Sourav Banerjee Aug 26 7 min read

Data Engineering Design Principles - Design Patterns for Data Pipelines

Updated: Aug 27

In software engineering, design patterns are repeatable solutions to common software design requirements. A design pattern is a type of abstraction that does not directly translate into executable code. It is a problem-solving template that can be used to build a solution on. Design patterns have numerous advantages, including reuse of thought processes, foundations for reusable code, accelerated design and development, and increased consistency and ease of maintenance.

> Perhaps it is time to incorporate pipeline design patterns into data engineering as a core discipline.

Some of the characteristics of Data Engineering Design Principals are:

- Reusable
- Accelerated
- Consistent

VARIETY OF DATA PIPELINES

A data pipeline's purpose is to transport data from one location to another. There are numerous types of data pipelines, including those for integrating data into a data warehouse, ingesting data into a data lake, and delivering real-time data to a Machine Learning application, among others. The variation in data pipelines is determined by several factors that influence the solution's shape. These elements are as follows:

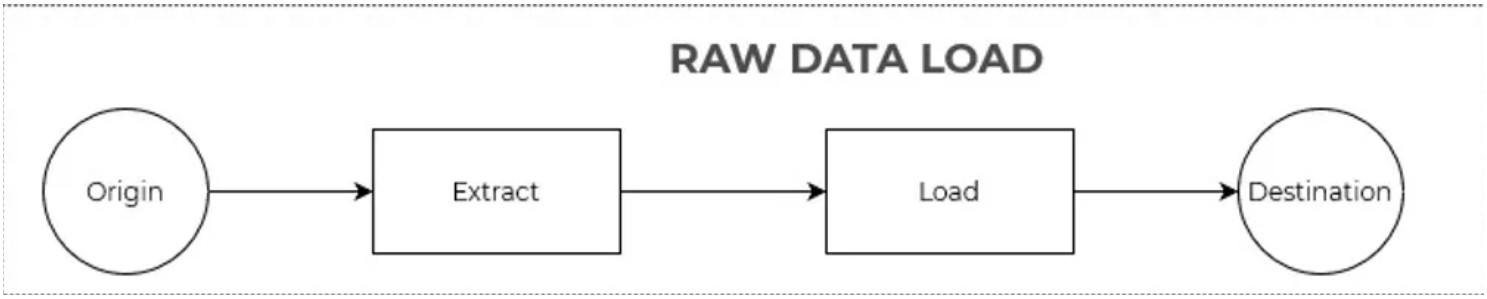
- What is the purpose of the pipeline?
- What data is being moved?
- From where to where is it being moved?
- Is this happening only once or repeatedly?
- What is the anticipated data volume?

- Is the data being stored or is it being streamed?
- Is it organized or unorganized?
- Usage — For what use cases is the data intended?
  Reporting
  BI
  Analytics
  Data Science
  AI/ML
  Automation
  Robotics
  IoT

> Considering these factors, it is obvious that there isn't a one-size-fits-all data pipeline. The data flows that load data from an ERP (Enterprise Resource Planning) system into a data warehouse are distinct from those that feed a real-time dashboard used to monitor a manufacturing facility. It is unlikely, however, that you'll encounter a data pipeline so unique that it has no similarities to other data pipelines. Those similarities are the basis of design patterns. With that in mind, I propose eight fundamental data pipeline design patterns as a practical place to start bringing the discipline of design patterns into data engineering.

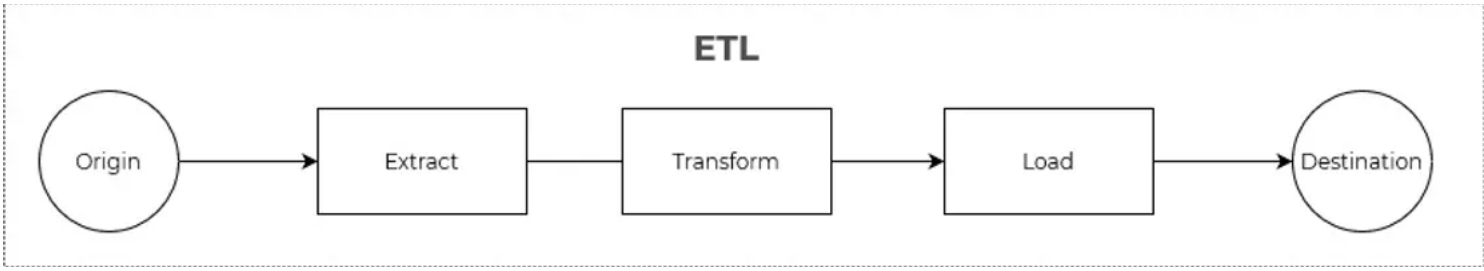
1. RAW DATA LOAD

A raw data load pipeline is built to move data from one database to another. These pipelines perform the bulk data movement that is needed for the initial loading of a database such as a data warehouse, or for migration of data from one database to another—from on-premises to cloud, for example. The term "raw data" implies data that has not been modified, so the raw data load pipeline pattern consists of two processes—extract and load—with no data transformation processing. Data load processing can be slow and time-consuming, especially with large data volumes. They are typically built for one-time execution and are rarely run on a periodic or recurring schedule.



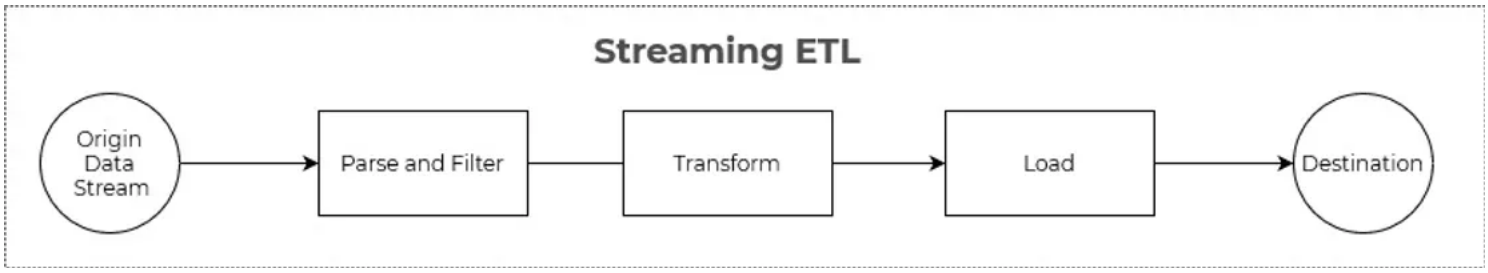
2. ETL

Extract-Transform-Load (ETL) is the most widely used data pipeline pattern. From the early 1990's it was the de-facto standard to integrate data into a data warehouse, and it continues to be a common pattern for data warehousing, data lakes, operational data stores, and master data hubs. Data is extracted from a data store such as an operational database, then transformed to cleanse, standardize, and integrate before loading into a target database. ETL processing is executed as scheduled batch processing, and data latency is inherent in batch processing. Mini-batch and micro-batch processing help to reduce data latency but zero-latency ETL is not practical. ETL works well when complex data transformations are required. It is especially well-suited for data integration when all data sources are not ready at the same time. As each individual source is ready, the data source is extracted independently of other sources.



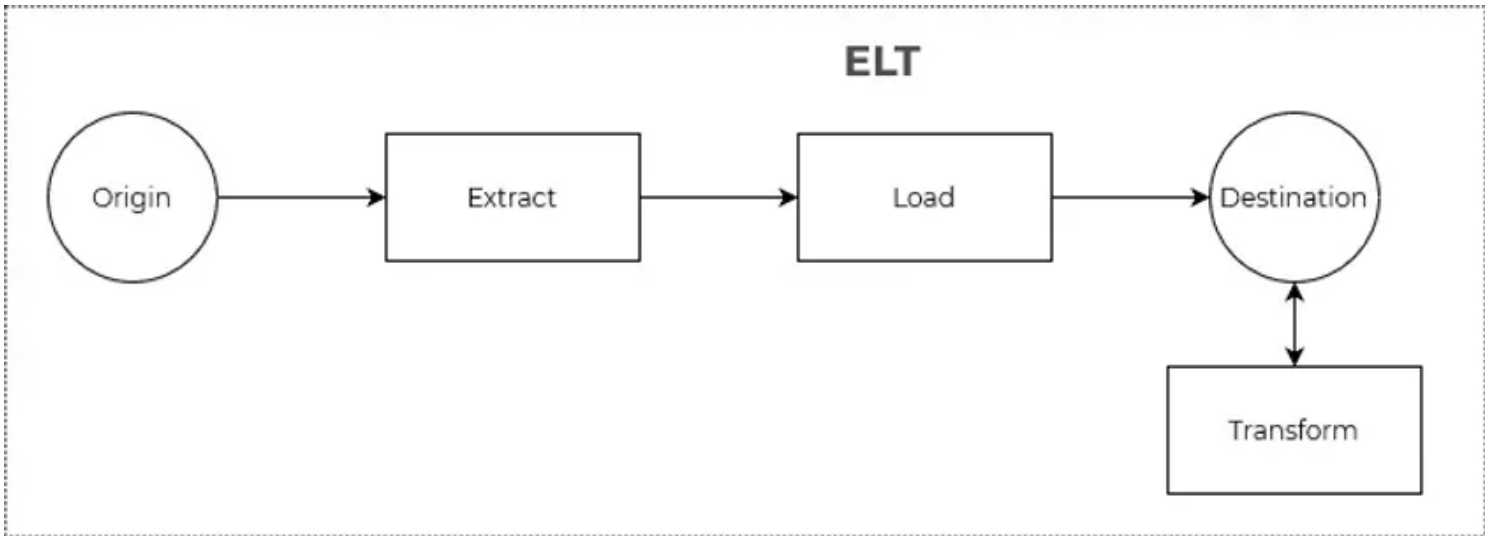
3. STREAMING ETL

The ETL pattern that is typically used with stored data can also be applied with some variation when the source is a data stream. Instead of extracting from a data store, streaming data is parsed to separate individual events into unique records, then filtered to reduce the data set to only the events and fields of interest for the use case. Parsing and filtering are followed with transformation, and then data is loaded into a data lake. Batch processing is replaced by stream processing using a stream processor such as Kafka. This pattern is particularly useful for machine learning use cases that often focus on a only a few fields in much larger data sets.



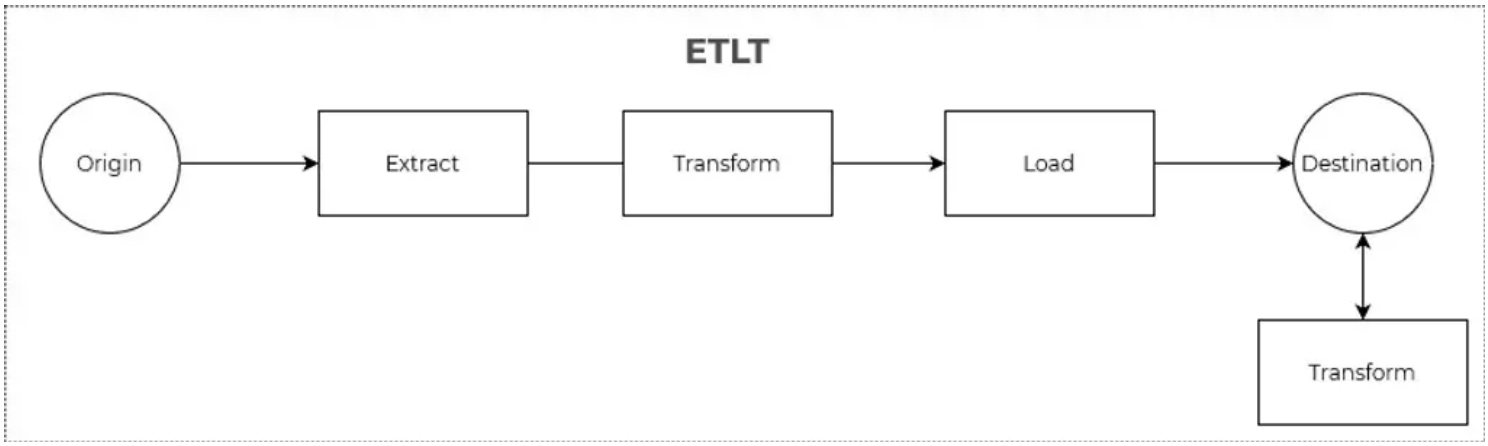
4. ELT

Extract-Load-Transform, is a variation on ETL that was conceived to offset some of the latency issues that results from pure ETL processing. Waiting for all transformation work to complete delays the availability of data for business use. Loading immediately after extract, then transforming data in place reduces the delay. ELT has the greatest impact on accelerated data availability when multiple sources that are not ready for extract simultaneously would be held in a staging area with ETL processing. With ELT the data warehouse serves the role of data staging so each source becomes available for use upon extraction and loading. Data transformation is performed in place in the data warehouse once all sources are loaded. Data quality and data privacy considerations are sometimes a concern with ELT processing. When data is made available for use without transformation, it is widely exposed without first performing data cleansing and sensitive data masking or obfuscation.



5. ETLT

The Extract-Transform-Load-Transform (ETLT) pattern is a hybrid of ETL and ELT. Each source is extracted when ready. A first stage of "light" transformations is performed before the data is loaded. The first stage transformations are limited to a single data source and are independent of all other data sources. Data cleansing, format standardization, and masking of sensitive data are typical kinds of first stage transformations. Each data source becomes available for use quickly but without the quality and privacy risks of ELT. Once all sources have been loaded, second stage transformation performs integration and other multi-source dependent work in place within the data warehouse.

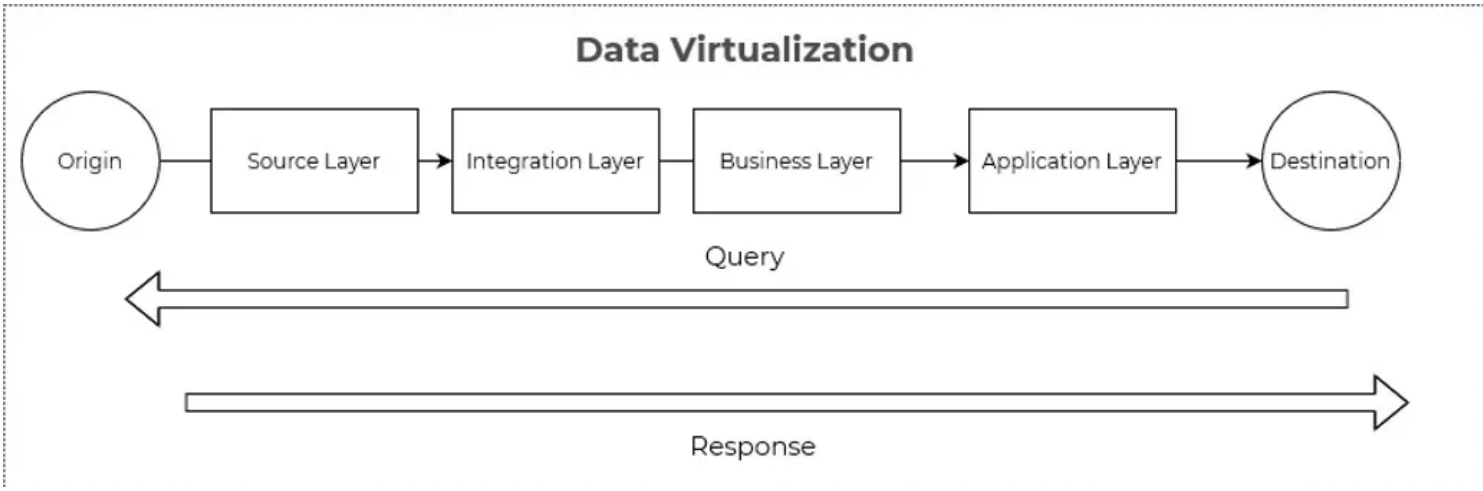


6. DATA VIRTUALIZATION

Data virtualization, which serves data pipelines differently than the other patterns. Most pipelines generate physical copies of data in a data warehouse, data lake, analytics data feed, and so on. Virtualization provides data as views rather than storing a separate version for each use case. Layers of data abstraction are used in virtualization. The source layer is the least abstract, providing connection views through which the pipeline sees the content of data sources. The integration layer combines and connects data from disparate sources providing views similar to the transformation results of ETL processing. The business layer presents data with semantic context, and the application layer structures the semantic view for specific use cases.

In contrast to ETL processing, which is started by a schedule, virtualization processes are started by a query. The query is issued in the application and semantic context, and it is translated through the integration and source layers to connect with the appropriate data sources. The response takes the opposite path, acquiring source data, transforming and integrating it, presenting a semantic view, and delivering an application view of the data. When people want the most recent and up-to-date data, virtualization can be useful. The data sources determine the degree of data freshness. The amount of historical data available is also determined by the sources.

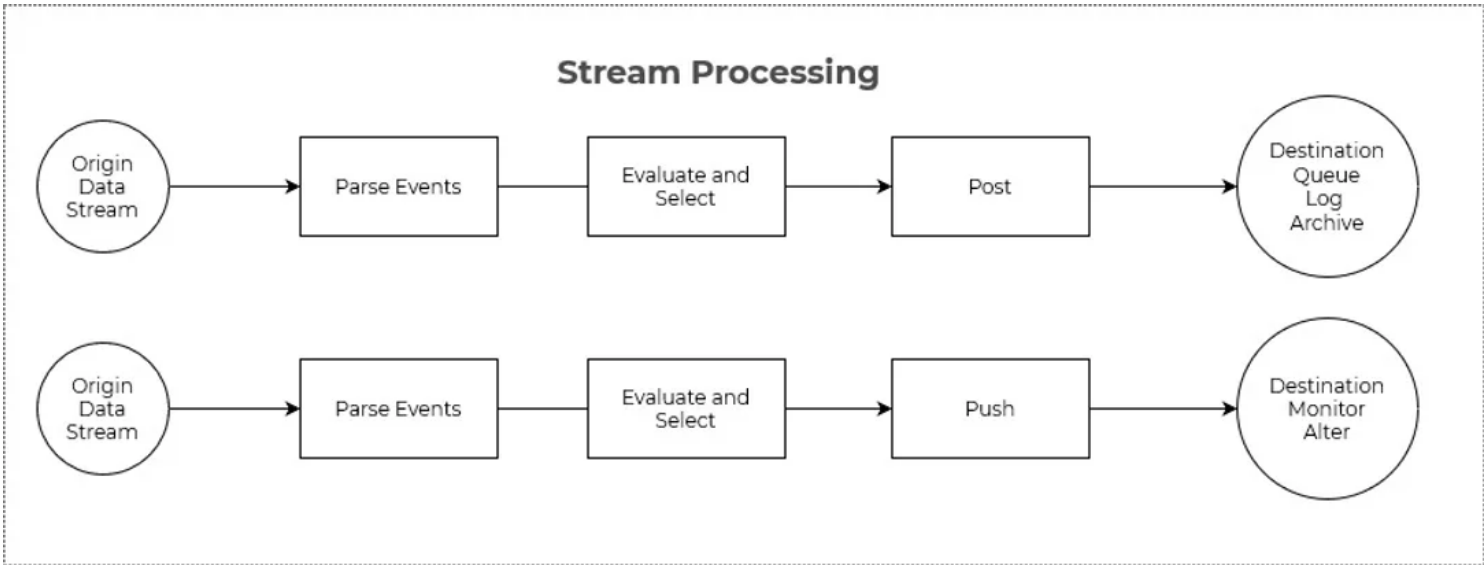
> Virtualization has the distinct advantage of integrating and transforming only the data that is requested and used, rather than all possible data. It works well with simple transformations and small data volumes, but it may struggle with complex transformations or when large amounts of data are required to respond to a query.



7. DATA STREAM PROCESSING

The data origin in both patterns is a stream with a continuous flow of event data in chronological order. The processing starts with parsing the events to isolate each unique event as a separate record. Individual events can then be evaluated in order to select only those that are relevant to the use case. In many cases, especially when dealing with large data volumes or a high percentage of unnecessary events, it is preferable to move parsing and selection to the network's edge—close to the sensors where event data is captured—rather than moving unnecessary data across the network.

The two patterns diverge slightly at the data flow's destination. The goal of some use cases is to send events to a message queue, an event log, or an events archive. When events are posted, they become the origin or input to a downstream data pipeline where data consumption is somewhat asynchronous. In other cases, the goal is to send events to a monitoring or alerting application, which will deliver real-time information about the state of a machine or other entity. Stream processing pipelines typically work with sensors and Internet of Things (IoT) data using high-volume, fast-moving data technology.



8. CHANGE DATA CAPTURE

Change Data Capture (CDC) pattern that can be used to increase the freshness of data that is typically latent. ETL is used to process many of the operational data sources that flow into data warehouses and data lakes. ETL is inherently latent. When applied to operational databases, CDC technology can detect data changes as they occur and deliver information about those changes in two formats. When changes are pushed to a message queue, they become available for downstream processes to work within mini, or micro-batch modes. CDC with a message queue, for example, can reduce data warehouse latency from days to hours or minutes.

HOW TO USE THE PATTERNS

Data architects and engineers should embrace pipeline design patterns as a core discipline.

You can start with the set of design patterns that are described here and add to them as your needs and experiences dictate. Knowing the data characteristics and the use-cases for each data pipeline is the first step you should take if you are a data engineer responsible for developing data pipelines.

Next, select the pattern that fits your needs the best and look for reusable code or a code framework to implement it. Use the framework you find and consider making improvements. Make an effort to create one as a byproduct of building the pipeline if you can't find one. As a data architect, assume responsibility for creating and disseminating frameworks.