Chapter 3 - Data Ingestion

The data ingestion step is where Data Engineering and Analytics Engineering overlap. While a Data Engineer cares more about the overall architecture, its cost efficiency, or its structure (monolith vs. microservices etc.), the Analytics Engineer cares more about the individual pipelines: Which data do I get, what is its lifecycle, which permissions or compliance go with it, which format does my data warehouse want the data in, which columns do I need for a certain metric, etc.

Data ingestion involves a source system—e.g. a CRM software, an database, or some third-party API, and a destination, mostly a database or a cloud-storage that a datawarehouse can access.

Fundamental Differences between Source Data Systems and Analytics Systems

These systems—and their owners and stakeholders—often have differing foci and concerns.

- **Data operation**: Source systems care about writing and updating records, while Analytics systems care about reading and aggregating data.
- **Database types**: Source systems often run on row-based OLTP databases or document storages, e.g. MySQL, PostgreSQL, MongoDB, or SQL Server, while Analytics systems run on columnar Data Warehouses like Snowflake, BigQuery, Redshift, Vertica, or Clickhouse.
- **File formats**: Source data often is lightweight and use to write and update, such as JSON, XML, or CSV. Analytics files heavily utilise compression and schema enforcement such as Parquet.
- **Focus of owners and maintainers**: If you are responsible for a source system, you care about the accuracy and correctness of individual records, each record needs to have the relevant properties and metrics, they need to be complete and consistent. If you are responsible for Analytics, you care about the aggregate of the data. A faulty record is not the end of the world, as long as a certain threshold is maintained. Instead, you want to make sure that your pipelines run smoothly and your data does not take up too much space or computation capacity.

The 8 Essential steps of a data ingestion pipeline

1. **Trigger**: How and when does a pipeline start?
2. **Connection**: What are the requirements for connecting the source system or database to the target or destination?
3. **State Management**: How do you keep track of what has and has not changed since your last ingestion? How do you handle changes to the data over time?
4. **Data extraction**: What is the actual process of extracting data? Do you query a database, access an API, or just read files?
5. **Transformations**: Is there any filtering, cleaning, reshaping, deduplicattion, or similar process that you need to apply to be able to move your data from the source to the destination?
6. **Validation and data quality**: How sure do you want to be that the data is correct? What kind of guarantees do you want to enforce and what kind of anomalies are you willing to accept?
7. **Loading**: Where do you store the data that you captured? How do you handle connecting and adjusting to your destination system?
8. **Archiving and retention**: How long should you sttore or keep the data you just captured? How do you handle expired data?

The 3 overarching topics

- **Scalability and resilience**: As the data grows over time, are you still able to handle the additional load? What is the fault tolerance ofthe entire pipeline and how often does it retry before giving up?
- **Monitoring, logging, and alerting**: How do you make sure you are aware and able to debug when there is an issue anyhwere in the ingestion process?
- **Governance**: What kind of agreements do you have with both the source data provider as well as the destination ownr? Are there any legal requirements you need to consider when handling this data?

Pipeline components

Now we are going through a data extraction / ingestion pipeline and talk about its individual components

Trigger

A pipeline can be triggered by two ways: On a **regular schedule** or on an **irregular event-basis**.

Schedule triggers: They are easy until they're not

If you schedule your pipeline to run regularly, the classic way of doing so was the **cron job**: You just specify an interval, like *daily at 10am*, or *every wednesday at 5 past the hour* and then you just check if the jobs were executed.

> **Downstream jobs need to wait when upstream jobs are late**
>
> If you have a sequences of jobs with strict dependencies (C can only happen once A and B are finished), then late cronjobs cause the whole DAG to break, especially if you can execute multiple jobs at the same time. When C depends on A and B being finished, and C has regularly started but A has NOT finished, C might execute with old data or it might break entirely.
> Thus, notifying downstream jobs and postponing them is important in some pipelines.

Thus, schedule triggers are good for initial jobs in a pipeline, such as "start pulling data from the source system at 1am", but should be avoided when there are hard dependencies in downstream jobs. And even for initial jobs, you might be able to find some event like a web hook or so to turn this into an event trigger, if it is not too complicated to create such an event.

Event Triggers: More complex but secure.

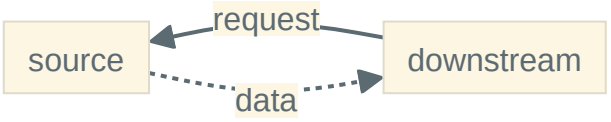
Orchestration tools like Airflow excel in handling task dependencies. They can notify and postpone downstream jobs, and they can re-try a failed job. For this, they need to register trigger events. Other than the simple "upstream job has finished" case, there are various other possible **event triggers**:

- New file added to watched folder
- New table added to database
- HTTP request to API/webhook
- Log or audit events (e.g. permission granted)

- Billing or consumption threshold reached
- Queue has reached a specific number of items
- Branch of version control system has changed

Event triggers can be **push-based** or **pull-based**. In the former case, the source system or upstream data source actively **pushes** data into the next stage, or notifies the next stage that the data is ready. In the **pull** case, the downstream model actively has to query the upstream source and ask for the data. Here, the upstream source remains passive and the downstream source has to **pull** the data by itself.
Push:

Pull:



Example

> We have discussed with the CEO that daily updates are enough for the insights. We already have a schedulingtool such as Airflow available and will use that to trigger our data ingestion functions at midnight.

Connections

Some source data are public such as Weather APIs or so, but most source data will require authentication: Your ad cost figures, your customer data, or your sales figures. Thus, if your pipeline wants to fetch this data automatically, it must authenticate with your source system. And indeed, it is a common hurdle and source of problems. Often, it is a conflict of interests: The source data providers want to be as safe as possible and introduce additional security features, whereas the downstream consumers want easy and stable access to their data.

⚠ Common authentication issues:

- Permissions for service account were revoked
- A personal user account was used and that person left the company

- New security features such as MFA were added

As a best practice, every tool accessing your data should have its own service account, or at least permissions level, so only necessary permissions can be granted.

Common types of connections

- HTTP `GET` or `POST` requests with an authentication token in the header
- **ODBC** (Open Database Connectivity) connection: Standardised way of connecting to most database systems.
- **gRPC** (Remote Procedure Calls) connection: A special communication protocol designed by Google using Protocol Buffers, mostly used in the context of small, independent services.

Example

> The connection for our CRM tool is simple. The tool itself proies daily exports as Parquet files o a storag bucket in our cloud provider. We make sure that the specific storage bucket we use only serves his tool to avoid any cross-contimation in temrs of security, and for authentication the tool provides a service account to which we can give access.

> For our ERP system, things are a bit more complex. We need to create a separate user in the system that only has view permissions, and the serverless function that we use first needs to connect to the on-premise server via a VPN as it cannot be reached from the outside. We store the credential for the user we created in the credential manager of our cloud provider. That way, we never have to reference a credential in our code.

State management

As hinted at earlier in the discussion about re-trying failed tasks and handling downstream dependencies, managing states is important in a data pipeline. Things don't always go according to plan and thus knowing which part of a task has been completed or which part of the data has already been imported is crucial for re-trying failed tasks.
This also involves caching or other means of not having to re-run the ENTIRETY of

your pipeline. This might be especially important for **expensive operations** such as pulling data from a **rate-limited API** or computationally expensive tasks when you run them on a cluster in the cloud that is billed by CPU usage.

Examples of state management in data pipelines

- Keeping track of the status of data extraction: What has been imported already?
- Keeping track of failed jobs and their retries
- Keeping track of pipeline job metadata such as start time, end time, and the elapsed time to compare different runs over time
- Tracking the diffs from a source system itself for auditing purposes.

Example

> With our CRM tool, there is not much we have to do in terms of state management. The tool itself will do a full export every day. For our ERP system, we need to be more caeful, however. Since the system is on-premise, resources are limited and we cannot just export everything. Our function will write to a manifest if the extraction for a specific date was successful. We can then use this manifest to retrieve the latest date, and in our ERP system, only select data from after that date. This way, we will limit the amount of data the system has to process

Data extraction

(These notes will not go into too much detail here, as much of it is already described in even more detail in the Data Engineering books in these Booknotes.)

Takeaways from Data extraction

The main task of data extraction is: Getting the source system to spit out my required data in a format that is the closest to how I need it. This involves

- the grain: How granular and broken down can I get the data? Can I get them *en bloc* or are some breakdowns mutually exclusive? Do I then need to extract them to multiple tables?
- the timespan: Can I query all my data or is there a limit for re-fetching historical data?

- Rate limits: Can I get as much data as I want or is there some hourly / daily limit?

Example

> As we've already established, our CRM tool will provide us with Parquet files. We can then write a SQL statement in our data warehouse to use this storage bucket as a so-called extenral table. We refresh this table daily, which allows us to query the Parquet files directly with SQL. This way, we can easily model the data and apply our business transformations while keeping the source data intact.

> For our ERP system, the extraction is a little bit harder: First, we set up what is called **peering** to connect the on-premise VPN to our private cloud environment. This allows us to connect cloud providers such as AWS, Azue, or GCP to the on-premise server. We can then use a Python library such as `pyodbc` to connect to the Microsoft SQL Server that the ERP system uses. We use a separate storage bucket for our manifest file t keep track of the last successful extractions and retrieve that to determine the start date in our SQL query. Besides the manifest, we keep another file, this time in our version-controlled code. This file contains all the databases, tables, and columns that we want to xtract as YAML code. This way, we will not have any trouble with newly added or changed columns that we do not need, while still being able to extract what we want in a controlled manner, tracking changes over time. As an added beneit, we can use this to create tables to load our data in our data warehouse as well as validate the data we ingest.

Data transformation

(Same here, most of it is either commonplace or described more in-depth elsewhere.)

Although most transformation steps are not executed in the data extraction pipeline itself, there are some basic transformations that are worth considering in the pipeline itself, as they might ease the weight of the pipeline and save cost or prevent easy-to-prevent errors:

Possible extraction/ingestion transformations

- **Filtering**: If you know there is data you will never need (such as superfluous columns or the like), it is better to never even ingest them in

the first place.

- **Light Cleaning**: This involves naming columns to prevent confusion or correcting simple delimiter or escaping mistakes in strings.
- **Joining / Denormalising**: Some source systems are heavily normalised and data warehouses tend to be more de-normalised so that reducing the number of tables to ingest by joining some prior to ingestion can be beneficial.
- **Enriching**: While most enrichments and cross-system joins are best performed as late as possible, some extra info will simply not change or will be easy look-ups. Examples include geolocations of IPs, the current timestamp, or the weather of the day.
- **Reshaping**: Some operations such as pivoting, or flattening JSONs are easier in languages like Python, so performing them in the ingestion step might be easier than using SQL in the Data Warehouse for this.
- **Deduplication**: Enforcing primary keys as early as possible is generally a good practice and you cannot start doing that too early in the pipeline.

Example

> Since our CRM tool is easy to load into our data warehouse, we will apply all transformations there. For our ERP system, the requirement is to not make any sensitive customer data available in our analytics system. Therefore, we will use the Pydantic implementation to apply a hashing function to any fields marked as sensitive in our YAML schema file.

Validation and data quality

Checking data quality, again, is a step that stretches to all parts of the data lifecycle and is not exclusive to the ingestion pipeline. Concretely in the pipeline, data validation focuses on **checking and enforcing the data schema** as well as **anomaly detection**.

(Schemas are heavily discussed in Data Engineering, so we will skip most of it here.)

Examples of post-ingestion data validation checks

- Enforcing a data schema
- basic integrity checks (primary or surrogate keys)

- Integrity checks (integrity between multiple normalised entities)
- anomaly detection (how much deviation from an average will we tolerate before sending an error?)

Example

> For our CRM tool, we have the benefit of Parquet files having an emebdded schema. This means we do not have to check indiviual valus; however, we dno need to check whether the schema of the ltest Parqurt file matches the schema of earlier Parquet files. Since this source is very reliable, we habe decided to only monitor for errors when the files are processed in our data warehouse.
> For our ERP tool, we can use the defined YAML schema and load it into a widely used Python library called Pydantic to valiate out data and apply necessary transformations.

Loading

The main point is that you don't need to ingest all data into your data warehouse directly (as tabular data). Most modern data warehouses can import Parquet, JSON, Avro, or CSV files directly as an **external table**. You can ingest your data into your data lake and leverage e.g. the compression properties of Parquet until you stage the contents in your data warehouse (sometimes referred to as the Data Lakehouse).

> **Quote**
>
> If you think about it, it is not hugely different from an actual data warehouse. You could think of a table in your data warehouse **as if it were based on files in folders**. If you apply partitioning to a column for example, a column with dates - each date will be a separate folder. If you also apply clustering - for example, on country- there could be another set of sub folders for different countries. Depending on the size of your table, you could split the data in the folder into one or more files that can then be processed independently when you execute a query. For example, looking up the sum of a metric, let's say sales, for a specific country in a specific date range means you can now just grab all the files from the relevant folders, calculate the sum per file independently, and then add the results together to get the final sum of sales.

Example

> Loading data is easy for our CRM tool. By using the concept of external tables, our data warehouse can connect to a storage bucket and read Parquet files directly. This way, we do not need to explicitly load data into our data warehouse. For our ERP system, it is a little bit harder. First of all, we need to use our schema file to make sure that tables and columns already exist in out data warehouse. If they do not, we'll have to create them. Then, we need to decide whether we want to load rows one by one, in a batch, or through a file. Since this is not transactional data such as sales transactions and our columnar data warehouse is not made for performance on transactions, it makes sense to use batch loading. Since we also want to minimise the load on our ERP system, we will first tore our extracted rows as Parquet files in a storage bucket. In this case, if anything happens during the loading phase, we do not have to go back to our ERP system but can leverage the stored Parquet files.

Archiving and retention

As the last part of the lifecycle, also your extracted source data needs some archiving, retention, and deletion strategy. Source data is especially relevant for backfilling, i.e. when some major logic has changed and you need to re-calculate some important metrics of historical data.

Example

> For both our CRM system and our ERP system, we care mostly about the analysis and insights we can get from the datam not the actual data itself, with all the potentially sensitive information such as e-mail addresses or invoices. Therefore, we decide to let our raw data expire after 30 days. this way, we can still reuse the raw data whenever there is an error during the loading, but we do not carry as much sensitive information and can also guarantee that, fir example, any data deletion request from a customer is completed within 30 days.

Managing quality and scalability of pipelines

As the load on the pipeline increases, the more it needs to adapt to handle the requirements of bigger loads.
Initially, most pipelines (especially if they are ETL) start off as a single-threaded application like a Python script. As the load increases, the more individual parts can break or the worse it is IF individual parts break. Thus, breaking up your pipeline scripts into **modular functions** helps scale the pipeline.

Another important feature of making scalable pipelines is to re-try jobs if they failed. If you have a lot of jobs then going through each of the failed jobs can be tedious, espcially because they sometimes just need a restart to work. A handy strategy for this is calld **exponential backoff**: the times between retries scale expnentially, e.g. 1 minute --> 4 minutes --> 16 minutes --> ...

Monitoring, logging, alerting, governance

Obviously, you want to be properly alerted about failures on an illustrative interface. You also want to store the logs of your jobs in a handy place where you can easily access them for troubleshooting.

On the Governance level, you want to establish treaties like a Service-Level-Agreement (SLA) with stakeholders so you have a benchmark for how old your stale data is allowed to be. This helps tremendously in triaging your errors and gauging their severity.