

# The ultimate guide to modern data movement on the Lakehouse

An all-in-one guide to the Fivetran way of  
data movement on the Databricks  
Lakehouse Platform



# Table of contents

<b>Introduction: Data movement is the future</b>	<b>3</b>
Data integration challenges	4
The automated data movement platform	5
<b>Chapter 1: Data movement</b>	<b>7</b>
Automated data movement	8
Incremental updates	11
Idempotence	16
Schema drift handling	21
Pipeline and network performance	24
Transformations	29
<b>Chapter 2: Destination Lakehouse</b>	<b>34</b>
Data Warehousing on the Lakehouse	35
<b>Chapter 3: Security</b>	<b>37</b>
Data security	37
Platform security	38
<b>Chapter 4: Governance</b>	<b>43</b>
Governance on the Lakehouse	45
<b>Chapter 5: Extensibility</b>	<b>46</b>
Tools	46
Use Cases	48

# Data movement is the future

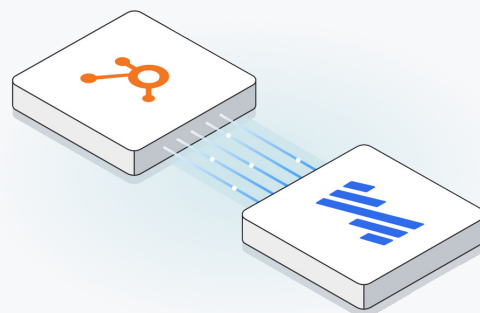
Since the advent of modern computing, organizations have sought to mine the data produced by their operations for insights to support decisions. The growth of the cloud has intensified both the need to harness data and the evolution of the tools directed to solve this problem. Use cases for data have grown from decision support to predictive modeling, direct support for large-scale operations and innovative, data-based products. These emerging use cases call for a change to the way we think and talk about the tools and processes used to prepare data for various end uses.

In light of the growing range of data use cases, data integration is needlessly limiting and is superseded by a broader conception of **data movement**. Data movement, a more general and inclusive concept than data integration, encompasses a wider range of important use cases for data.

For instance, there are operational use cases for data movement. To ensure performance, an organization might want to replicate data across operational systems that are distributed across different regions. Similarly, to prevent disruptions to operations in the event of failures, an organization might want to replicate data in real time from production servers to failover instances. Other use cases for data movement require productizing or activating data by moving data models back into applications and operational systems.

As data-related use cases become ever more complex and varied, additional concerns crop up regarding how to scale data usage safely and responsibly. These concerns include governance, security and extensibility. Answering these concerns requires a much more comprehensive suite of capabilities than can be provided through a single point solution.

In this guide, we will discuss the Fivetran approach to data movement and how an automated data movement platform can support your organization's future needs as you scale your operations and pursue new products. We will also discuss the importance and attributes of new destinations, particularly the data lakehouse, that offer expanded capabilities over traditional data warehouses and data lakes. If you are a data professional with a hand in designing or building your organization's data architecture, particularly a data engineer or data architect, this guide is essential reading.



# Data integration challenges

Data integration is a data management process that focuses on extracting raw data from a variety of disparate sources and combining it into a single, unified view for analysis and management. The growth of the cloud and cloud-based technologies have radically changed the realities of data integration, enabling the easy movement of data from any source to any destination.

Before the modern cloud, the standard approach to data integration was a data pipeline featuring Extract, Transform, Load (ETL), often using on-premise hardware. ETL demands considerable engineering effort in planning, design, construction and continued maintenance. This makes ETL an engineering-centric process, imposing a barrier between analysts and the functionality they depend on to model data.

Its bespoke nature makes it difficult to scale and expand, as well as slow to respond to changing business needs. It features two major failure states that require redesigns and rebuilds of the data pipeline.

1. Upstream schemas at the source may change, breaking downstream processes that depend on specific configurations of fields, tables, etc.



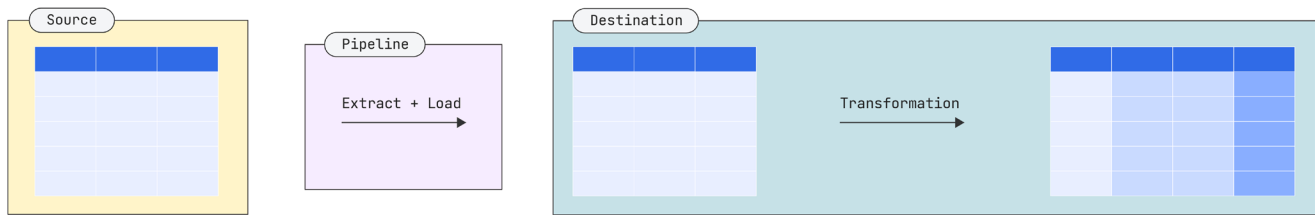
2. Downstream business and analytical needs may change, requiring new transformations to produce new data models.



These two common failure states make ETL extremely brittle, leading to downtime and stale data, absorbing precious engineering hours and limiting the effectiveness of an organization's analytics.

## The automated data movement platform

The modern approach to data movement is cloud-based Extract, Load, Transform (ELT). In contrast to ETL, ELT defers transformations to the end of the process. ELT architecture loads data with minimal alteration from source to destination, while transformations are decoupled from the data pipeline and performed within the destination. This prevents the two failure states commonly imposed by ETL (i.e., changes to upstream schemas and downstream data models) from interfering with extraction and loading, leading to a simpler, more robust and more scalable data pipeline. A simpler and more robust data pipeline results in lower costs, downtime and engineering burden as well as fresher data.



There are several implications to shifting data modeling and transformations from the pipeline to the destination. The most important is that decoupling extraction and loading from transformation enables the data pipeline to be fully outsourced and automated, as there is no longer any need to build a bespoke solution for every downstream data model needed by an organization. Another implication is that transformations are now accessible to analysts. Finally, loading raw data from source to destination enables access to a singular, unadulterated source of truth that isn't obscured by transformation.

**ELT is the central mechanism of a broader suite of tools and processes used to efficiently move data through the cloud, called the automated data platform.** With the power of a modern data movement platform, organizations can not only integrate data but migrate critical data infrastructure to the cloud, promote self-service analytics and turn data assets into new products. The modern data movement platform features four key pillars:

The modern data platform features four key pillars:



#### Data movement

an organization may need to move data from applications and data-bases to data warehouses, data lakehouses across clouds, from lakes to warehouses and more.



#### Governance

As organizations and their data grow, it becomes all the more important for data teams to understand what data is being loaded so it can be protected and monitored for compliance.



#### Security

The growing volume and complexity of data carries a high risk of exposure and misuse, complicating regulatory compliance, proper access and deployment practices.



#### Extensibility

Cloud-based data movement is enhanced by programmatic control and coordination with other cloud-based tools, and can also serve as the foundation for products.

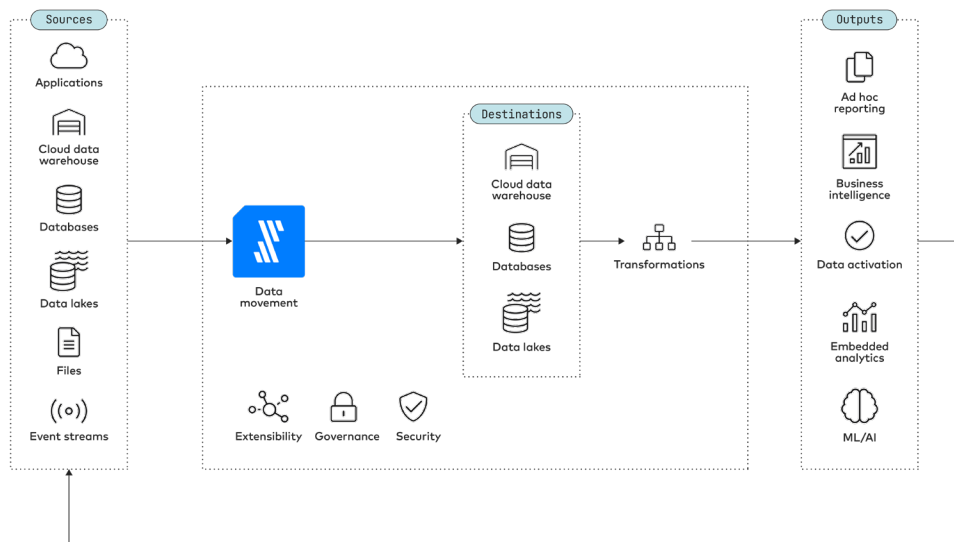
# Data movement

Data movement refers to moving data from any source to any destination, across any type of deployment – cloud, on-premise or hybrid.

Data movement supports all uses of data, including:

- 1 Business intelligence and decision support
- 2 Predictive modeling, machine learning and AI
- 3 Real-time exposure of data, business process automation and data-driven products

As illustrated by the diagram below, data movement can be multidirectional. The sources can include applications, files, event streams and databases as well as data warehouses and data lakes. Data from sources is either aggregated or replicated in destinations including cloud data warehouses, data lakehouses, databases and data lakes. Data from sources is either aggregated or replicated in destinations including cloud data warehouses, data lakehouses, databases and data lakes.



Most commonly, data is moved from applications, databases and file systems into data warehouses or lakehouses in order to support analytics. Centralizing data into a single destination is necessary because analytics should not be performed on source systems, which are heavily loaded with operational tasks. Moreover, many analytics tasks require data from a variety of different sources to be consolidated. For instance, to analyze marketing data, you may need to combine data from a variety of advertising applications and operational systems in order to construct customer journeys and compare the efficacy of different marketing initiatives.

You may also move data from one database to another, typically for reasons of redundancy or performance. Your production architecture might involve replicating a production database to a failover instance for disaster recovery and to prevent stoppages to operations. Or, you might handle traffic across multiple server regions, and rather than route all traffic to a server on one continent you might replicate to servers across the world so that your users can access servers closer to them with less latency.

Moving data from source to destination requires a high-performance system that can be deceptively complex to engineer. Considerations include properly scoping and scaling an environment, ensuring availability, recovering from failures and rebuilding the system in response to changing data sources and business needs. Many common data integration tools provide frameworks for approaching these tasks but still require a considerable degree of configuration and engineering work from end users.

These complications impose significant costs in time, labor and money on organizations that attempt to move data using an in-house solution. **An automated data movement solution obviates the need for an organization to build such a solution in-house.**

## Automated data movement

At Fivetran, we believe that data movement should be fully managed and automated from the standpoint of the end user. From the perspective of the end user, the ideal data movement workflow should consist of little more than:



Step 1:  
Selecting connectors for data  
sources from a menu



Step 3:  
Specifying a schedule



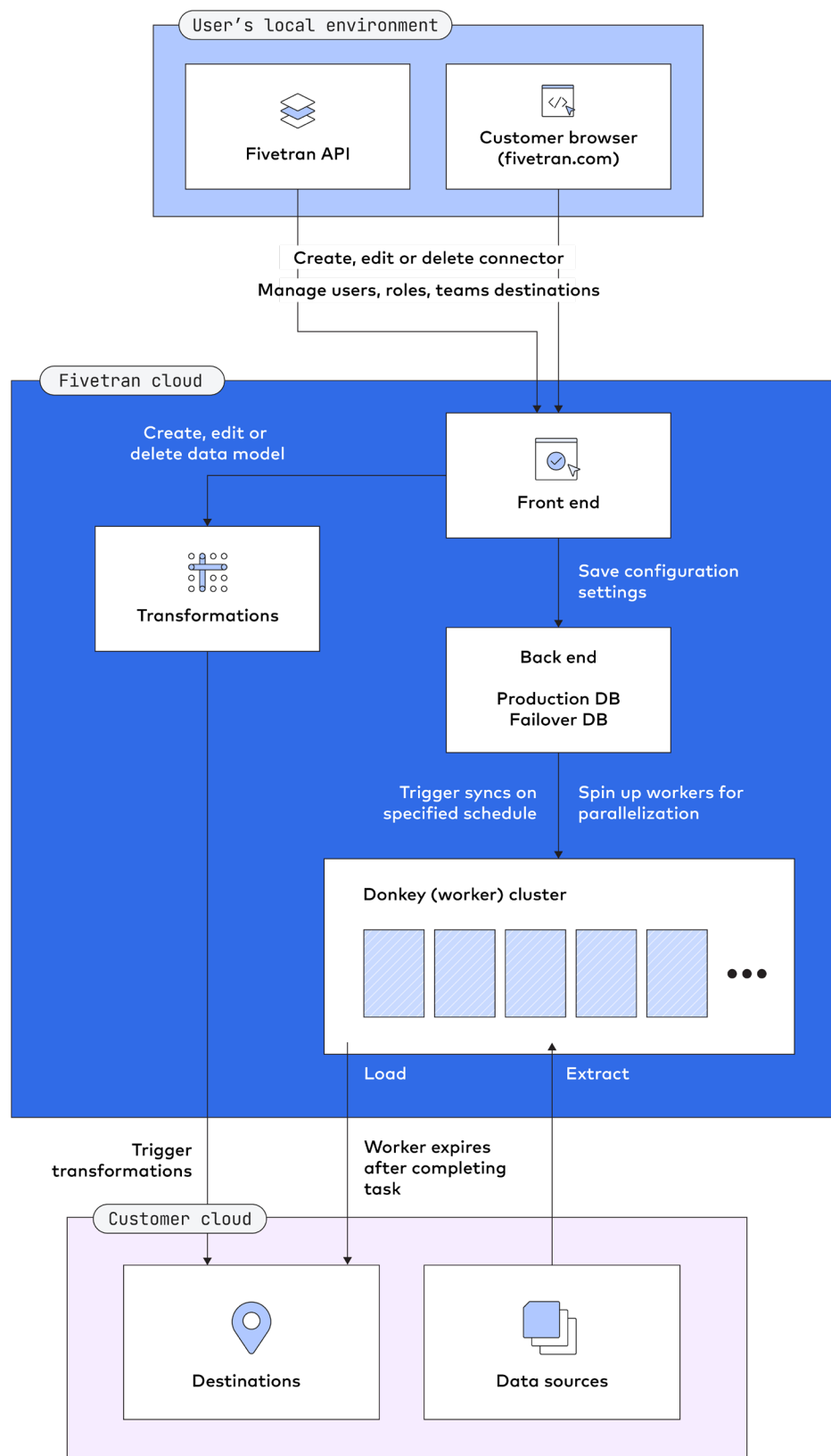
Step 2:  
Supplying credentials



Step 4:  
Pressing a button to  
begin execution

The simplicity of this workflow belies considerable complexity under the hood. The Fivetran architecture is strictly divided between a user's local environment, Fivetran cloud and customer cloud. This division is essential for ensuring both security and performance. In terms of security, the strict separations between the front end, back end and customer cloud ensure there are no ways sensitive data can be exposed through the front end. For the sake of performance, as a cloud-native tool, Fivetran makes extensive use of on-demand parallelization.

The following architectural diagram lays out the Fivetran approach to automated data movement:



Note that this diagram illustrates the standard cloud-based solution, which is a low-maintenance, fully managed experience appropriate for most cases. For organizations with security requirements that limit the ability to use cloud-based SaaS solutions, Fivetran also offers hybrid and on-premises architectures.

A typical workflow follows these steps:

- 1 The user accesses the Fivetran front end through the Fivetran.com dashboard or API.
- 2 The user creates and configures connectors.
- 3 The user's choices are recorded in the Fivetran production database.
- 4 Based on the settings saved in the production database, the Fivetran backend spawns a number of workers on a schedule.
- 5 Each worker extracts and loads data, with some light processing.
- 6 Transformations to produce analyst-ready data models are separately triggered and run on the destination.

The smooth operation of this workflow involves a number of design considerations that aren't easily captured in an architectural diagram. In the coming sections we will discuss:

#### **Incremental updates**

Ensuring timely updates and minimal disruption to source systems.

#### **Schema drift handling**

Accurately representing data even as sources change. Schema drift handling also involves data type detection and coercion, balancing accurate replication and preservation of data with the reliable functioning of data pipelines.

#### **Idempotence**

The ability of the data pipeline to easily recover from failed syncs. Idempotence depends on deduplication, which we will also discuss.

#### **Pipeline and network performance**

Minimizing latency and performance bottlenecks.

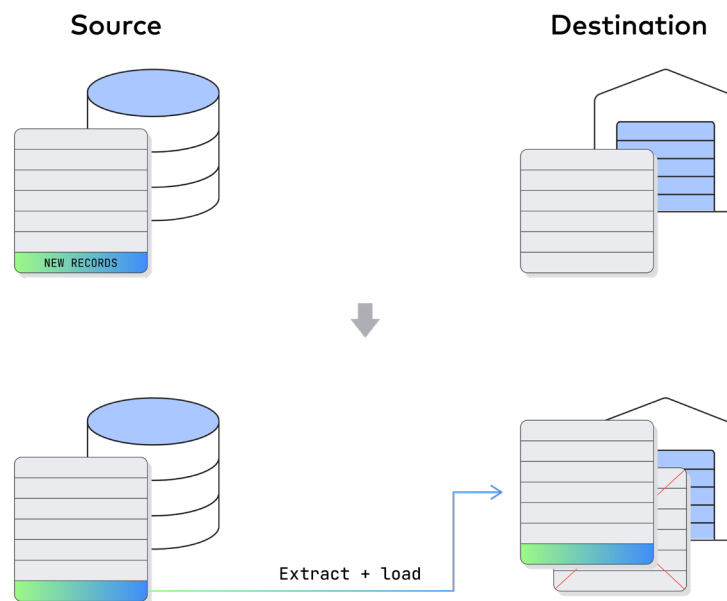
We will also explore how Fivetran handles **transformation**.



# Incremental updates

Whenever possible, Fivetran incrementally updates records from every data source. That is, instead of extracting and loading the entire data source, Fivetran detects new or modified records and reproduces the changes at the destination. Incremental updates are a key feature of an efficient data pipeline.

Full syncs are necessary to capture the full data set during an initial sync, and occasionally to fix corrupted records or other data integrity issues, i.e. resyncing tables or columns.

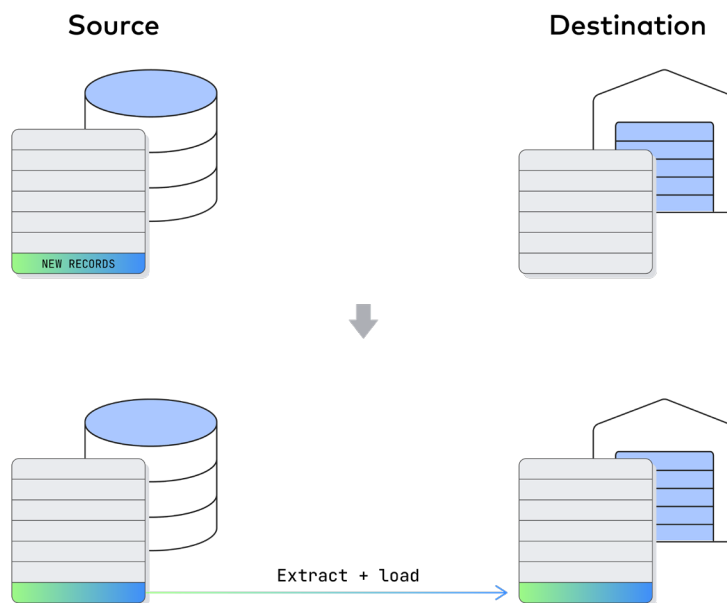


However, full syncs are inefficient for routine updates because they:

- 1 Often take a long time, especially for large data sources. This makes full syncs incapable of supporting timely, real-time updates. This point is exemplified by the common practice of daily snapshots, where "daily" itself describes the slow rate of syncs.
- 2 Can bog down both the data source and the destination, consuming resources otherwise required for operations and analytics, respectively.
- 3 Consume excessive network bandwidth and compute expense.

As data sources grow and add records, the problems listed above naturally grow, as well.

Incremental updates require the ability to identify changes made to a previous state of the source. This practice is often referred to as **change data capture** (CDC). There are several methods of change data capture, but two of the most prominent methods are the use of logs and last-modified timestamps. At small enough increments or batches, change data capture enables real-time or streaming updates.



## ► INCREMENTAL UPDATES FROM A CHANGE LOG

Log-based change data capture (CDC) is most commonly used to make incremental updates from databases. The initial sync from a relational database is always a full sync using a `SELECT *` query, as it must return all records and all values. By contrast, subsequent syncs should only retrieve new records and values.

Fivetran database connectors parse the transaction log or **change log**. Change logs contain a full history of updates, new records and deleted records as a list of updates.

These updates may be whole-row, in which a record's unique primary key and all values are tracked, offering a full snapshot of each record. They may also be partial-row, in which only a primary key and changed values are tracked.

The sync workflow looks like so:

- 1 Import a full table using a `SELECT *` query. This initial sync provides a snapshot of the table's entire contents, primary keys and all, at a particular moment in time. Make note of the primary key column; this will later be essential for designating the correct updates and preventing duplicate or conflicting records.  
Suppose we imported a table (right) at 2022-07-30 13:23:04:

id	username	status
1	"D_artagnan"	"inactive"
34	"Goeman"	"active"
45	"Ichabod_Crane"	"active"
94	"jon_rico"	"inactive"
101	"Drosselmeyer"	"active"

- 2 After the specified interval has elapsed, query the change log and identify all the changes that occurred since your import. Our update corresponds with all transactions before `transaction_id = 3`, so we'll start from there.

transaction_id	transaction	timestamp_completed	table	row_id	column	value
1	INSERT	2022-07-30 13:23:02	user	45	status	"active"
2	UPDATE	2022-07-30 13:23:03	user	45	username	"Ichabod_Crane"
3	UPDATE	2022-07-30 17:45:30	user	1	status	"active"
4	DELETE	2022-07-30 17:59:03	user	34	NULL	NULL
5	DELETE	2022-07-30 18:12:59	user	94	NULL	NULL
6	INSERT	2022-07-30 18:23:03	user	13	status	"inactive"
7	INSERT	2022-07-30 18:23:03	user	13	username	"capt_ahab"

- 3 Start reading the change log from the position you identified in the previous step. As you progress through the change log, based on unique row identifiers:
  - Merge updates with the corresponding records in the destination
  - Add new records as new rows with new primary keys
  - Remove deleted records from the destination

id	username	status
1	"D_artagnan"	"active"
13	"capt_anab"	"inactive"
101	"Drosselmeyer"	"active"
45	"Ichabod_Crane"	"inactive"

The incremental data sync workflow depends on idempotence, which we will discuss later. Incremental updates often involve a small amount of backtracking to ensure the complete replication of data. Without idempotence and the ability to properly attribute rows using primary keys, replication can introduce several versions of the same row, depending on how many updates occurred in the interval between the log and the snapshot. This introduces a serious danger of duplicate and conflicting rows.

## ► INCREMENTAL UPDATES FROM A LAST-MODIFIED TIMESTAMP

Data sources that expose data through an API may not have change logs, but may instead have timestamps that indicate when a record was last changed. The goal is to identify, extract and load records that have been added or updated since the last sync.

The process of incrementally updating from a timestamp looks like so:

### Step 1:

For each entity, find the "last modified timestamp," "last modified date," or a similar attribute that denotes when a record was last updated.

### Step 2:

After the initial update, make subsequent updates based on the timestamp being "greater than" the initial timestamp value, called a cursor. This cursor is updated with every subsequent update.

Timestamps offer a simpler approach to incremental updates than change logs, may be considered more intuitive and are more commonly used. The downsides to timestamps are that:

- 1 Unlike change logs, which track actions, timestamps may not track deletes, because a deleted record may simply be absent (if it isn't soft-deleted).
- 2 Since you have to query and scan an entire table or feed, it imposes a heavy load on the source, leading to potential slowdowns.
- 3 Updating from a timestamp only allows you to access a snapshot of data from any given time. If you ever need to reconstruct every change ever made to a record, you can't use this method without taking and separately storing a succession of snapshots. Even then, there is a possibility that the same row may have been updated multiple times between each snapshot, leading to missed values.

## ► HOW TO SOLVE THE INCREMENTAL UPDATE CHALLENGE

**Incremental updates are challenging to properly execute in several regards.**

**One is of scale.** If a data source grows quickly enough, even incremental syncs can become so large and lengthy that the duration of a sync exceeds the specified interval for updating, creating a form of growing data sync debt. Suppose, for instance, that a data connector is scheduled to sync incrementally every five minutes, but the size of the sync is such that it will take seven minutes. Once seven minutes have elapsed, the pipeline is two minutes late starting the next sync. This means that two minutes of extra data has accumulated at the source. This adds additional records to the next sync, which will then take even longer than seven minutes. The sync interval will continue to snowball as more and more data accumulates. The solution to this problem is to write more performant code and more efficient algorithms to minimize sync times.

**Another challenge concerns the granularity of timestamps.** Suppose the "last modified" field in an API feed is only accurate to the second, and your latest timestamp was 5:00:01 PM. You may not know where between 5:00:01 PM and 5:00:02 PM you ended. If you use the "greater than" approach described earlier, your next sync is likely to miss some records that were updated in the interval from 5:00:01 PM to 5:00:02 PM. These missing or outdated records are easily overlooked until they cause serious problems because few records tend to be affected at any particular time. It can be impossible to recover the missing data. Here, as with change logs, you must use "greater than or equal" logic, erring on the side of introducing duplicate records into the workflow.

Sometimes, APIs contain records that are modified at a particular timestamp but aren't available to read until after that timestamp has already elapsed, leading them to be missed. You may have to use your discretion to determine how many records you are willing to resync in order to catch late arrivals.

Finally, sometimes parts or all of a data source will lack change logs or timestamps. In the best-case scenario, you might be able to identify a reasonable proxy for a timestamp, like a transaction or serial number. In the worst-case scenario, you may be forced to schedule full syncs for part or all of the data source.

Incremental updates and duplication lead us to our next topic, idempotence. ————— ►

# Idempotence

Idempotence means that if you execute an operation multiple times, the final result will not change after the initial execution.

One way to mathematically express idempotence is like so:

$$f(f(x)) = f(x)$$

An example of an idempotent function is the absolute value function `abs()`.

$$\text{abs}(\text{abs}(x)) = \text{abs}(x)$$

$$\text{abs}(\text{abs}(-11)) = \text{abs}(-11) = 11$$

A common, everyday example of an idempotent machine is an elevator. Pushing the button for a particular floor will always send you to that floor no matter how many times you press it.



In the context of data movement, idempotence ensures that if you apply the same data to a destination multiple times, you will get the same result. This is a key feature of recovery from pipeline failures. Without idempotence, a failed sync means an engineer must sleuth out which records were and weren't synced and design a custom recovery procedure based on the current state of the data. With idempotence, the data pipeline can simply replay any data that might not have made it to the destination. If a record is already present, the replay has no effect; otherwise, the record is added.

Fivetran offers an idempotent approach to data movement. The Fivetran approach to idempotence leverages:

- 1 Cursors to track progress through batches of data syncs, including failed syncs
- 2 Row identifiers (such as primary keys) in order to identify and remove duplicate or conflicting records

## ► HOW DATA SYNCS FAIL AND WHY FAILURES MATTER

Idempotence comes into play whenever data syncs fail. Data syncs can be interrupted or otherwise fail at several stages of the data movement process.

<b>Source</b>  A data source unexpectedly becomes unavailable, interrupting the sync. This can be the result of a network stoppage or a failure at the source itself.	<b>Pipeline</b>  Sometimes the pipeline itself hiccups and stops working. There are many reasons pipelines can fail: <ul style="list-style-type: none"><li>• Infrastructure failures, i.e. servers going down</li><li>• Wrong or missing credentials</li><li>• Resource limitations, i.e. memory leaks</li><li>• Software bugs</li></ul>
<b>Destination</b>  Failed queries to a data warehouse or lakehouse can interrupt syncs, as can upgrades and migrations. This may happen because of resource constraints, such as busy nodes and scheduled downtime.	

A data pipeline will inevitably experience failures over a sufficiently long period of operation. In every such case, data values that are already loaded may be reintroduced to the destination, creating duplicate or conflicting records.