Gonna Go Back in Time with Apache Iceberg

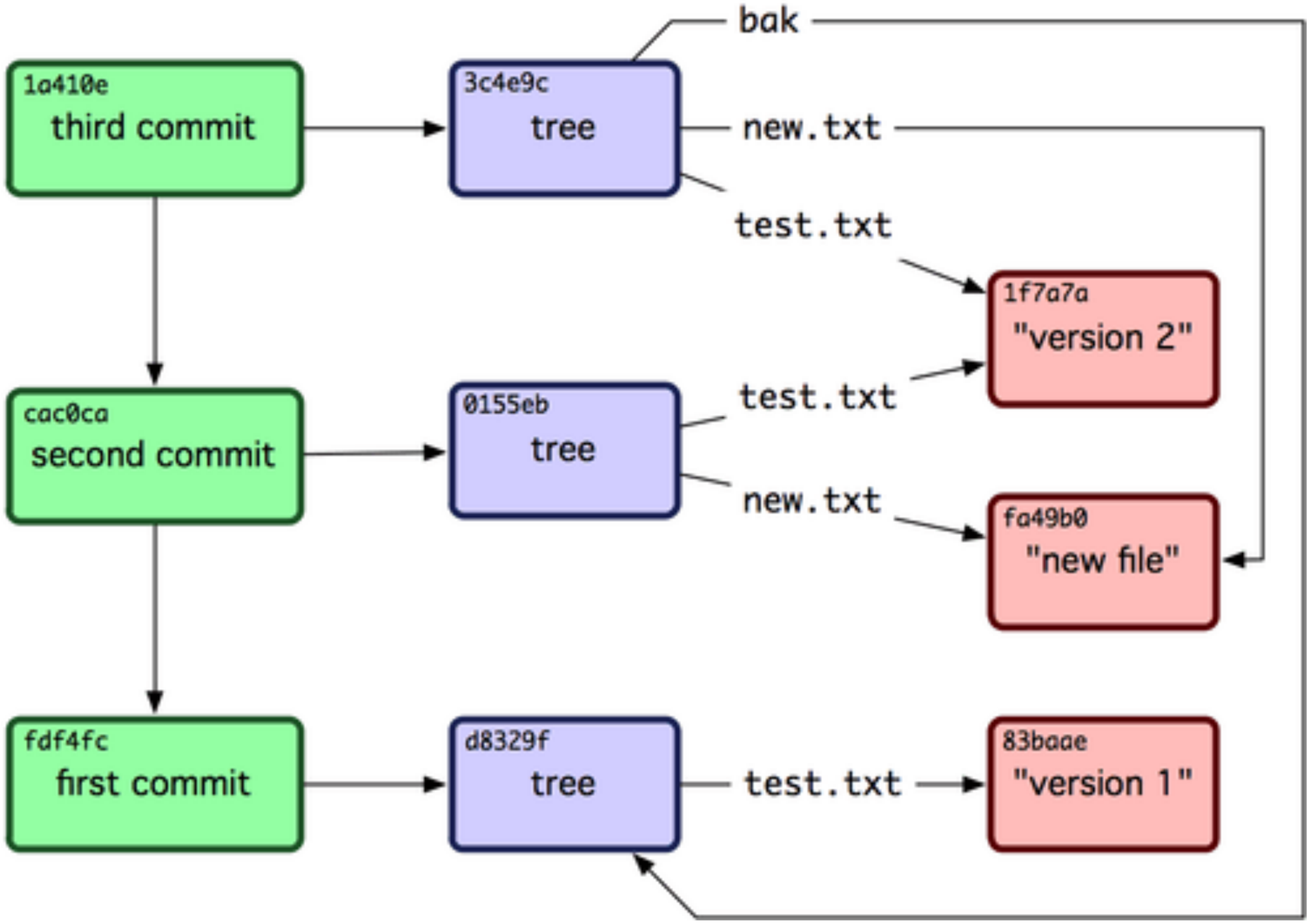
Daniel Hinojosa

In this Workshop

- History of Data Storage
- Object Storage
- Introduction to Iceberg
- Big Data Infrastructure
- Iceberg Internals
- Exercises
- Other Querying Engines
- Data Mesh and Open Metadata

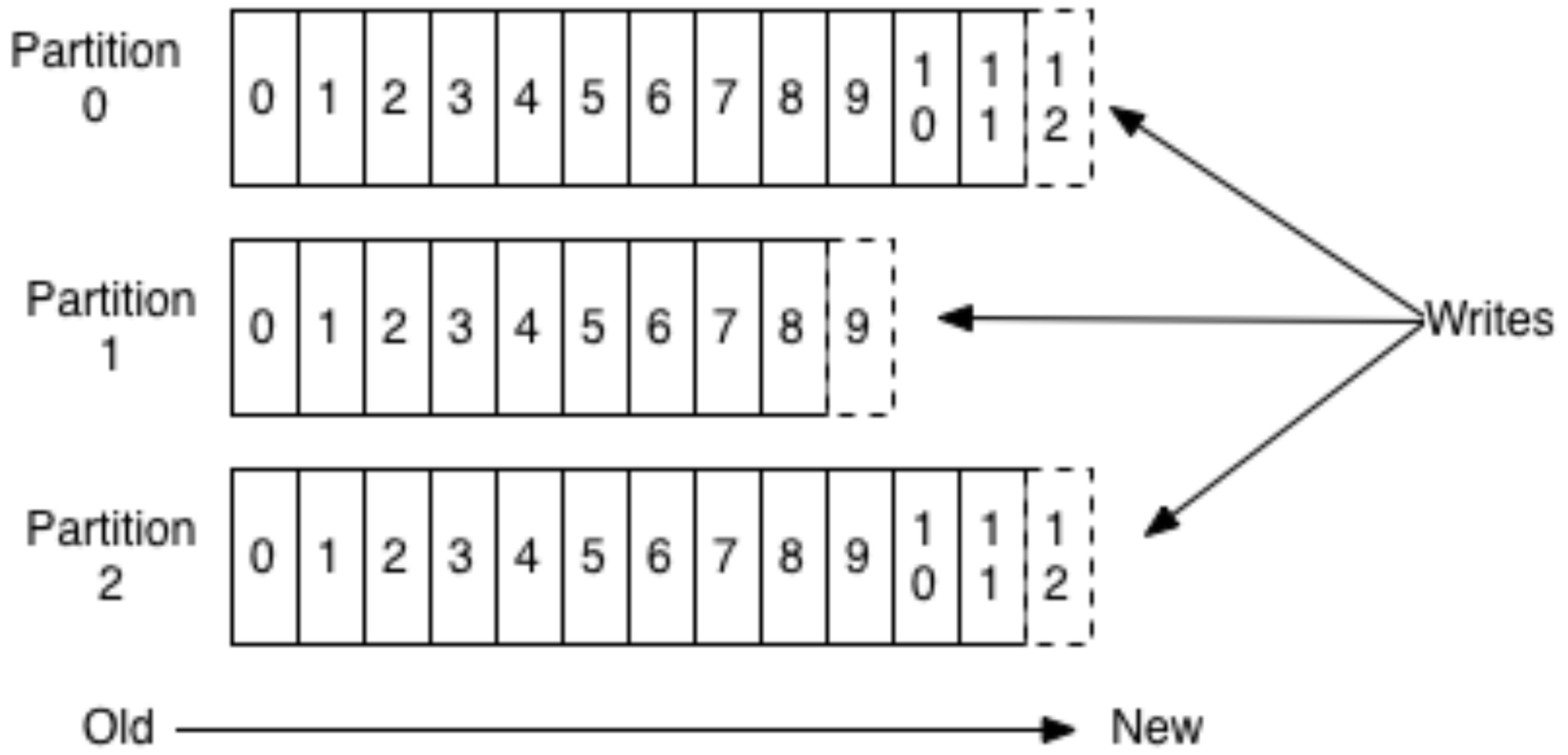
Slides and Material:
https://github.com/dhinojosa/iceberg-workshop

A Familiar Pattern

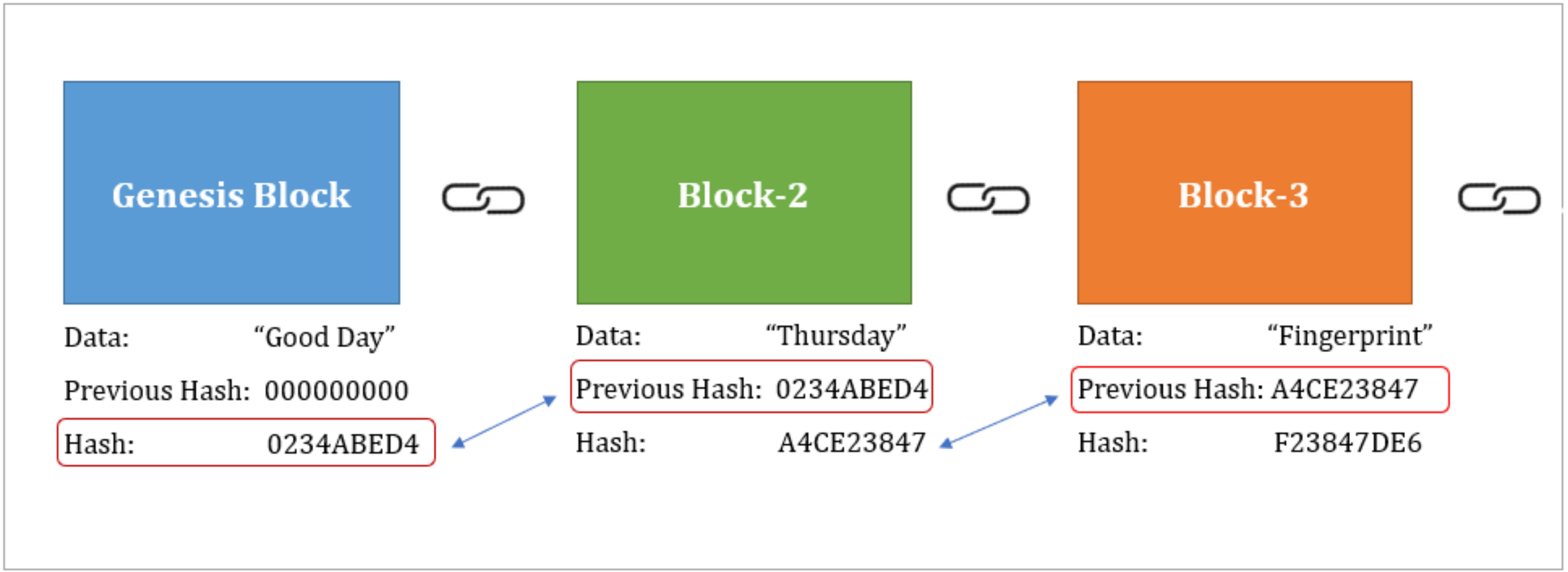

1a410e
third commit
3c4e9c
tree
bak
new.txt
test.txt
1f7a7a
"version 2"
cac0ca
second commit
0155eb
tree
test.txt
new.txt
fa49b0
"new file"
fdf4fc
first commit
d8329f
tree
test.txt
83baae
"version 1"


kafka

Anatomy of a Topic

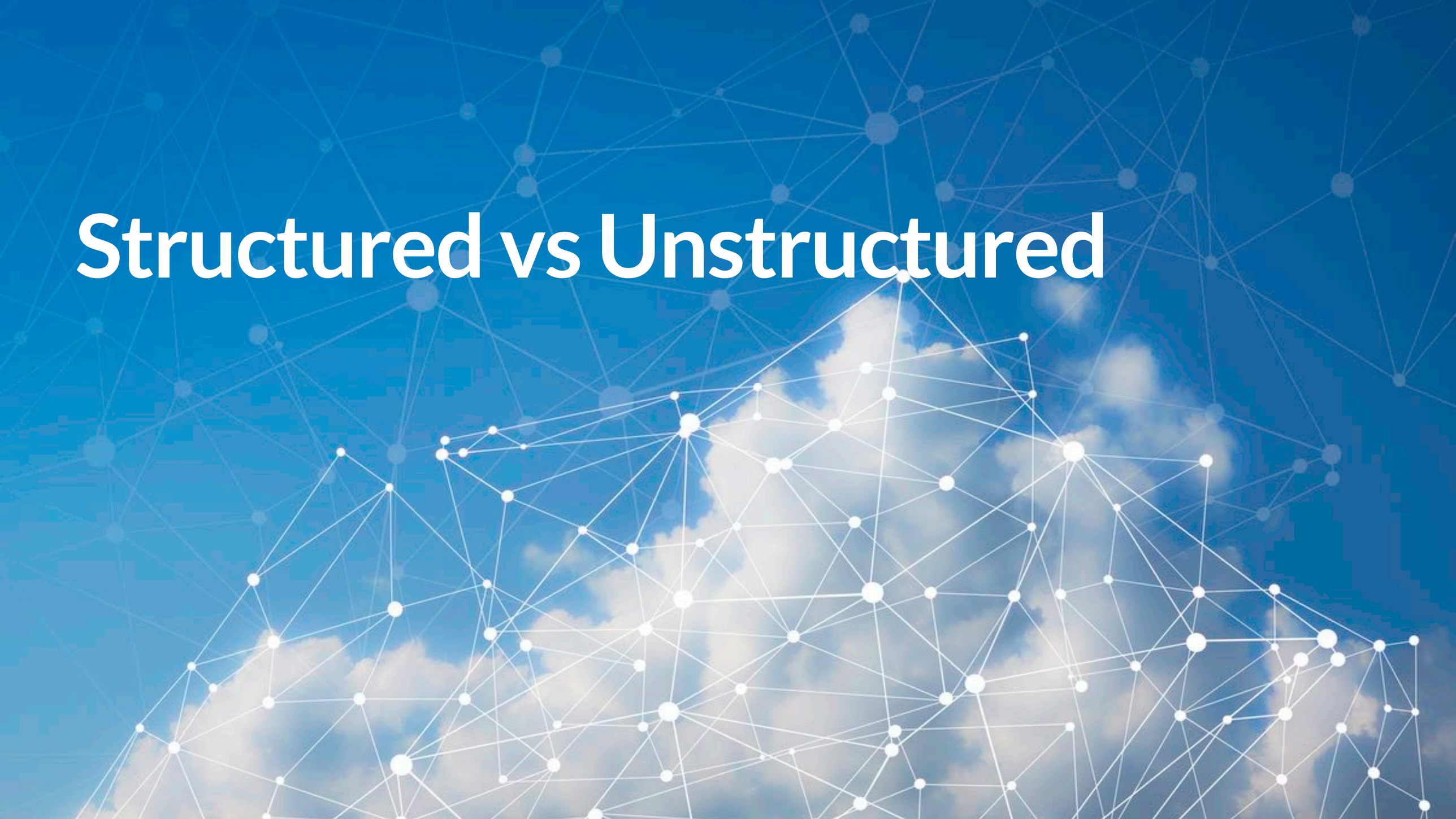


Blockchain



https://metlabs.io/en/what-is-a-hash-in-blockchain/

Structured vs Unstructured



Structured vs Unstructured Data

- **Structured:**
  - Data that conforms to a predefined schema or structure, making it easily searchable and queryable.
  - The schema defines the types, relationships, and constraints of the data.
  - Can exist in non-tabular formats like JSON, XML, or Avro if a schema is defined.
- **Unstructured Data:**
  - Data that lacks a predefined schema or consistent organization.
  - Examples: Text files, images, videos, audio files, and logs without consistent structure.

Structured vs Unstructured Data

Unstructured Logging

```
6:01:00 accepted connection on port 80 from 10.0.0.3:63349
6:01:03 basic authentication accepted for user foo
6:01:15 processing request for /super/slow/server
6:01:18 request succeeded, sent response code 200
6:01:19 closed connection to 10.0.0.3:63349
```

Structured Logging

```
time="6:01:00" msg="accepted connection" port="80" authority="10.0.0.3:63349"
time="6:01:03" msg="basic authentication accepted" user="foo"
time="6:01:15" msg="processing request" path="/super/slow/server"
time="6:01:18" msg="sent response code" status="200"
time="6:01:19" msg="closed connection" authority="10.0.0.3:63349"
```

Structured vs Unstructured Data

Structured vs. Unstructured

Aspect	Structured Data	Unstructured Data
Storage	SQL/NoSQL databases (e.g., MySQL, JSON, Parquet)	Object stores or file systems (e.g., S3, HDFS)
Querying	SQL or tools like Presto/Spark with schema	Advanced NLP, computer vision, or ML
Format	Tabular, JSON, XML with schema	Free-form (e.g., raw text, videos, images)
Schema Dependency	Schema-defined before or during use	No schema or schema inferred dynamically
Cost to Store	Higher per unit due to indexing, schema enforcement, and redundancy requirements	Lower per unit because raw data requires minimal processing or indexing

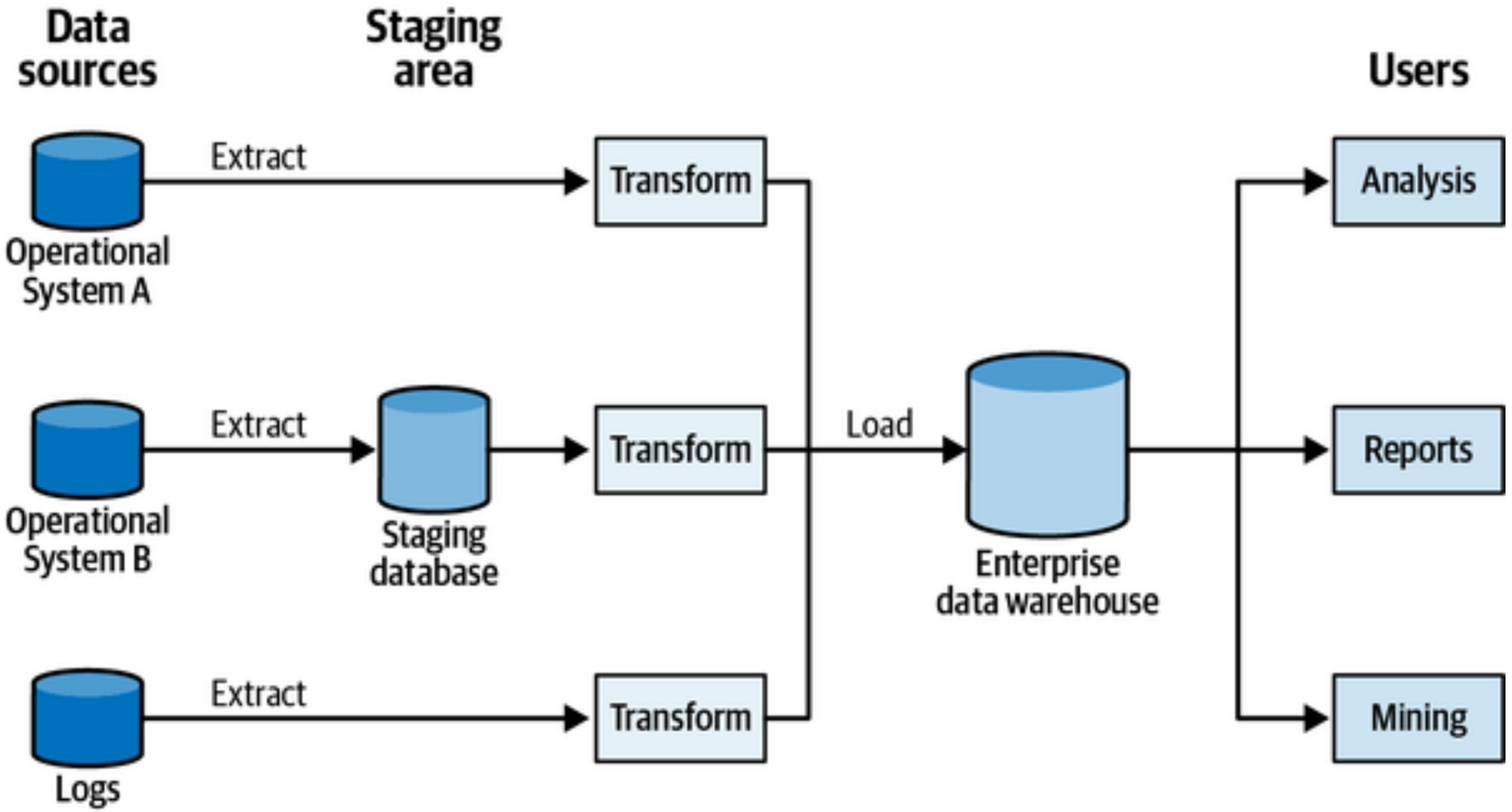
History of Data Storage

Data Warehousing

Data Warehousing

- A Data Warehouse, a centralized repository for *structured data*, designed for querying and analysis
- Consolidates data from multiple sources.
- **Schema-on-Write: Data schema is applied when stored.**
- Optimized for read-heavy analytical workloads.
- Supports historical analysis through snapshot storage.
- Single source of truth for analytics.
- Enables complex reporting and BI tools.
- Provides pre-aggregated, summarized data for fast querying.

The Diagram



Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy - Vlad Khononov

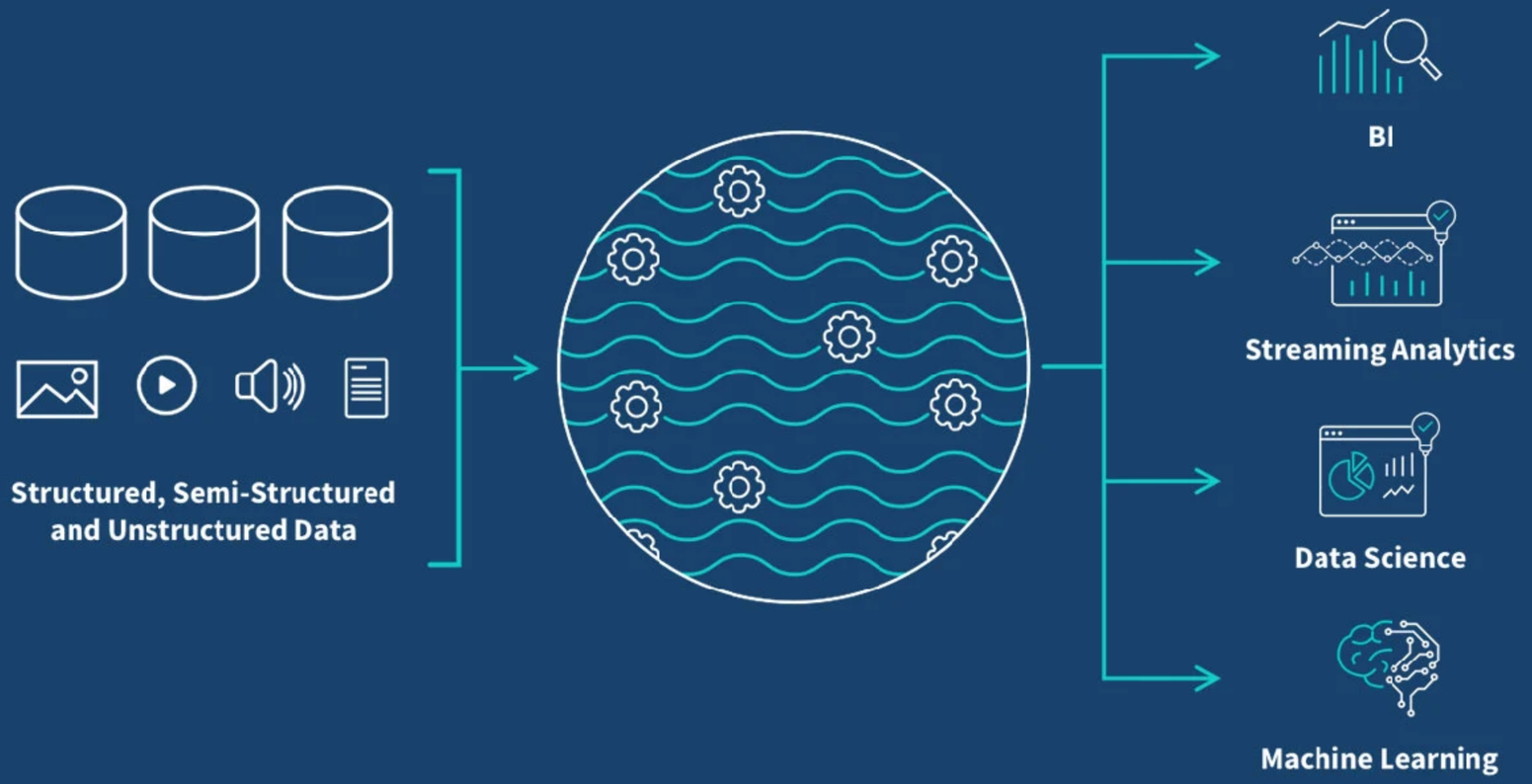
What was the problem?

- Large Storage, expensive operational costs
- Difficult Schema Evolution
- Batch Oriented Pipelines made real-time analysis difficult
- High Cost & Vendor Lock In
- Scaling Issues
- **You used your vendor’s compute engine**

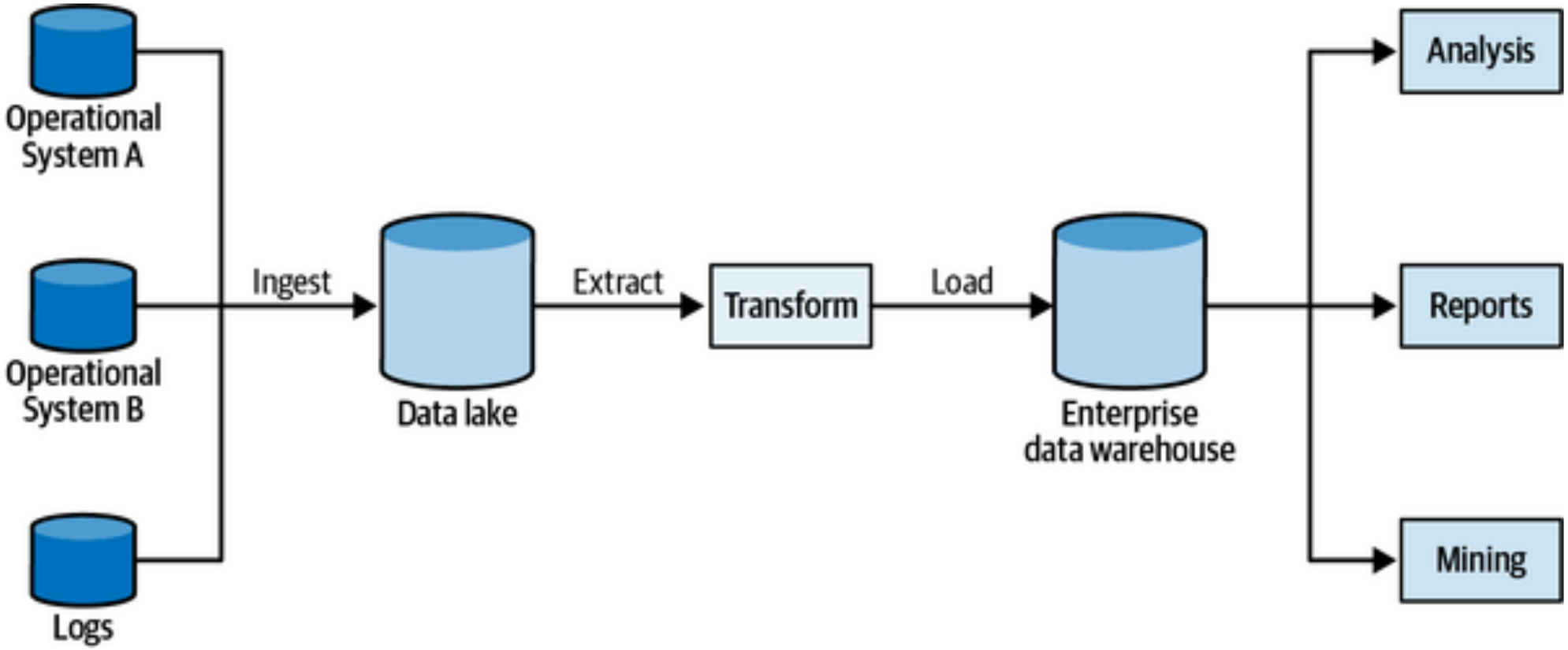
Data Lakes

Data Lakes

- **A Data lake is a unified repository for all data**
- A data lake is a centralized repository that **stores raw, unprocessed data in its native format.**
- **Store everything:** Handles structured, semi-structured, and unstructured data.
- **Schema-on-Read:** Data schema is applied when queried, not when stored.
- **Scalability:** Designed to scale with increasing data volumes.
- **Cost-Effective:** Often built on cheap, scalable storage like object stores (e.g., Amazon S3).
- You use whatever engine you would like: Spark, Flink, Trino, Presto, etc.


Structured, Semi-Structured
and Unstructured Data
BI
Streaming Analytics
Data Science
Machine Learning

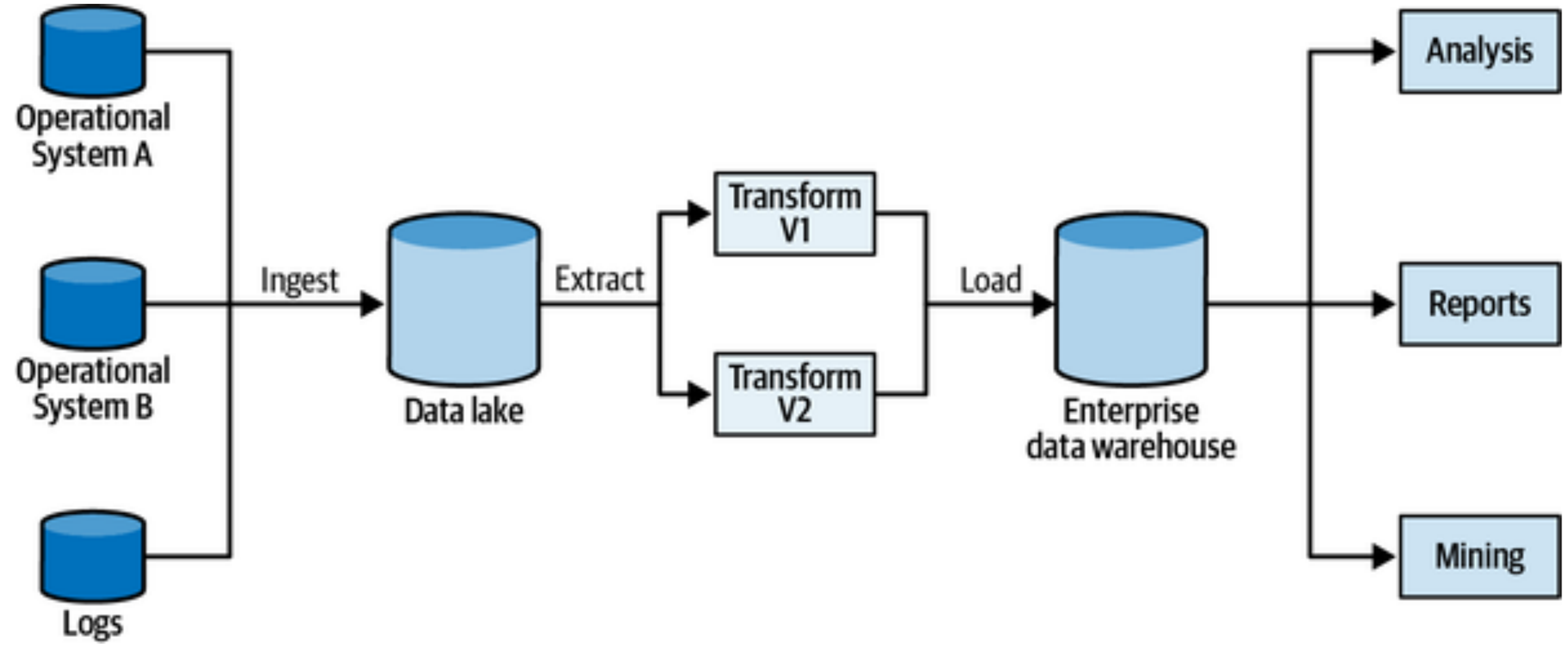
Data Lake



Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy - Vlad Khononov

Data Lake

It’s not uncommon for data engineers to implement and support multiple versions of the same ETL script to accommodate different versions of the operational model



Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy - Vlad Khononov

Small Details

- Supports large volumes of data in diverse formats: CSV, JSON, Parquet, Avro, video, images, logs, etc.
- Typically implemented using object storage (e.g., Amazon S3, Azure Data Lake Storage).
- Integrates with tools like Kafka, Flume, or AWS Glue for ingesting data.
- Supports AI/ML workflows, ad-hoc analysis, and business intelligence tools.
- Implements data governance and security policies (e.g., AWS Lake Formation).

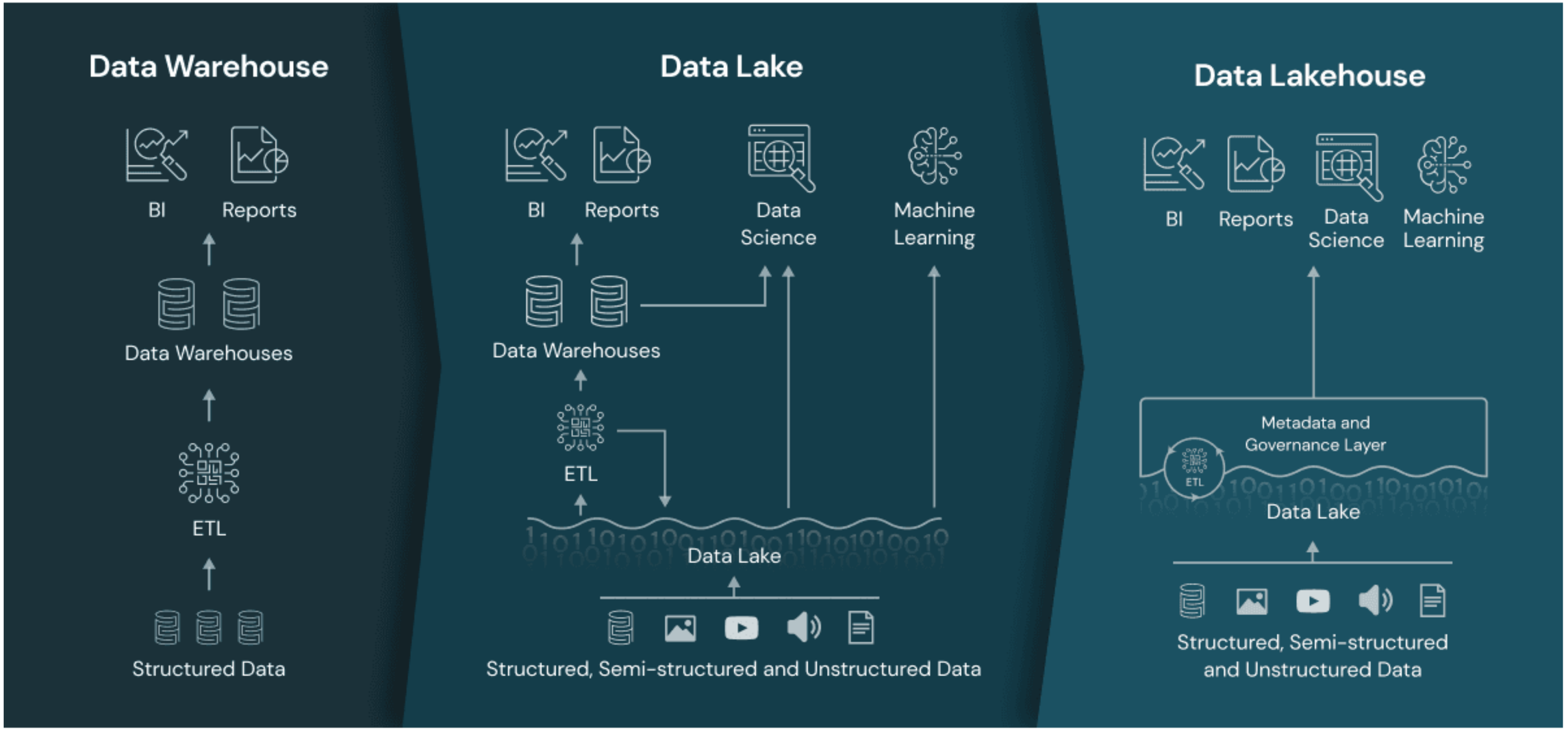
Tradeoffs

- **Advantages:**
  - Cost-effective for storing raw, unstructured data.
  - Flexible schema-on-read approach.
  - Scalable for petabyte-scale storage and analytics.
  - Facilitates AI/ML and real-time processing.
- **Disadvantages:**
  - Risk of becoming a “Data Swamp”: Poorly managed data lakes can lead to unorganized, low-quality data.
  - Higher complexity for users to derive value (requires robust querying and data governance).
  - Latency in querying raw data compared to structured warehouses.
  - Security challenges with managing diverse data formats and access policies.

Data Lakehouse

Data Lakehouse

- **A Data Lakehouse bridges this gap**
- A data lakehouse combines the best of data lakes and data warehouses into a single system.
  - Handles structured, semi-structured, and unstructured data.
  - Provides high-performance analytics and supports AI/ML workloads.
  - Enforces data governance while retaining the scalability and flexibility of data lakes.



https://www.databricks.com/glossary/data-lakehouse

Small Details

- **Unified Architecture: Single system for storage and analytics.**
  - **Open Storage Format:** Supports Parquet, ORC, or Delta for easy access and compatibility.
  - **Real-Time Data Processing:** Enables faster insights with streaming capabilities.
  - **Integrated Governance:** Ensures quality, lineage, and compliance across all data.

Open Table Format

Open Table Format

- Category of technologies for how big data tables are stored and managed
- Decouples compute from storage
- Enables ACID guarantees on Data Lakes
- Supports Schema Evolution and Partition Evolution
- Provides Time Travel and Rollback with metadata snapshots

https://www.teradata.com/insights/data-platform/what-are-open-table-formats

Competitors

- **Apache Iceberg:** Excels in schema and partition evolution, efficient metadata management, and broad compatibility with various data processing engines.
- **Apache Hudi:** Best suited for real-time data ingestion and upserts, with strong change data capture capabilities and data versioning.
- **Apache Paimon:** A lake format that enables building a real-time lakehouse architecture with Flink and Spark for both streaming and batch operations.
- **Delta Lake:** Provides robust ACID transactions, schema enforcement, and time travel features, making it ideal for maintaining data quality and integrity.

Who runs the Open Table Format

- Since it is technology category, Apache Iceberg, Delta Lake, and Apache Hudi, are all maintained by their respective open-source communities.
  - Iceberg is managed by the Apache Software Foundation.
  - Delta Lake is developed by Databricks
  - Hudi is maintained by the Apache Software Foundation

https://aws.amazon.com/blogs/big-data/choosing-an-open-table-format-for-your-transactional-data-lake-on-aws/

https://delta.io/blog/open-table-formats/

Schema Formats

Avro

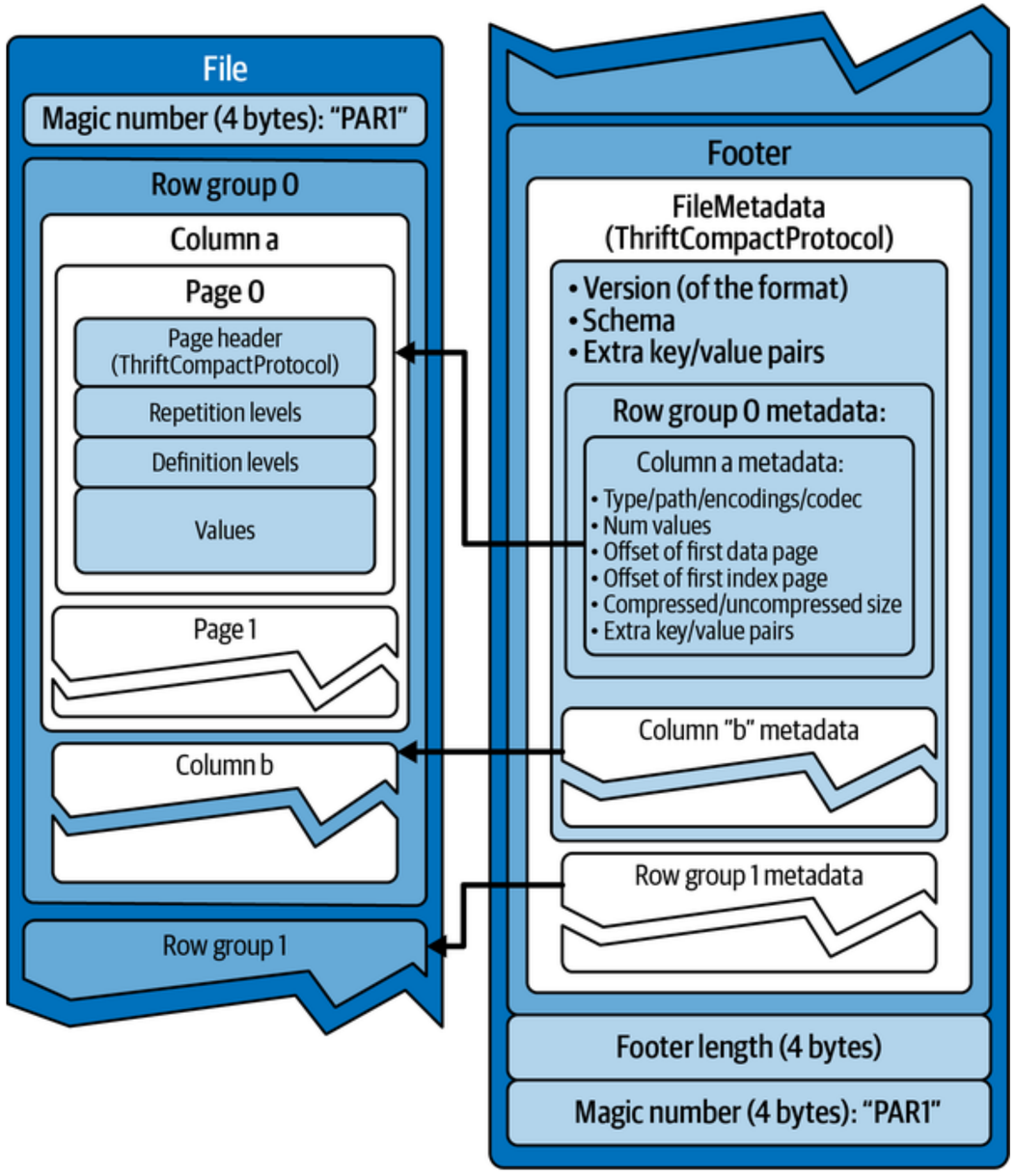
Avro

- Created by Doug Cutting; Creator of Hadoop
- Serialization is defined by schema
- Schemas are JSON Based
- Codegen available at command line
- Codegen available by Maven, Gradle, SBT Plugin
- Supports the following languages:
- C, C++, Java, Perl, Python, Ruby, PHP
- Named after WWI, WWII Aircraft Company, A.V. Roe

Parquet

Parquet

- Introduced by Twitter and Cloudera in 2013
- Designed for efficient columnar storage on Hadoop ecosystem
- Row-based formats (like CSV, JSON) had poor performance for analytics
- Column-based formats enables easy selective read of columns during queries
- Built for analytics and OLAP workloads



Apache Iceberg: The Definitive Guide Tomer Shiran, Jason Hughes, Alex Merced

Internal Structure

- **File → Row Groups → Column Chunks → Pages**
- **Row Groups:** Horizontal partitioning of data (like batches of rows)
- **Column Chunks:** Data for one column within a row group
- **Pages:**
  - Data Page – actual values
  - Dictionary Page – optional
  - Index Page – optional (used in advanced filters)

OLTP vs OLAP

OLTP vs OLAP

- Online analytical processing (OLAP) and online transaction processing (OLTP) are data processing systems that help you store and analyze business data.
- OLAP combines and groups the data so you can analyze it from different points of view
- OLTP stores and updates transactional data reliably and efficiently in high volumes
- OLTP databases can be one among several data sources for an OLAP system.

What is the difference between OLTP and OLAP

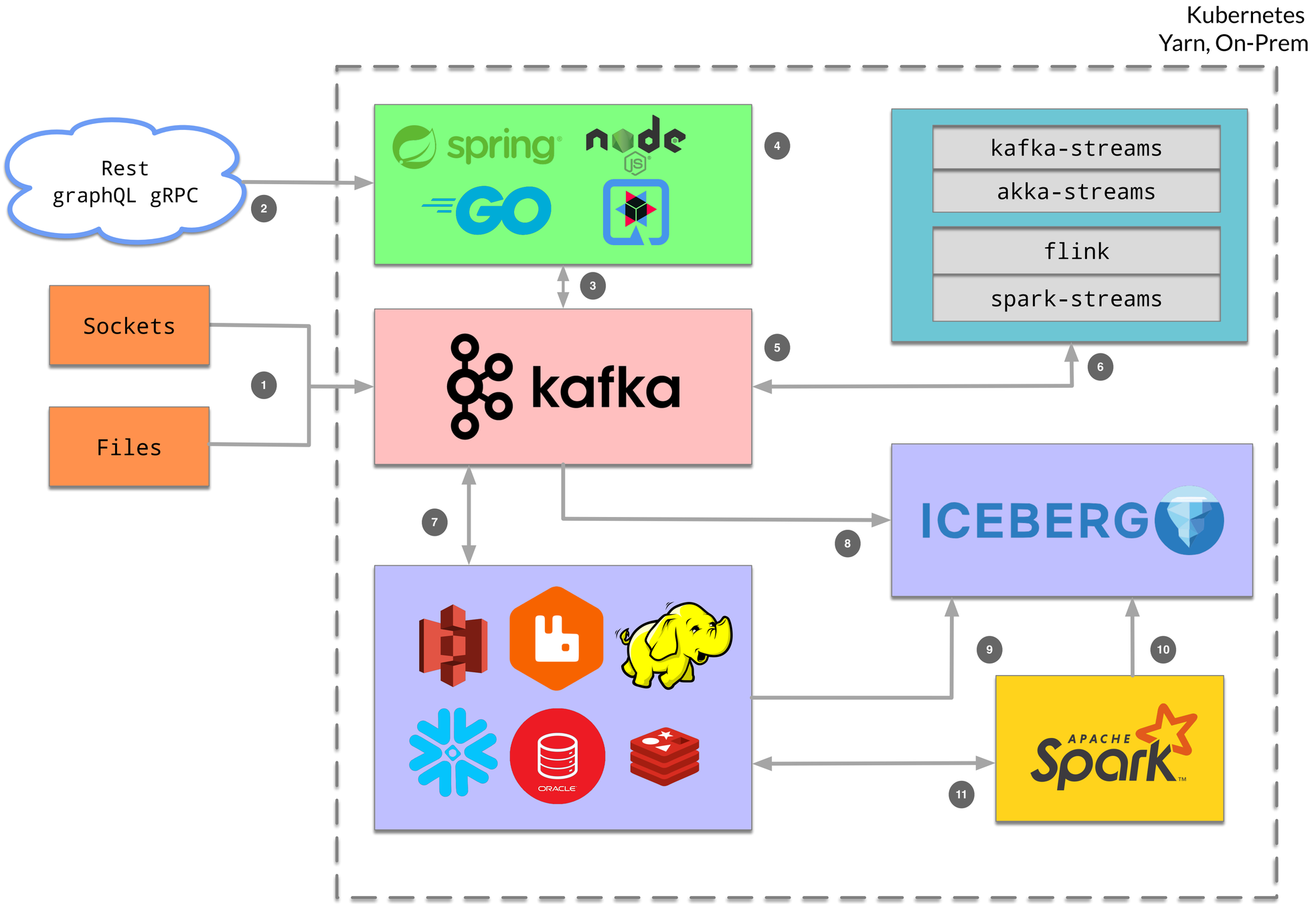
OLTP vs OLAP

OLAP vs OLTP

Aspect	OLAP (Analytical)	OLTP (Transactional)
Purpose	Analytics, reporting, decision-making	Real-time transaction processing
Query Complexity	Complex, multi-dimensional queries	Simple, fast queries
Data Type	Historical, aggregated, and derived data	Current, detailed, and transactional data
Operations	Read-intensive	Read and write-heavy
Schema Design	Star or snowflake schema	Normalized schema (3NF or higher)
Performance Goal	Optimized for query speed	Optimized for transaction speed
Concurrency	Few concurrent users	Many concurrent users
Example Use Cases	BI dashboards, trend analysis, forecasting	Order processing, inventory tracking
Examples	Snowflake, Google BigQuery, Tableau	MySQL, PostgreSQL, Oracle DB

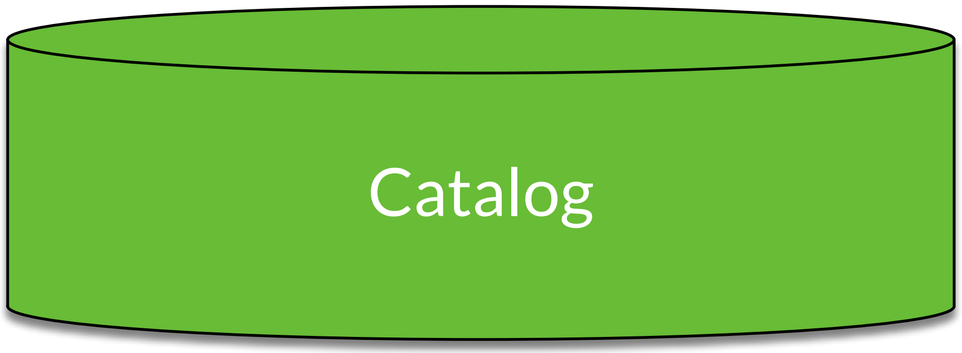
Apache Iceberg

- Iceberg is an open table format.
- Defines how data is stored
  - **How partititions are managed**
  - **How a schema evolves**
  - **How time travel works**
- Iceberg organizes files like Parquet, Avro. Typically in Parquet due to it’s columnar structure
- It is engine agnostic - Iceberg lets multiple engines read from and write from the *same table* safely
  - Spark, Flink, Trino, Dremio, etc Data is stored in object storage (S3, GCS, etc)
- ACID Compliant
- OLAP - Meant for Large Scale Analytical Queries with Fast Scans and evolving Data Sets


Kubernetes
Yarn, On-Prem
Rest
graphQL gRPC
Sockets
Files
spring
node
GO
kafka
kafka-streams
akka-streams
flink
spark-streams
ICEBERG
ORACLE
APACHE
Spark
1
2
3
4
5
6
7
8
9
10
11

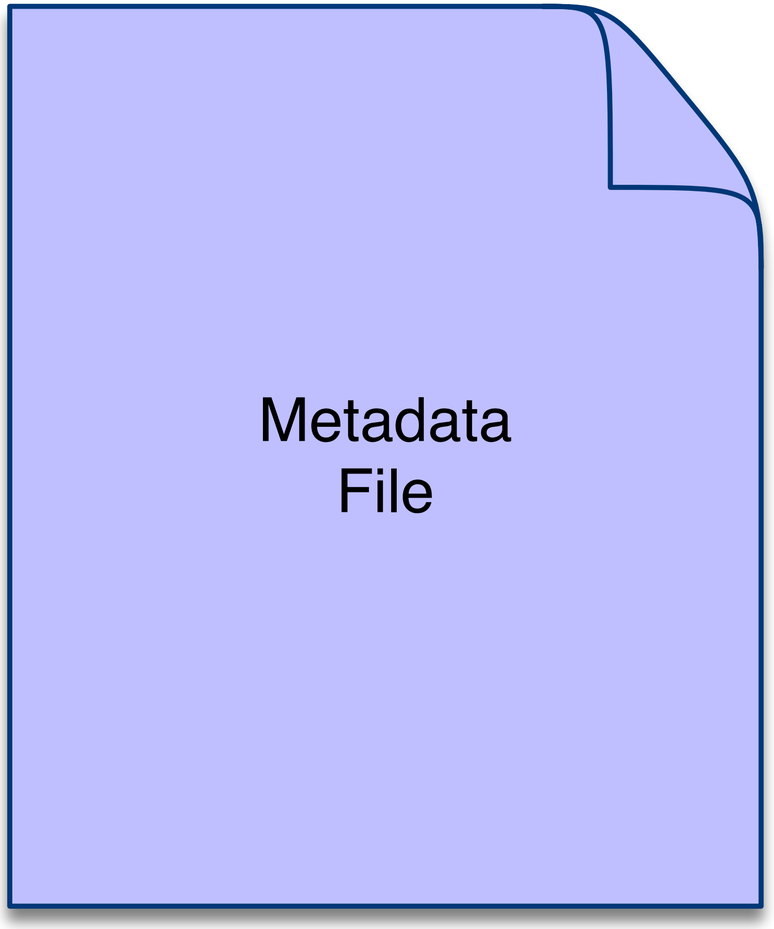
Apache Iceberg Visualized

Catalog



- Manages the metadata for Iceberg tables and namespaces
- It tracks all iceberg tables and their metadata locations like `analytics.orders`
- **Multiple Implementations:** Hive, Hadoop, Rest, Nessie
- Coordinates atomic table operations using snapshot
- Entry point for data engines like Flink, Spark, Trino, etc.

Metadata File

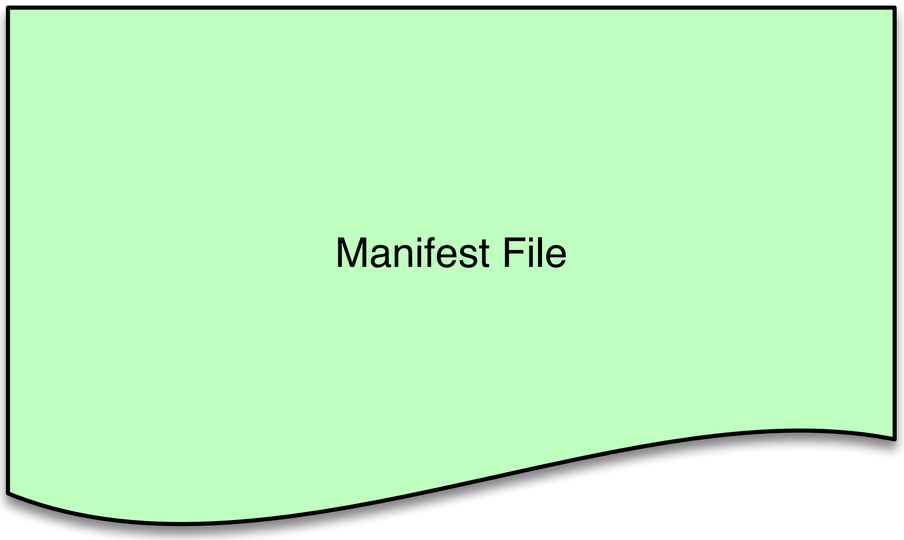


- Current State and Structure of the Iceberg Table
- Stored as JSON
- Contains References to All Snapshot which of consistent views of the table
- Store current schemas and table schemas
- **Points to the Manifest Lists**

Manifest List

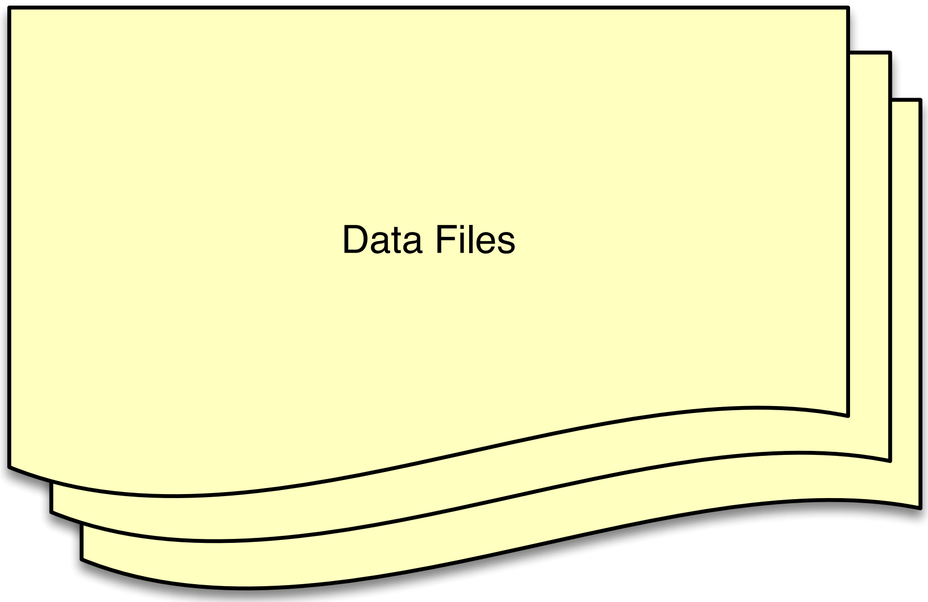
- A manifest list is a metadata file that **lists all the manifest files for a single snapshot.**
- Stored in Avro
- Contains metadata like file counts, partition bounds, and stats used for query planning and pruning.
- Immutable

Manifest File

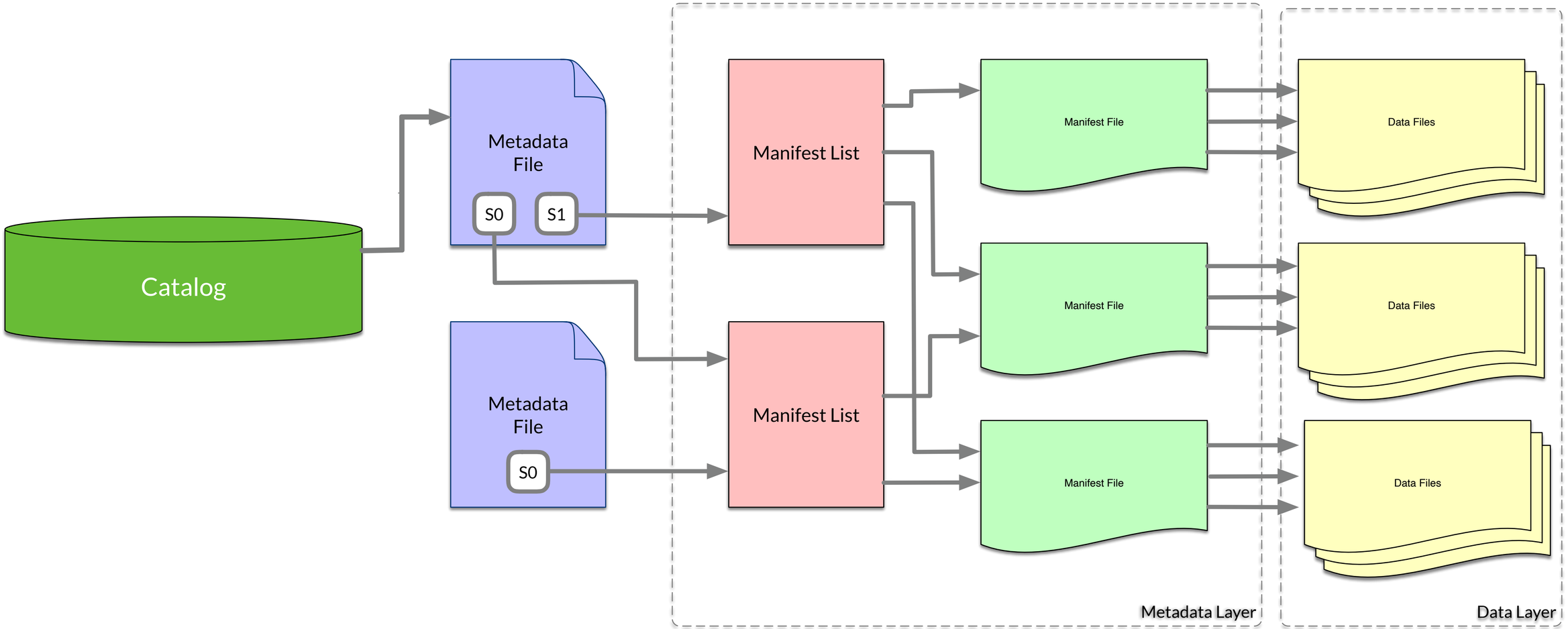


- Each manifest file contains a list of data files (and delete files) that belong to a partition or snapshot
- Grouped by Partition: Manifest files are typically organized by partition
- Stores stats like row counts, column bounds (min/max), null counts, and file size
- Stored as Avro
- Immutable and Append Only

Data Files



- Storage of actual table data
- Columnar File Format like Parquet
- Immutable


Catalog
Metadata File
S0
S1
Metadata File
S0
Manifest List
Manifest List
Manifest File
Manifest File
Manifest File
Data Files
Data Files
Data Files
Metadata Layer
Data Layer

Compute Engines

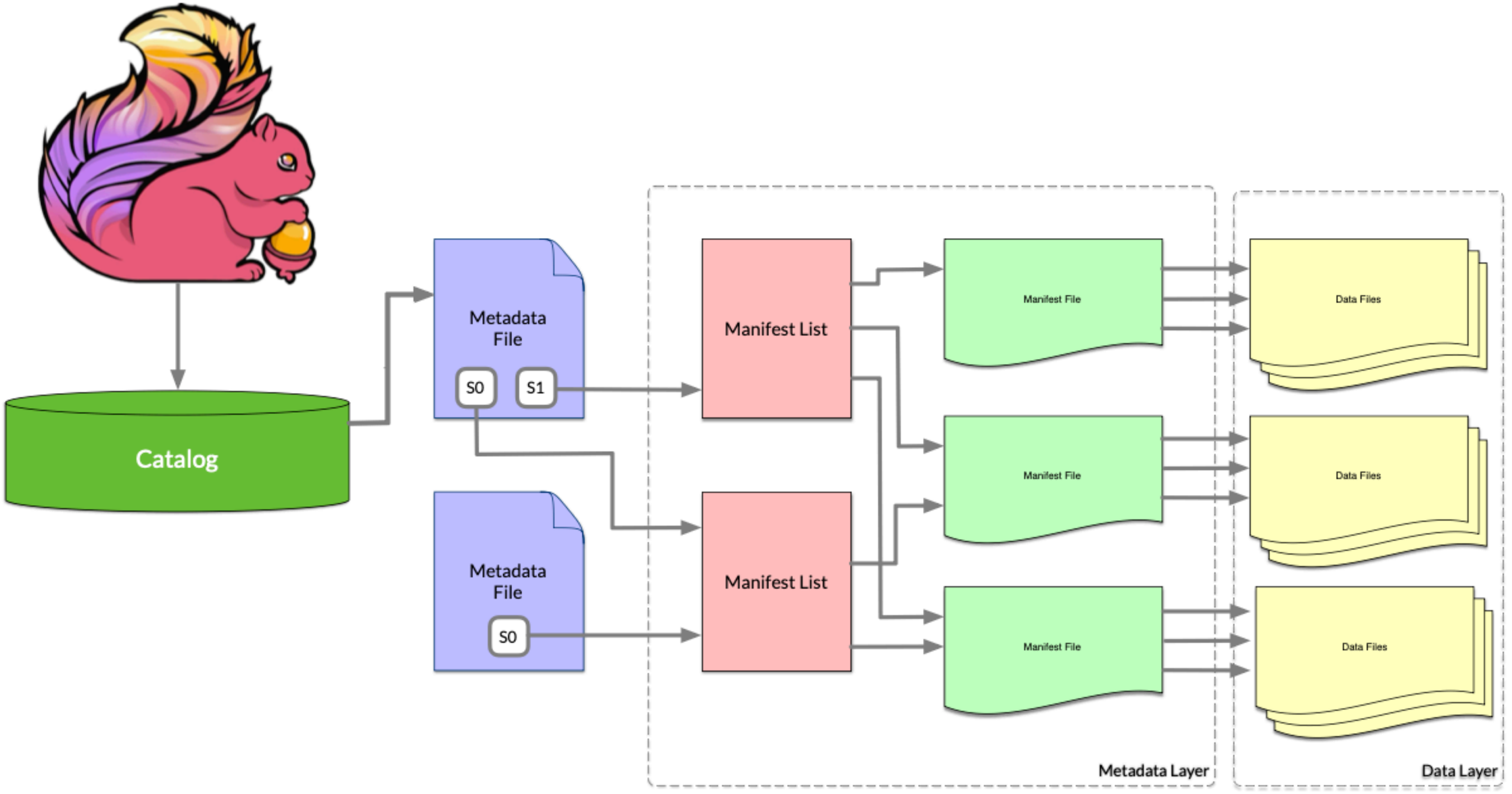
DuckDB

presto

APACHE Spark™

dremio

trino


Catalog
Metadata File
S0
S1
Metadata File
S0
Manifest List
Manifest List
Manifest File
Manifest File
Manifest File
Data Files
Data Files
Data Files
Metadata Layer
Data Layer

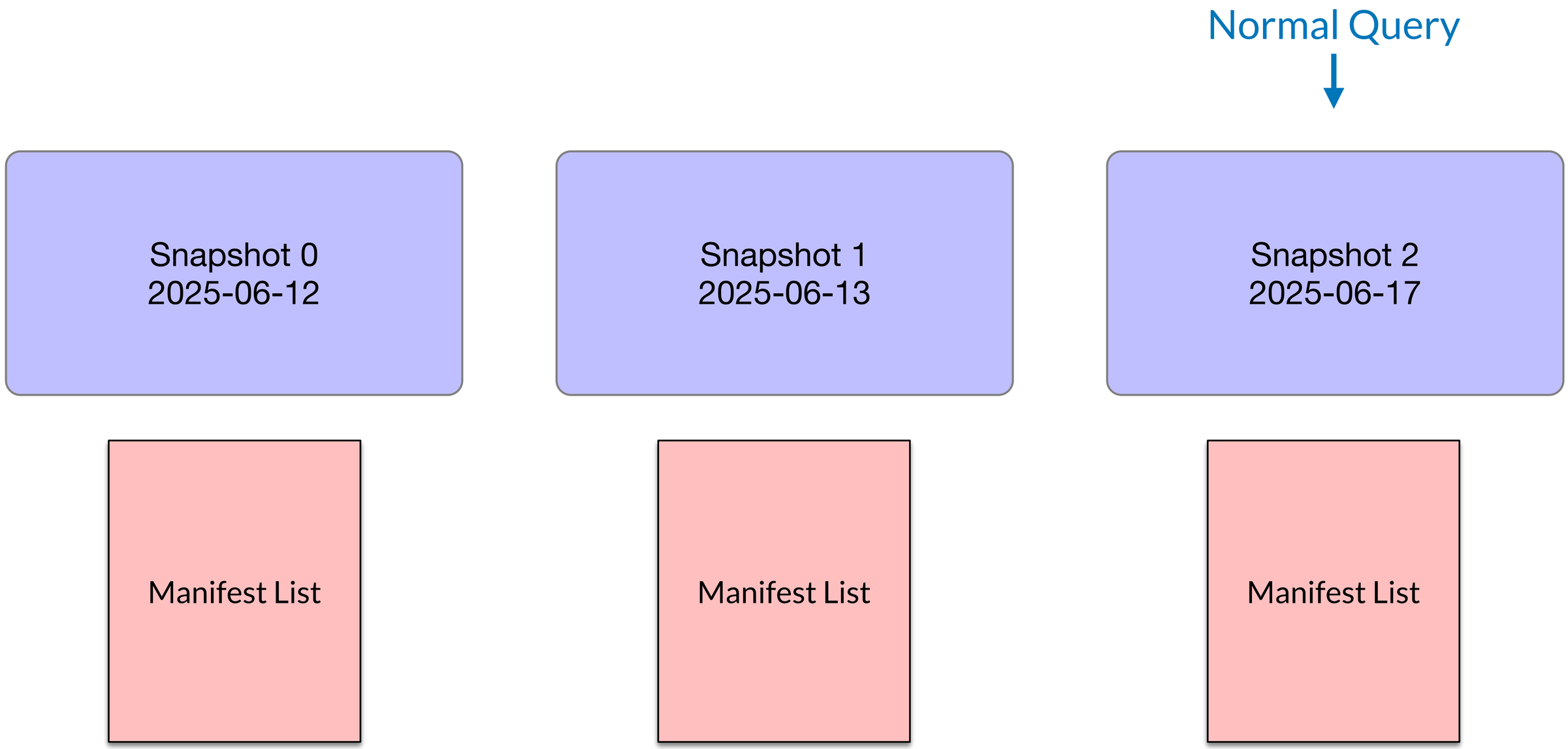
Lab: Apache Iceberg Intro

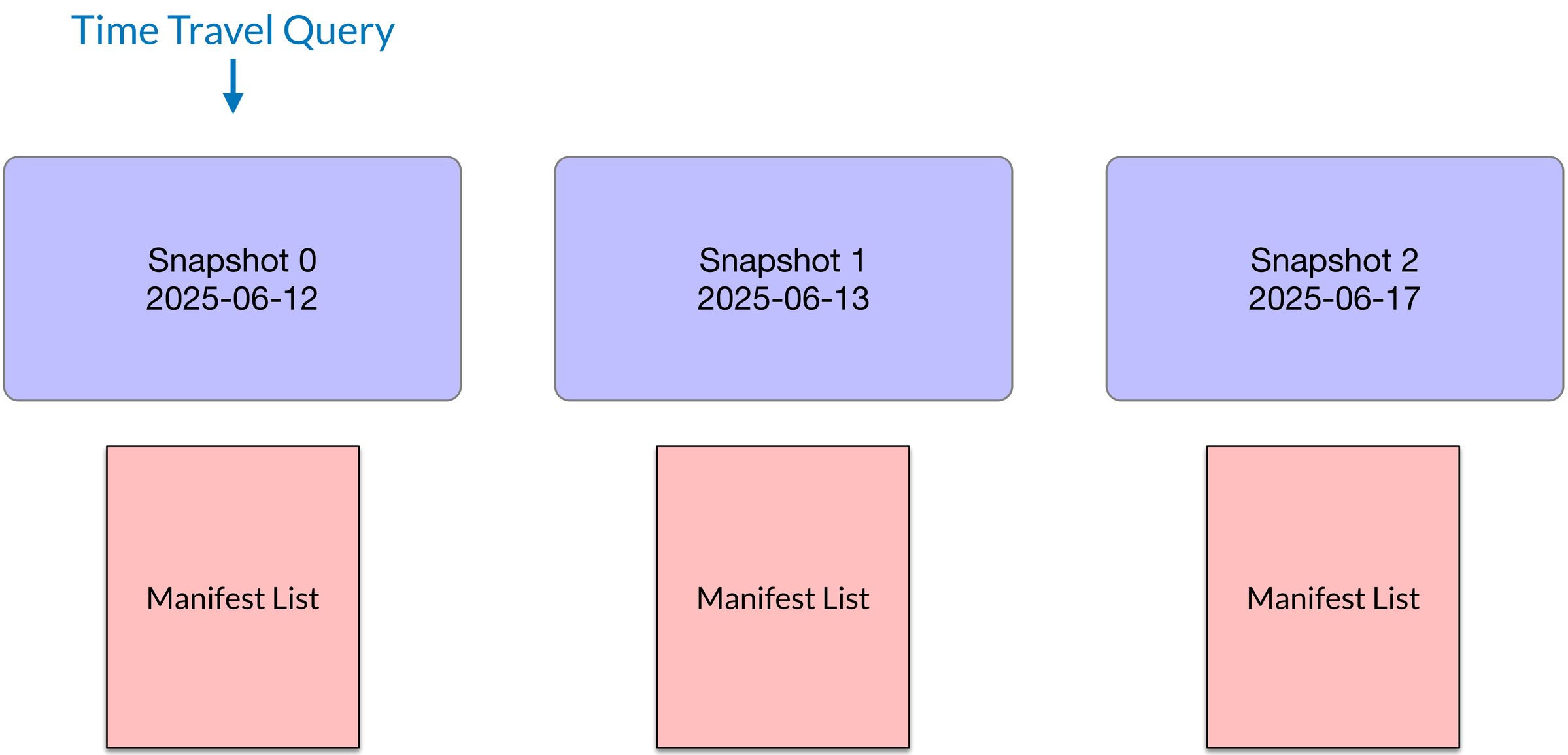
- Let’s see how we can run Apache Iceberg and all the components involved
- We will then insert some data, and view the snapshot
- We can then view the snapshots

Time Travel

Time Travel in Iceberg

- We can query data as it existed in a previous point in time
- Because we have *immutable* snapshots we can roll back in time
- Use `AS OF VERSION (snapshot ID)` or `AS OF TIMESTAMP` in SQL


Normal Query
Snapshot 0
2025-06-12
Snapshot 1
2025-06-13
Snapshot 2
2025-06-17
Manifest List
Manifest List
Manifest List


Time Travel Query
Snapshot 0
2025-06-12
Snapshot 1
2025-06-13
Snapshot 2
2025-06-17
Manifest List
Manifest List
Manifest List

Can it be made permanent?

- You can make the call: `CALL system.rollback_to_snapshot('my_table', 1234567890);` to move to the latest snapshot (for Spark only)
- This will make the snapshot the new current version
- It does not delete newer data or snapshots

What if I really want to “Hard Reset”?

- Rollbacks are metadata-level operations.
- Previous and newer snapshots still exist—unless explicitly expired.
- You can still time travel to newer (post-rollback) snapshots unless:
- You call `expire_snapshots` to remove them permanently.
- Retention policies prune them automatically
- For Permanent Rollback use:

```
CALL system.expire_snapshots('my_table', retain_last = 1);
```

Lab: Time Travel

- Let’s rewind to either the snapshot or the date and see if we see a different perspective

Hidden Partitioning

Hidden Partitioning

- Traditional table formats require explicit partitioning columns
- Iceberg introduces hidden partitioning to simplify queries and avoid user errors
- No need to include partition columns in queries (e.g., no `WHERE year = 2024`)
- Partition values are derived using transforms on columns

Hidden Partitioning Reasoning

- Prevents data skipping mistakes when users forget to filter on partition columns
- Enables automatic filtering during query planning
- Makes schema evolution and partition evolution safer and easier
- Encourages query simplicity: filters are applied without knowing internal partitioning

Hidden Partitioning Examples

- `identity(col)` – default (no transformation)
- `bucket(col, 16)` – evenly distributes data into 16 buckets
- `truncate(col, 4)` – truncates string/numeric value
- `year(ts)` / `month(ts)` / `day(ts)` – extract date parts
- Enables fine-grained partitioning without over-exposing schema details

Schema Evolution

Schema Evolution

- Iceberg allows safe, in-place schema evolution
- Supports adds, drops, renames, reorders, and type promotions
- **No need to rewrite existing data files**
- Schema changes are versioned in *metadata snapshots*
- Old queries on historical snapshots will use the schema at the time

What can I do with a schema change?

- Add, Drop, Rename, Reorder Columns
- Promote Types; e.g from `int → long`, `float → double`

What can I not do with a schema change?

- Demote Types; e.g from `long → int`
- Change Column Id

Backward and Forward Compatibility

- Iceberg tracks column IDs, not just names – enables safe renames
- New reads must handle old data (backward compatible)
- Old readers can skip new columns (forward compatible)
- Schema must be evolution-safe: don’t drop critical fields required by old readers

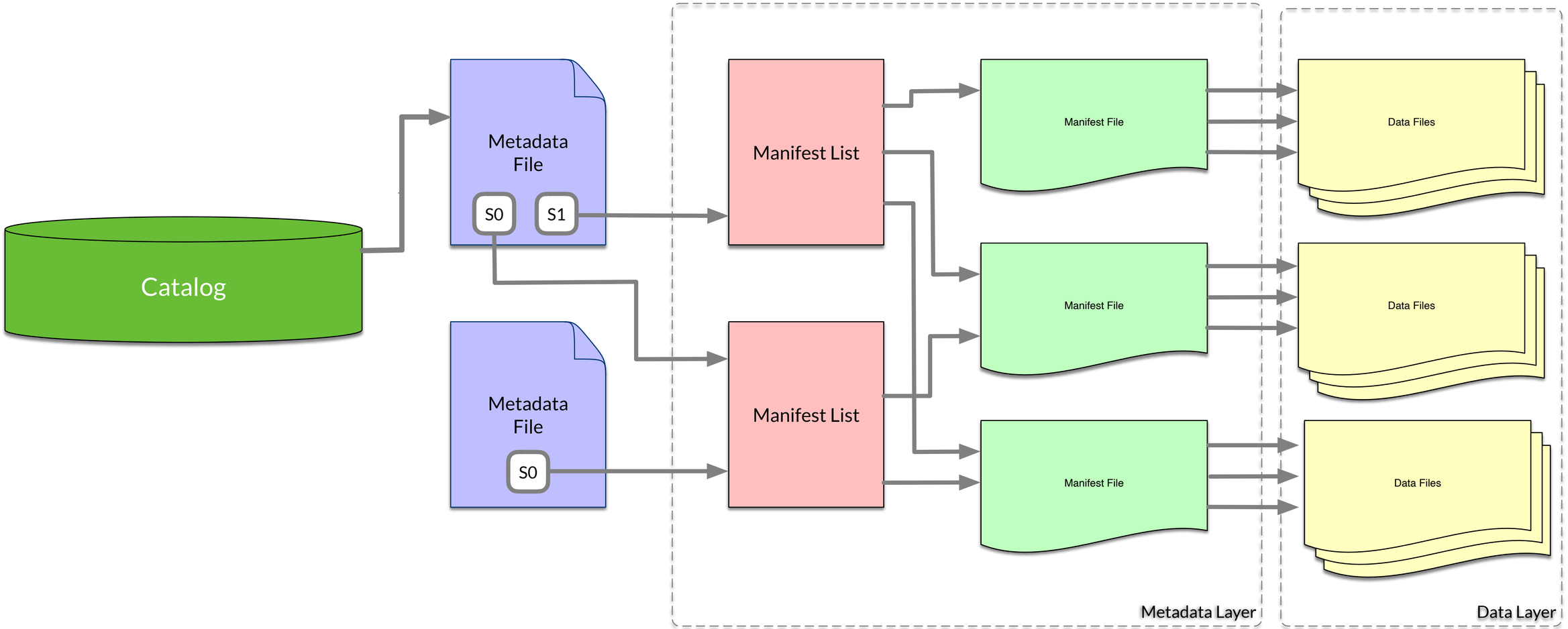
Lab: Changing the Schema

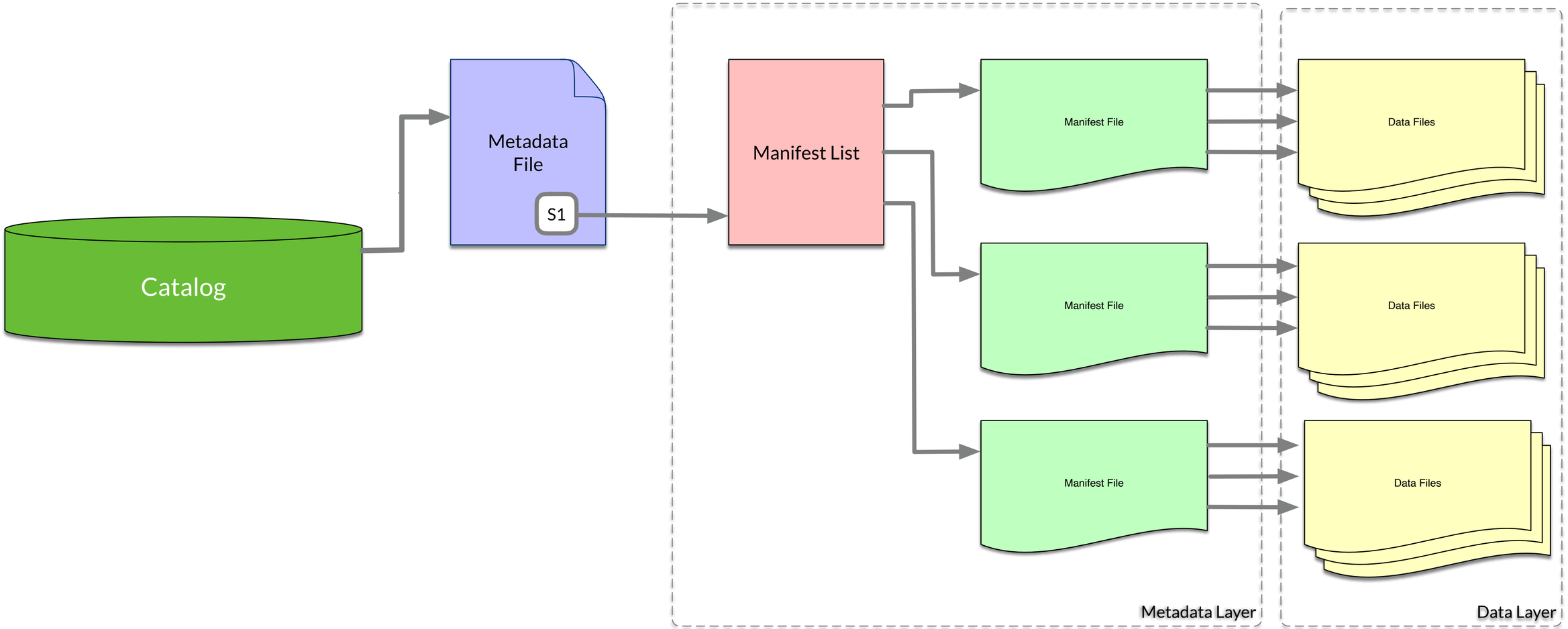
- Let’s change the schema
- Then let’s go back in time and see the data without the previous schema

Pruning

Pruning and Expiration Policies

- Snapshots track the state of a table over time (version history).
- Use `CALL system.expire_snapshots(...)` to manually expire old snapshots.
- Expiration removes **snapshot metadata** and **orphaned manifest and metadata files**
- You can express the criteria by which to prune using `retain_last` or `older_than`
- Helps storage footprint and performance
- Can be scheduled and supports incremental cleanup


Catalog
Metadata File
S0
S1
Metadata File
S0
Manifest List
Manifest List
Manifest File
Manifest File
Manifest File
Data Files
Data Files
Data Files
Metadata Layer
Data Layer


Catalog
Metadata File
S1
Manifest List
Manifest File
Manifest File
Manifest File
Data Files
Data Files
Data Files
Metadata Layer
Data Layer

Query Pruning

Query Pruning

- Queries scan only relevant files using partition & metadata information — no full table scans.
- Filters like `WHERE vendor_id = 4` eliminate all files not matching that partition.
- Manifests track min/max values and partition info — entire manifests can be skipped.
- Iceberg stores column-level stats (min, max, null count) per file — files outside filter bounds are excluded.
- Reduces I/O and increases performance

Delete Files

Delete Files

- Important to remember that data is immutable
- Deletes are tracked using separate files: delete files, which maintain row-level or file-level deletion information.
- This allows efficient updates and deletions with minimal I/O cost.

Types of Delete Files

- Equality Deletes:
  - Contain rows that match certain column values (e.g., `id = 42`).
  - Typically used in row-level deletes.
- Position Deletes:
  - Specify the exact row position in a specific data file to delete.
  - Useful when deleting known rows (like from compaction or merge-on-read engines).

How Deletion Works

1.User Issues a `DELETE`: `DELETE FROM customers WHERE id = 42;`

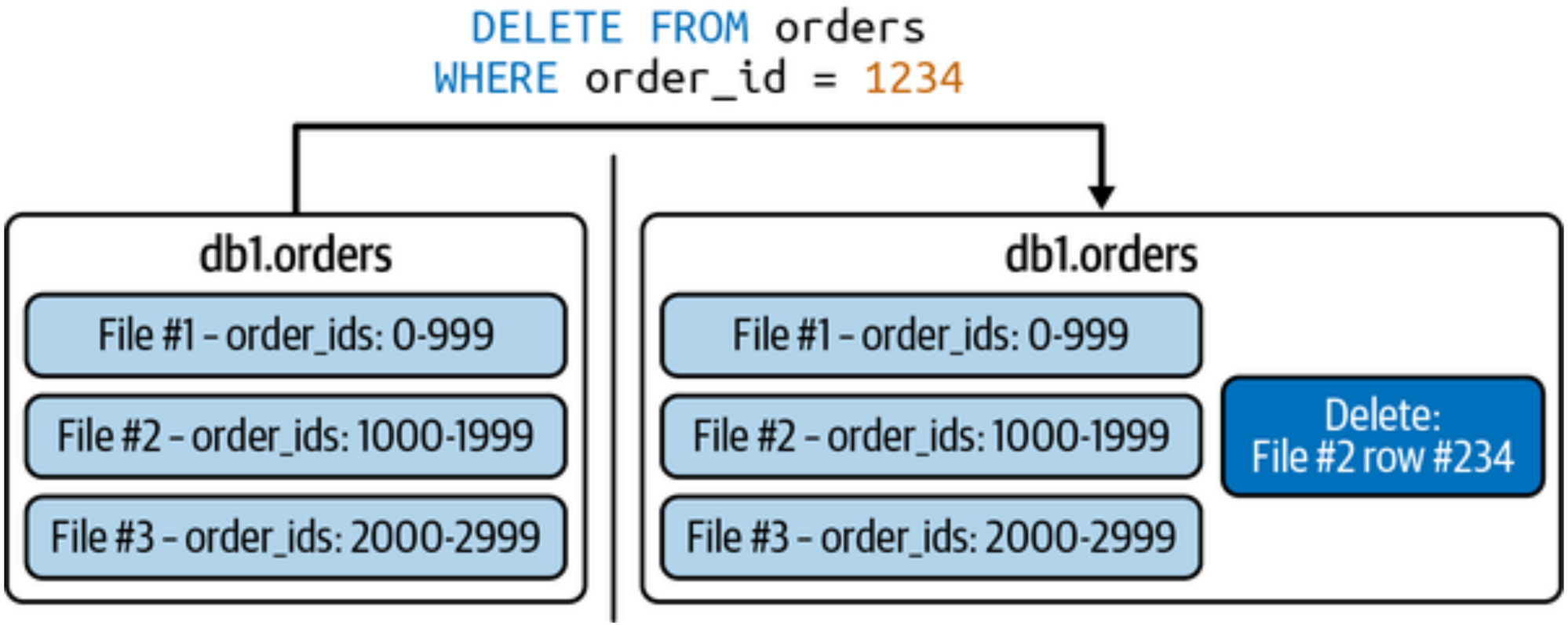
2.Nothing is actually deleted; a delete file is created in its place

3.New Snapshot Is Created:

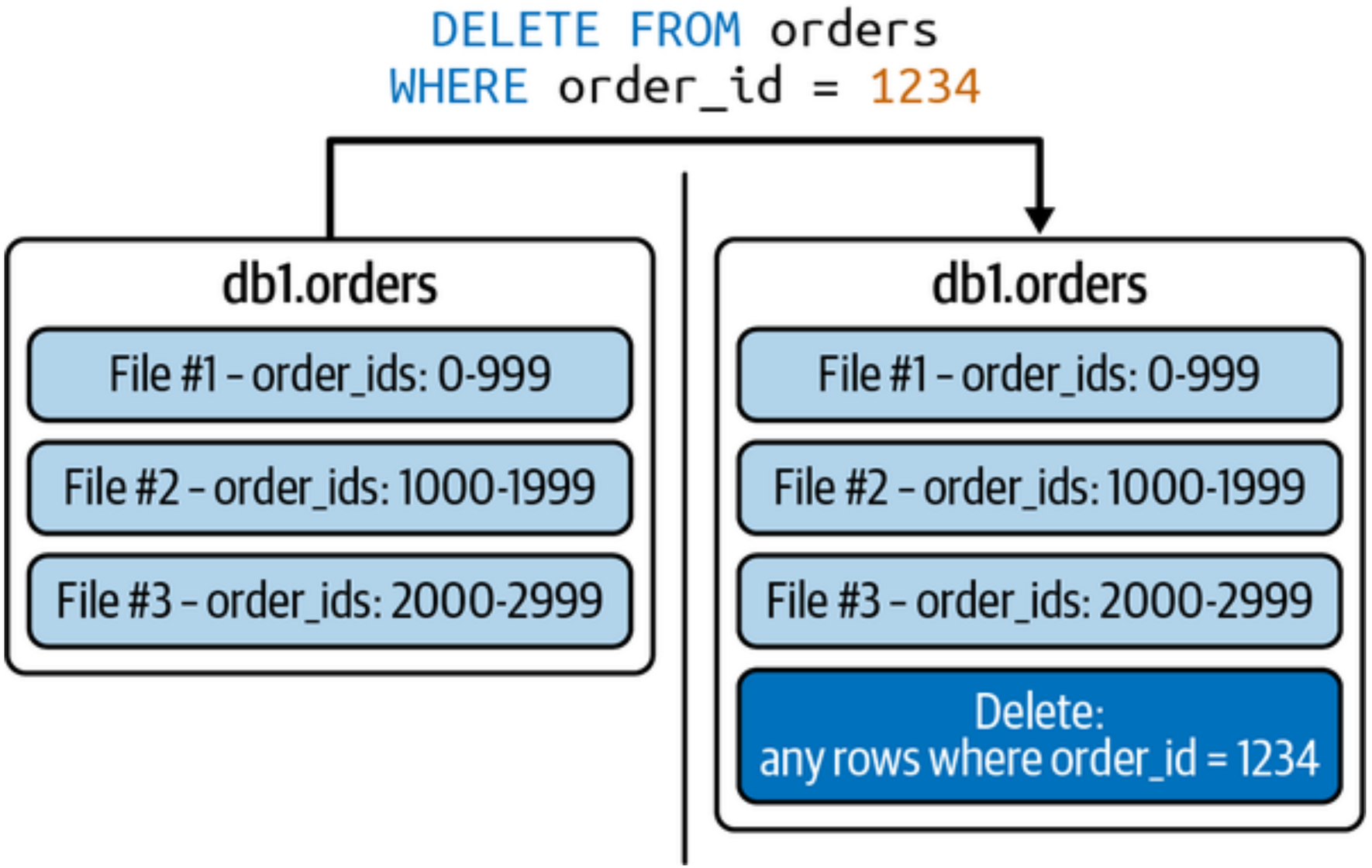
- The snapshot includes:
- The original data files
- The new delete file

4.At Read Time:

- Delete files are applied on-the-fly.
- Engines skip or filter out rows marked for deletion.



Apache Iceberg: The Definitive Guide Tomer Shiran, Jason Hughes, Alex Merced

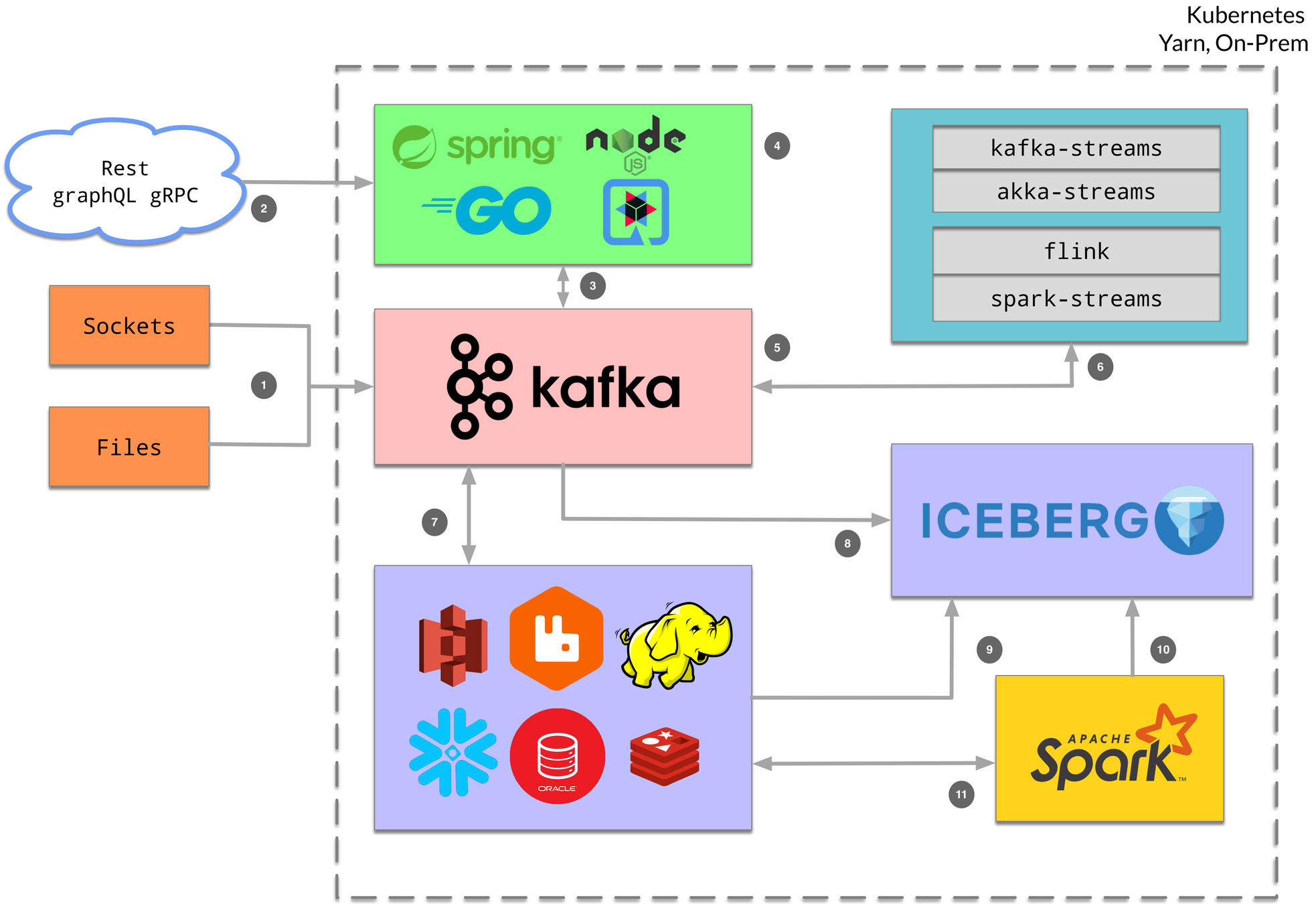


Apache Iceberg: The Definitive Guide Tomer Shiran, Jason Hughes, Alex Merced

Lab: Deletes

- Let’s see how we can run Apache Iceberg performs deletes and see the corresponding files

Live Data Feed


Kubernetes
Yarn, On-Prem
Rest
graphQL gRPC
Sockets
Files
spring
node
GO
kafka
kafka-streams
akka-streams
flink
spark-streams
ICEBERG
ORACLE
APACHE
Spark
1
2
3
4
5
6
7
8
9
10
11

Loading Live Data into Iceberg

- There are many strategies to consider in getting in live data into iceberg
  - Apache Flink
  - Apache Spark Streaming
  - Kafka Connect
  - Debezium Iceberg Connector
  - Confluent Cloud ($)
  - Likely More!

Lab: Live Data

- Let’s fill in the rest of the live data by creating some more table which we will constantly feed.

Copy On Write vs Merge On Read

Copy-on-Write Versus Merge-on-Read

- When you are adding new data, it just gets added to a new datafile, but when you want to update preexisting rows to either update or delete them, there are some considerations.

- Recall everything is immutable, so it you're updating 10 rows, there is no guarantee they are in the same file, so you may have to rewrite 10 files and every row of data in them to update 10 rows for the new snapshot.

Copy on Write

- The default approach is referred to as copy-on-write (COW). In this approach, if even a single row in a datafile is updated or deleted, that datafile is rewritten, and the new file takes its place in the new snapshot.

Apache Iceberg: The Definitive Guide Tomer Shiran, Jason Hughes, Alex Merced

Merge On Read

- The alternative to copy-on-write is merge-on-read (MOR), where instead of rewriting an entire datafile, you capture in a delete file the records to be updated in the existing file, with the delete file tracking which records should be ignored.
- **If you are deleting a record:**
  - The record is listed in a delete file.
  - When a reader reads the table, it will reconcile the datafile with the delete file.
- **If you are updating a record:**
  - The record to be updated is tracked in a delete file.
  - A new datafile is created with only the updated record.
  - When a reader reads the table, it will ignore the old version of the record because of the delete file and use the new version in the new datafile.

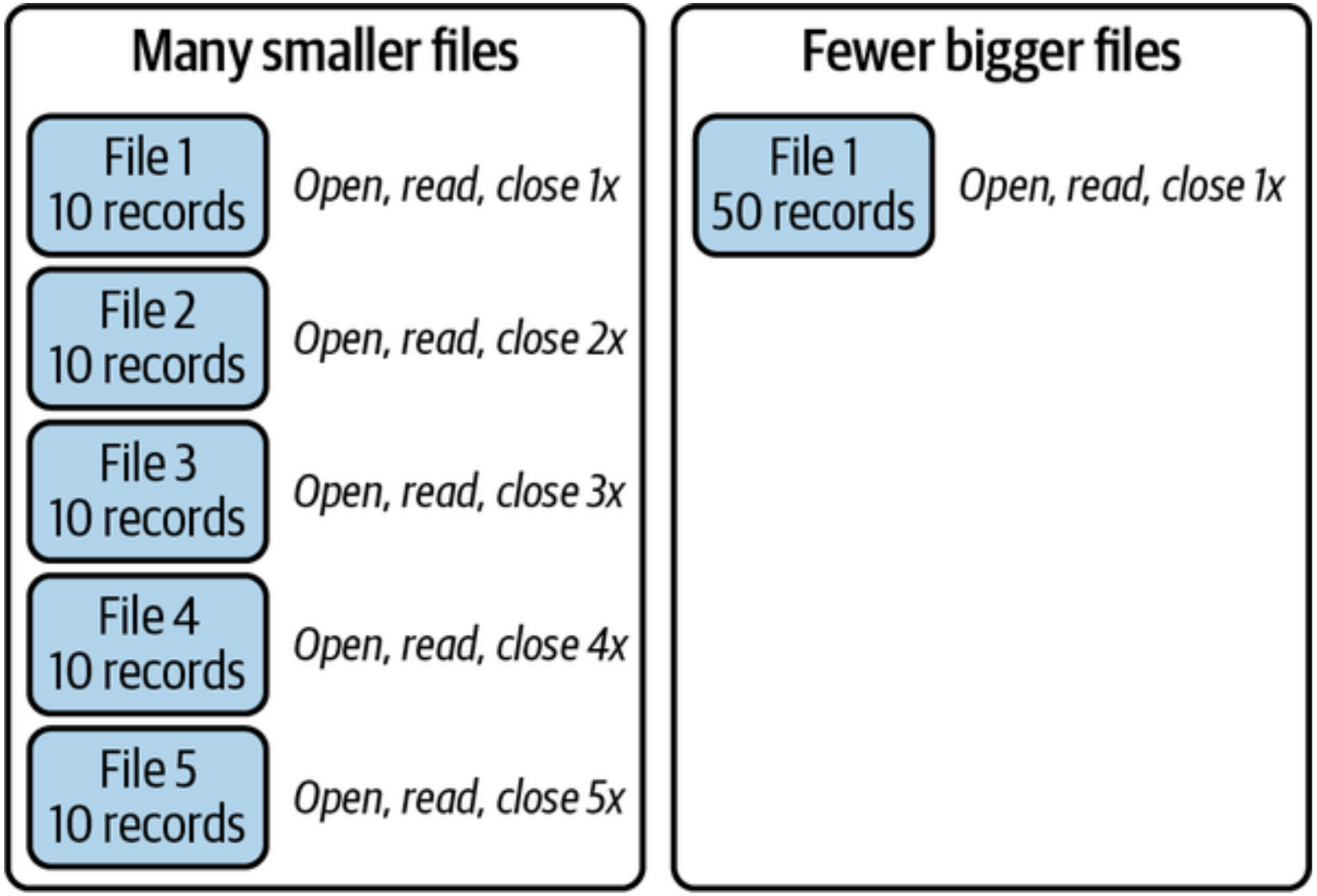
Configuring COW or MOR

- The following table properties determine whether a particular transaction is handled via `COW` or `MOR`:
  - `write.delete.mode` - Approach to use for delete transactions
  - `write.update.mode` - Approach to use for update transactions
  - `write.merge.mode` - Approach to use for merge transactions

Compaction

Compaction

- Process of rewriting small or fragmented files into larger, optimized files.
- Helps reduce metadata overhead and read amplification.
- Not required immediately—lazy and configurable.
- Especially useful after many small writes or deletes.


Many smaller files
File 1
10 records
Open, read, close 1x
File 2
10 records
Open, read, close 2x
File 3
10 records
Open, read, close 3x
File 4
10 records
Open, read, close 4x
File 5
10 records
Open, read, close 5x
Fewer bigger files
File 1
50 records
Open, read, close 1x

Why is Compaction Required?

- Improves query performance by reducing the number of files to scan.
- Enhances read efficiency (fewer I/O operations).
- Reduces planning and metadata load (fewer manifest entries).
- Cleans up old delete files and snapshot metadata.

Types of Compaction

- File Compaction (Data File Compaction):
  - Merges small Parquet/Avro files into larger ones.
- Rewrite Deletes:
  - Rewrites data files by applying delete files.
  - Produces clean, delete-free files.
- Manifest Rewrite:
  - Rewrites the manifest files to compact metadata structure.

When Does it Happen?

- Manually triggered using Flink, Spark, or Iceberg APIs.
- Can be scheduled as part of maintenance jobs.
- Often done after ingest jobs, particularly for streaming writes.

```
CALL catalog_name.system.rewrite_data_files(
  table => 'db_name.table_name',
  strategy => 'binpack',  -- or 'sort'
  options => map(
    'min-input-files', '5', -- only compact if more than 5 small files
    'target-file-size-bytes', '134217728'  -- 128MB target
  )
);
```

ACID Transactions

ACID Transactions

- Iceberg uses snapshot isolation, not locking, to handle ACID guarantees
- Atomic writes: pipelined file additions (data, manifests, metadata updates) occur all-or-nothing, avoiding partial commits
- Batch ingest safety: if an ingestion job fails mid-stream, no partial state is visible
- Schema evolution is transactional: applying schema changes creates a new snapshot; failure means no schema change is recorded
- Row-level updates/deletes are supported atomically, with conflict detection if simultaneous writers update/delete is atomic for target rows

Joins

Joins

- Iceberg provides table format support for joins — **but execution depends on the query engine** (e.g., Spark, Flink, Trino).
- What Iceberg enables:
  - Schema evolution and partition pruning for efficient joins
  - Metadata-aware planning (using snapshots and statistics)
  - Support for joining Iceberg tables with other sources (Parquet, Hive, etc.)

Types of Joins

- Since the joins are governed by your query engine of choice we can’t define all the types of joins you may have, but you should expect most of the following:
  - Inner Join - Matches rows in both tables
  - Left / Right Outer Join - Preserves rows from one side
  - Full Outer Join - Preserves all rows
  - Cross Join - Cartesian product
- You may have some other specialized joins too:
  - Semi / Anti Join - Existence or exclusion checks
  - Broadcast Join - Optimized small-table join

Lab: Joins

- Let’s have some fun with some joins and view some of the queries we can use

Aggregates

Aggregates

- Iceberg supports aggregations through the query engine — it stores data efficiently for fast scans and pruning, while the engine (Spark, Flink, Trino) executes the actual aggregation logic.
- What Iceberg enables:
  - Metadata-based partition and file pruning for faster aggregations
  - Column stats to skip unnecessary reads
  - Schema evolution without breaking aggregate queries
  - Works seamlessly with SQL aggregate and grouping functions

Types of Aggregates

- Since the aggregates are governed by your query engine of choice we can't define all the types of joins you may have, but you should expect most of the following:
  - `COUNT(expr)` - Count rows or non-null values
  - `SUM(expr)` - Add up numeric values
  - `AVG(expr)` - Compute average
  - `MIN / MAX(expr)` - Find extremes
  - `HAVING` - Further filtering after grouping

Special Types of Aggregates

- You may also have some special types of aggregates given your query engine
  - `CASE WHEN ... THEN ... END` - Conditional aggregation logic
  - `ROLLUP(...)` - Hierarchical grouping totals
  - `CUBE(...)` - Multi-dimensional aggregates

Lab: Aggregates

- Let’s view some of the power we have when it comes to aggregates and querying Iceberg

Windows

Windowing

- Iceberg supports windowed aggregations when combined with streaming engines (e.g., Flink, Spark Structured Streaming).
- Windows define how data is grouped over time before being written or queried
- . What Iceberg enables:
  - Acts as stateful sink for windowed streams
  - Works with event-time and processing-time semantics

Windowing

- Some common windows types that you can expect:
- Tumbling Window - Fixed-size, non-overlapping windows Every 5 minutes
- Sliding Window - Overlapping windows that “slide” by smaller intervals - 10-min window, slide every 2 min
- Hopping Window - Similar to sliding, with overlapping hops of fixed step - Hop 5 min, size 15 min
- Session Window - Dynamic window closed by inactivity gap - Session gap = 10 min
- Cumulative Window - Expands over time until triggered - From 0 → N minutes

Tumbling Window

Hopping Window

Session Window

Session Window

the last element has a occurred; now we wait

Session Window

waited 5 minutes = make it a window

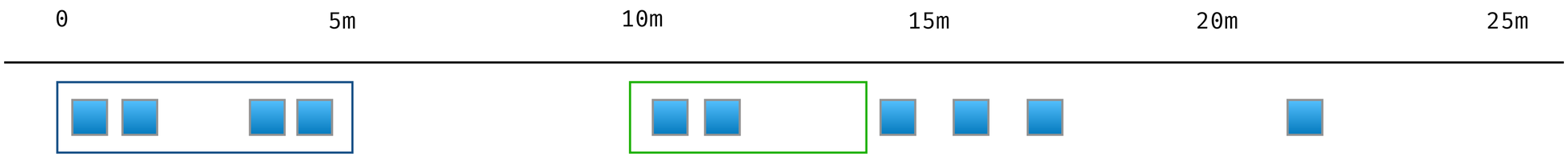
Session Window

we have a new element; we start a new window

Session Window

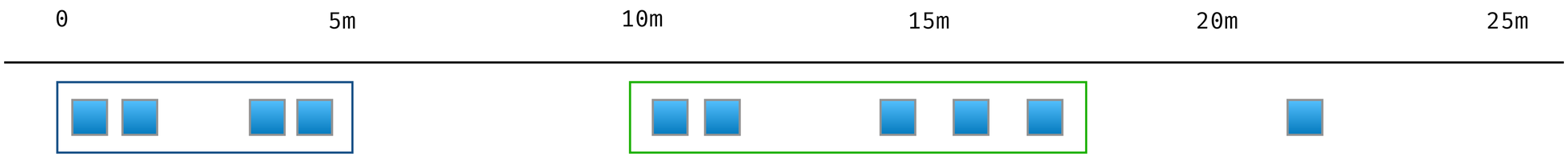
we wait for 5 minutes

Session Window



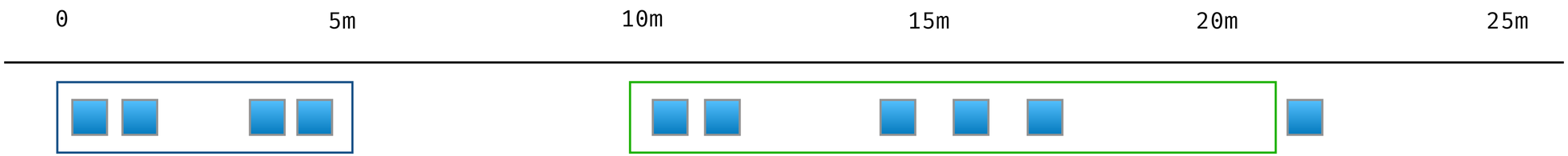
2 minutes elapsed; no windowing yet

Session Window



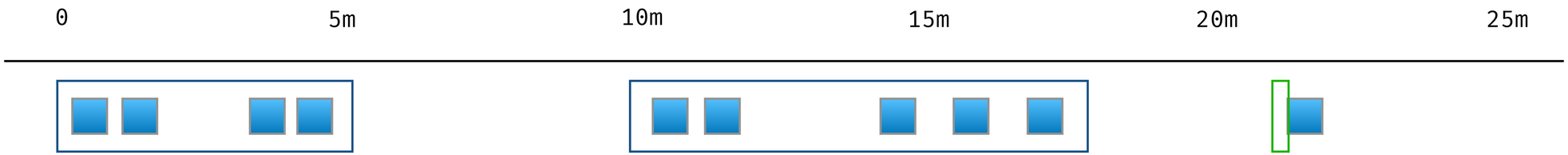
we wait five minutes

Session Window



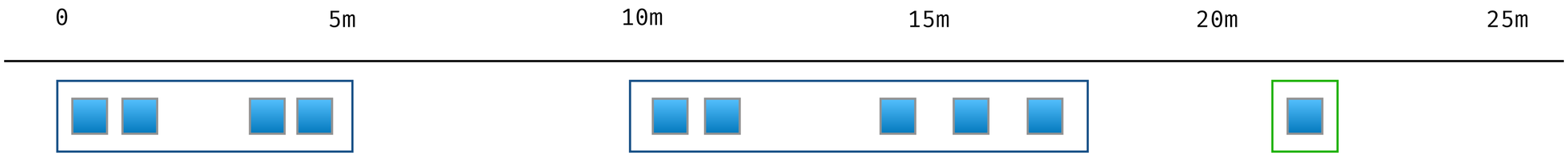
no new elements; we make it a window

Session Window



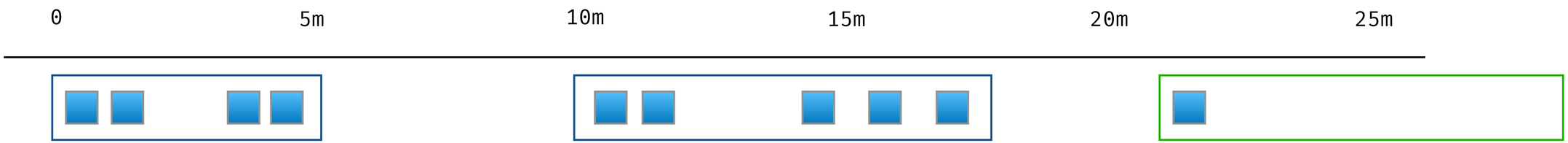
we see something new

Session Window



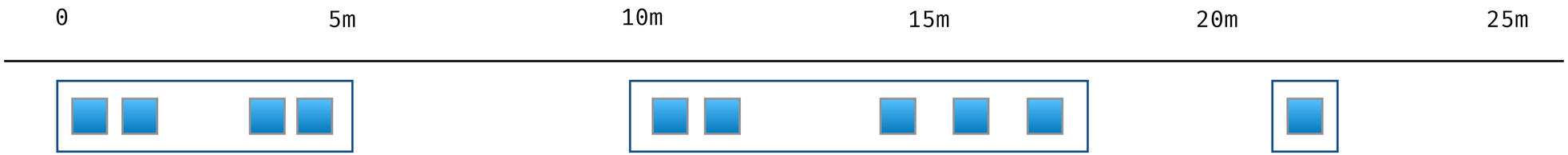
last element; we wait five minutes

Session Window



nothing else; we wait 5 minutes

Session Window



that's a new window

Lab: Windowing

- Let’s now perform some querying based on a timestamp and try some powerful querying based on time

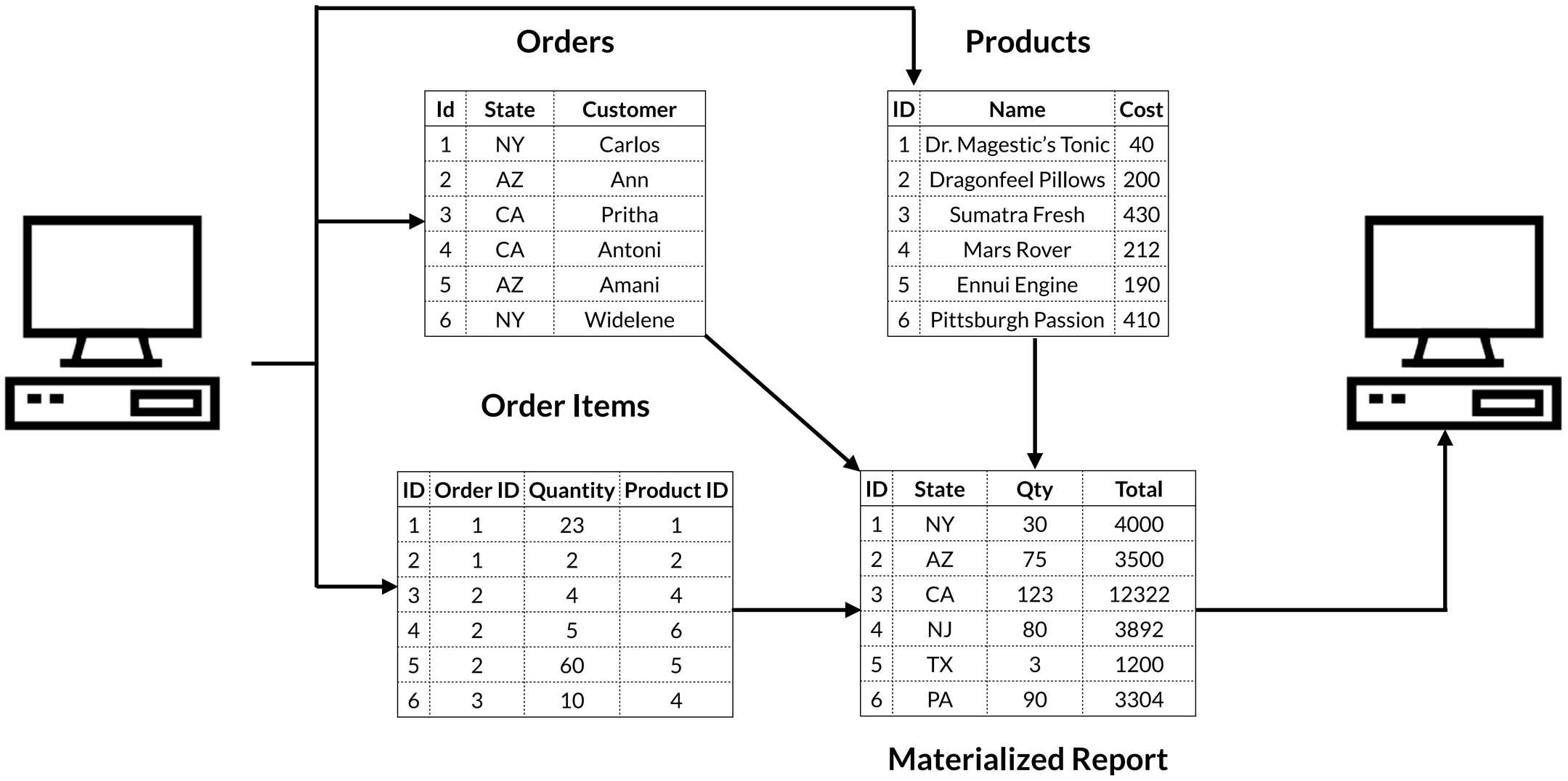
Materialized Views

Materialized Views

- Currently much of the data revolves on how it is stored, not how it is read
- **Data when read needs to transformed and prepared**
- Each entity typically has too much information for it to be queryable and usable
- For example, one data entry in a document style database can have other aggregates that are not absolutely necessary
- Data may also need to be joined, cleansed, or engineered for a particular purpose, like machine learning

Materialized Views

- Create a different perspective of the data that you need
- This can be implemented by the datastore itself:
  - Oracle
  - PostgreSQL
- Can be performed by Stream Processing Frameworks:
  - Kafka
  - Spark
  - Flink
- Used in an OLAP Data Source like Iceberg



Id	State	Customer
1	NY	Carlos
2	AZ	Ann
3	CA	Pritha
4	CA	Antoni
5	AZ	Amani
6	NY	Widelene

ID	Name	Cost
1	Dr. Magestic's Tonic	40
2	Dragonfeel Pillows	200
3	Sumatra Fresh	430
4	Mars Rover	212
5	Ennui Engine	190
6	Pittsburgh Passion	410

ID	Order ID	Quantity	Product ID
1	1	23	1
2	1	2	2
3	2	4	4
4	2	5	6
5	2	60	5
6	3	10	4

ID	State	Qty	Total
1	NY	30	4000
2	AZ	75	3500
3	CA	123	12322
4	NJ	80	3892
5	TX	3	1200
6	PA	90	3304

Lab: Materialized Views

- Let’s now create a materialized view based on other data tables.

Integrating for ML

Integration for ML

- Use Trino, Spark, or Flink SQL clients (via JDBC or Python APIs) to query Iceberg tables from Python notebooks.
- Fetch the most recent snapshot or partitioned slices of data for training ML models using Pandas, Scikit-learn, or TensorFlow.
- Use Iceberg's snapshot-based time travel to re-train or validate models using consistent historical data.
- Add or modify input features in Iceberg tables as model requirements change—without breaking existing pipelines.

Lab: Using Jupyter Notebooks with ML

- Let’s finally see how we can integrate with an ML Notebook and query some of the data

Integrating for DataMesh

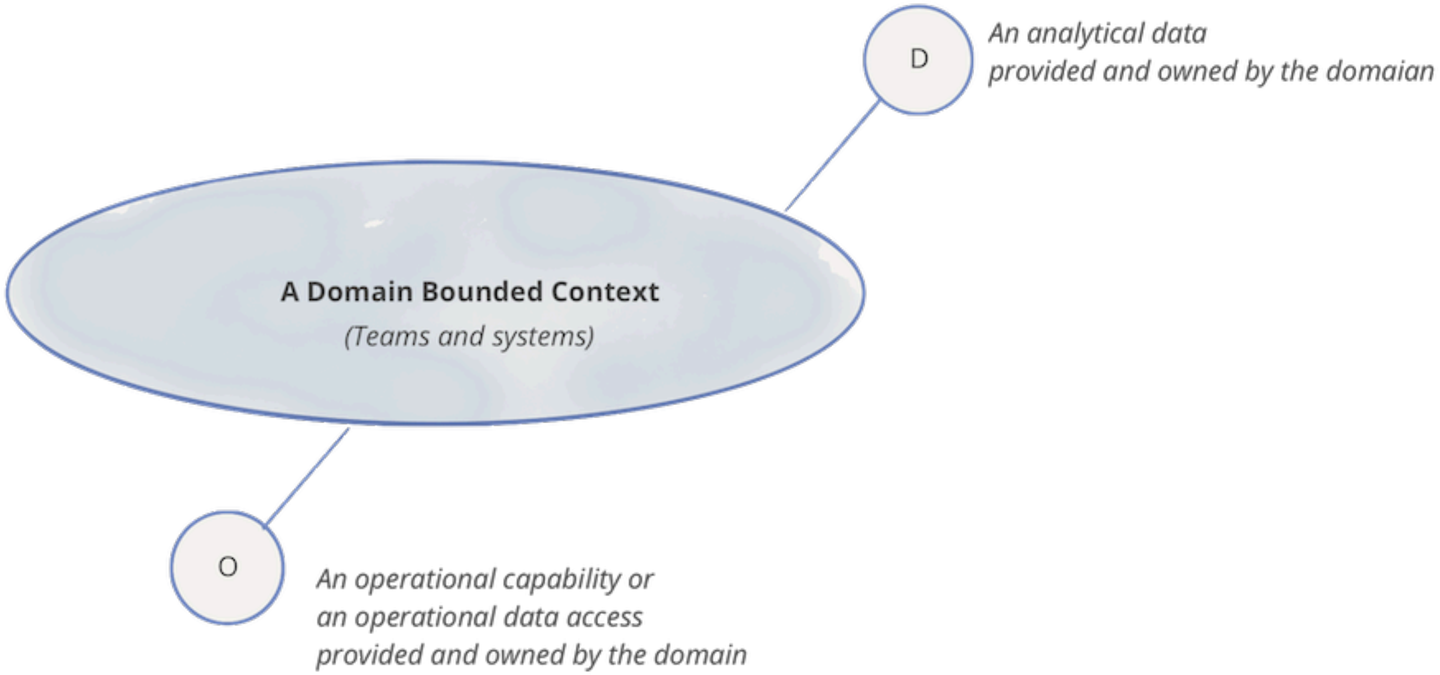
Data Mesh

- There are two patterns: Database per Service, and Shared Database
- Data Mesh is an attempt at bringing not only the database but owning all data
- Data Mesh is about providing data ownership per context, and follows these principles:
- Domain-oriented decentralized data ownership and architecture
- Data as a product
- Self-serve data infrastructure as a platform
- Federated computational governance
- The accountability of data quality shifts upstream as close to the source of the data as possible.

The corps was composed of all arms of the service, was **self-sustaining**, and could fight on its own until other corps could join in the battle. The corps itself was a headquarters to which units could be attached. **It might have attached two to four divisions of infantry with their organic artillery, it had its own cavalry division and corps artillery, plus support units**. With this organization a corps was expected to be able to hold its ground against, or fight off an enemy army for a least a day, when neighboring corps could come to its aid. "Well handled, it can fight or alternatively avoid action, and maneuver according to circumstances without any harm coming to it, because an opponent cannot force it to accept an engagement but if it chooses to do so it can fight alone for a long time."

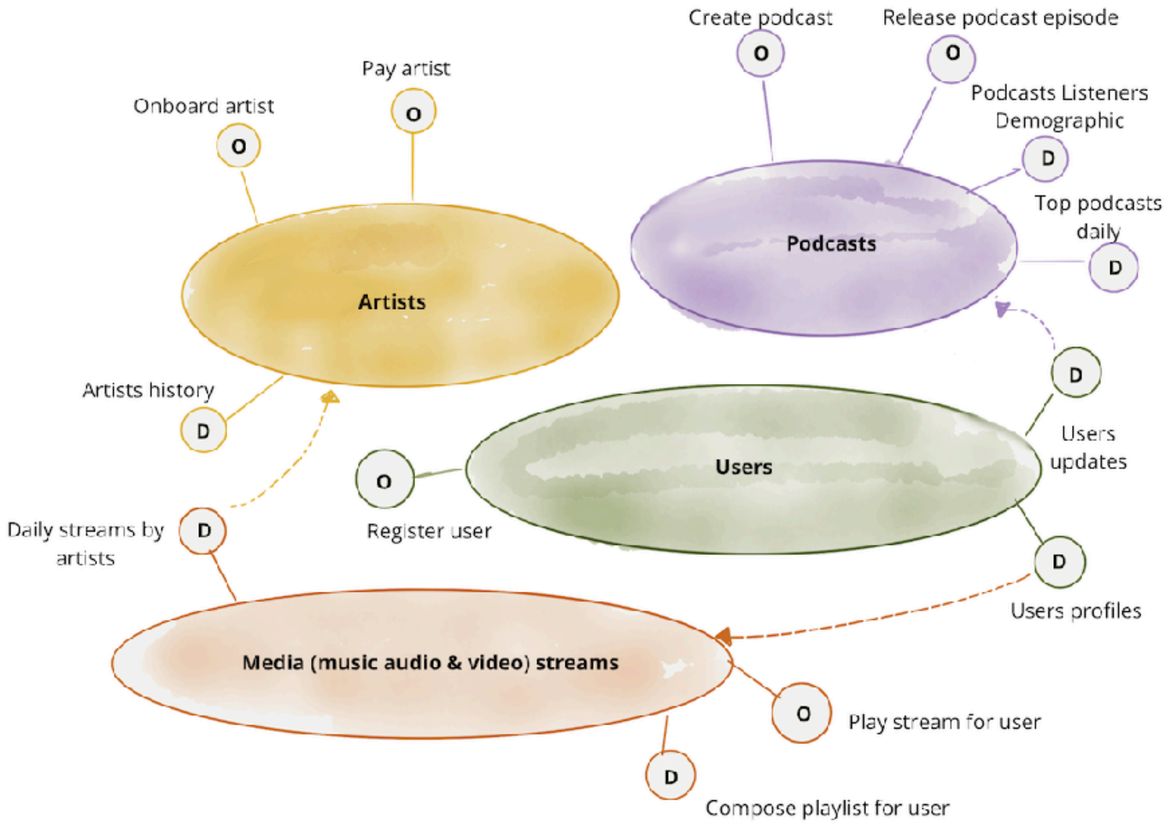
https://www.napoleon-series.org/military-info/organization/c_armycorps.html

The Diagram



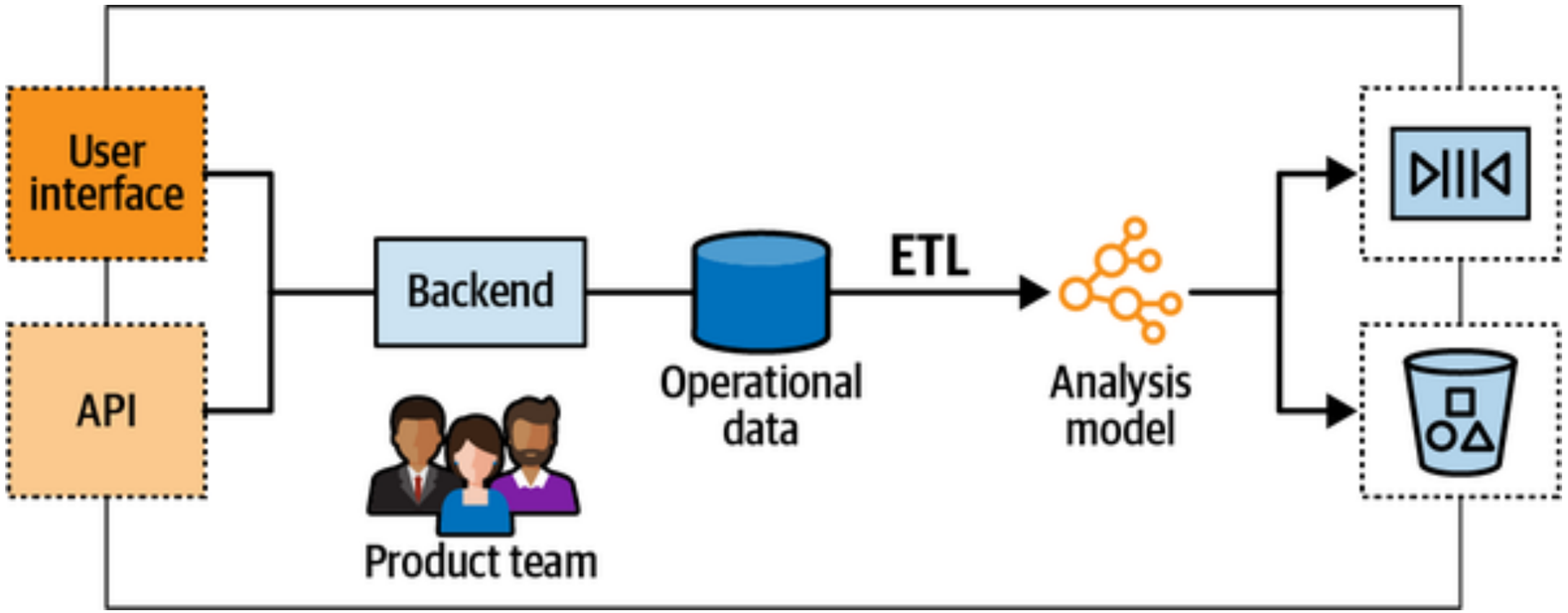
https://martinfowler.com/articles/data-mesh-principles.html

The Diagram



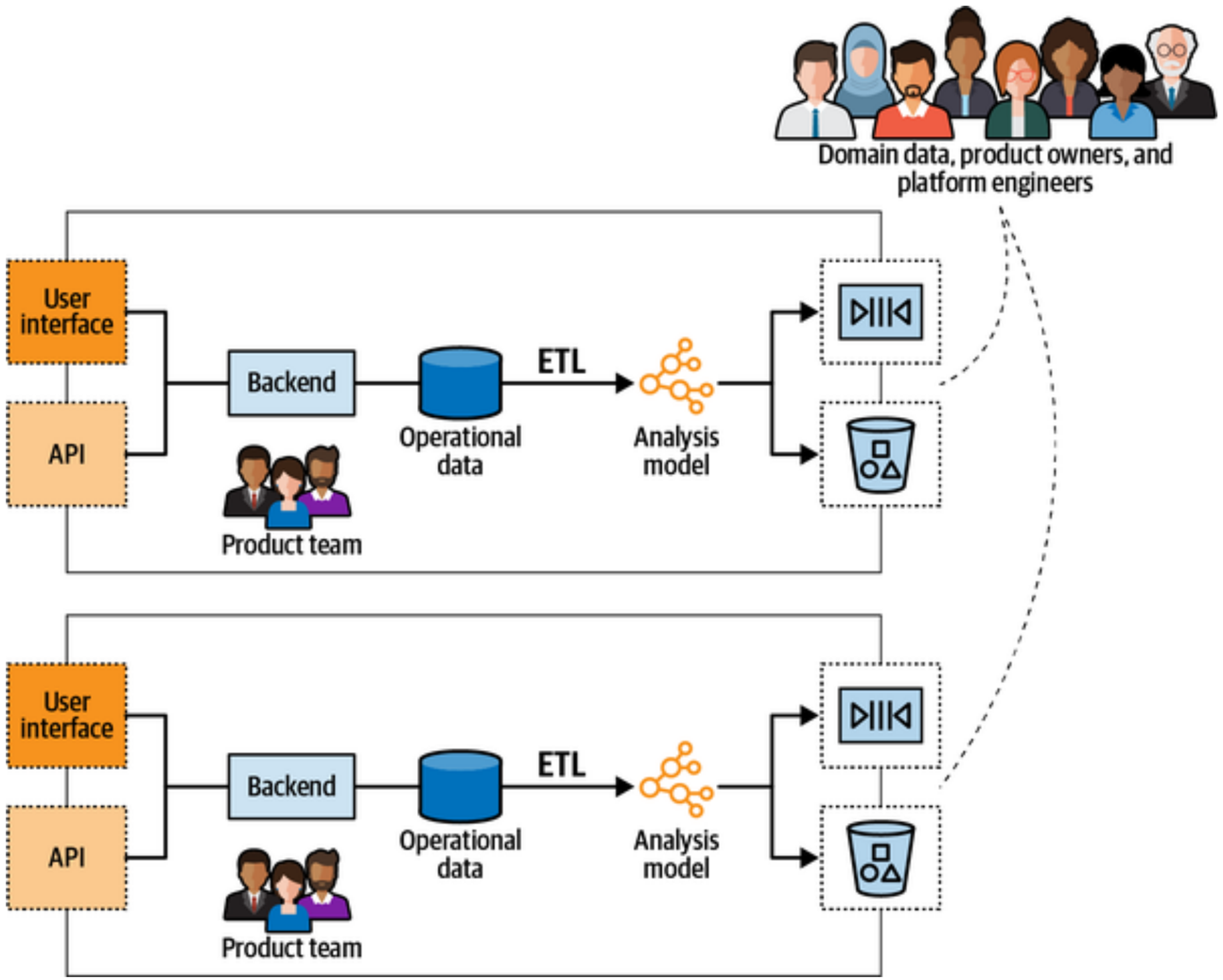
https://martinfowler.com/articles/data-mesh-principles.html

The Diagram

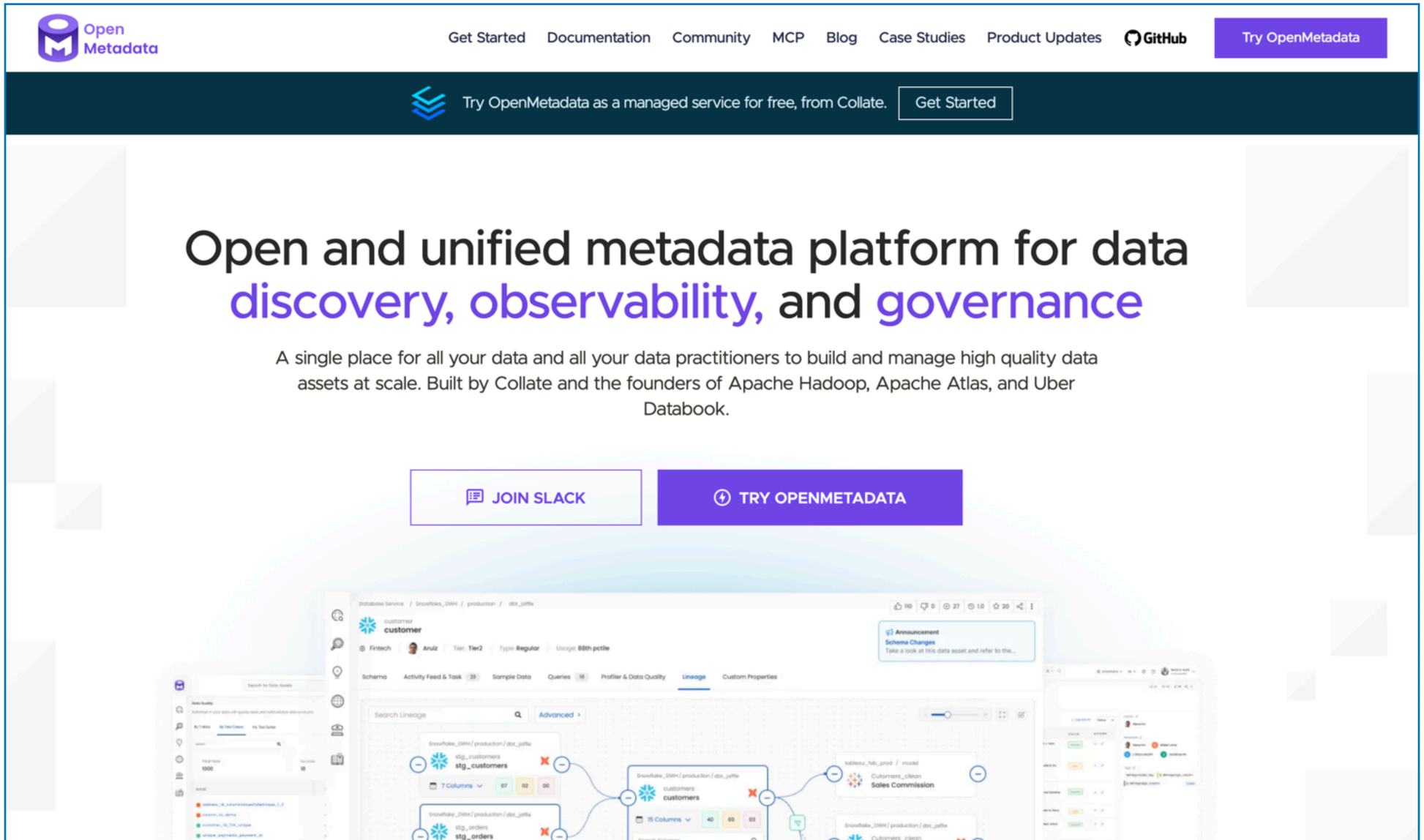


Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy - Vlad Khononov

The Diagram


Domain data, product owners, and platform engineers
User interface
API
Backend
Operational data
ETL
Analysis model
Product team
User interface
API
Backend
Operational data
ETL
Analysis model
Product team

Open Metadata



What is Open Metadata?

- Open-source metadata and data catalog platform
- Built for data discovery, governance, lineage, and observability
- Supports modern data stacks: Flink, Iceberg, Spark, Trino, Kafka, etc.
- Built for cataloging data, search and discovery, lineage tracking, governance, quality, schema versioning and more.

Iceberg, DataMesh, and Open Metadata

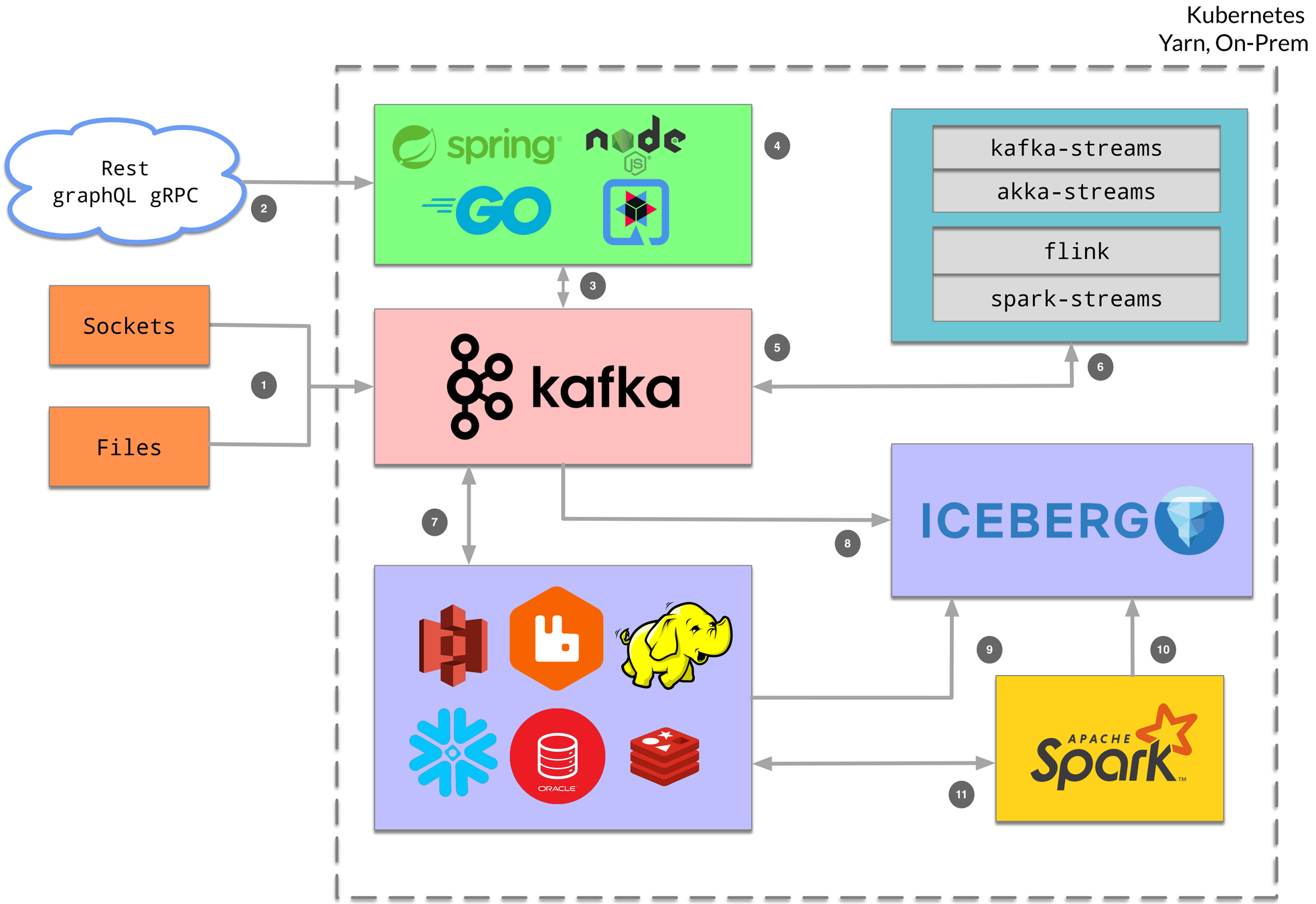
- Iceberg tables can be discovered, described, and governed through OpenMetadata’s unified catalog.
- Visualize data lineage: track how Iceberg tables are created, modified, or queried by different engines (e.g., Flink, Spark, Trino).
- Monitor schema evolution and detect breaking changes across environments.
- Govern access, usage, and retention policies for Iceberg-managed data assets across teams.

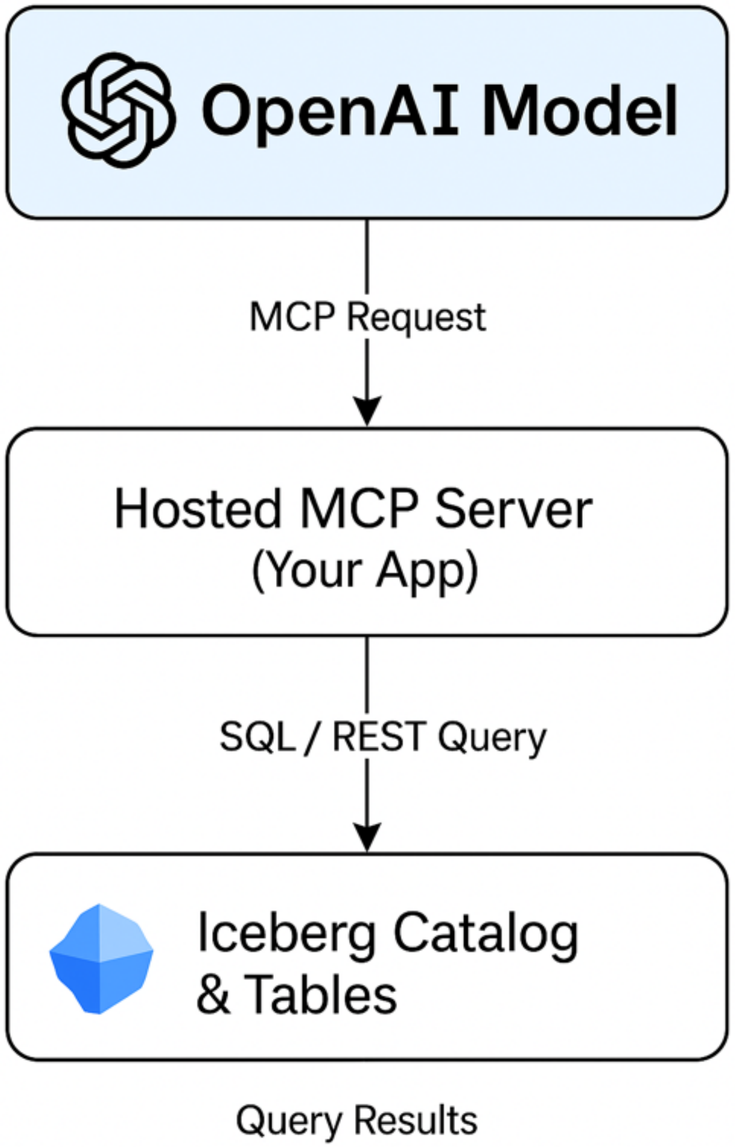
MCP and Ai

Model Context Protocol

- MCP is a protocol that lets models (like LLMs) securely access, retrieve, and reason over external data — giving them context awareness beyond their pretraining.
- **“MCP turns your infrastructure, databases, and APIs into context providers for AI.”**
- It standardizes how models:
  - Discover available data and tools
  - Retrieve structured context (SQL, JSON, embeddings, etc.)
  - Invoke external computations (e.g., analytics, business logic)
  - Stay stateless — no data is baked into the model itself

Can we ask Ai, “Who are our top customers?”


Kubernetes
Yarn, On-Prem
Rest
graphQL gRPC
Sockets
Files
spring
node
GO
kafka
kafka-streams
akka-streams
flink
spark-streams
ICEBERG
ORACLE
APACHE
Spark
1
2
3
4
5
6
7
8
9
10
11


OpenAI Model
MCP Request
Hosted MCP Server
(Your App)
SQL / REST Query
Iceberg Catalog
& Tables
Query Results

Service Discovery

`GET /.well-known/mcp`

```
{
  "name": "example-corp-mcp",
  "version": "1.0.0",
  "description": "ExampleCorp unified MCP registry for domain services.",
  "services": [
    {
      "name": "orders-mcp",
      "description": "Access to order analytics and metrics.",
      "manifest_url": "https://api.example.com/orders/.well-known/mcp"
    },
    {
      "name": "customers-mcp",
      "description": "Customer data service with profile and segmentation info.",
      "manifest_url": "https://api.example.com/customers/.well-known/mcp"
    }
  ]
}
```

Schema Location

GET https://api.example.com/orders/.well-known/mcp

```
{
  "name": "orders-mcp",
  "version": "1.0.0",
  "endpoints": { "rpc": "https://api.example.com/orders/mcp/rpc" },
  "capabilities": [
    {
      "name": "orders.topOrders",
      "description": "Retrieve top N orders by total amount.",
      "params": { "$ref": "#/schemas/TopOrdersRequest" },
      "returns": { "$ref": "#/schemas/TopOrdersResponse" }
    },
    {
      "name": "orders.trends",
      "description": "Get total sales trend over time."
    }
  ],
  “schemas”: {...}
}
```

Request Schema

```
"schemas": {
    "TopOrdersRequest": {
      "type": "object",
      "properties": {
        "period": { "type": "string", "description": "e.g. '7d', '30d'" },
        "limit": { "type": "integer", "default": 10 }
      },
      "required": ["period"]
    },
...
}
```

Response Schema

```
"schemas": {
    "TopOrdersResponse": {
      "type": "object",
      "properties": {
        "orders": {
          "type": "array",
          "items": {
            "type": "object",
            "properties": {
              "order_id": { "type": "string" },
              "customer_name": { "type": "string" },
              "total_amount": { "type": "number" }
            }
          }
        }
      }
    }
}
```

MCP Transaction

`POST /mcp/rpc`

```
{
  "jsonrpc": "2.0",
  "id": "42",
  "method": "orders.topOrders",
  "params": { "period": "7d", "limit": 5 }
}
```

```
{
  "jsonrpc": "2.0",
  "id": "42",
  "result": {
    "orders": [
      { "order_id": "A123", "customer_name": "Acme Corp", "total_amount": 12000.00 }
    ]
  }
}
```

Conclusion

- OLAP and Data Lake Houses are for extensive and fast querying
- Technology (Avro, and Parquet) enabled to perform fast querying
- Curated and governed by DataMesh governance board, categorized by medallion status
- Streamed in from multiple sources of data
- Capable of materialized views of the data.
- Strong candidate for MCP to provide context for Ai, possibility of creating chatbots, provide analysis with plain text and NLP
- Strong candidate to perform ML

Thank You

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext