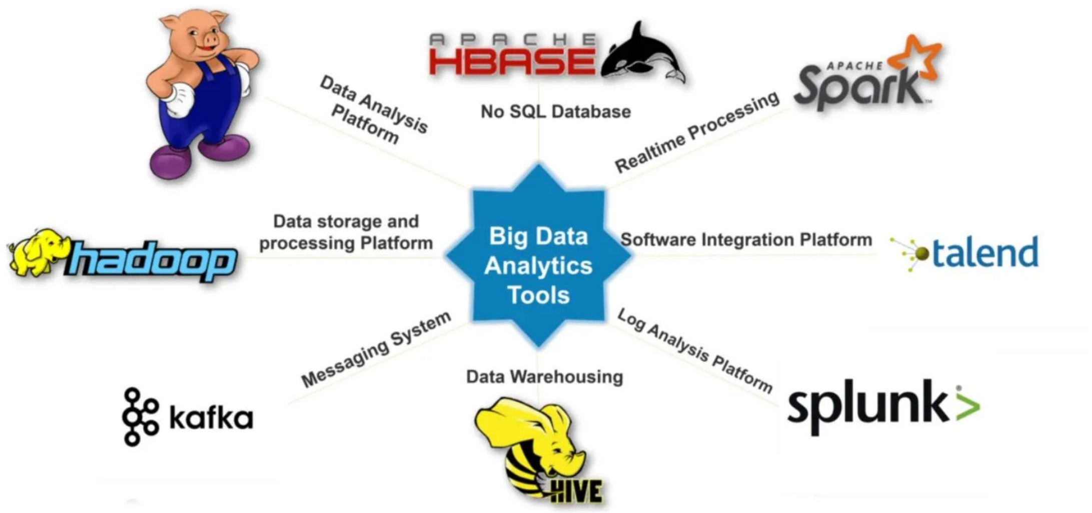
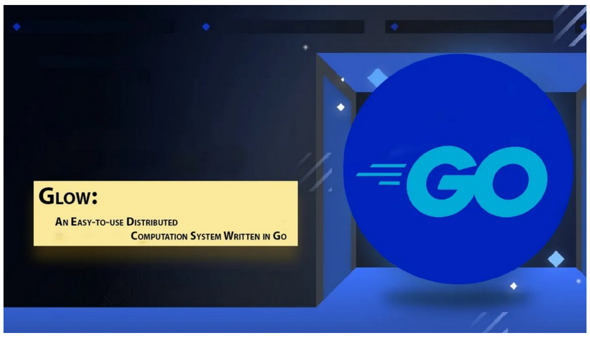
BrandLovrs

Jefferson Marchetti - Head of Data
Março/2024

Pros/Cons of Data Lake Stacks

Comparisons for Decision-making





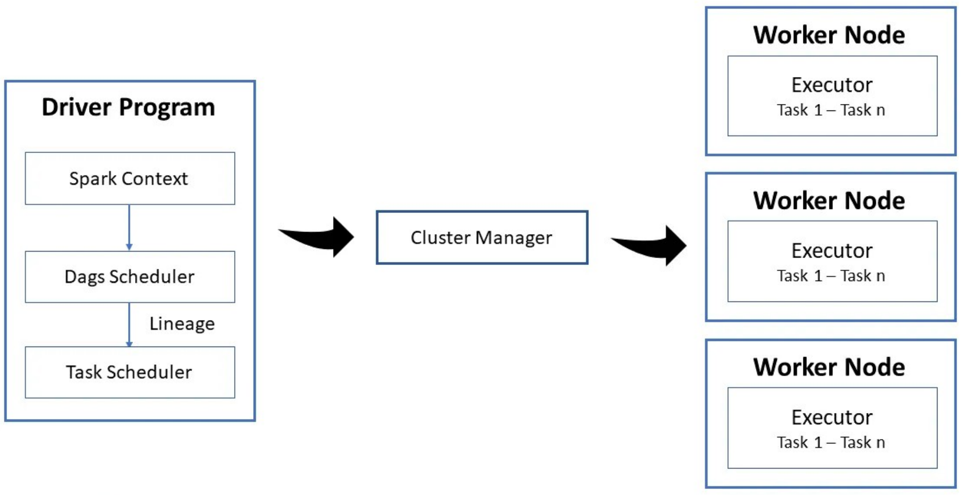
1 - Definitions:

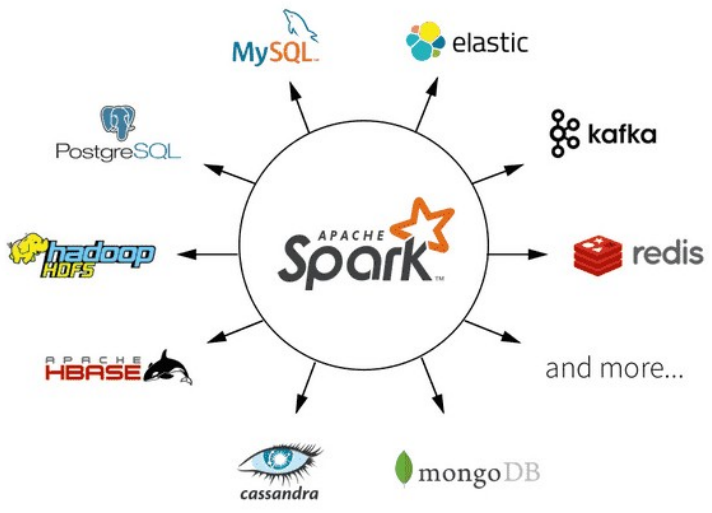
Apache Spark / Hadoop / Scala

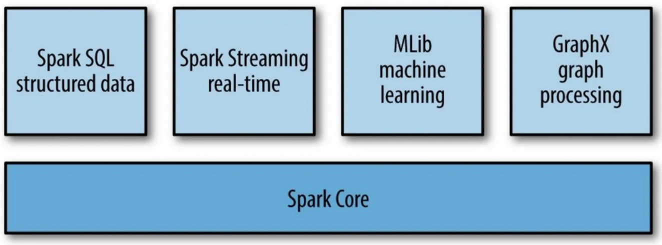
Apache Spark is a unified computing engine and set of libraries for parallel data processing across computer clusters.

One of the great advantages of Spark is being able to work in a distributed way. This means that when dealing with very large data sets or when new data is input very quickly, it can become too much for a single computer.

Instead of trying to process a huge data set, these tasks are divided between several machines in constant communication with each other. In a distributed computing system, each individual computer is called a node and the collection of them all is called a cluster.

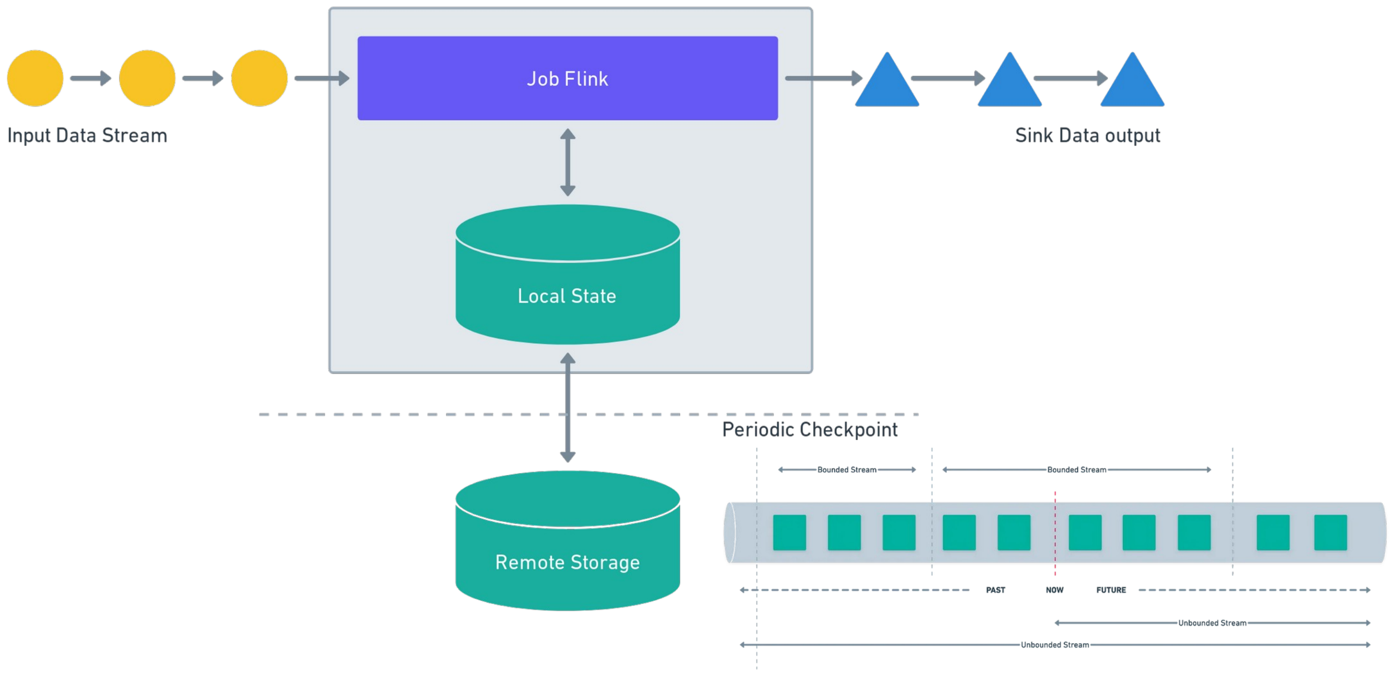






Apache Flink

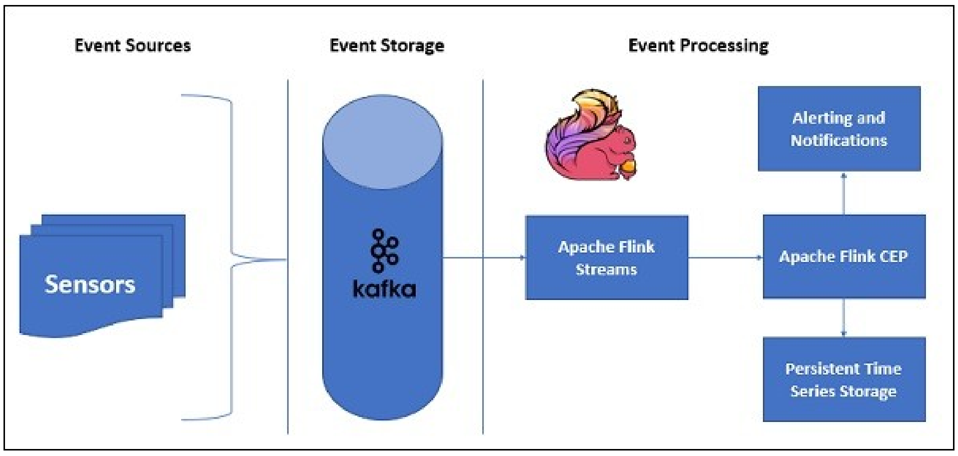
Apache Flink is a framework and distributed processing engine for **stateful** computations on **unbounded** and **bounded** data streams. Flink is designed to run in all common cluster environments and perform processing at **memory speed** and at any **scale**.

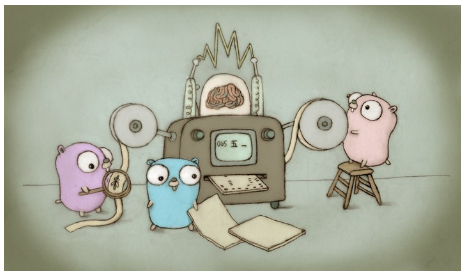


In addition to the architecture itself, there are other concepts that are fundamental and are present in every application that uses Spark.***

The main one is RDD (Resilient Distributed Dataset), which is basically a collection of objects partitioned across several machines in the cluster and which can be processed in parallel. In other words, this is the structure responsible for loading the data, which are partitioned across several machines.

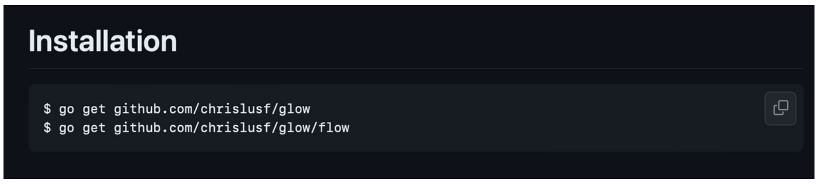
It is important to mention that RDDs are immutable objects, once created they cannot be edited.





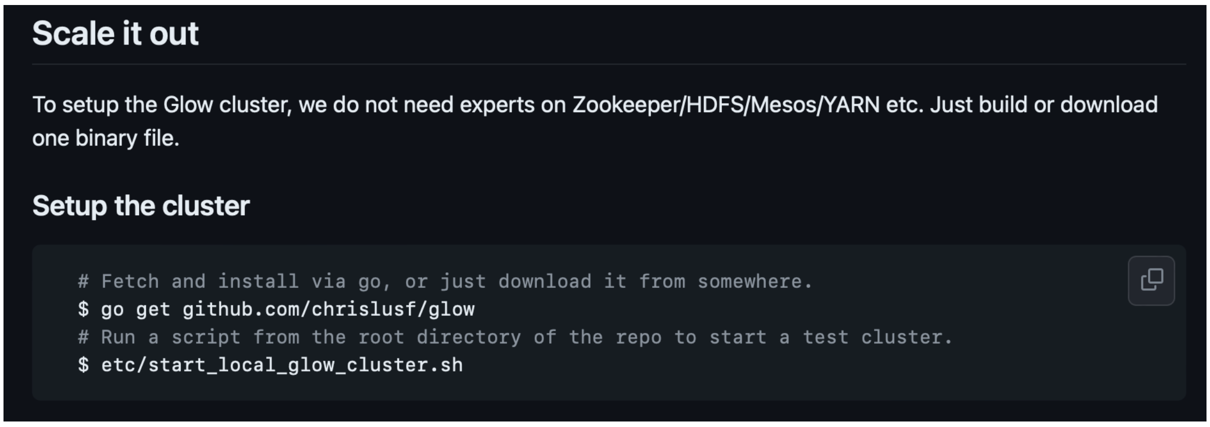
Golang Glow/Gleam

Glow provides a library to easily compute in parallel threads or distributed to clusters of machines.
This is written in pure Go.
I am also working on another pure-Go system, https://github.com/chrislusf/gleam, which is more flexible and more performant.



Installation

```
$ go get github.com/chrislusf/glow
$ go get github.com/chrislusf/glow/flow
```

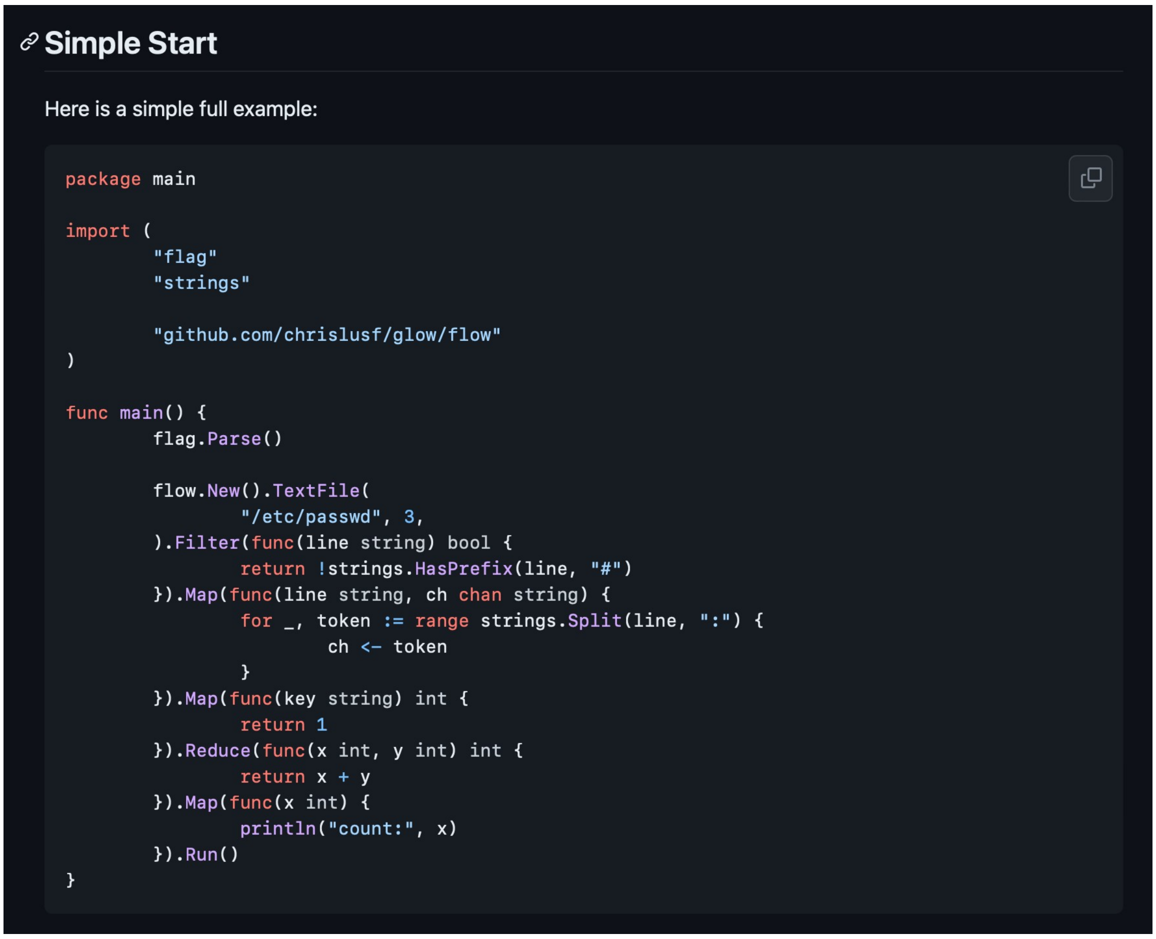


Scale it out

To setup the Glow cluster, we do not need experts on Zookeeper/HDFS/Mesos/YARN etc. Just build or download one binary file.

Setup the cluster

```
# Fetch and install via go, or just download it from somewhere.
$ go get github.com/chrislusf/glow
# Run a script from the root directory of the repo to start a test cluster.
$ etc/start_local_glow_cluster.sh
```



Simple Start

Here is a simple full example:

```go
package main

import (
        "flag"
        "strings"

        "github.com/chrislusf/glow/flow"
)

func main() {
        flag.Parse()

        flow.New().TextFile(
                "/etc/passwd", 3,
        ).Filter(func(line string) bool {
                return !strings.HasPrefix(line, "#")
        }).Map(func(line string, ch chan string) {
                for _, token := range strings.Split(line, ":") {
                        ch <- token
                }
        }).Map(func(key string) int {
                return 1
        }).Reduce(func(x int, y int) int {
                return x + y
        }).Map(func(x int) {
                println("count:", x)
        }).Run()
}
```

2 - Pros and Cons:

Apache Spark

Pros:

- Ease of Use: Spark provides high-level APIs for various languages (Scala, Java, Python, and R), making it accessible to a wide range of developers and data scientists.
- In-Memory Processing: Spark's in-memory computing capabilities enable fast data processing for iterative algorithms and interactive queries.
- Can write natively in all Data Lakes formats such as **Iceberg**, **Delta 2.0**, **Hudi** and native formats such as **Avro**, **Parquet, ORC, CSV, Binary and JSON.**Versatility: Supports batch processing, interactive queries, machine learning, graph processing, and stream processing through Spark Streaming and Structured Streaming.
- Rich Ecosystem: Spark has a large and mature ecosystem with various libraries (MLlib, GraphX, Spark SQL) and integration with tools and protocols like Hive, HBase, Cassandra, **ScyllaDB**, **Akka Actor Models**, **ZIO**, GraphQL, **JDBC native**,
  Prometheus, Elastic Search, Kafka, e recursos nativos da AWS.
- Fault Tolerance: Resilient Distributed Datasets (RDDs) provide fault tolerance by tracking lineage information and enabling data recovery in case of failures.
- Community: Spark has a strong open-source community, resulting in active development, continuous improvement, and a rich set of resources.
- Full support for Kubernetes and Helm Executors, while it has a lot of support in AWS such as Apache EMR, a cloud environment fully prepared and optimized for Spark.

Cons:

- Latency: While Spark Streaming and Structured Streaming offer real-time processing capabilities, they may not provide the same low-latency and high throughput as specialized stream processing frameworks like Flink or PolaRS.
- Complexity: Tuning Spark for optimal performance can be complex, especially for large-scale applications.
- Memory Overhead: Spark's in-memory processing can require substantial memory resources, which may lead to higher hardware requirements and costs.

The main goals of Spark are: It has a complete structure suitable for the most diverse data processing. Native support for many industry standard connectors, file systems, and protocols and a lot of maturity and stability working with huge data lake already in production environments

Apache Flink

Pros:

- ✓ **Low Latency and High Throughput**: Flink is designed for low-latency stream processing and can handle high-throughput real-time data with minimal processing delays.
- ✓ Exactly-Once Processing: Flink provides strong support for exactly-once processing semantics, ensuring data integrity and consistency in event-driven applications.
- ✓ Event Time Processing: Flink has built-in support for event time processing, which is crucial for accurate analysis of time-sensitive data.
- ✓ Dynamic Scaling: Flink supports dynamic scaling, allowing you to adjust the number of processing resources based on workload demands.
- ✓ State Management: Flink offers flexible state management options, making it well-suited for applications with complex stateful computations.
- ✓ Batch and Stream Processing: Flink seamlessly supports both batch and stream processing, making it a versatile choice for hybrid workloads.

Cons:

- ✓ **Learning Curve:** Flink's APIs and concepts may require a steeper learning curve, especially for users new to stream processing paradigms.
- ✓ **Ecosystem:** While Flink's ecosystem is growing, it might not be as extensive as Spark's, which has a broader range of libraries and integrations.
- ✓ **Resource Management:** While Flink supports dynamic scaling, optimizing resource management and fine-tuning performance can still be challenging.

The main goals of Flink are Excels in low-latency, high-throughput stream processing scenarios.

Interesting details:

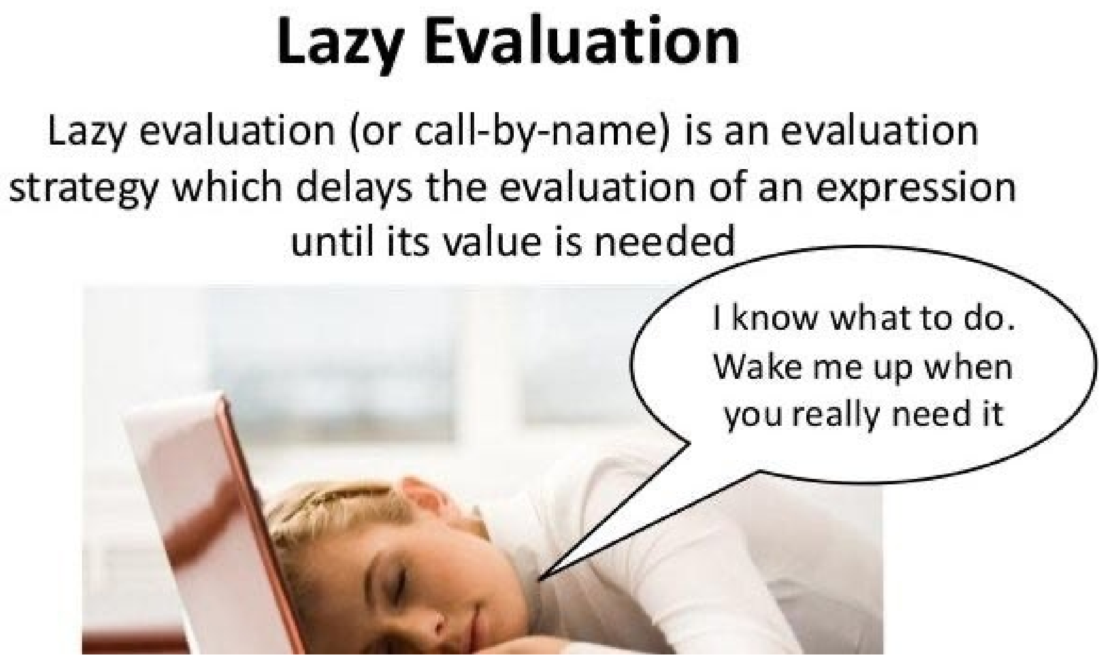
- Spark has been around for a long time.
- Spark has a lot of overhead when running on a single node
- Golang Gleam has not been around for a long time.
  Gleam is going to be faster on a single machine
- Spark's features and functions probably outstrip Gleam, but the average use case wouldn't notice.
- Gleam is easier to install and use on any platform basically.
- Gleam is going to Rust.
- Spark is going to be Scala or Python.

Languages for data comparisons (Golang vs. Scala)

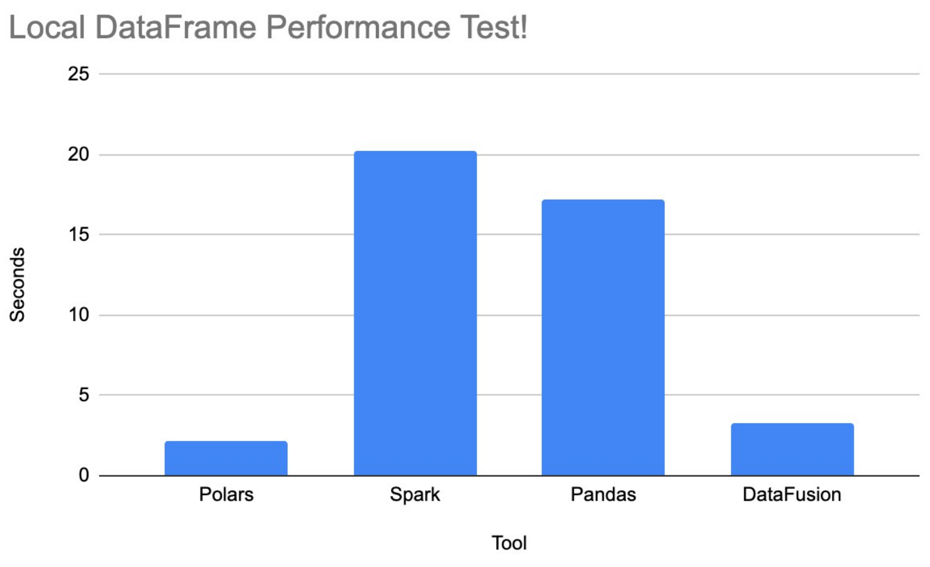
Golang(cons)	**Scala(pros)**
It does have some limitations, including a lack of libraries and limited support for generics to working well with data transformation and treatment of data over schema evolutions.	Functional Paradigm Language, allows you to apply advanced techniques such as Lazy Evaluation, Immutability, monads, strong parallelism, reading, and writing in larger structures and data volumes.
Golang does not have native integration with Python and therefore does not offer good support for market-standard tools such as Spark, Pandas, Polars, Flink, etc.	Scala Is Used to Build Modular and Scalable Software
Now only Kafka has a proper golang client.	Compatible and/or Native with the vast majority of data tooling available on the market today. (Market standard)
Because Golang is procedural, minimalist, and strongly typed. Support for mathematics and generics tends to be more limited compared to more extensive functional languages.	Strong support from the community and large companies that have been working with the Java ecosystem (JVM) and adjacent tools for years.
Due to the lack of mature libraries and frameworks, Golang may become less interesting for working on complex transformations in complex schemas and variant types.	The best option for applying Transformation and Loads to immense and complex volumes of data.
Doesn’t have the native support to Iceberg data lake house.	It has powerful frameworks for resilience, events and high concurrency such as Finagle, Akka, Finatra, ZIO, Play, Cats, Lagom
Doesn’t have the native support to Hudi data lake house.	The language is highly rich and efficient, both for general purposes and mainly for transforming large data.
	Scala Is Used for Building Data-Intensive, Distributed Applications and Systems (**Uber**, **Netflix**, **Twitter** and **LinkedIn** use a lot **Scala**)
	Has the native support to write in the Iceberg, Delta and, Hudi Datalake house.

Golang (pros)	**Scala (cons)**
It runs faster, compiles quicker, it's easy to maintain and support, and allows for shorter software development lifecycles	As it depends on Virtual Machines and large frameworks, memory consumption tends to be always high, resulting in high expenses in the cost of machines and cloud resources.

Golang (pros)	Scala (cons)
Golang compiles directly to machine code as it doesn't use Virtual Machines, providing an even better speed advantage and less use of memory	As it is a complex language with a functional paradigm, the learning curve tends to be high and it is difficult to find professionals easily available on the market.
Concurrency: Go is designed for concurrency, making it easy to write programs that can perform multiple tasks simultaneously.	Scala is hugely complex with surprisingly little payoff in expressiveness. This is especially true for the type system. We get the complexity of type inference multiplied by that of implicits / sub-typing / variants / etc. The complexity of the type system is combinatorial!
It has built-in features for handling concurrency, such as goroutines and channels, which make it easier to write efficient and scalable code.	In scala, is very common we find terms like, variance, invariance, contravariance, covariance, monoids, monads, implicits, Currying, Fold left, OMG!!!!!
Golang allows you to quickly write your own codes that are more business-friendly, without depending on large frameworks, large infrastructure resources, or third-party tooling.	
Golang is excellent for managing processes and execution of goroutines, such as cron, and schedulers, and thus performing specific readings and writings (Extract) supporting different protocols.	


Lazy Evaluation

Lazy evaluation (or call-by-name) is an evaluation strategy which delays the evaluation of an expression until its value is needed

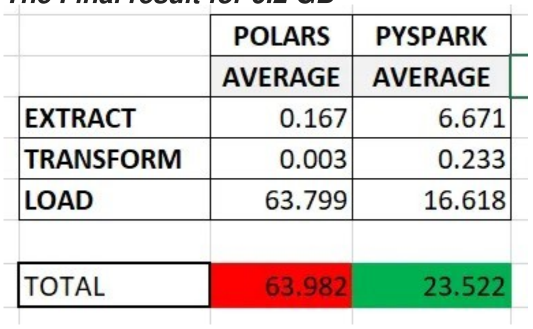


Based on some tests applied on local **Data lakes**, we have the following results comparing the following aspects: Load, Extract, and Transformation:

I found an article on the following topic:

New Polars code to allow lazy evaluation;
New time results with 1.2Gb parquet file;
New test results with a folder with just parquet files and a total of 6.2 GB.

The Final result for 6.2 GB



	POLARS	PYSPARK
	AVERAGE	AVERAGE
EXTRACT	0.167	6.671
TRANSFORM	0.003	0.233
LOAD	63.799	16.618
TOTAL	63.982	23.522

Then Spark takes longer to start but then (for big data) it is faster!

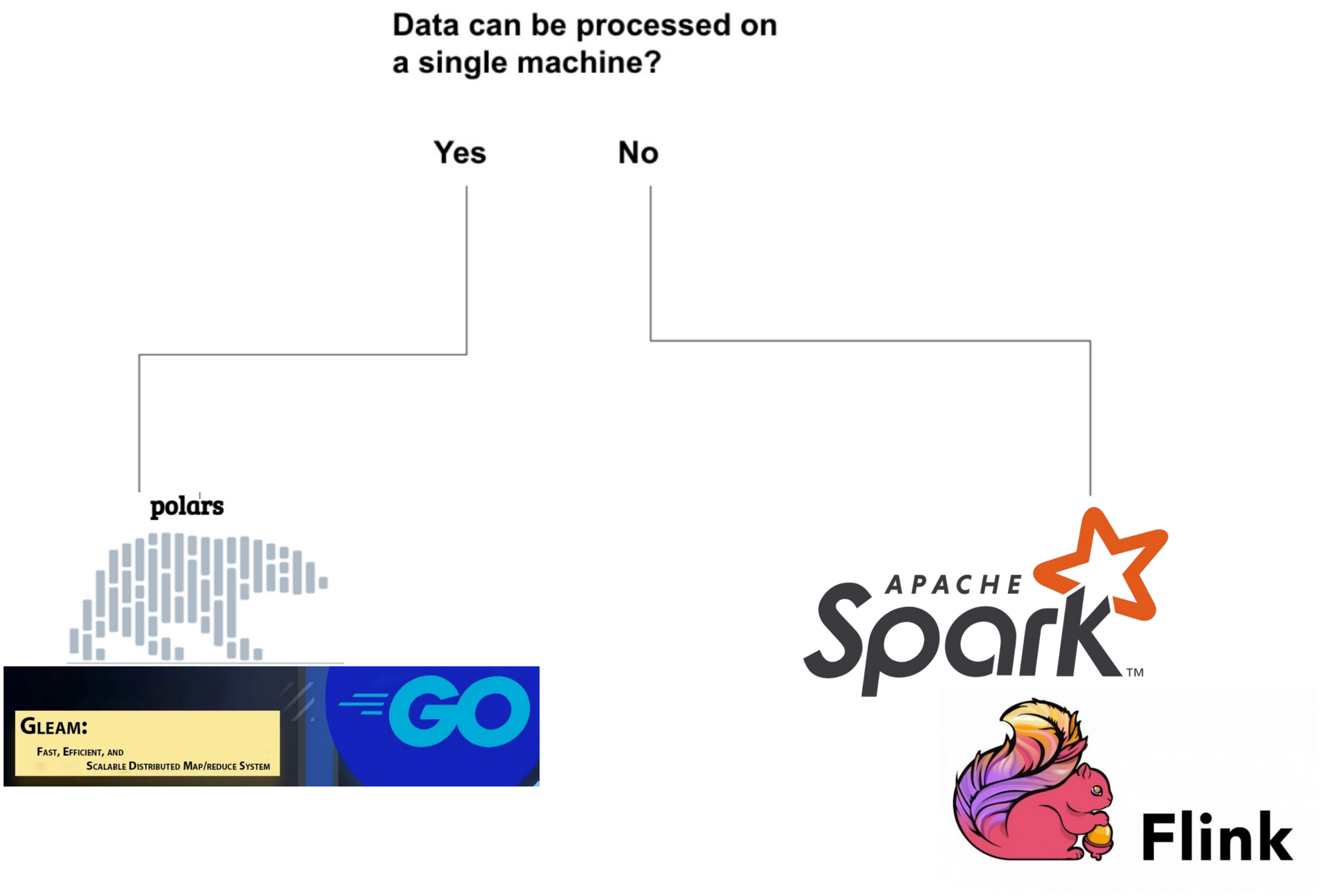
This is the opinion of the author who made these tests during some research:

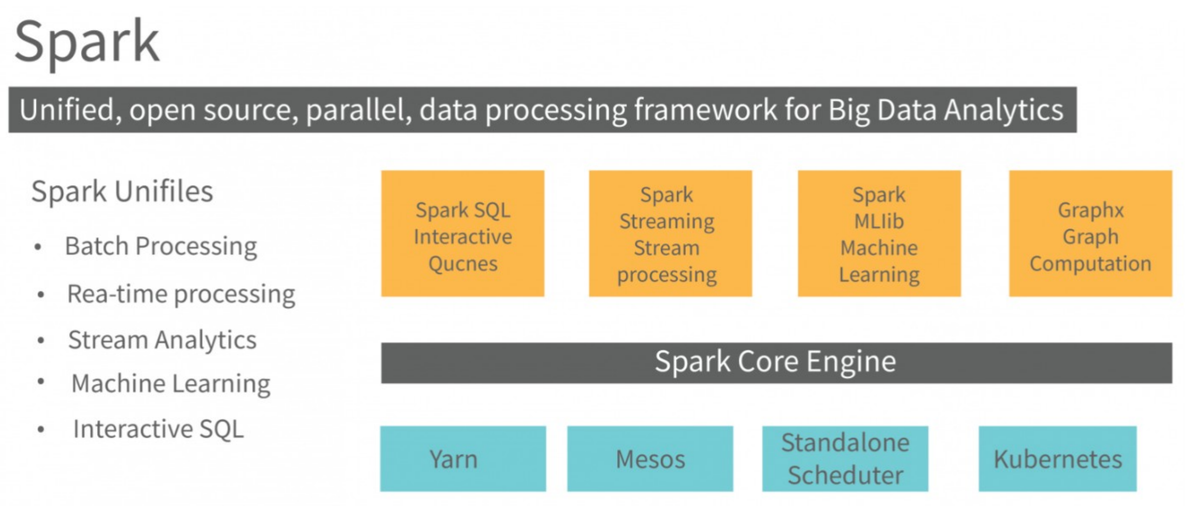
"I feel great potential in Polars 😎 but Spark is still THE big data tool!"

Taking the Polars as an example, I believe from the numbers we have, that our tool created in Golang called Workers, can be very capable of scheduling runs using GoCron and performing large "Extraction" tasks only!!!

Assigning the concepts of Load and Transformation to Spark/Scala/Python to do.

The final question is:


Data can be processed on a single machine?
Yes
No
polars
GLEAM: Fast, Efficient, and Scalable Distributed Map/Reduce System
GO
APACHE Spark
Flink



Spark

Unified, open source, parallel, data processing framework for Big Data Analytics

Spark Unifiles

- Batch Processing
- Rea-time processing
- Stream Analytics
- Machine Learning
- Interactive SQL

Spark SQL Interactive Qucnes | Spark Streaming Stream processing | Spark MLlib Machine Learning | Graphx Graph Computation

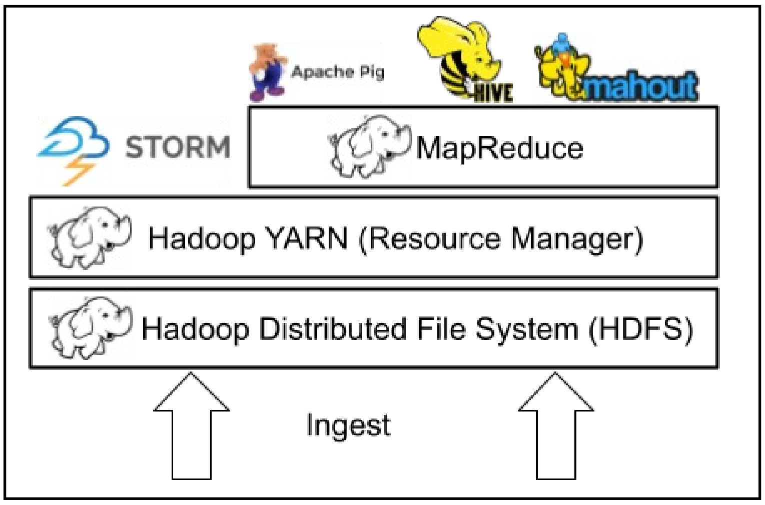
Spark Core Engine

Yarn | Mesos | Standalone Scheduter | Kubernetes

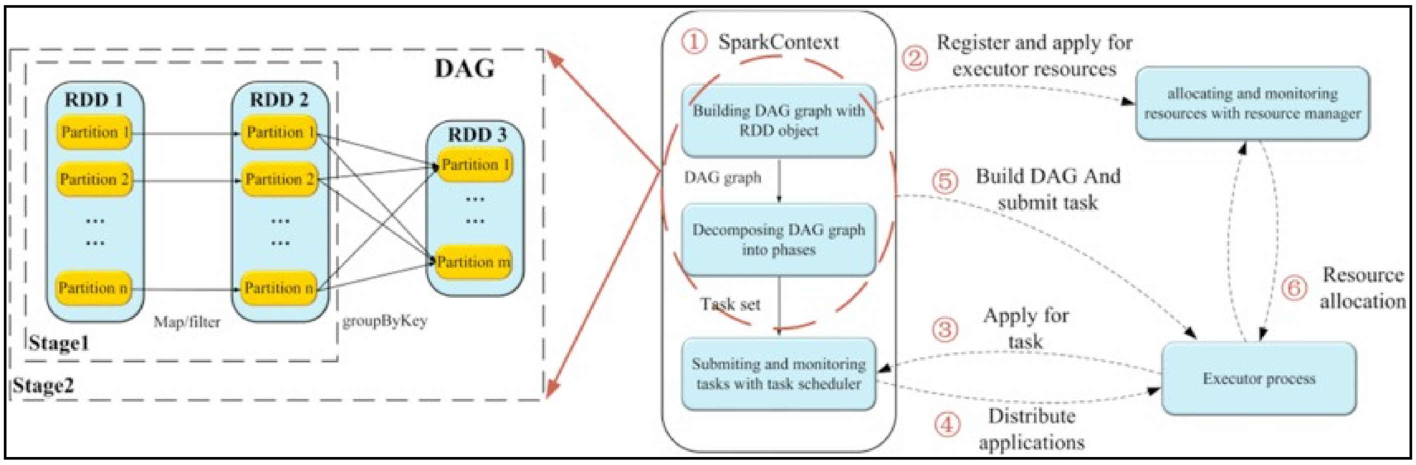
If your previous answer was **No** to single machine and **Yes** to cluster, let's talk a little about Hadoop Ecosystem.

Apache Hadoop

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high availability, the library itself is designed to detect and handle failures at the application layer, delivering a highly available service on top of a cluster of computers, each of which may be prone to failures.


Apache Pig
HIVE
mahout
STORM
MapReduce
Hadoop YARN (Resource Manager)
Hadoop Distributed File System (HDFS)
Ingest

Example of basic pipeline builded in with Spark/Haddop


DAG
RDD 1
RDD 2
RDD 3
Partition 1
Partition 2
Partition n
Partition m
Map/filter
groupByKey
Stage1
Stage2
① SparkContext
Building DAG graph with RDD object
DAG graph
Decomposing DAG graph into phases
Task set
Submiting and monitoring tasks with task scheduler
② Register and apply for executor resources
allocating and monitoring resources with resource manager
⑤ Build DAG And submit task
⑥ Resource allocation
③ Apply for task
④ Distribute applications
Executor process

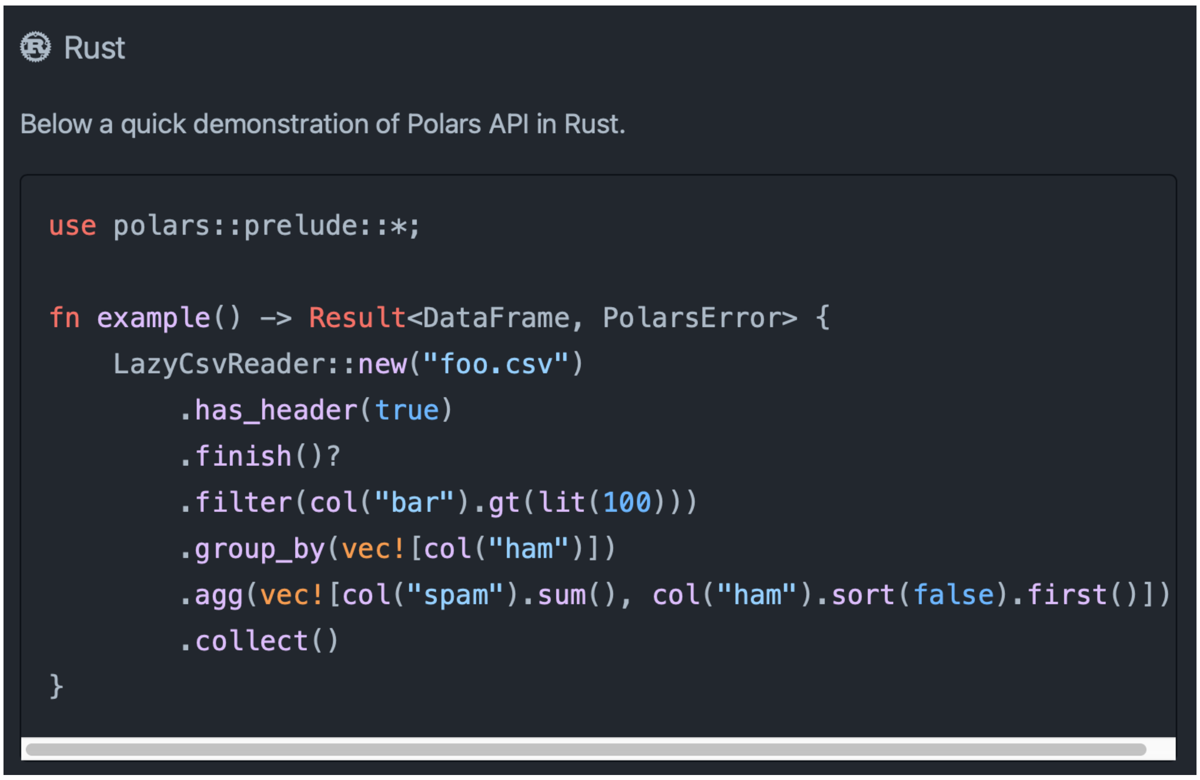
If your previous answer was **Yes** to single machine and **No** to cluster, let's talk a little about **Alternatives** to Hadoop Ecosystem.

For a batch and Dataframes solution:

Polars is a lightning-fast DataFrame library/in-memory query engine. Its embarrassingly parallel execution, cache-efficient algorithms, and expressive API make it perfect for efficient data wrangling, data pipelines, snappy APIs and so much more.

Polars was developed in **Rust** and supports native **Rust** implementations and full **Python** language support.

Example in Rust:



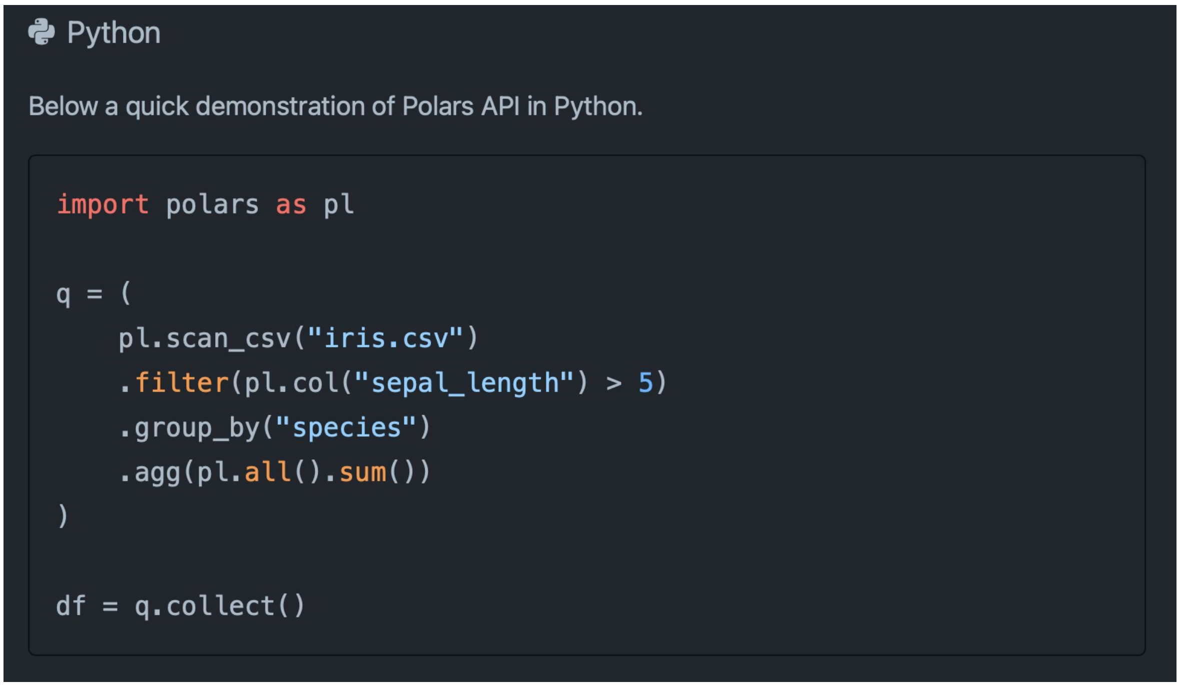
Rust

Below a quick demonstration of Polars API in Rust.

```rust
use polars::prelude::*;

fn example() -> Result<DataFrame, PolarsError> {
    LazyCsvReader::new("foo.csv")
        .has_header(true)
        .finish()?
        .filter(col("bar").gt(lit(100)))
        .group_by(vec![col("ham")])
        .agg(vec![col("spam").sum(), col("ham").sort(false).first()])
        .collect()
}
```

Example in Python using Polars:



Python

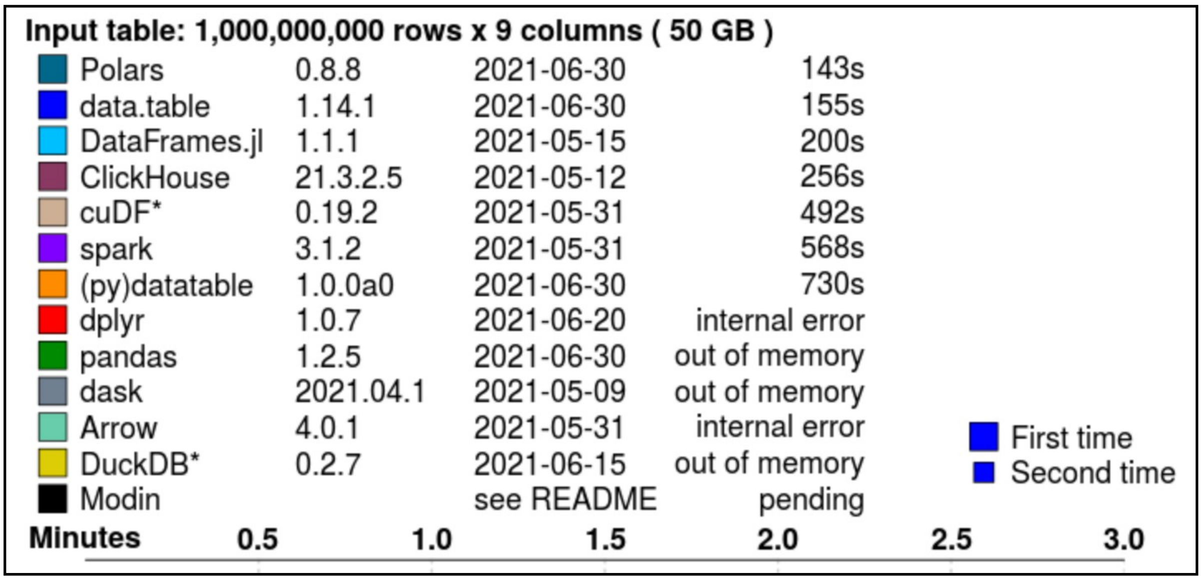
Below a quick demonstration of Polars API in Python.

```python
import polars as pl

q = (
    pl.scan_csv("iris.csv")
    .filter(pl.col("sepal_length") > 5)
    .group_by("species")
    .agg(pl.all().sum())
)

df = q.collect()
```

Performance and Benchmark examples:



Input table: 1,000,000,000 rows x 9 columns (50 GB)

🟦	Polars	0.8.8	2021-06-30	143s
🟦	data.table	1.14.1	2021-06-30	155s
🟦	DataFrames.jl	1.1.1	2021-05-15	200s
🟪	ClickHouse	21.3.2.5	2021-05-12	256s
🟫	cuDF*	0.19.2	2021-05-31	492s
🟪	spark	3.1.2	2021-05-31	568s
🟧	(py)datatable	1.0.0a0	2021-06-30	730s
🟥	dplyr	1.0.7	2021-06-20	internal error
🟩	pandas	1.2.5	2021-06-30	out of memory
⬜	dask	2021.04.1	2021-05-09	out of memory
🟩	Arrow	4.0.1	2021-05-31	internal error
🟨	DuckDB*	0.2.7	2021-06-15	out of memory
⬛	Modin		see README	pending

Big Data and the Data Lakes

What Is Big Data?

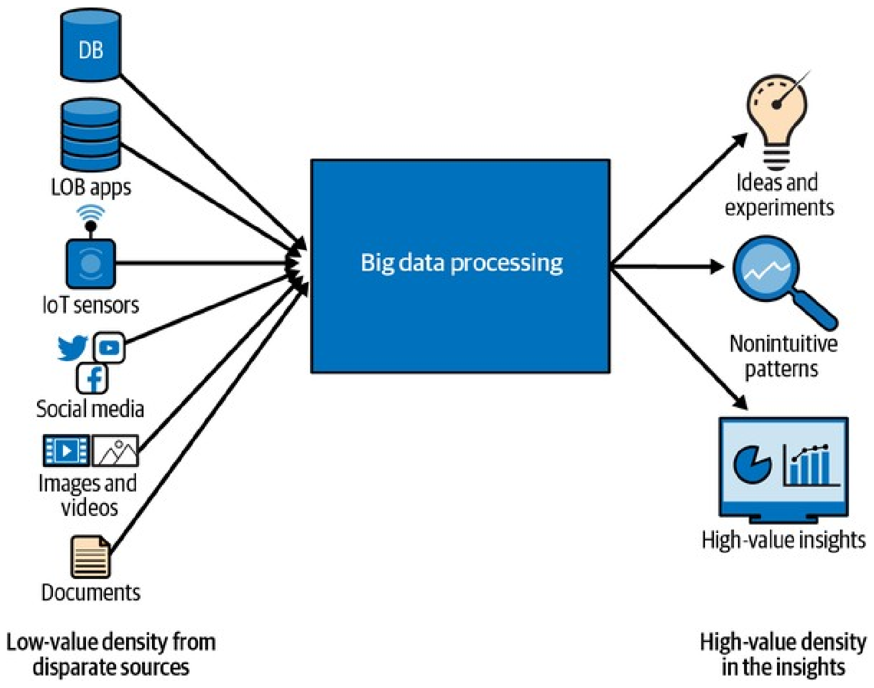
All of the examples given previously share a few things in common:

These scenarios illustrate that data can be explored and consumed in a variety of ways, and when the data is generated, there is not really a clear idea of the consumption patterns. This is different from traditional online transaction processing (OLTP) and online analytical processing (OLAP) systems, where the data is specifically designed and curated to solve specific business problems.

Data can come in all kinds of shapes and formats: it can be a few bytes emitted from an IoT sensor, social media data dumps, files from line of business (LOB) systems and relational databases, and sometimes even audio and video content.

The processing scenarios of big data are vastly different—whether they are data science, SQL-like queries, or any other custom processing.

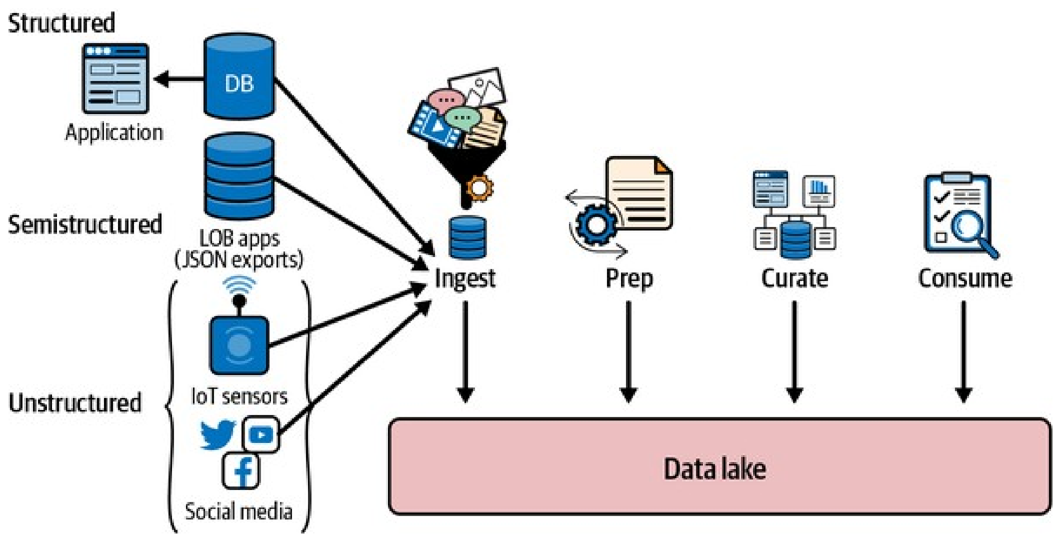
As studies show, big data is not just high volume but can also arrive at various speeds: as one large dump, such as data ingested in batches from relational databases, or continuously streamed, like clickstream or IoT data.



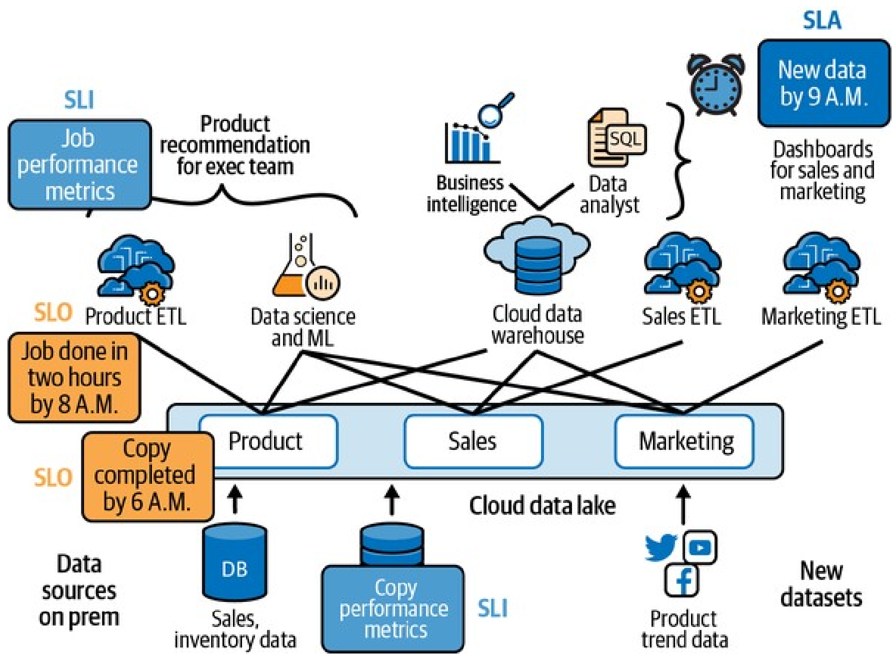
Why do we need a DataLake House? By the way, what is a Data Lake?

As we saw in “**What Is Big Data?**”, the big data scenarios go way beyond the confines of traditional enterprise data warehouses. **Data lake architectures** are designed to solve these exact problems since they were designed to meet the needs of the **explosive growth of data** and their **sources** without making any assumptions about the source, formats, size, or quality of the data. In contrast to the problem approach taken by traditional data warehouses, **Data lakes take a data-first approach**. In a Data lake architecture, all data is considered to be useful—either immediately or to meet a future need.

The first step in a Data architecture involves ingesting data in its raw, natural state, **without any restrictions on the source, size, or format of the data**. This data is stored in a Data lake, a storage system that is **highly scalable** and can **store any kind of data**. This raw data has variable quality and value and needs more transformations to generate high-value insights.



The processing systems on a **Data lake** work on the data that is stored in the data lake and allow the data developer to define a **schema** on demand, and describe the data at the time of processing.



These processing systems then operate on low-value unstructured data to generate high-value data that is often structured and contains meaningful insights.This high-value, structured data is then either **loaded** into an **enterprise data warehouse** for consumption or consumed directly from the data lake.

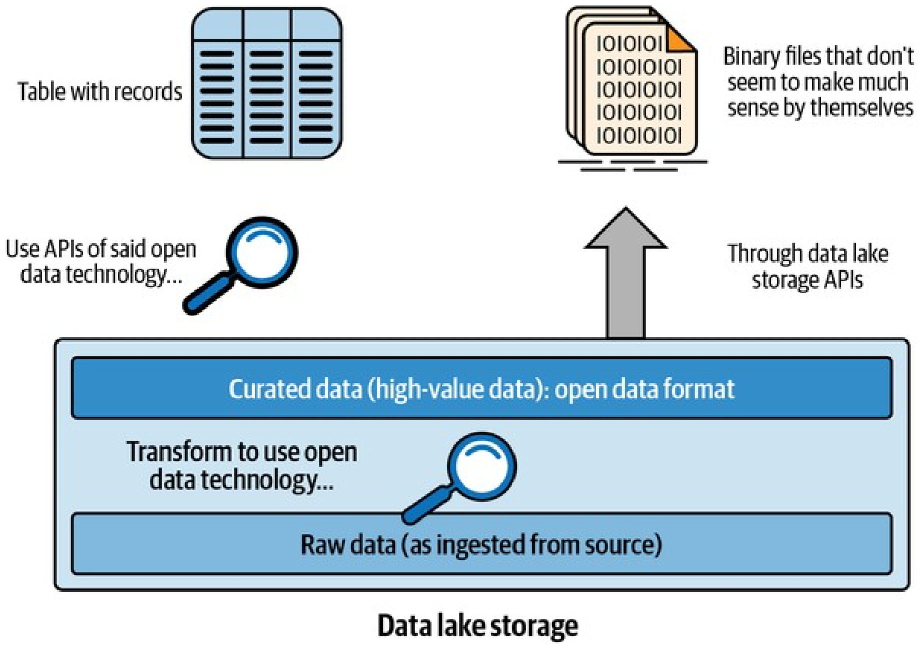


Table with records
Binary files that don't seem to make much sense by themselves
Use APIs of said open data technology...
Through data lake storage APIs
Curated data (high-value data): open data format
Transform to use open data technology...
Raw data (as ingested from source)
Data lake storage

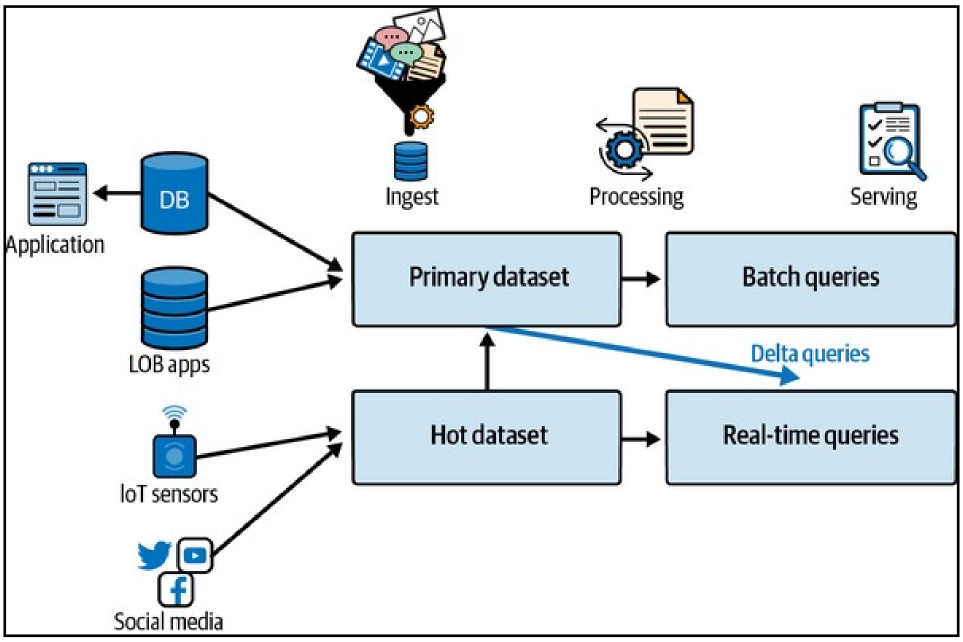
What are the main DataLake House open platforms?

- Delta Lake
- Iceberg
- Hudi

1 - Delta Lake

Delta Lake is an open data format incubated and maintained by Databricks, the company started by the founders of Apache Spark. As we saw in “Apache Spark”, Apache Spark enabled a unified programming model across a variety of scenarios, such as batch processing, real-time streaming, and machine learning scenarios on a unified platform in a cloud data lake architecture.

The final piece of the puzzle was to remove the silo of needing a data warehouse for BI scenarios. Delta Lake was the underpinning of the data lakehouse pattern that Databricks popularized, where, in addition to batch, real-time, and machine learning scenarios, organizations could also run BI scenarios directly on the cloud data lake storage, without requiring a separate cloud data warehouse.



How Does Delta Lake Work?

Delta Lake is an open storage format used to store tabular data in data lake storage systems, offering ACID guarantees. A Delta Lake table consists of the following components:

Data objects

The actual data in the table is stored as Parquet files. You can review the concepts behind Apache Parquet in "Exploring Apache Parquet".

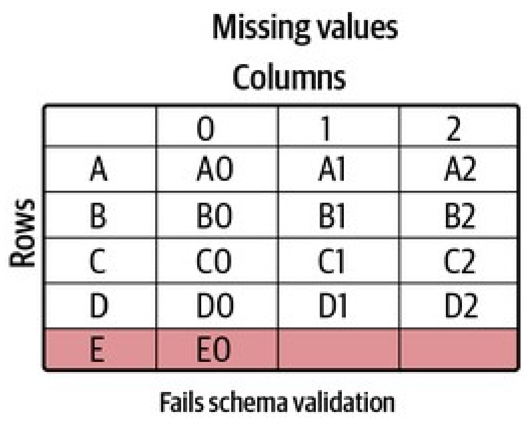
Log

A transaction log, or a ledger, keeps track of changes to the data in the table. These changes are called actions and are stored in JSON format. Delta log keeps track of changes to the data itself; the inserts, deletes, or updates; and the changes to the metadata or the schema, where columns are added or removed from the table.

Log checkpoints

A compressed version of the log that contains nonredundant actions up to a certain point in time. As you might imagine, given the number of actions that happen to data over time, the log could grow a lot, so the log checkpoints serve as an optimization for performance.

The Delta Lake documentation page provides detailed instructions for how to work with Delta tables. When you create a Delta table, a log is also created for that table. All changes to the table are recorded in the log, and this log is crucial to maintaining the integrity of the data in the table, thereby offering the guarantees we discussed.

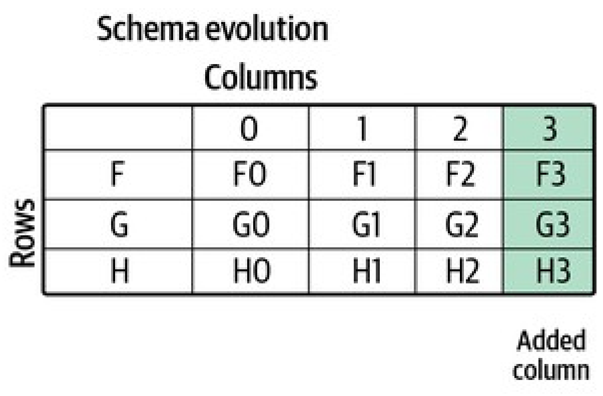


Missing values

Columns

Rows	0	1	2
A	A0	A1	A2
B	B0	B1	B2
C	C0	C1	C2
D	D0	D1	D2
E	E0		

Fails schema validation



Schema evolution

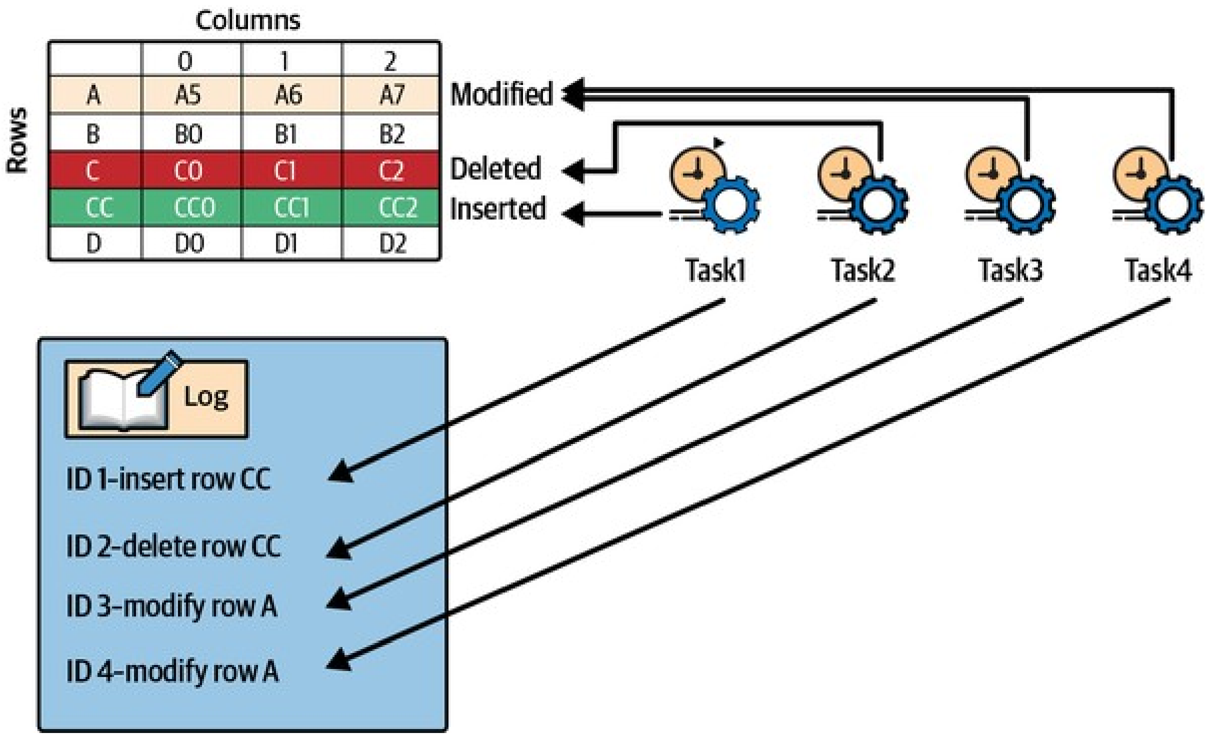
Columns

Rows	0	1	2	3
F	F0	F1	F2	F3
G	G0	G1	G2	G3
H	H0	H1	H2	H3

Added column

The Delta Lake documentation page provides detailed instructions for how to work with Delta tables. When you create a Delta table, a log is also created for that table. All changes to the table are recorded in the log, and this log is crucial to maintaining the integrity of the data in the table, thereby offering the guarantees we discussed.

Make the updates to the data objects by modifying the Parquet files.



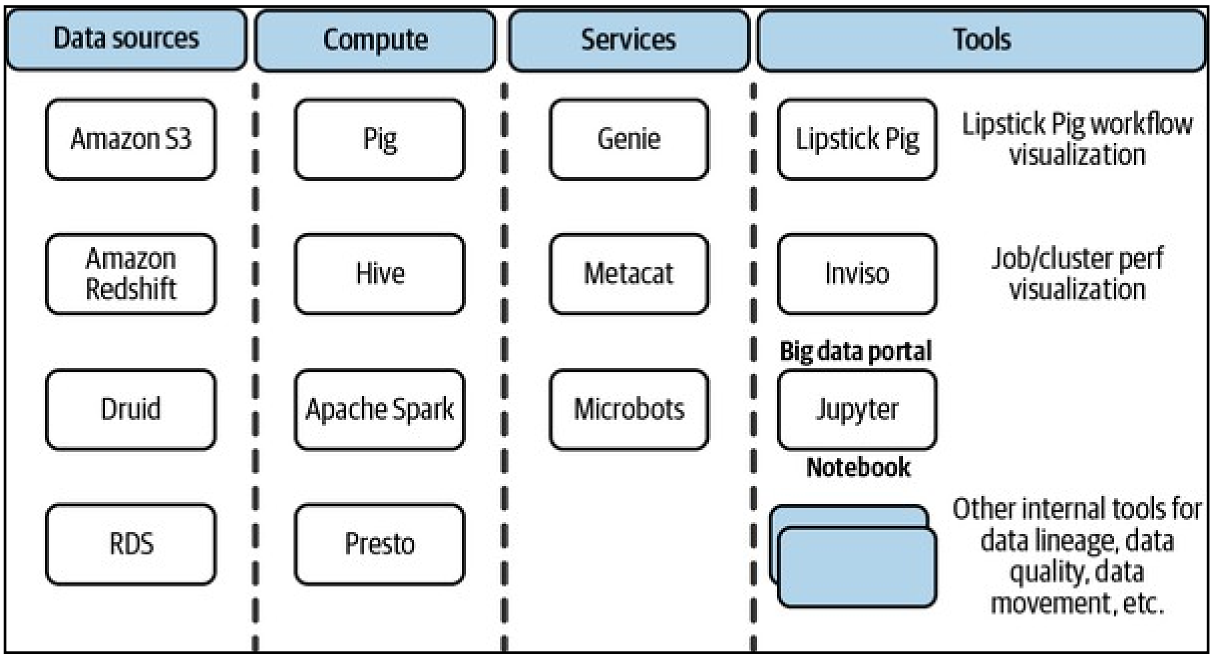
Update the Delta log and associate that modification with a unique identifier in the Delta log.

When Do You Use Delta Lake?

Delta Lake provides stronger guarantees to data that resides in general-purpose object storage. One thing you need to remember is that for data stored in the Delta Lake format, you need to leverage compute engines that understand the format to take advantage of its capabilities. I recommend that you leverage Delta Lake on the data that you expect to run SQL-like queries on or datasets that power machine learning models where you need to be able to preserve the versions. If you use Apache Spark, you can largely leverage your existing pipelines with minimal modifications to convert your existing data to Delta Lake format.

2 - Apache Iceberg

Apache Iceberg was incubated by Netflix as it was powering its business-critical applications on top of the data lake storage, with the shortcomings described in "Why Is It a Problem to Store Tabular Data in a Cloud Data Lake Storage?".



Netflix data architecture diagram.

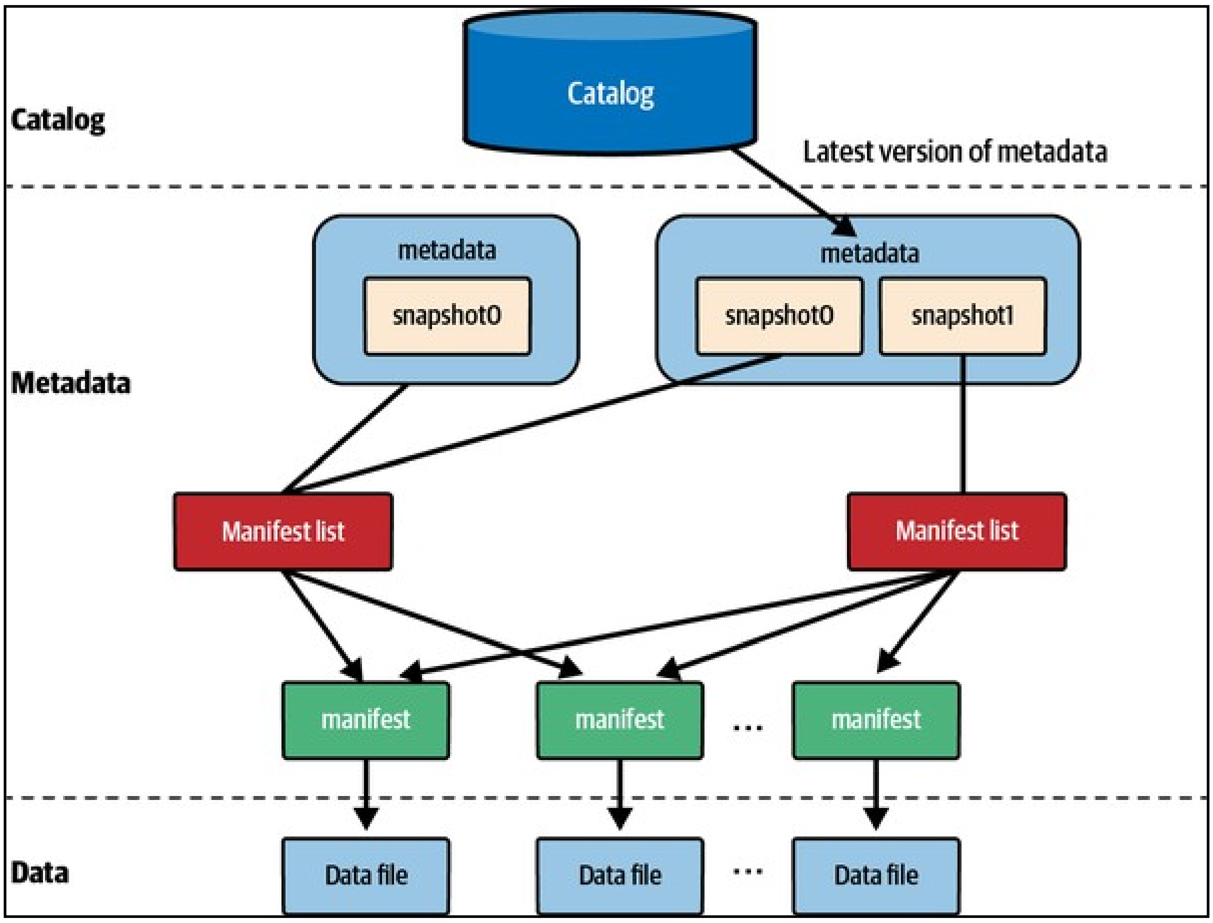
How Does Apache Iceberg Work?

Apache Iceberg, interestingly, builds on top of existing data formats, so you could use it on top of your existing data. The physical data is stored in open data formats like Apache Parquet and Apache ORC.

The simplest way to describe Apache Iceberg is that it is a translation layer between the physical storage of data (Apache Parquet or Apache ORC) and how it comes together and is structured to form a logical table.

Apache Iceberg stores the files that contribute to the table in a persistent tree structure. The state of the table is stored in multiple files that describe the metadata, as follows:

- A catalog file that has the pointers to the latest version of the metadata and is the primary source of truth for where the latest version of the metadata is
- A snapshot metadata file that stores the metadata about the table, such as the schema, the partitioning structure, and so on
- A manifest list for this snapshot, where there is an entry for each manifest file associated with the snapshot
- A set of manifest files that contains a list of paths to the files that actually store the data as well as metrics about the data itself, such as the minimum and maximum values of columns in the dataset

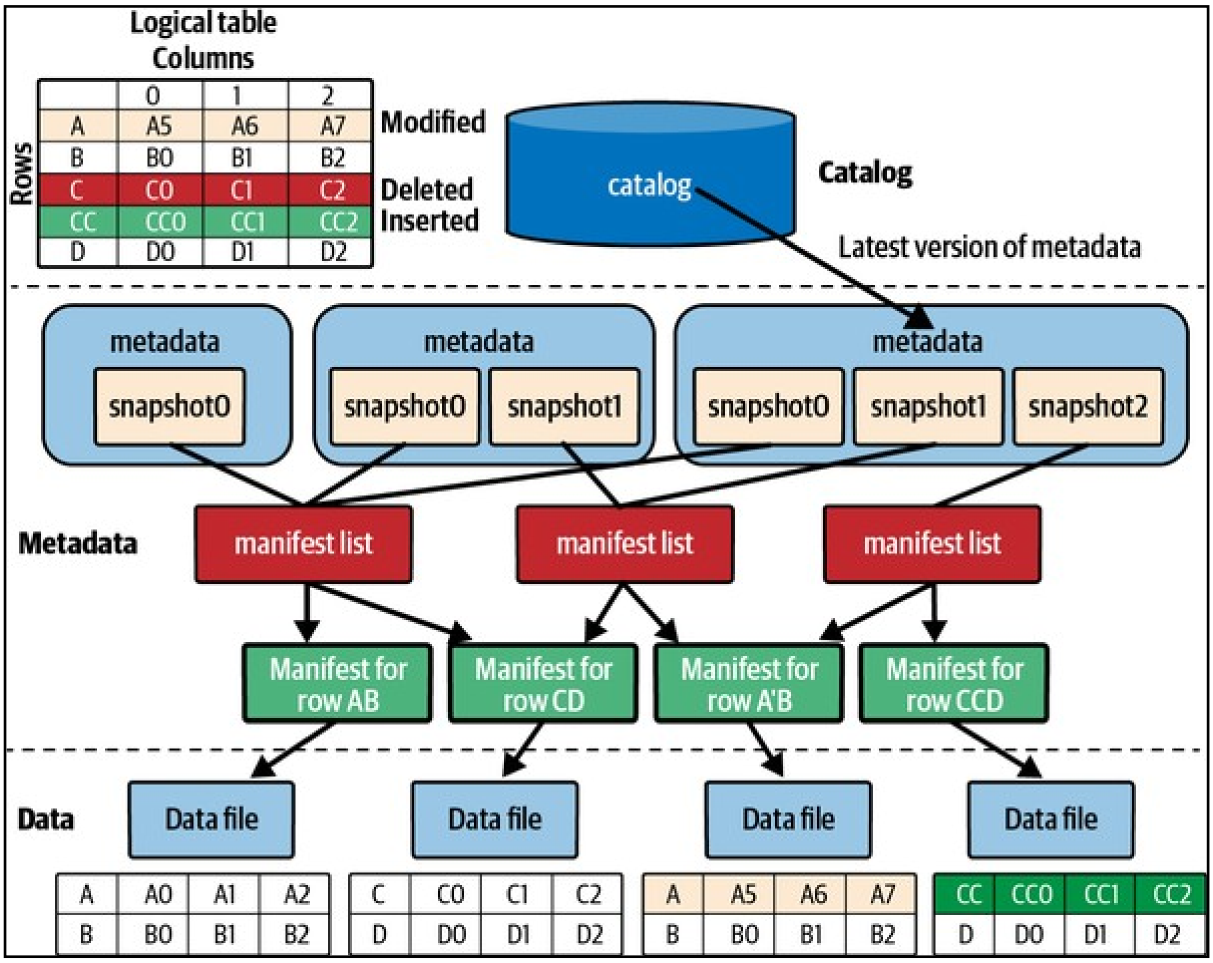


As you can see, initially the Apache Iceberg table has rows A, B, C, and D and is stored across two data files. There are manifest files that point to these data files and a manifest list that contains pointers to these two manifest files. Let me walk you through the sequence of how a change to this dataset is managed in Apache Iceberg:

- When row A is modified first, that data is written in a new data file, and a new manifest file is created to point to this new row. A new snapshot is created and preserved in a manifest list file. Apache Iceberg then updates the catalog to point to this new manifest list file. The write succeeds when all of these succeed, and the writes are completed only when the catalog file updates the metadata pointer to the newest version; this controls multiple writes.

- When row C is deleted and row CC is inserted, a similar process is repeated, and the catalog now points to the newest version of the data.

- When there is a read query that filters only for rows A and B, the manifest files help indicate that the data files containing rows CC and D can be skipped, thereby avoiding unnecessary reads of files and increasing query performance.



When Do You Use Apache Iceberg?

Similar to Delta Lake, Apache Iceberg is used for curated data and enriched data zones, where data assumes a tabular format, and also for queries. Prioritize the data where strong guarantees are expected. Since this is update-heavy, typically this is the curated data zone in the table.

Apache Iceberg is best suited for when you need structural guarantees on your data without having to require a specific format to be used for your underlying data. Apache Iceberg offers the following capabilities that you can leverage in your architecture:

Support for schema evolution

When the schema is updated—that is, new columns are added or existing columns are deleted—these updates are preserved in the snapshots, and you can leverage them to understand how the schema has evolved over time.

Data partition optimization

As we saw in "Data Formats", placing similar data closer together offers increased performance for your queries because you are optimizing for the smallest number of transactions to get to the data that you want. When your underlying data changes, you can continue making changes to how the data files are organized and partition the data better, while the applications continue to call into the manifest without having to understand these optimizations, thereby allowing the flexibility to optimize your data layouts without worrying about breaking your applications.

Time travel or rollback

Specifically, in machine learning scenarios, models are built using a certain version of the data, and as the data changes, the model behavior also changes. In Apache Iceberg, you have snapshots that preserve specific versions of data, so you can tie your application to a version of the data by associating it with a specific snapshot or version. This concept is called snapshot isolation.

Apache Iceberg is very similar to Delta Lake in offering a tabular structure for your data in the data lake; however, the way it accomplishes that is with the metadata layer. Apache Iceberg supports non-Parquet file formats as well, so if you need data in varied formats to have a tabular representation, Apache Iceberg works great

3 - Apache HUDI

Apache Hudi was incubated by Uber, a company that started as one of the first ride-sharing applications.

Data and advanced machine learning capabilities power critical Uber business operations, such as predicting drivers' estimated times of arrival, making meaningful recommendations to Uber customers for ordering their next dinner, and ensuring rider and passenger safety, to name a few. The timeliness of these features—that is, the real-time recommendations or actions—are critical to ensuring a great customer experience.

In addition to other challenges involving the importance of real-time insights, Uber incubated Apache Hudi to offer strong guarantees and timely insights on data that resides in a data lake. Apache Hudi was designed to support these operations and data guarantees at scale, supporting around 500 billion record updates per data on a 150 PB data lake as of the year 2020, which is only growing as Uber continues to scale as a business.

Why Was Apache Hudi Founded?

The motivations for Apache Hudi are largely similar to those for Apache Iceberg and Delta Lake, in that it was founded to overcome the inherent limitations of the general-purpose object storage used to power the data lake storage. Specifically, Uber wanted to address the following scenarios:

Upserts for efficient writes

Upsert refers to a concept where data needs to be written as an insert operation when it doesn't already exist or as an update operation if the row already exists. In a scenario where upserts are not supported, there are multiple rows instead of one, and a separate computation needs to be written to fetch these rows and filter for the most recent data. This solution costs more as well as takes more time for processing. Object storage systems are largely append only, and they do not inherently support the concept of upserts.

Understand incremental modifications

As we saw in previous chapters, data processing pipelines run jobs that process large numbers of datasets to generate highly curated data that constitutes aggregated, filtered, and joined versions of the input data. Typically, whenever the input data is changed, these processing engines recompute the curated data over the entire dataset. To speed up the time to insights, instead of reprocessing the whole batch, there was an opportunity to run the recomputations only on the datasets that changed, thereby processing only the incremental changes. To do this, there was a need to understand what had changed since the last job, and this operation is not naturally supported by the data lake storage as is.

Support real-time insights

As we saw in "Why Was Delta Lake Founded?", while there are programming models that offer unified computation across real-time and batch streaming, the actual architectures and implementations are different. As an example, when there are decisions to be made, such as finding the best drivers to call for a particular ride, the real-time data about the driver's location is combined with the batch data of maps and traffic optimizations to book the best driver. Similarly, when serving recommendations in Uber Eats, the real-time clickstream data of what the customer is browsing at that moment is combined with the recommendations data, which is possibly in a graph database, to serve the right recommendations. Once again, the data lake storage as is was not natively designed for these scenarios.

Apache Hudi was primarily designed to drive efficiencies with support for upserts and incremental processing that enable data freshness in minutes as opposed to recomputing the entire dataset from scratch.

How Does Apache Hudi Work?

Apache Hudi, like other formats, is an open data format that is used to store tabular data. In a lambda architecture that supports both real-time streaming scenarios and batch scenarios, you have two kinds of write patterns on the data:

- Continuous ingestion of large volumes of data—think of a ton of Uber vehicles transmitting information in real-time.
- Batch ingestion of data in bulk—think of dumps of sales or marketing data that are done with daily jobs.

To support these patterns, Apache Hudi offers two kinds of tables:

Copy on write
There is one source of truth that both readers and writers of the table interact with. Every write is immediately written as an update to the Apache Hudi table, and the updates are reflected in near real-time to the readers. The data is stored in a columnar format optimized for reads, such as Apache Parquet.

Merge on read
Every write is written into a buffered zone in a write-optimized data format (a row-based data format such as Apache Avro), and this is later updated to the table that serves the readers, where data is stored in a columnar format (such as Apache Parquet).

Apache Hudi consists of three main components that are stored in a table:
Data files
The files containing the actual data. For copy-on-write tables, the data is stored in a columnar format. For merge-on-read tables, data is stored as a combination of incremental writes stored in row-based formats and the full dataset stored in columnar formats.

Metadata files
This is a complete set of all transactions stored as an ordered timeline of activities on a table. There are four types of transactions on an Apache Hudi table:

Commits
An atomic write operation of a batch of records into a dataset stored in an Apache Hudi table.

Delta commits
An atomic write operation of a batch of records into a delta log, which needs to later be committed to the dataset; this operation is supported only on the merge-on-read type of tables.

Compaction
A background process of optimizing data stored by reorganizing its file structure, where the delta files are merged into the columnar format in the dataset.

Cleans
A background process where older versions of data that are no longer required are deleted.

Index
A data structure that enables efficient lookups of the data files belonging to the transactions.

Three types of reads are supported on Apache Hudi tables:

Snapshot queries
Query a specific snapshot of the data stored in the Apache Hudi table. A *snapshot* refers to the version of the data for a given time. This query returns all the data that matches your query.

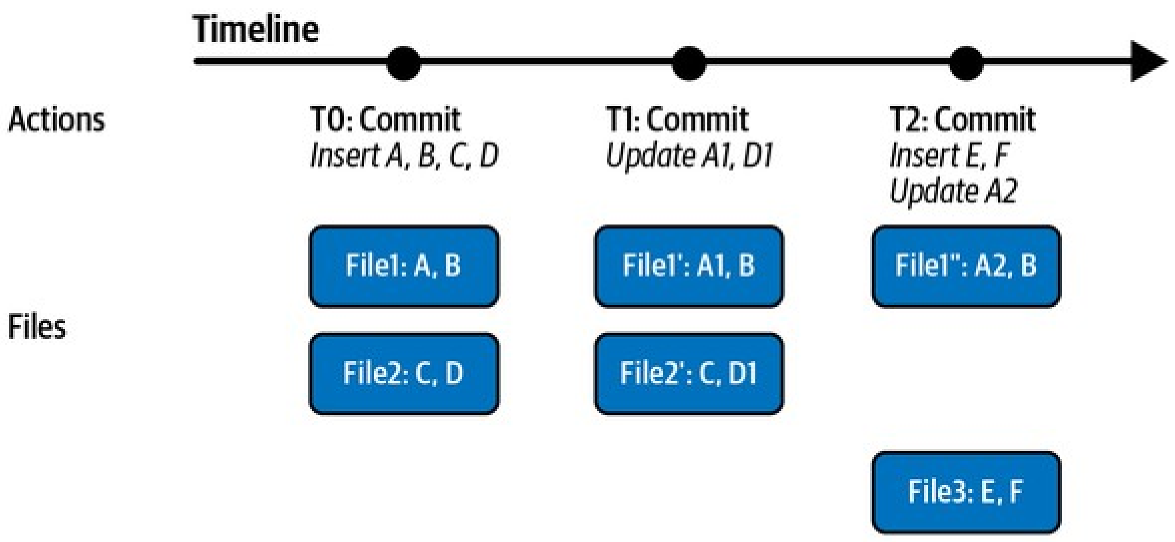
Delta queries
Query data that has changed over a given time period. If you are only interested in what changed, you will use delta queries.

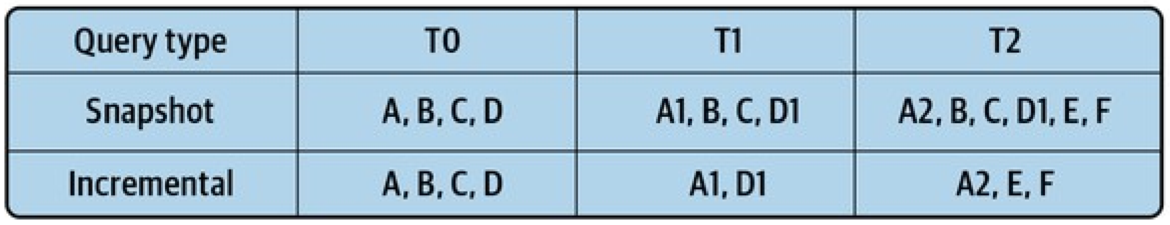
Read optimized queries
This query type is supported for the merge-on-read tables and returns the data that is stored in a format optimized for reads. This does not include the data in the delta files that is not yet compacted. This query is optimized for faster performance and comes with the trade-off of data not being the freshest.

Copy-on-write tables

In the copy-on-write tables, every write—whether it is an insert operation where new rows are inserted, an update operation where existing rows are updated, or a delete operation where existing rows are deleted—is treated as a commit.

There is one source of truth for the dataset preserved in a columnar format, and every write is an atomic operation where the dataset is updated. Snapshots of this truth are preserved, where you see what the data was at a particular point in time.



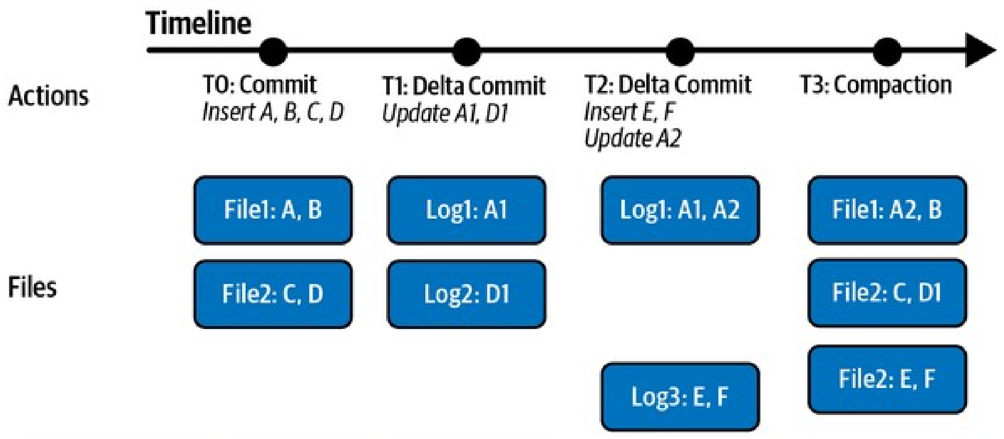


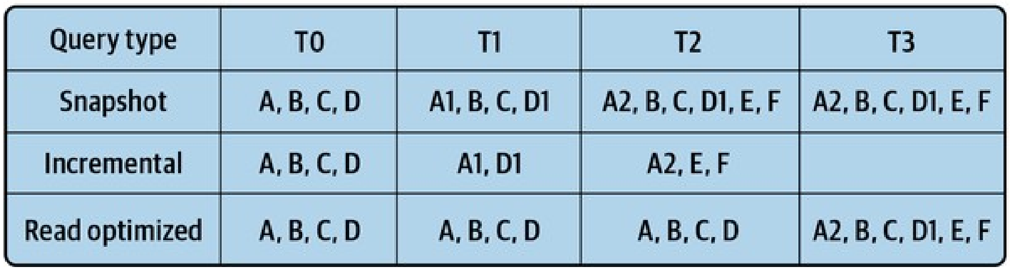
Query type	T0	T1	T2
Snapshot	A, B, C, D	A1, B, C, D1	A2, B, C, D1, E, F
Incremental	A, B, C, D	A1, D1	A2, E, F

Merge-on-read tables

In the merge-on-read tables, there is a dataset in a columnar data format optimized for reads, and writes are stored in delta logs in a write-optimized format, which is then compacted into the primary dataset in a columnar format.

Three types of queries are supported on these tables: a snapshot query that returns the version of the dataset as of a particular time, a delta query that returns only the data that changed, and a read-optimized query that returns the dataset from the columnar format, which offers faster performance but does not include the data that has not yet been compacted.





Query type	T0	T1	T2	T3
Snapshot	A, B, C, D	A1, B, C, D1	A2, B, C, D1, E, F	A2, B, C, D1, E, F
Incremental	A, B, C, D	A1, D1	A2, E, F	
Read optimized	A, B, C, D	A, B, C, D	A, B, C, D	A2, B, C, D1, E, F

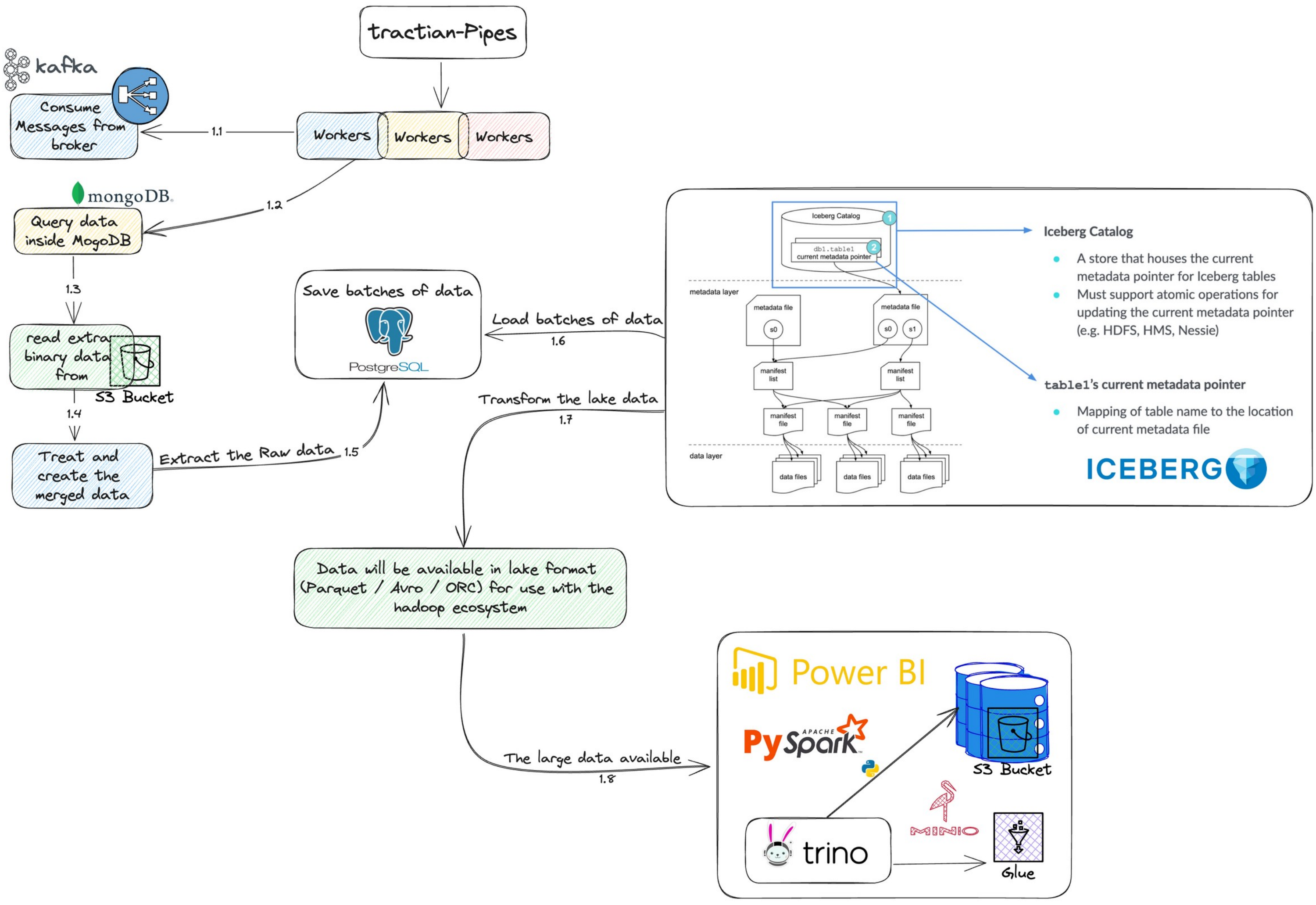
When Do You Use Apache Hudi?

Apache Hudi offers more flexibility with its different types of tables that support different types of queries while providing strong data guarantees with atomic writes and versioning of data. Apache Hudi was incubated by Uber but since then has been gaining strong adoption across other companies, such as Amazon, Walmart, Disney+ Hotstar, GE Aviation, Robinhood, and TikTok. A recently founded company, Onehouse, offers a managed platform built on Apache Hudi.

Apache Hudi is designed to support tables that need to handle a high frequency of writes in both real-time and batch fashion. It also offers the flexibility of using different types of queries based on the requirements you have for performance and data freshness.

We decided to move forward with the Iceberg strategy and below is our first solution architecture diagram.

(version 1.0.0)



tractian-Pipes
Workers
Workers
Workers
kafka
Consume Messages from broker
1.1
mongoDB
Query data inside MogoDB
1.2
1.3
read extra binary data from
S3 Bucket
1.4
Treat and create the merged data
Extract the Raw data
1.5
Save batches of data
PostgreSQL
Load batches of data
1.6
Transform the lake data
1.7
Iceberg Catalog
db1.table1
current metadata pointer
metadata layer
metadata file
s0
s1
manifest list
manifest file
data layer
data files
Iceberg Catalog
A store that houses the current metadata pointer for Iceberg tables
Must support atomic operations for updating the current metadata pointer (e.g. HDFS, HMS, Nessie)
table1's current metadata pointer
Mapping of table name to the location of current metadata file
ICEBERG
Data will be available in lake format (Parquet / Avro / ORC) for use with the hadoop ecosystem
The large data available
1.8
Power BI
APACHE PySpark
S3 Bucket
trino
MINIO
Glue