realm, like calling a google maps API to get back long/lat for a given address or transformations implemented in 3GL languages. A method that does not support 100% of all current and future transformation requirements cannot be used.
- [ ] provide low latency integration, say less than 1 minute delay, between a change in the source and the data arriving in Fabric Lakehouse - obviously source system specific.
- [ ] load the target tables with a commit timestamp and a soft delete flag. The commit instead of the change timestamp is important, because otherwise if a change was made at 09:00 and committed at 09:15 would not be visible for a delta run at 09:10. And the next delta run will read all changes from 09:10 onward and hence skip the row changed at 09:00. The requirement is simply, we copied the sales order header and sales order line item table into Fabric and now those two tables get joined to load the sales order fact table. This must be implemented via a delta logic.
- [ ] be implemented quickly, for simple and complex logic. Transformation logic has the same requirements as classic programming languages: modularization, reuse of common code, inline documentation.
- [ ] copy not only the data but also the metadata. For all 1:1 mappings the column metadata like exact datatype (NVARCHAR(1), not just String) is preserved as well as column comments, for more complex mappings the metadata can be derived (case when the col1 else col2 end; data types and comments can be derived). At table level the metadata is table comments, partition information, primary key, foreign keys.
- [ ] generate the impact/lineage information on table and column level from the sources to the reports. The user must be able to answer questions like "Where is the source table being used?", "If the source modifies the definition of a column, what target tables and columns are impacted?", "The value in the target looks wrong, where does it come from and what kind of transformations have been applied?"
- [ ] support reading and transforming deeply nested structures, e.g. a Json document with multiple levels or the result of a RestFul API call.
- [ ] process the data fast, throughput should be in 10k rows per second and better.
- [ ] process the data with little costs, meaning the cost of running the dataflow should be fractions of cents per million of source rows. For example, starting an entire Spark cluster, just to be told that the source has no new records, would be quite an overhead. Even if Fabric encapsulates all, we still pay per CUs.
- [ ] work with a CICD pipeline. A developer works on the code, saves it in an incomplete state but that does not interfere with the currently deployed version. When he commits the code it gets deployed. When the code is committed into the prod branch, it is ready to be deployed and is deployed at a suitable time manually by somebody else.
- [ ] work with git. Two people working on the same artifact at the same time and later the code is merged. History shows what has been changed. Code reviews with pull requests.
- [ ] provide operational statistics. How many record read, transformed and written in what time. Provide measures to identify hot spots (profiling info).

- [ ] proactively trigger alerts when something happens, run time too long, error, daily delta not executed although it should (somebody deactivated the schedule?).
- [ ] differentiate between temporary technical errors (e.g. source system down) and data errors (e.g. field missing).
- [ ] recover from errors automatically for temporary technical errors and notify support about data errors immediately.
- [ ] support schema evolution out of the box.
- [ ] trigger all dataflows that depend on the loaded target tables, so that they can process the changes.
- [ ] provide data sensitivity and applicable data regulation for each target table.
- [ ] support test automation. For example, instead of reading the current data in the source, use a mock source with stable data, apply all the transformations and load into a mock target to compare the actual result with the expected result via asserts.
- [ ] support switching from initial to delta loads and re-run an initial load of all or parts at any time.

At the end it is a cost comparison question.

- Speed and hence cost of development.
- Availability and cost of resources with a certain skill set.
- Number of resources required to keep the system alive.
- Cost of maintaining and enhancing the system, e.g. source got a new column.
- Agility to react to new demands.
- Reuse of the data for other projects.
- Scaling the cloud resource to the demand, e.g. during initial load a massive amount of resources will be needed, the daily delta little and if there were no changes in the source the data movement should cost zero.

By using Python and DuckDB, we get the flexibility of Python with the processing power of an in-memory, columnar ANSI SQL database. Coupled that with Kafka as event broker to trigger dependent processes checks every single requirement and enables a future proof solution.