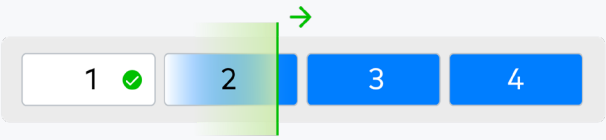
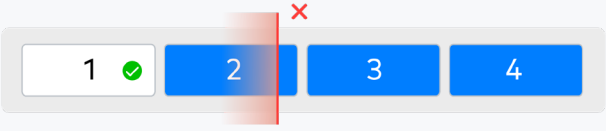
Suppose you have four micro batches of data to be synced to a destination . The cursor, symbolized by a green checkmark, is basically a bookmark that records progress. In the below example, it indicates that the first batch has been completed. The pipeline is partway through the second batch:



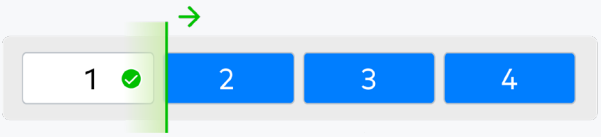
All records from the first batch ("foo," "bar," "baz," "qux," and "corge"), and some from the second batch ("grault," "garply," and "waldo"), have been loaded:

1	foo ✓
	bar ✓
	baz ✓
	quz ✓
	corge ✓
2	grault ✓
	garply ✓
	waldo ✓

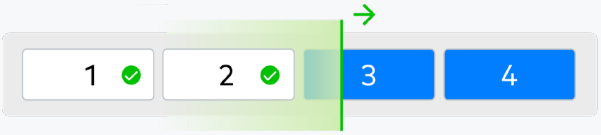
Then, the sync is interrupted:



When the data pipeline resumes, it resumes from the last location indicated by the cursor, recognizing that the first batch has been completed but the second has not.



As the second batch is completed, the cursor is updated to the end of the second batch.



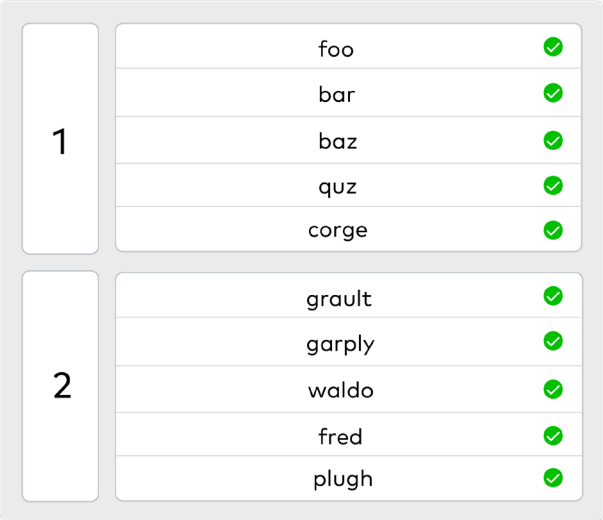
Records that were previously synced from the second batch (in this case, "grault," "garply," and "waldo") are reintroduced to the destination. Without idempotence, the records are duplicated (red rows):

1	foo ✓
	bar ✓
	baz ✓
	quz ✓
	corge ✓
2	grault ✓
	garply ✓
	waldo ✓
	grault ✗
	garply ✗
	waldo ✗
	fred ✓
	plugh ✓

Duplicate records can pose serious problems for business intelligence and analytics:

1. **Misleading or erroneous insights –** you will not have correct rank orders, counts or sums of records. Duplication may also alter medians, averages, distributions and other summary metrics.
2. **Broken operations downstream –** any systems that rely on a one-to-one correspondence between records and identifiers may throw an error or produce the wrong behavior.
3. **Wasted storage space and computational bandwidth –** extra records don't add value yet still have to be accounted for in queries (and disk space).

The challenge is to identify every unique record and ensure there is no duplication so that the output of the process we illustrated above looks like so:



DEDUPLICATING DATABASE RECORDS

Deduplicating records depends on identifying unique entities within records. Databases may explicitly declare primary keys, enabling easy programmatic identification.

As previously discussed, incremental syncs of databases leverage change logs, which contain lists of updates. Without unique row identifiers and idempotence, the destination will create new rows for all log entries, duplicating some rows in the corresponding tables.

When primary keys are not explicitly declared, the data pipeline must use another identifier. One alternative is to impute a primary key by hashing the contents of the entire row; the drawback to this approach is that the data pipeline can only prevent exact duplication. It has no way to prevent new, conflicting rows from being created as individual values in a row change.

Change logs offer an opportunity to detect effective changes so that the data pipeline isn't forced to sequentially reproduce every single operation done to a record. That is, a record that goes through an UPDATE, then UPSERT, then DELETE is ultimately just deleted. Fivetran sorts change log records by DELETEs and primary keys and uses a number of rules to collapse the records into effective changes.

DEDUPLICATING SAAS RECORDS

There is typically no programmatic way to identify primary keys when extracting data through API endpoints for SaaS applications. Instead, they must be found in the documentation or reverse-engineered using knowledge of the source's underlying data model, then constructed from a field or combination of fields that are fixed and unique to a particular entity.

This can be highly error-prone, as seemingly fixed and unique fields, such as email addresses, may actually change. In order to determine a usable primary key, you have to understand the exact behavior of the application that uses the data, specifically which fields can and can't be changed.

For instance, a naive and tempting approach is to identify a user from contact information such as username or email address. But it isn't safe to assume that usernames and emails can't change. Consider the following basic user data:

There are plenty of practical reasons a person might change any of these (moving, marriage, new phone plan, etc.). You must use a fixed reference point for a unique identifier, and therefore determine exactly which fields are fixed within that particular system and which are not. In addition, you need to ensure that any fields you use as primary keys don't (and won't ever) contain null values.

Without documented primary keys, you will need to reverse-engineer the underlying data model and functionality of the application in order to identify the fields that really are both unique and immutable.

If it turns out that there are no unique or immutable fields, then you have no choice but to treat each full record as unique. In this case, the primary key would be imputed as a hash of the entire row. This at least prevents duplication on recovery from pipeline failure, but not from updates to existing records.

Wherever possible, Fivetran identifies primary keys for every table from every source. It can be burdensome to build and maintain a system that makes primary keys of paramount importance, especially as schemas at the source continue to change. It is tempting to simply read records from the source and dump them to destinations, but doing so introduces serious problems with duplication and general data integrity.

The more sustainable approach requires understanding the underlying data model of each source and assigning the correct primary keys. As sources grow in number and complexity, the challenges associated with identifying primary keys grow accordingly.

Schema drift handling

Schemas drift when an application's data model evolves and its columns, tables and data types change. Faithfully preserving the original values of data and ensuring its smooth passage from source to destination, even as the source schema changes, is a particularly vexing data movement challenge. Schema changes are one of the chief failure conditions for traditional ETL, often causing the data pipeline to fail and require an adjustment that requires scarce and highly specialized expertise.

NET-ADDITIVE DATA MOVEMENT AND LIVE UPDATING

One method Fivetran preferred in the past for avoiding pipeline breakages and ensuring faithful reproduction of data was net-additive data movement. The basic behavior is as follows:

- When a column or table is added to the source schema, it is added at the destination as well.
- When a column or table is removed, it is kept in the destination but no longer updated.
- When a column or table is renamed, it is duplicated at the destination under its new name, so that both the old version with the old name and the new version with the new name are present. The old version is subsequently no longer updated but retained for the sake of completeness.

The chief characteristic of net-additive data movement is that columns or tables are never removed despite schema changes. This means that schema changes never lead to lost data. This preservation of old data extends to individual records, as well. Rows that are deleted at the source are soft-deleted or flagged in the destination so that analyses that depend on old data still function.

Fivetran's currently preferred alternative to net-additive data movement is live updating. Here, the basic behavior is as follows:

- When a column or table is added to the source, it is added to the destination as well.
- When a column or table is renamed, it is renamed at the destination.
- When a column or table is removed, it is removed at the destination.

Live updating dispenses with the retention of old schema elements by perfectly matching the data model in the destination with the data model in the source. This extends to individual records as well. Rows that are deleted at the source are deleted at the destination.

New schemas, tables and columns may include sensitive information. For the sake of security, Fivetran supports the ability to exclude or include new schemas, tables and columns – more on that later!

Depending on whether the data source features a change log, some behaviors may not be possible. For both net-additive data movement and live updating, the ideal behaviors concerning renamed columns and tables are only possible if the data source **1) has a change log** and **2) tracks schema changes.** Otherwise, renamed columns and tables are usually recognized as a removal combined with an addition.

HISTORY MODE

One downside of both net-additive and live updating data movement is that values that change but are not deleted are updated in both the source and destination. Without an additional solution, this means the loss of values that change over time.

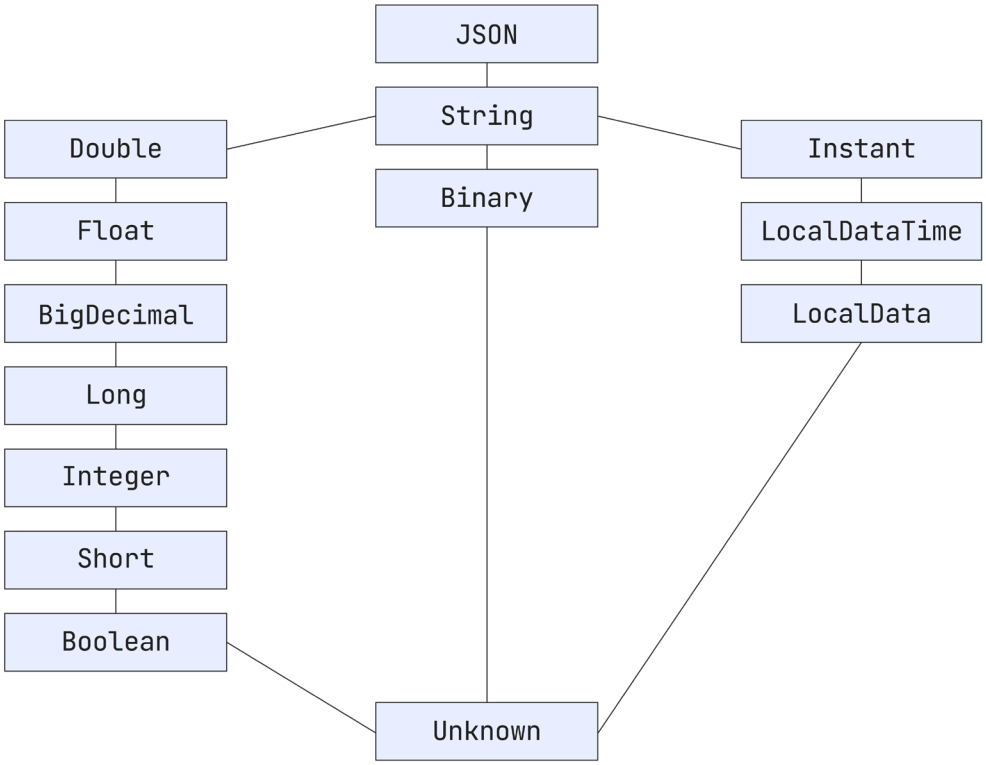
The Fivetran solution to retaining historical values is history mode. History mode retains the current and all previous versions of all rows in a table. A history table specifically contains "start" and "end" timestamps for every version of every value, in which the most recent version has a NULL (or equivalent) "end" timestamp. Deletions are signified when all rows with a particular primary key have a known end time stamp, meaning there is no current "live" version.

Note that keeping track of changes to the schema is a different matter, and schema changes must be represented separately from history tables.

History mode can be costly due to the sheer volume of data that is retained. A good practice is to allow users to selectively apply it to certain tables rather than entire schemas. It can readily coexist with either net-additive data movement or live updates, applied toward the same table.

THE HIERARCHY OF DATA TYPES

Besides a change in name, columns can also change by data type. The key to accommodating a data type change is to assign a new data type that is inclusive enough to handle both old and new values in that column; that is, assigning a supertype.



The chart above, from top to bottom, illustrates a hierarchy from the most inclusive data types (top) to the least (bottom).

Let's say a column consists of integers but fractional values/decimals are added to the column later on. At the destination, we will update the column's data type to "double" instead of "integer," because "double" can accommodate both types. Conversely, if a column consists of decimals that are converted to integers, its counterpart in the destination will remain a double because it's more inclusive.

Reliable data replication requires keeping a data pipeline running and ensuring faithful replication of source data in the midst of schema drift. It is more than a matter of simply copying and pasting data from one place to another, especially when large volumes of data are concerned and analysts require the ability to examine historical data.

Pipeline and network performance

The workflows associated with extracting, loading and transforming data feature a number of possible performance bottlenecks.

Fivetran uses three basic methods to overcome bottlenecks in a data pipeline:

1. **Algorithmic improvements and code optimization** don't involve major changes to the software's architecture.
2. **Architectural changes** can be used to parallelize simple, independent processes.
3. **Pipelining** involves architectural changes to separate the data movement process into distinct, sequential stages that can each simultaneously handle a workload, like an assembly line.

The full workflow of extracting, loading and transforming data can be a complex, path-dependent process. Architectural changes to software affect both how the code is organized and the hardware requirements of the system. This means parallelization and pipelining are potentially costly ways to improve performance and should be pursued only when algorithmic optimization has been exhausted.

ALGORITHMIC OPTIMIZATION

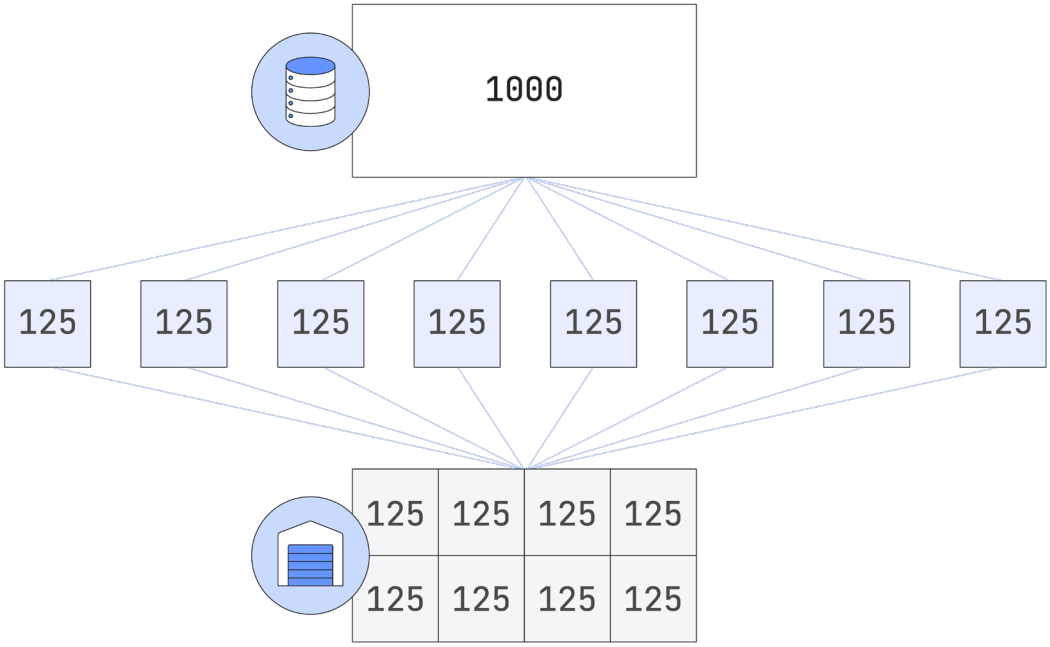
Algorithmic optimization is all about using the most efficient method for every computation. There are better and worse ways to perform every computation. It is also one of the few ways to directly squeeze monetary savings out of improved performance, as parallelization and pipelining require additional infrastructure.

> Suppose you want to test whether a figure is divisible by 2. The hard, slow way is to perform the actual computation, i.e., to divide the figure by 2. A better approach is to see whether the number is even or not by reading the trailing bit of the binary for that number, which should be 0.

In a data pipeline, suppose some data arrives as a JSON and you want to determine if a key has already been converted into a field. Rather than looping through a list of all of the keys, you can create an index out of your fields and then match the key against the index using search or lookup methods, like hash maps.

PARALLELIZATION

Parallelization is about splitting and distributing work into parcels to execute simultaneously rather than sequentially. Suppose you have 1,000 records and spawn eight processes. The original data can be split into eight queues of 125 records each that are executed in parallel rather than 1,000 in sequence.



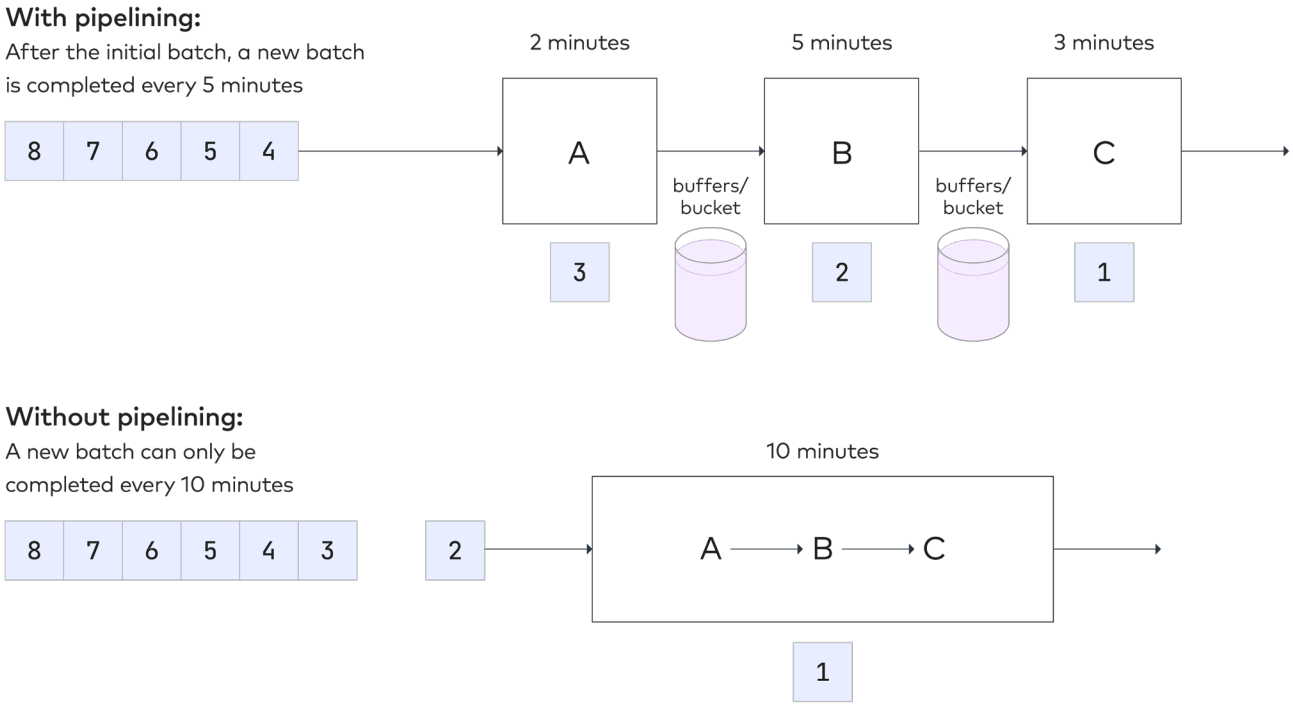
A concrete example is making API queries in parallel. Network request-response times impose a more-or-less fixed interval on each query. To prevent request-response times from adding up sequentially, it's better to send a pool of requests across the network at once.

PIPELINING

The full data movement workstream involves a number of intermediate steps, including deduplication and some forms of aggregation. Operations like deduplication affect entire data sets. They must be performed on monolithic blocks of data where the parts can't be separated in order to produce the correct outputs.

This means that, for some units of work, another approach is necessary for optimizing performance. Rather than splitting the work into pieces of parallel execution whose outputs are reassembled later, the key is to carefully separate the process into steps that can all be simultaneously populated, with buffers between them to ensure that files are fully written before they are handed off to the next stage.

Consider a queue with 8 separate batches that must each be processed en bloc through steps A, B and C:



NETWORK PERFORMANCE

Network performance is a subtopic of pipeline performance, concerning the movement of data between sources, nodes and destinations in a pipeline.

For traffic from most sources, particularly commodity SaaS applications, network performance generally matters very little because the overall volumes of data are low and data is quick to ingest regardless of network performance. At Fivetran, we commonly experience the intentional throttling of API bandwidth, especially in response to repeated queries.

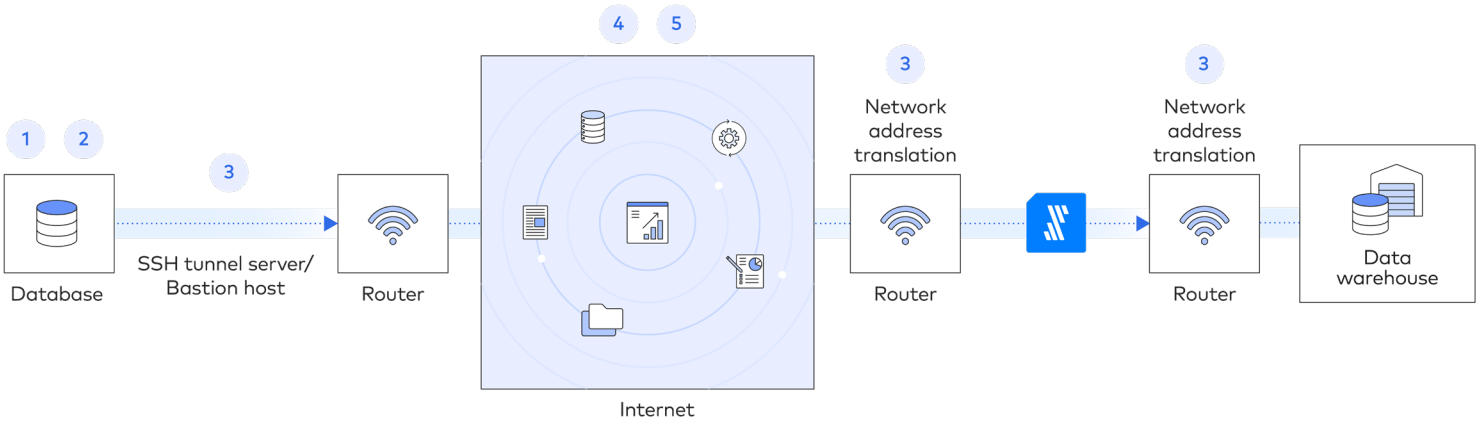
However, network performance is a far more serious matter for high-volume data sources, specifically operational databases. The huge volume of records stored in an operational database heavily impacts the speed and turnaround time of both historical and incremental updates.

Historical syncs, which ingest the full contents of a data source and are necessary both to initialize a data connector and to recover from serious errors, are especially impacted by network performance. Syncs that take many days or weeks can interfere with a company's operations and deter a company from modernizing its data movement infrastructure altogether. With very large datasets, even incremental syncs can bog down.

HOW NETWORK TRAFFIC FROM DATABASES GETS BOTTLENECKED

When the source is a database, there are several common ways syncs can be slowed down:

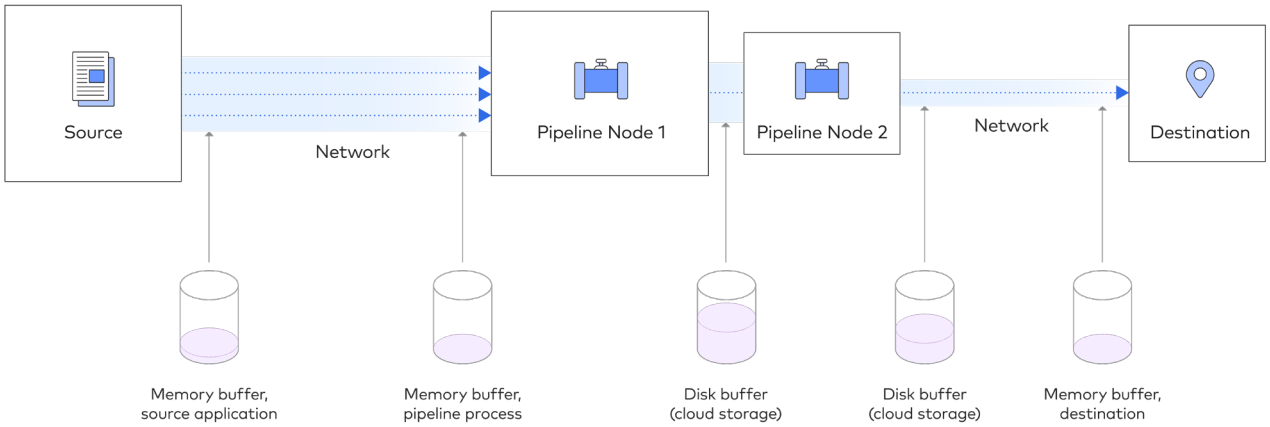
1. Database instances can be too small and lack the resources (i.e. processing power, RAM, disk space, local network bandwidth) to quickly transfer large amounts of data.
2. High transactional traffic can also leave databases too busy to service data extraction.
3. Databases can be gated behind other servers. Security protocols using SSH tunneling, bastion servers, port-forwarders and network address translation (NAT) often intentionally route traffic through narrow checkpoints in order to prevent sensitive data from being intercepted.
4. Inter-region transfer (e.g. from AWS East to AWS West) is slower than intra-region transfer, meaning that you can face slowdowns by moving data from one region to another.
5. Inter-cloud transfer (e.g. from AWS to GCP), or moving data from one network to another, can cause slowdowns.



Note that the example above is just one hypothetical. The relative severity of each chokepoint can vary, and it will take careful troubleshooting to identify which one impacts your data movement workflow the most.

ADDITIONAL CONSIDERATIONS

The measurement of bandwidth and throughput can be complicated by the presence of buffers.



Buffers are staging areas that temporarily contain data, either in memory or on disk, whenever data is set to be handed off between machines. The more bottlenecked a subsequent stage of a handoff in comparison to the previous stage, the more the buffer will fill.

The presence of buffers can radically alter the behavior of a network. Without buffers, the movement of data along every node of the network takes place at a constant rate and is limited by the slowest bottleneck. With buffers, the flow is less brittle as nodes can act as staging areas to store batches of data.

Observing how full buffers get (and how quickly they fill) can offer clues to where your network has bottlenecks.

Another important consideration is compression. Compressing data reduces memory usage and is nearly always worthwhile, especially when supported by the power of modern CPUs. However, compression can complicate how you measure transit through your system, requiring some judgment calls. If a 100 GB file reduces to 40 GB, did 100 GB move through the network or 40 GB?

HOW TO IMPROVE NETWORK PERFORMANCE

Every point of transition from one platform to another has the potential to throttle traffic. The ways Fivetran addresses network performance are similar to the ways Fivetran addresses pipeline performance. The first, and most obvious, is simply to scale up infrastructure, i.e. widen every part of the path. The second is to re-architect the data movement process so that there are fewer points of transition. Yet another way is to optimize algorithms and parallelize execution to speed up processing. Finally, it is important to properly leverage compression and decompression as data moves between platforms.

Transformations

ELT data pipelines extract and load raw data to a destination. Raw data often isn't especially usable to analysts and other users. **Data transformation** consists of all processes that alter raw data into structures called **data models**, collections of tables that directly support analytical and operational uses. These include:

Revising – values sometimes must be corrected or organized in a manner that supports their intended use.

Separating – data values are sometimes combined in the same field because of idiosyncrasies in data collection, but may need to be separated from others for more granular analysis.

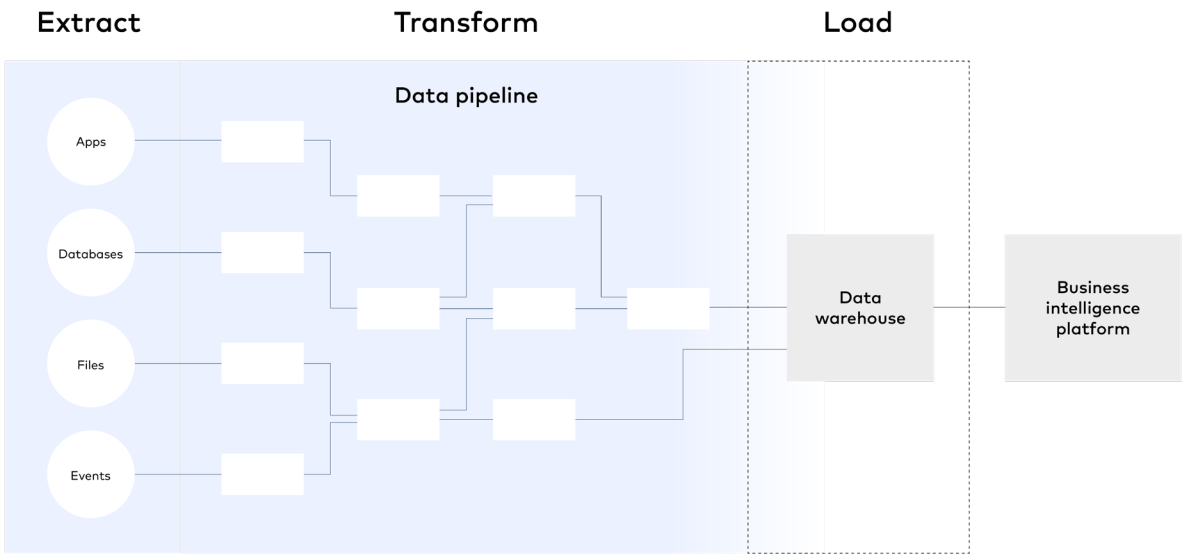
Computing – it is often necessary to turn raw numbers into rates, proportions, summary statistics and other important figures. For machine learning, unstructured data, such as media files, must be vectorized.

Joining – data must sometimes be linked across tables to build a full picture of some phenomenon.

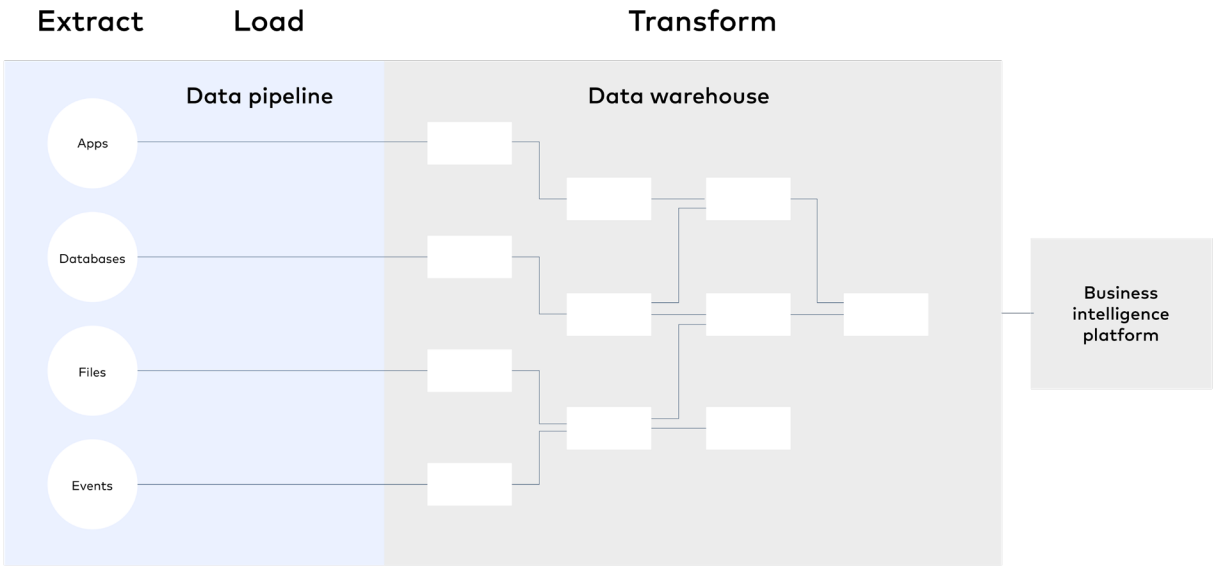
Analysts and other end users depend on the data models produced by transformation to produce visualizations, dashboards and reports. Transformation is also essential to supporting use cases such as embedded analytics, machine learning and data activation.

Traditional ETL features two main pitfalls. The first is that it makes transformation an engineering-centric process. The system requires specialized engineering skills to build and maintain, and imposes a technical barrier between analysts and the functionality they need to perform their role.

The second is that closely coupling transformation with the extraction and loading of data means that the moment upstream data sources or downstream data requirements change, the pipeline will need to be adjusted, resulting in downtime and consuming additional engineering hours.



By contrast, the Fivetran approach to ELT makes transformation accessible to analysts by bringing it into the data warehouse or lakehouse. Transformations inside the warehouse / lakehouse are written using SQL, a more accessible language than Python or Java and the prevailing standard in database operations.



DATA MODELS

Fivetran applies light transformations in the data pipeline to clean and **normalize** data. This keeps data close to its raw form while eliminating redundancy and enhancing interpretability. With Fivetran **data models**, we push the data further along to a usable state. This enables analysts to self-serve with less effort and turnaround time.

Fivetran data models are the product of our integration with **dbt™** to power **transformations**. The SQL-based dbt™ data transformation tool enables analysts to build data models iteratively and automate data transformation. Users model their data using SQL SELECT statements, create relationships and dependencies between models and then materialize those models as tables and views in a data warehouse or lakehouse. From there, the analytics-ready tables can be used to produce reports and dashboards.

A dbt project is a directory of SQL and YAML files hosted on a user's GitHub repository. The YAML file contains project configuration information, while each SQL file either contains SELECT statements that transform data in some way or macros that enable code reuse.

With dbt, users can:

- Leverage an extensive open source **library of pre-built data models**
- Practice continuous integration/continuous development (CI/CD), collaboration and version control
- Customize or produce data models by writing, testing and performing SQL-based transformations
- Create and share documentation

The Fivetran approach to data modeling strives to lower the barrier to entry by reducing the **use of the command line** and configuration in several environments. Fivetran Quickstart Data Models allow you to utilize our pre-built data models directly through the platform. This further lowers the technical barrier to entry, bringing yet more modularity to data movement. As with creating new connectors, the process is performed entirely through a graphical user interface.

ORCHESTRATION, INTEGRATED SCHEDULING AND DATA LINEAGE

Data orchestration is the process of coordinating the execution and monitoring of data movement workflows. It is an essential feature of a fully automated data pipeline. In the context of transformations, the main need is to ensure that data is transformed in the correct sequence (and that the process is triggered at the correct time) in order to minimize latency and maintain freshness.

Orchestrations can involve one data source or many. Fivetran automates orchestration for off-the-shelf data models from our library as well as data models custom-built by users. This completely replaces processes that were traditionally conducted using cron jobs, traditional ETL tools like Informatica, scripting and orchestrating with Python and Airflow, etc.

A related consideration to data orchestration is to automatically trigger sequences of transformations based on the status of a connector. This is called integrated scheduling.

In **fully integrated** scheduling, Fivetran automatically runs all transformations downstream of a connector as soon as we finish loading data in your destination.

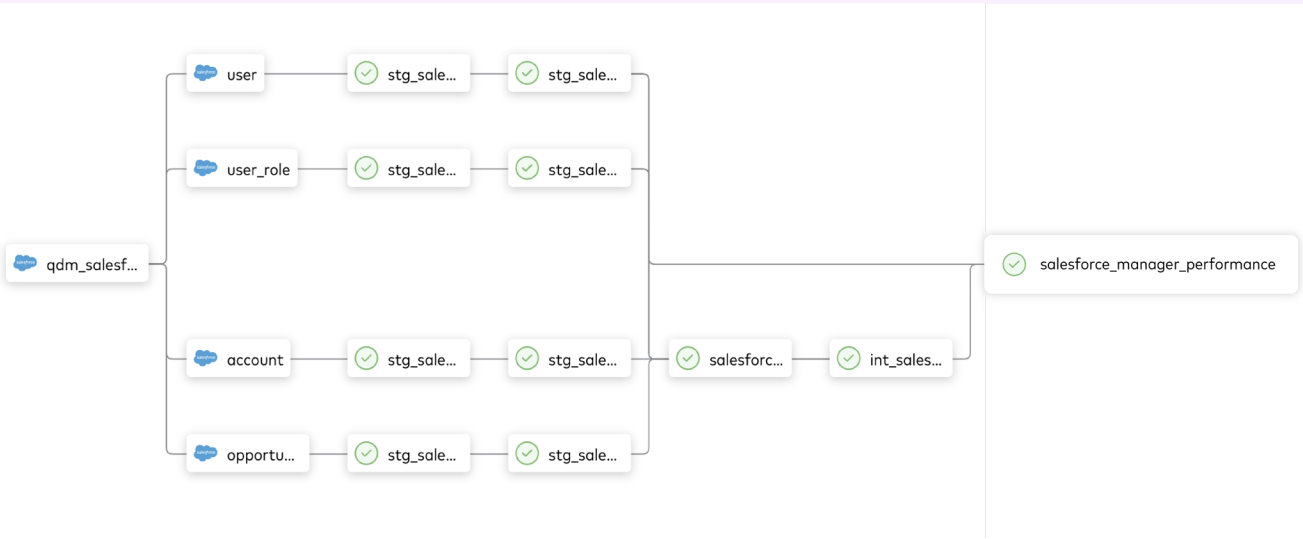
In **partially integrated** scheduling, Fivetran will run transformations according to a schedule you set, except when its schedule overlaps with the associated connector's schedule. In that case, we wait to run the transformation until the connector finishes running.

In **independent** scheduling, Fivetran will run transformations on a fixed schedule determined by the user without consideration for the status of upstream connectors.

Ensuring that models run in the correct sequence and schedule prevents unnecessary syncs, saving compute costs and fostering data integrity by preventing discrepancies between related tables.

It is important to be able to visualize the process of orchestration. Once your data pipeline is set up and automated, data lineage graphs illustrate your data pipeline end-to-end, showing the dependencies between dbt models so that you can track the flow of data through intermediate models. Data lineage graphs also display the run status for each connector and model in the pipeline, enabling you to troubleshoot failed runs.

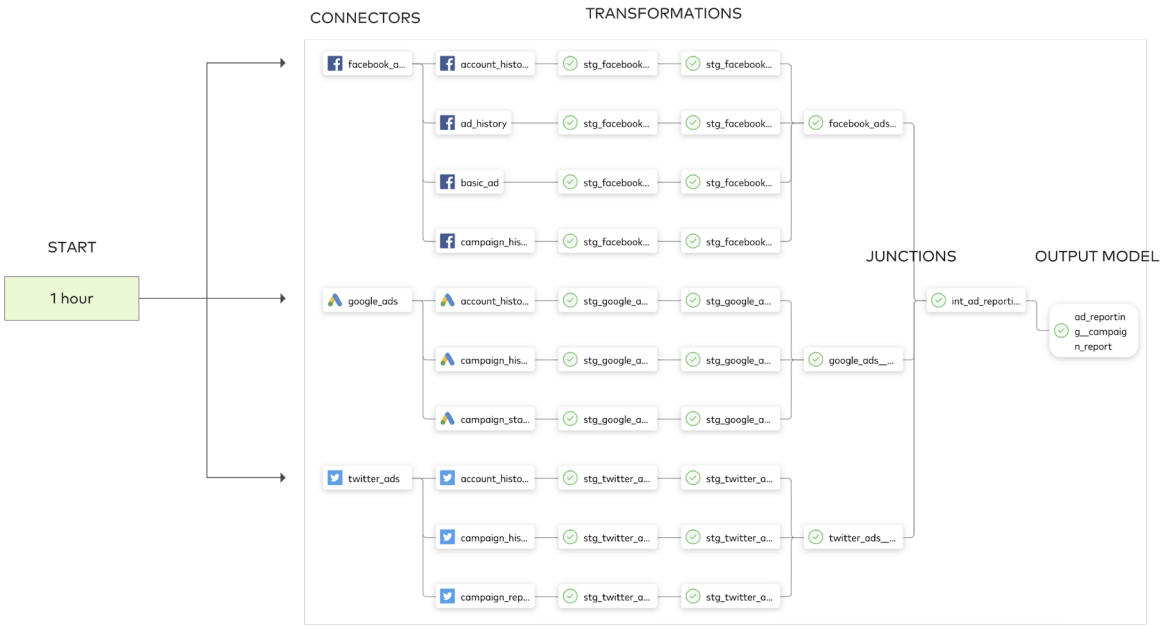
The example below illustrates the data lineage of our Salesforce data model for manager performance:



The graph consists of:

- **Source tables (with the Salesforce logo) –** The starting tables in the sequence
- **Intermediate models (everything else except the output model) –** All models in between the source and output models
- **Output models (at the end, labeled "salesforce_manager_performance") –** The final model analysts use for analytics

A more complex example might involve multiple data sources. Suppose you want to combine data from your various marketing channels to get a view of your return on advertising spend (ROAS). The lineage might look like so:



This graph consists of the following elements:

- The **start** is the interval that initiates the pipeline.
- A **junction** waits for multiple connectors to finish syncing before it triggers a dbt transformation.
- A **connector** updates source tables in the destination.
- An **output** model generates an analytics-ready table. It is typically a leaf node on your data lineage graph.
- A **transformation** is a model or a collection of models that updates downstream tables in the destination.

These visual representations, combined with alerts and notifications, expose dependencies between data models and are essential to tracking the flow of data from your connectors to your destination and helping pinpoint failure points and causes of stoppages.

CHAPTER 2

Destination Lakehouse

Fivetran's automated data movement platform provides secure, scalable, high performing, real-time data movement integration for analytics and AI on the lakehouse. A lakehouse architecture is the ideal data architecture for data-driven organizations. It combines the best qualities of data warehouses and data lakes to provide a single solution for all major data and AI workloads. Pioneers of the lakehouse category, Databricks offers a unified, open and scalable lakehouse platform for data, analytics and AI.

Unified

One platform for your data, consistently governed and available for all your analytics and AI

Open

Built on open standards and integrated with every cloud to work seamlessly within your modern data stack

Scalable

Scale efficiently with every workload from simple data pipelines to massive LLMs

The lakehouse is a critical component of the modern data stack as it eliminates the data silos that traditionally separate and complicate data engineering, analytics, BI, data science and machine learning. And, its unified common approach to data management, security and governance helps you operate more efficiently and innovate faster.

With automated data movement from Fivetran, **Delta Lake** transforms your data lake into the destination for all your structured, semi-structured and unstructured data. Delta Lake, a 100% open source storage format provides reliability through ACID transactions to ensure that your data is always in your control. All data in Delta Lake is stored in open Apache Parquet format, allowing data to be read by any compatible reader. Delta Lake enables you to scale reliable data insights throughout the organization and run analytics and other data projects directly on your data lake.