Data Warehouse

Before starting with data warehouse learn what is olap and oltp, olap is online analytical processing and oltp is online transaction processing. Oltp is responsible for entering the data into data warehouse, it is responsible for keeping all the data of transactions, now the data goes to data warehouse where it structured and it runs multiple queries which is used to get insight s from data. Olap is responsible for running multidimensional complex queries on data to have knowledge from it to determine our future decisions and give the results

Oltp>>>>>data warehouse>>>>>>olap .

📖

- *OLTP = Short Notebook 📓 → Keeps quick, real-time records (transactions, updates).*
- *Data Warehouse = Encyclopedia 📚 → Stores everything in an organized way, holding data from different sources.*
- *OLAP = Librarian 👩🏫 → Helps analyze, summarize, and fetch the right insights from the encyclopedia.*

ETL

(Extract-Transform-Load)

First data is extracted from various sources and sent to the etl tools(many available in market space), now the data is cleansed{Data Cleansing (Data Cleaning) Explained

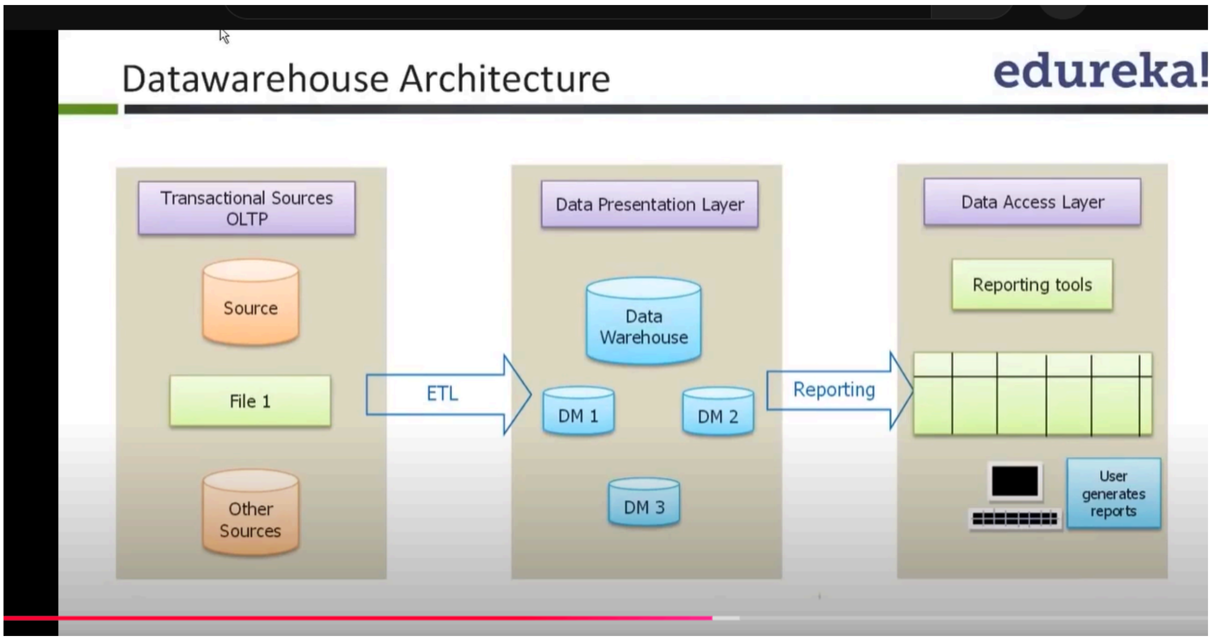
Data Cleansing is the process of detecting, correcting, and removing inaccurate, incomplete, or irrelevant data from a dataset to improve its quality and reliability.}

Now the data is loaded onto a staging area. Now all business rules will be applied on data.

1)all data should always be in master table record

And certain other steps to further ensure data authenticity and then loaded in the dimension

In the architecture of a data warehouse, first to etl that from different sources to either data mart or data warehouse then it is loaded subsequently in data warehouse or data mart respectively, whichever way you choose then there is data access layer where you get reporting tool like jasper



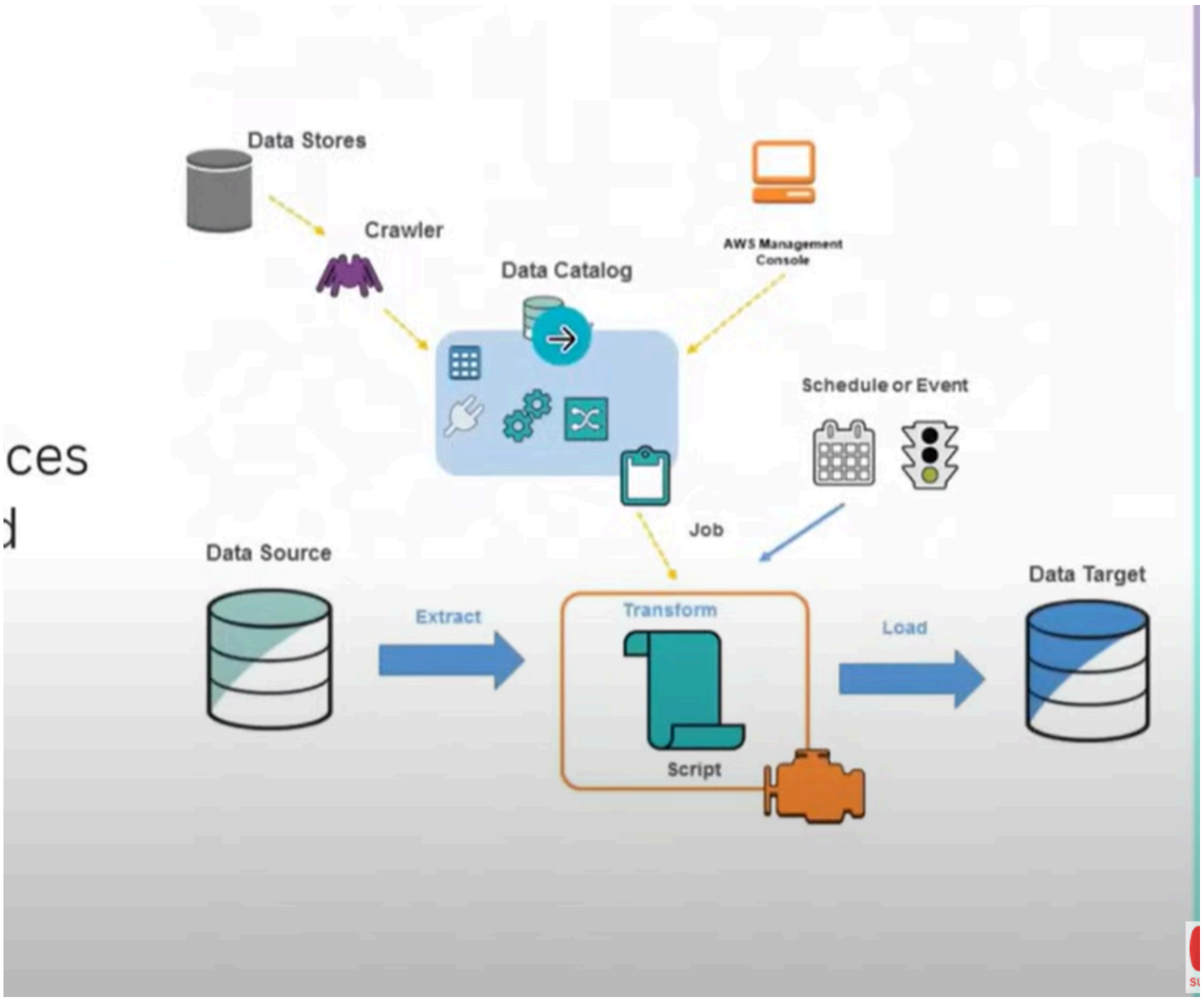
Data marts are customized reporting from data warehouse or to be uploaded to data warehouse.

Aws connections

We use connections to connect to different databases.

You have to use an etl tool to ready and clean the data some etl tools are informatica, Talend, and Apache NiFi other.

Although a lot of issues will be faced when downloading some ETL tools so AWS GLUE is preferred one as it more reliable available and easy to access.



Two main features of GLUE are data catalog and spark etl engine.

Can connect to 70 different data sources, and manage in a centralized way

Main components of AWS GLUE are :

- Database
- Crawlers
- Connections
- Tables
- Etl jobs
- Triggers

- Workflows

TABLES

object withing the aws data catalog that defines the schema of the unstructured data we added in s3 or db or rds, these table store the metadata of the actual tables stored in the sources,

Plays a crucial role in organizing the data in a queried way.

S3 – data storage

Dynamo DB- NoSQL of AWS

RDS- SQL of AWS

Crawlers

It is a program that connects to the data source and automatically scans data in various sources to determine its schema and create metadata tables in aws glue data catalog

{{so can edit the table schema ourselves after creating databases and tables or use crawlers to infer automatically if we are not ensuing for precision in terminal }}

Aws connections

We use connections to connect to different databases to RDS, Dynamo db

Aws glue ETL jobs

Aws visual ETL works the same way as the diagram in in any etl tool like talend and informatica works, here you take data, therefore choose a source select what you want to do with data hence choose transform and at last target of where to store the altered data.

Aws glue can use data sources from s3, redshift and others as sources and even on premises server

While working on aws glue you also have to setup **IAM(Identity & Access Management)** roles and later assign it to that service so that service first can access the databases and sources to take data and secondly to log that data in cloudwatch logs,

Now will discuss some other ETL tools in the aws glue

1.) ***ETL*** jobs -you can create what you want to do with your data and can do it in different types, visual, code, etc

2) ***Connectors***- Google BigQuery is available as a connector in AWS Glue to provide users with the ability to seamlessly move data between Google BigQuery and other data sources or destinations in AWS, enabling a more integrated, multi-cloud data processing and analytics workflow. This can be especially useful for organizations that use services from both Google Cloud and AWS, or for those that want to leverage the strengths of both cloud platforms.

3) ***Data Catalog tables***- which stores metadata of your sourced tables.

{METADATA- data about the table itself, not the data stored within it. It includes information like table names, column names and types, primary and foreign keys, indexes, constraints, and relationships with other tables.}

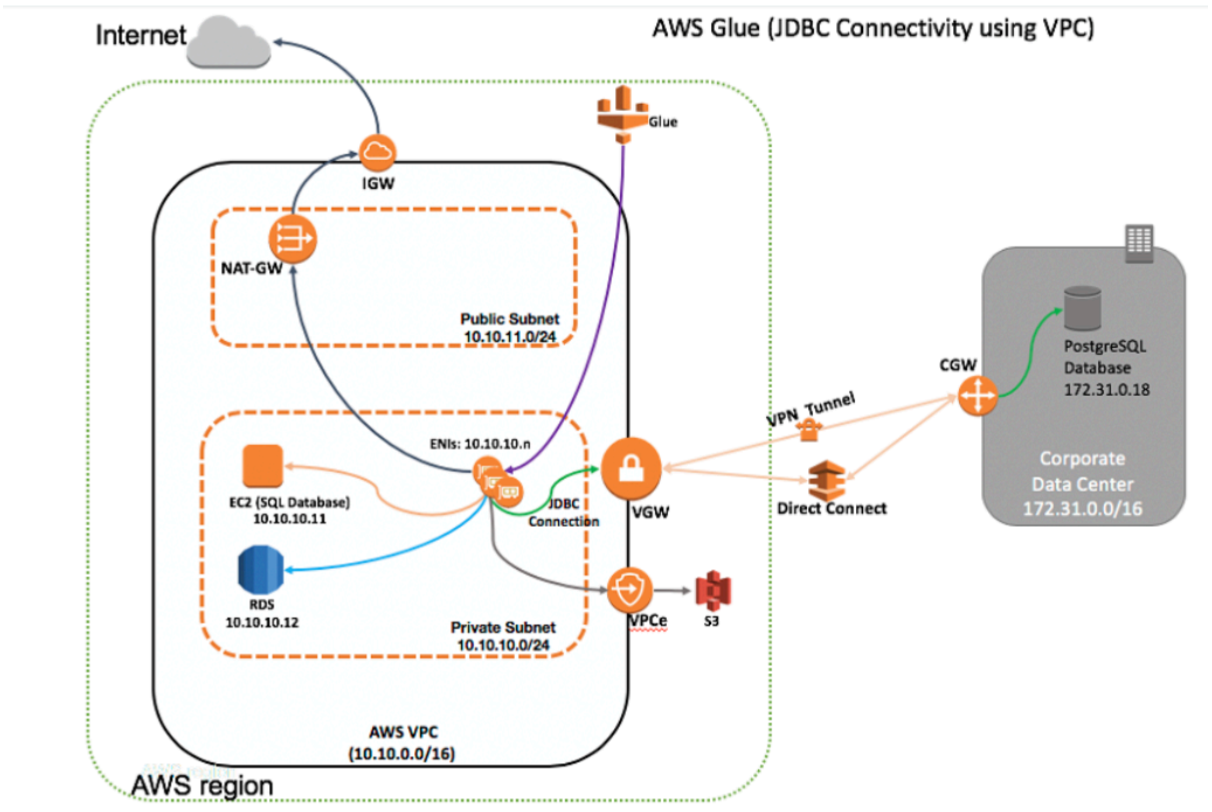
4) ***Workflow***- A workflow is a collection of multiple dependent AWS Glue jobs and crawlers that are run to complete a complex ETL task.

{when you create a job it will be added as one, when you trigger for that job it will be added as another.}

5)***Zero ETL integration***- Zero-ETL integration in AWS Glue refers to a set of fully managed integrations that minimize or eliminate the need to build traditional ETL pipelines for data movement and replication. Instead of extracting, transforming, and loading data, zero-ETL allows direct access to data in-place or replication to destinations like Amazon Redshift and Amazon SageMaker Lakehouse without the need for intermediate transformation steps.

6)AWS Glue Stream schema registry- It tracks and record the schemas of our tables to make sure consistency is maintained across glue and every data is standardized.

XX->How to connect aws glue to on-prem server of data-sources



To have this connections completed follow these steps:

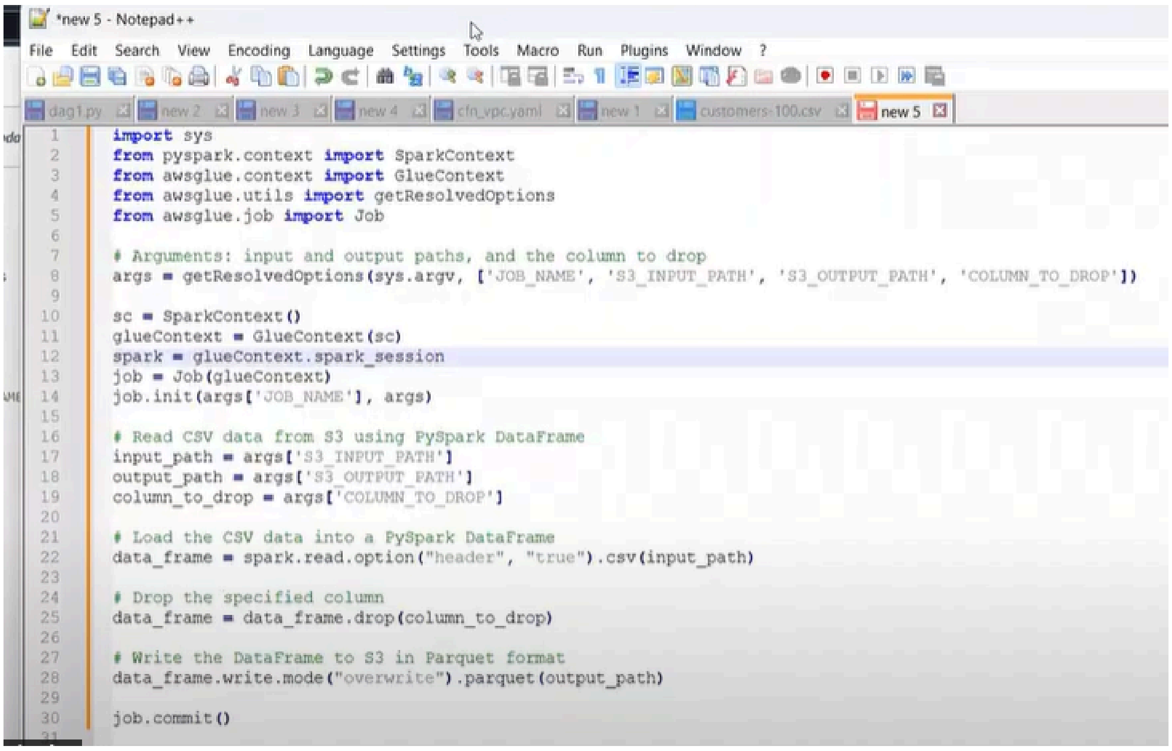
JDBC(java database connectivity)

https://aws.amazon.com/blogs/big-data/how-to-access-and-analyze-on-premises-data-stores-using-aws-glue/

1) Create a security group for where in inbound rules you allow all tcp for all
2) For outbound allow all tcp, port 5432 for SQL(your device ip) , DNS udp and tcp all, and also open it for s3 (so add s3 ip) in ip prefix list as well as https for

3) $ curl -s https://ip-ranges.amazonaws.com/ip-ranges.json | jq -r '.prefixes[] | select(.service=="S3") | select(.region=="us-east-1") | .ip_prefix' for ip of s3
4) Then create an iam role as told
5) Go to connection, and add a jdbc connection you will find it, in there add a jdbc URL which is jdbc:protocol://host:port/db_name
6) Protocol is port is 5432 db_name you will tell ChatGPT it will tell you what the protocol will be
7) jdbc: postgresql://172.31.0.18:5432/glue_demo
8) open apt firewall ports in on-prem so that data from private subnets can come to it,
9) now test jdbc connection
10) and other some steps like adding crawler you can learn it from the website

now to have the etl job create by pyspark code script ::::


*new 5 - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
dag1.py
new 2
new 3
new 4
cfn_vpc.yaml
new 1
customers-100.csv
new 5

```python
import sys
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.utils import getResolvedOptions
from awsglue.job import Job

# Arguments: input and output paths, and the column to drop
args = getResolvedOptions(sys.argv, ['JOB_NAME', 'S3_INPUT_PATH', 'S3_OUTPUT_PATH', 'COLUMN_TO_DROP'])

sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args['JOB_NAME'], args)

# Read CSV data from S3 using PySpark DataFrame
input_path = args['S3_INPUT_PATH']
output_path = args['S3_OUTPUT_PATH']
column_to_drop = args['COLUMN_TO_DROP']

# Load the CSV data into a PySpark DataFrame
data_frame = spark.read.option("header", "true").csv(input_path)

# Drop the specified column
data_frame = data_frame.drop(column_to_drop)

# Write the DataFrame to S3 in Parquet format
data_frame.write.mode("overwrite").parquet(output_path)

job.commit()
```

in here whatever has been greyed out you need to specify in details which are the parameters, where you will specify where to where from , what to etc.

getting back on warehousing !!!!!!!!!!!

Data marts: smaller versions of data warehouse, they basically deal with data from limited sources.

Therefore time efficient in creation.

3 types of DM:
1: dependent data mart
Oltp>>>data ware>>>data mart
2: independent data mart
Oltp>>>data mart
3: hybrid data mart
Oltp>>>data mart<<<<data warehouse

While creating Data warehouse there are 2 approaches:
1: top down (dw>>>>dm) (Inmon approach)
2: bottom up(dm>>>dw)
{dw- data warehouse, dm- data mart}
[>>> indicating flow direction of data]

ODS (operational data store)
It is small window of data between oltp and data warehouse, it keeps real time data, no backing memory,
It is either continuous or snapshot ods.

Some important things in data warehousing
1: dimension table (Table of content)
2: subject (topic of table)
3: dimensions (diff subtopics)
4: attributes (subtopics well defined)

Fact table

A measure that can be manipulated summed or multiplied, averaged. It contains a dimension a dimension key and a measure

Such as product	product_id	unit_sold

Types of facts
1: additive facts
2: semi-additive facts
3: non-additive facts
4: fact less fact
5: transaction fact table
6: snapshot fact table
7: accumulated fact table

Slowly Changing Dimensions
Dimensions which don’t change for a long time are slowly changing dimensions, but when need to be changed they are amended or edited in 3 types
1:overwrite the old value
2:add a new row
3:add a new column

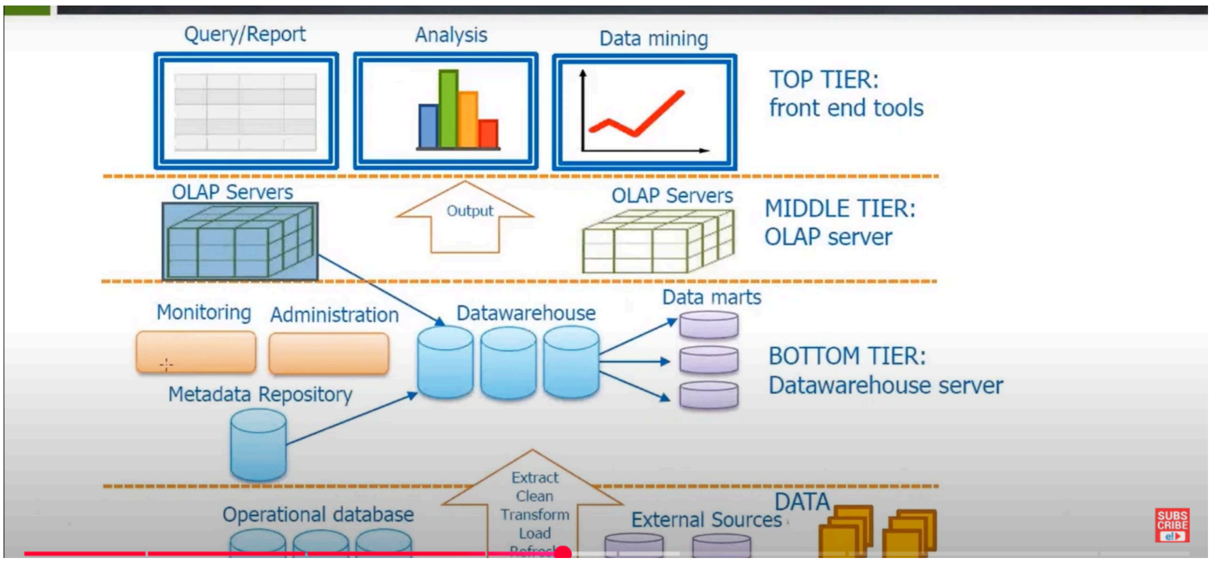
Conformed dimension: sharing related
Degenerated dimension: not related to anyone
Junk dimension: A junk dimension is essentially a collection of random transactional codes, flags, and/or text attributes which are unrelated to any particular dimension

Role playing dimension:A role-playing dimension is a dimension table that can be used in multiple contexts, each with a different meaning. For example, a date dimension table can be used to create order date, ship date, and delivery date dimensions.

TYPES OF ANOMALIE:
1: insert anomaly
2: update anomaly
3: delete anomaly
Therefore, to avoid these anomalies we use the NORMALIZATION:
Arranging the data in small groups
Remove data redundancies
Updating the table should change only at one place not all things updates
There must be no impact on speed accuracy and integrity of the data

HEIRARCHIES:
Where dimensions have a parent child relationship among each other, 3 types of hierarchies

1: balanced
2: unbalanced
3: Ragged
1:->->where same level dimensions have same no. of child
And so, on other could be intercepted that way
3: where levels are skipped



(MIDDLE TIER)
1. **ROLAP (Relational OLAP)**
   - **Storage**: Uses relational databases (RDBMS) to store data in tables.
   - **Query Processing**: Translates OLAP queries into SQL and retrieves data dynamically.
   - **Performance**: Slower for complex queries because it depends on SQL execution in relational databases.
   - **Scalability**: Can handle large datasets efficiently since it leverages relational databases.
   - **Example Tools**: IBM Cognos, MicroStrategy, SAP BusinessObjects.
2. **MOLAP (Multidimensional OLAP)**
   - **Storage**: Uses a precomputed, multidimensional data cube to store data.
   - **Query Processing**: Retrieves data directly from pre-aggregated cubes, making queries much faster.
   - **Performance**: High speed for complex analytical queries since data is already pre-processed.
   - **Scalability**: Limited by storage, as data cubes can become very large.

- **`Example Tools`**`: Microsoft Analysis Services (SSAS), Oracle Essbase, IBM TM1.`

DATA LAKES

A data lake is like a massive data repository, designed to store any kind of data or big data which can be structured, semi-structured and unstructured data. And it makes possible to store data in its original as-it forms.

A data lake holds data volumes before a specific use case has been identified. For better decision making, there is no need to be concerned about structuring the data first or having different types of analytics execution from dashboard and visualization, real time analytics and machine learning for big data processing.

- **Analytical Sandboxes:** An analytical sandbox layer is secured, testing environment. This layer provides a dedicated environment within a data lake architecture, where data scientists, analysts or researchers tests and experiment and explore the data and derive insights, without compromising the integrity or quality of the data. Transformed data and raw data both imported into the analytical sandboxes.
- **Data Consumer:** As we delve further into the architecture, we reach the Data consumer layer. At this layer, the data is accessible and available once all preceding processes have been completed. This is where the data is ready for analysis and insights generation.

Steps:(by aws lake formation, Athena , s3, glue)

Step 1: Create IAM User

- Search and select below permissions:
  - **AmazonS3FullAccess**
  - **AmazonAthenaFullAccess**
  - **AWSCloudFormationReadOnlyAccess**
  - **AWSGlueConsoleFullAccess**
  - **CloudWatchLogsReadOnlyAccess**

Step 2: Create IAM Role

- After creating IAM User, now we have to create IAM Role, to catalog the data which is stored in Amazon S3 Bucket for our Data Lake.

Next, select "**AWS Service"** option**.** and type **"Glue"** as the AWS service **. add permissions, search for "PowerUserAccess" policy**

Step 3: Create S3 Bucket to Store the Data

- We have successfully created our IAM users and IAM role for our AWS Data Lake, now to store our data we need to create Amazon S3 Bucket. in this demonstration we are uploading data manually into the S3.

Upload the files

Step 4: Data Lake Set Up using AWS Lake Formation

- Our data is ready to ingest into the data lake. now will begin to set up our Data Lake. in data lake we will create a database. Search and navigate to the **AWS Lake formation console.**
- Add administrator that performs administrative tasks of data lake. click on "**Add Administrators**" button to add administrators for your data lake (if you are working with AWS Lake Formation for the First time, only then "**Add administrators**" window will pop up).
- Administrator is added, now it's time to create a database. you will find the option to create a database in left hand side menu click on "**Databases**" and under databases click on "C**reate database**" button.
- Enter **Database Name** as per your wish. after that you have to browse and provide your S3 bucket path in which your data is stored, in the "**Location"** box.
- Also make sure to uncheck the "**Use Only IAM Access Control for New tables in this database**" checkbox. after that click on "**Create Database**" button. and here you go your database is created in no time.
- Database is created, now we have to register our S3 bucket as a storage for our data lake. for that find and click on "**Data Lake locations**" option from the left-hand side menu, click on "**Register Location**", browse and enter S3 bucket path where data is stored. after giving S3 path, choose IAM role as " **AWSServiceRoleForLakeFormationDataAccess**" by default and click on "**Register Location**".

Step 5: Data Cataloging using AWS Glue Crawlers

- While building the Data Lake, it is essential for data in the data Lake should be catalogued. using AWS Glue the process of data cataloging becomes easy.

- AWS Glue provides ETL (Extract, Transform, Load) service, meaning AWS Glue first transform, cleanse and organize data coming from multiple data sources before loading data into the Data Lake. AWS Glue makes data preparation process efficient by automating the ETL jobs.
- AWS Glue offers crawlers which automates the data catalog process, for better discovery, search and query big data.
- To create a data catalog in the Database, AWS Glue Crawler will use IAM role which we have created in previous step.
- Go back to the AWS Lake Formation console again, click on "Databases" option you will see your previously created database. select your database and you will see an "**Action**" button, under **Action** Dropdown menu click on "**Grant**" option.
- On the next window, you have to choose your previously created IAM Role for "**IAM Users & roles**". scroll down you will see **Database Permissions** field, check boxes for only "**Create Table**" and "A**lter**" permissions and click on "**Grant**" button.
- Navigate to the AWS Lake Formation console, click on "**tables**" from the menu, you can check here also **table** is created.

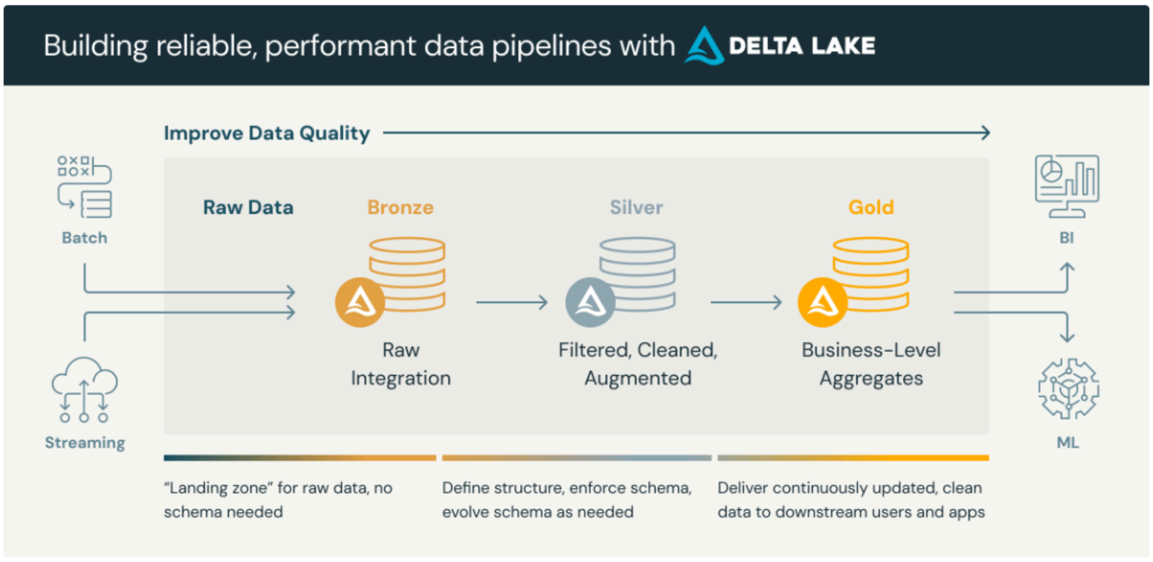
Step 6: Data Query with Amazon Athena

- Amazon Athena is a Query Service offered by AWS, Amazon Athena allows us to analyze data which stored in Amazon S3 Bucket efficiently using Standard SQL.
- When we are working with a large amount of data, we need some sort of querying tool for analyzing the data or big data, and here is where Amazon Athena comes into play, using Amazon Athena makes it easy for analyzing the data present in Amazon S3 Bucket.
- When we are using Amazon Athena, we don't need to be good at SQL (Structured Query Language) for querying data, by default Athena supports Standard SQL Query language, because of that data analysts, data scientists and organizations are able to perform analytics and derive valuable insights from the data.
- Amazon Athena allows user to query data stored in Amazon S3 in its original format. Navigate to the Amazon Athena Console.
- Click on "Query Editor", select Database which we have created in the earlier steps, but before executing any query we need to provide "**Query Result Location"** which is Amazon S3 Bucket.
- Amazon Athena stores Query Output and Metadata for each Query which executes in "**Query Result Location**".
- we have to create S3 bucket to store our Query results in this bucket, click on "**Set up a query result location in Amazon S3"** tab and provide S3 bucket's path and hit the "**Save**" button.
- We have added the **"Query Result Location"**, Now we can Run our Queries in Amazon Athena Query Editor.
- Run the following MySQL Query and click on "**Run**" button.

(by s3, Athena, aws cli, docker or gui)

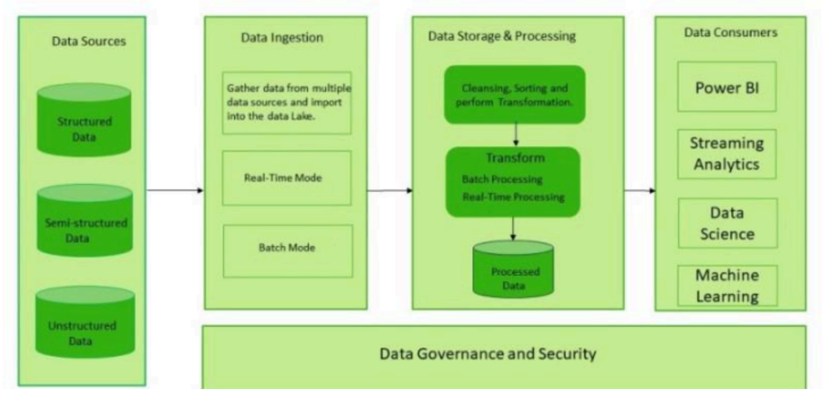
Go to gary Stradford youtube, amazon.

DATA LAKEHOUSE


Building reliable, performant data pipelines with DELTA LAKE
Improve Data Quality
Batch
Streaming
Raw Data
Bronze
Silver
Gold
Raw Integration
Filtered, Cleaned, Augmented
Business-Level Aggregates
BI
ML
"Landing zone" for raw data, no schema needed
Define structure, enforce schema, evolve schema as needed
Deliver continuously updated, clean data to downstream users and apps

Data Lake Architecture

The following diagram illustrates the AWS Data Lake Architecture and its components are discussed clearly in the below sections:



(Fig.) MEDALLION ARCH.

Bronze layer (raw data)

The **Bronze layer** is where we land all the data from external source systems. The table structures in this layer correspond to the source system table structures "as-is," along with any additional metadata columns that capture the load date/time, process ID, etc. The focus in this layer is quick

Change Data Capture and the ability to provide an historical archive of source (cold storage), data lineage, auditability, reprocessing if needed without rereading the data from the source system.

Silver layer (cleansed and conformed data)

In the **Silver layer** of the lakehouse, the data from the Bronze layer is matched, merged, conformed and cleansed ("just-enough") so that the Silver layer can provide an "Enterprise view" of all its key business entities, concepts and transactions. (e.g. master customers, stores, non-duplicated transactions and cross-reference tables).

The Silver layer brings the data from different sources into an Enterprise view and enables self-service analytics for ad-hoc reporting, advanced analytics and ML. It serves as a source for Departmental Analysts, Data Engineers and Data Scientists to further create projects and analysis to answer business problems via enterprise and departmental data projects in the Gold Layer.

In the lakehouse data engineering paradigm, typically the ELT methodology is followed vs. ETL - which means only minimal or "just-enough" transformations and data cleansing rules are applied while loading the Silver layer. Speed and agility to ingest and deliver the data in the data lake is prioritized, and a lot of project-specific complex transformations and business rules are applied while loading the data from the Silver to Gold layer. From a data modeling perspective, the Silver Layer has more 3rd-Normal Form like data models. Data Vault-like, write-performant data models can be used in this layer.

Gold layer (curated business-level tables)

Data in the **Gold layer** of the lakehouse is typically organized in consumption-ready "project-specific" databases. The Gold layer is for reporting and uses more de-normalized and read-optimized data models with fewer joins. The final layer of data transformations and data quality rules are applied here. Final presentation layer of projects such as Customer Analytics, Product Quality Analytics, Inventory Analytics, Customer Segmentation, Product Recommendations, Marking/Sales Analytics etc. fit in this layer. We see a lot of Kimball style star schema-based data models or Inmon style Data marts fit in this Gold Layer of the lakehouse.

So you can see that the data is curated as it moves through the different layers of a lakehouse. In some cases, we also see that lot of Data Marts and EDWs from the traditional RDBMS technology stack are ingested into the lakehouse, so that for the first time Enterprises can do "pan-EDW" advanced analytics and ML - which was just not possible or too cost prohibitive to do on a traditional stack. (e.g. IoT/Manufacturing data is tied with Sales and Marketing data for defect analysis or health care genomics, EMR/HL7 clinical data markets are tied with financial claims data to create a Healthcare Data Lake for timely and improved patient care analytics.)

What is an ACID Transaction? 🔥

An **ACID transaction** ensures that database operations are **reliable, consistent, and fault-tolerant**. ACID stands for:

1️⃣**A - Atomicity** 💣

- "All or nothing" rule: Either the entire transaction completes successfully, or nothing happens.
- If one part of the transaction fails, the whole operation is rolled back.
- Example: In banking, if you transfer money, **both debit & credit must happen together**—or neither happens.

2️⃣C - Consistency ✅

- The database must move from **one valid state to another**.
- No corrupted or invalid data should be stored.
- Example: If a transaction follows business rules (e.g., no negative balances), consistency ensures **the rules are never broken**.

3️⃣I - Isolation 🔀

- Multiple transactions running at the same time should **not interfere** with each other.
- Ensures transactions are independent, even under high concurrency.
- Example: If two users buy the last ticket at the same time, **only one should succeed**.

4️⃣D - Durability 💾

- Once a transaction is committed, it is **permanently saved**, even if the system crashes.
- Ensures no data loss after a successful transaction.
- Example: If you book a flight and the system crashes, **your booking should still exist after restart**.

- **ACID Transactions in a Lakehouse (Delta Lake, Iceberg, Hudi)**
- Traditional **data lakes lack ACID transactions**, making them unreliable for structured analytics.
  A **Lakehouse (using Delta Lake, Apache Iceberg, etc.) adds ACID properties** to big data storage, ensuring:
  ✅ Reliable **data updates & deletes** (important for GDPR compliance).
  ✅ **Concurrency control**, preventing dirty reads & write conflicts.
  ✅ Faster **real-time analytics**, like a traditional database.
- Would you like an example of ACID in **Databricks, Delta Lake, or SQL**? 🚀

THE MEDALLION ARCH IS FOR LAKEHOUSE. IT IS BASICALLY TRANSITION OF DATA LAKES IN LAKEHOUSE LIKE BRONZE TO SILVER- GOLD

✅ What Makes a True Lakehouse? ✅

A **Lakehouse** is more than just a **data lake with SQL**—it must combine the **scalability of a data lake** with the **governance and performance of a data warehouse**.

🔥 Key Features of a True Lakehouse

1️⃣ Open & Scalable Storage (Like a Data Lake)

✔ Stores **structured, semi-structured, and unstructured** data (JSON, Parquet, CSV, images, etc.).
✔ Uses **cheap cloud storage** (S3, ADLS, GCS) instead of expensive warehouse storage.
✔ Schema flexibility → Can **handle raw, unstructured data** without strict schemas.

2️⃣ ACID Transactions (Like a Data Warehouse)

✔ **Ensures data consistency & reliability** (Insert, Update, Delete safely).
✔ **Prevents data corruption & ensures rollback** if something fails.
✔ Unlike traditional data lakes, a **Lakehouse supports transactions** like a database.

3️⃣ Schema Enforcement & Governance

✔ Unlike data lakes (**schema-on-read**), a Lakehouse supports **schema-on-write**.
✔ Ensures **quality, validation, and versioning** of data as it moves from raw → refined.
✔ **Role-based access control (RBAC)** for security and compliance (GDPR, HIPAA).

4️⃣ Performance & Indexing (Like a Warehouse)

✔ Uses **caching, indexing, and optimized storage** to speed up queries.
✔ Unlike raw data lakes (which are slow), a Lakehouse supports **fast analytics**.
✔ Supports **BI tools, dashboards, and ML workloads** (Databricks SQL, Snowflake, etc.).

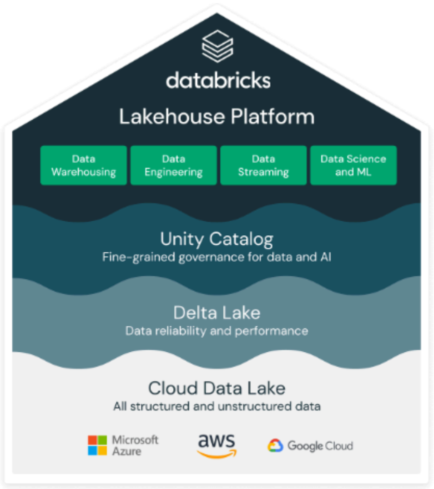
5️⃣ Supports BI, AI, & ML in One Place

✔ Unlike a traditional warehouse (only for structured data), a Lakehouse supports:

- Business Intelligence (**BI**) with SQL.
- Machine Learning (**ML**) with notebooks and Python.
- Real-time analytics & streaming.
  ✔ Removes the need for **separate Data Lake + Warehouse**, simplifying architecture.

A quick primer on lakehouses

A **lakehouse** is a data platform architecture paradigm that combines the best features of data lakes and data warehouses. A modern lakehouse is a highly scalable and performant data platform hosting both raw and prepared data sets for quick business consumption and to drive advanced business insights and decisions. It breaks data silos and allows seamless, secure data access to authorized users across the enterprise on one platform.



1️⃣ ACID Compliance: Handling Refunds & Order Updates

❌ **Data Lake Approach (S3-based Storage)**

Situation:

A customer orders a **laptop (Order ID: 12345)** but cancels it **5 minutes later**.

- The order data is stored in S3 as a raw JSON file.
- The refund system queries the order database **while a batch job is still processing cancellations.**
- Since S3 does **not support transactions**, some queries return **"Order 12345 exists"** while others return **"Order 12345 not found"** at the same time!

Impact:

- The **finance team processes a refund twice**, leading to a financial error.
- Analysts get **incorrect revenue reports**, showing the laptop sale even though it was refunded.

✅ **Lakehouse Approach (Delta Lake / Apache Iceberg)**

Situation:

With **Lakehouse**, the data is stored in **Delta Lake on top of S3.**

- **ACID transactions ensure** that updates are atomic.
- A rollback mechanism prevents double refunds.
- The refund system sees a **consistent** view of the order data at any time.

Impact:

- No financial errors.
- Analysts get the **correct revenue reports**.
- Customer support sees **only valid, up-to-date order details**.

2️⃣ Schema Enforcement: Handling Product Catalog Changes

❌ Data Lake Approach (Schema-on-Read)

Situation:
The product catalog team updates the **product schema**, adding a new field:

- Old schema: ProductID, Name, Price, Stock
- New schema: ProductID, Name, Price, Stock, Discount

Since the **schema is applied only when queried**, analysts running reports get **mixed data formats**:

- Some products **show the new "Discount" column**, others do not.
- Reports fail due to **missing columns** or **data type mismatches**.

Impact:

- Business reports show **incomplete discounts**.
- The analytics team needs to **write custom scripts** to handle schema mismatches.

✅ Lakehouse Approach (Schema-on-Write)

Situation:
With a **Lakehouse**, schema changes are **validated before ingestion**.

- If a new "Discount" column is added, it must follow **predefined data types**.
- If old data is missing this field, it gets **a default value (e.g., NULL or 0%)**.
- Queries always return a **consistent schema**.

Impact:

- No broken reports.
- The BI team **trusts the data**.
- New product features **launch faster** since the data is always clean.

3️⃣ Data Governance: Access Control on Customer Data

❌ Data Lake Approach (Basic IAM Policies)

Situation:
The **marketing** and **finance** teams both need customer purchase data.

- **Marketing:** Needs **email + purchase history** to send personalized ads.
- **Finance:** Needs **purchase + payment details** for revenue tracking.

With **S3-based Data Lake**, access control is **broad**:

- Either **everyone gets access** or **no one gets access**.
- If marketing gets access, they **also see payment details → Privacy risk!**
- If finance gets access, they **also see customer emails → Unnecessary exposure!**

Impact:

- Increased **data leaks & compliance risks (GDPR, PCI DSS violations)**.
- Engineers **manually filter data**, increasing workload.

✅ Lakehouse Approach (Fine-Grained Access Control)

Situation:
With **Lakehouse (Databricks Unity Catalog or AWS Lake Formation)**:

- **Marketing** gets access to CustomerEmail, PurchaseHistory but **not payment details**.
- **Finance** gets access to PurchaseAmount, PaymentMethod but **not customer emails**.
- Policies are **enforced at the table level**, so queries always return **only authorized data**.

Impact:

- No **data privacy violations**.
- Compliance with **GDPR & PCI DSS** is maintained.
- No need for **manual filtering**, reducing engineering overhead.

4️⃣ Query Performance: Analyzing Website Click Data

❌ Data Lake Approach (Slow Queries on S3)

Situation:
The company tracks **website clicks** to analyze which products are trending.

- Data is stored as **raw logs in S3 (JSON format)**.
- Analysts use **Athena (SQL on S3)** to query click data.
- Since S3 lacks **query optimization**, even **simple reports take 20+ minutes**.

Impact:

- **Delayed insights**, making real-time product recommendations impossible.
- Engineers try **manually partitioning data**, increasing complexity.

✅ Lakehouse Approach (Optimized Queries)

Situation:

With **Lakehouse (Delta Lake / Iceberg / Hudi)**:

- Data is **stored in Parquet format with indexing**.
- Queries **automatically use partition pruning**.
- The same clickstream analysis **now runs in seconds instead of minutes**.

Impact:

- **Real-time recommendations** are possible.
- No manual indexing or tuning required.
- Analysts can explore data interactively **without waiting for hours**.

Final Summary: When to Use What?

Feature	**Data Lake (S3, HDFS, GCS)**	**Lakehouse (Delta, Iceberg, Hudi)**
ACID Transactions	❌ No, inconsistent updates	✅ Yes, prevents data corruption
Schema Handling	❌ Schema-on-Read, error-prone	✅ Schema-on-Write, enforced
Data Governance	❌ Basic IAM & Bucket Policies	✅ Fine-grained access control
Performance	❌ Slow queries (S3 scans everything)	✅ Optimized (Indexing & Caching)
Use Case	✅ Cheap storage, ML pipelines	✅ Structured analytics, BI

Which One Should You Use?

- **If you just need a large, scalable storage solution for ML & batch processing → Use a Data Lake.**

- **If you need structured querying, ACID transactions, and governance → Use a Lakehouse.**

💡 **For enterprises requiring both raw storage & analytics, a hybrid approach (Data Lake + Lakehouse) works best.** 🚀

Here's how to impose **table-level policies using the GUI (Console) interface** in **AWS Lake Formation, Databricks Unity Catalog, and Google BigQuery**.

1️⃣ AWS Lake Formation (Table-Level & Column-Level Policies)

Steps to Restrict Access in AWS Lake Formation

✅ **Use Case:** Restrict the payment_method column in the transactions table.

1. **Go to AWS Lake Formation Console**
   - Open [AWS Lake Formation]()
   - Navigate to **Data Catalog → Databases**
   - Click on the database containing your table (transactions).
2. **Manage Table Permissions**
   - Go to **Tables** → Click on transactions.
   - Click **Actions** → Select **Grant**.
3. **Grant Specific Permissions to Roles**
   - Under **IAM users and roles**, select:
     - **Finance Role → Full Table Access**
     - **Marketing Role → Select Specific Columns**
   - Uncheck payment_method for **marketing role**.
   - Click **Grant**.

✅ Now, the **Marketing team can only see selected columns**, while **Finance has full access**.

2️⃣ Databricks Unity Catalog (Table & Column Access)

Steps to Restrict Access in Databricks

✅ **Use Case:** Allow only the **finance team** to access the full table, while **marketing sees only a restricted view**.

1. **Go to Databricks Console**
   - Open [Databricks]() and navigate to **Data Explorer**.
   - Click on **Unity Catalog → Databases → sales schema**.
2. **Set Table-Level Permissions**
   - Click on the transactions table.

   - Go to **Permissions**.
   - Click **Grant**.
   - Select:
     - **Finance Role → Full Access**
     - **Marketing Role → Custom Access →** Uncheck payment_method.
3. **Create a Restricted View for Marketing**
   - Go to **SQL Editor** and create a view:

sql
CopyEdit
```
CREATE VIEW sales.transactions_marketing AS
SELECT user_id, order_id, purchase_amount
FROM sales.transactions;
```

   - Go back to **Data Explorer → transactions_marketing → Grant Access**.
   - Assign the **Marketing Role** to this view only.

✅ **Marketing team can now access only the safe view** without payment details.

✅ **Steps to Enable ACID Transactions in an S3 Data Lake**

Option 1: Apache Iceberg (Recommended for AWS)

🚀 **Best for AWS Lake Formation + Athena + Redshift Spectrum**

🔹 **Step 1: Convert Your S3 Data to Iceberg Tables**

1️⃣ **Register S3 Bucket in AWS Glue & Lake Formation**

2️⃣ **Create an Iceberg Table**

sql
CopyEdit
```
CREATE TABLE lakehouse.transactions (
    id STRING,
    amount DOUBLE,
    status STRING
)
USING iceberg
LOCATION 's3://company-data/lakehouse/';
```

3️⃣ **Enable ACID Transactions in Athena**

sql
CopyEdit
```
ALTER TABLE lakehouse.transactions SET TBLPROPERTIES
('format-version'='2');
```

🔹 **Step 2: Perform ACID Transactions**

- **INSERT (Safe Write)**

sql

CopyEdit

```
INSERT INTO lakehouse.transactions VALUES ('txn_1', 100.0,
'pending');
```

- **UPDATE (No conflicts)**

sql
CopyEdit

```
UPDATE lakehouse.transactions
SET status = 'completed'
WHERE id = 'txn_1';
```

- **DELETE (Transactional Delete)**

sql
CopyEdit

```
DELETE FROM lakehouse.transactions
WHERE status = 'failed';
```

- **TIME TRAVEL (See past data)**

sql
CopyEdit

```
SELECT * FROM lakehouse.transactions FOR SYSTEM_TIME AS OF
TIMESTAMP '2024-03-15 10:00:00';
```

✅ **Now, Iceberg ensures your S3-based table has full ACID guarantees!**