🔹 Key Fabric Workloads

- **Data Factory** → ETL pipelines (replacement for Azure Data Factory inside Fabric).
- **Synapse Data Engineering** → Spark-based big data engineering.
- **Synapse Data Warehouse** → Cloud-scale SQL warehouse (replacement for Synapse SQL Pools).
- **Real-Time Analytics (KQL DB)** → For streaming/log data (replacement for Azure Data Explorer).
- **Data Science** → ML, AI experimentation.
- **Power BI** → Visualization & dashboards.
- **OneLake** → Universal data lake (replacement for Azure Data Lake).

Azure Data Factory (ADF) → becomes Fabric Data Factory.

Azure Synapse Analytics (SQL Pools + Spark Pools) → becomes Fabric Synapse (Data Warehouse + Data Engineering).

Azure Data Factory = full standalone ETL/ELT orchestration service.

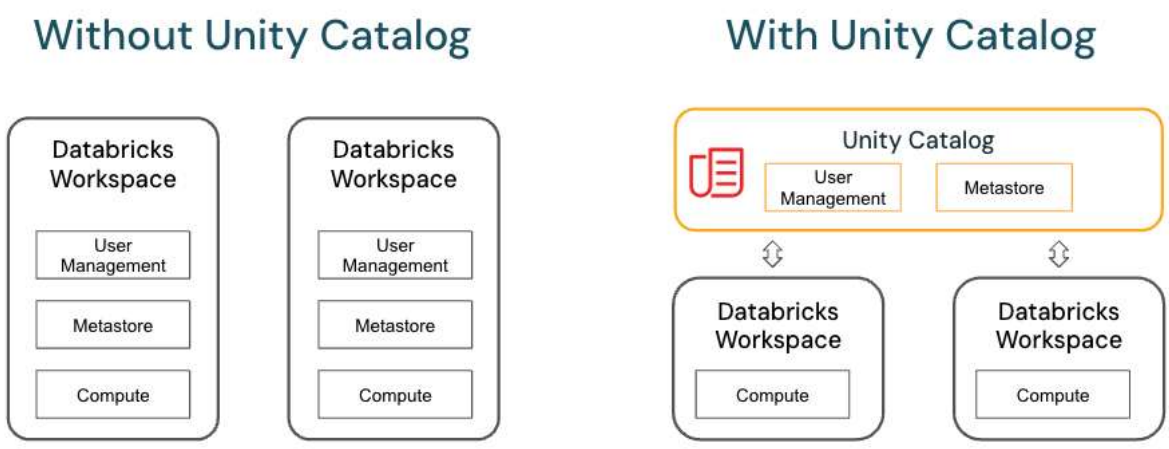
Synapse Analytics = warehouse + big data + a lightweight version of ADF (pipelines only).

Azure Synapse Analytics = Azure Data Warehouse (SQL Pools) + Azure Data Engineering (Spark Pools) + Synapse Pipelines (subset of ADF)

Unity Catalog (UC)

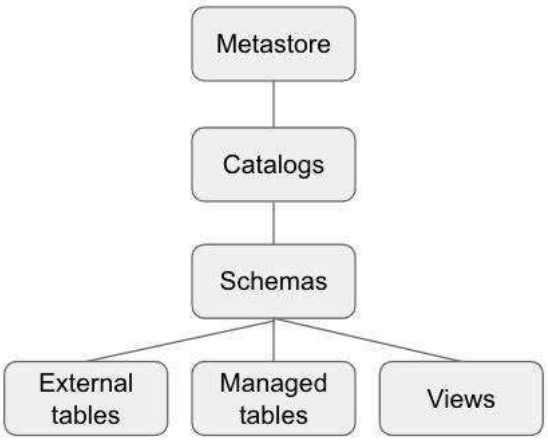
Unity Catalog is **Databricks' centralized governance and data catalog solution**. Think of it as the **"single place to manage all your data assets, permissions, and metadata"** across Databricks and connected systems.

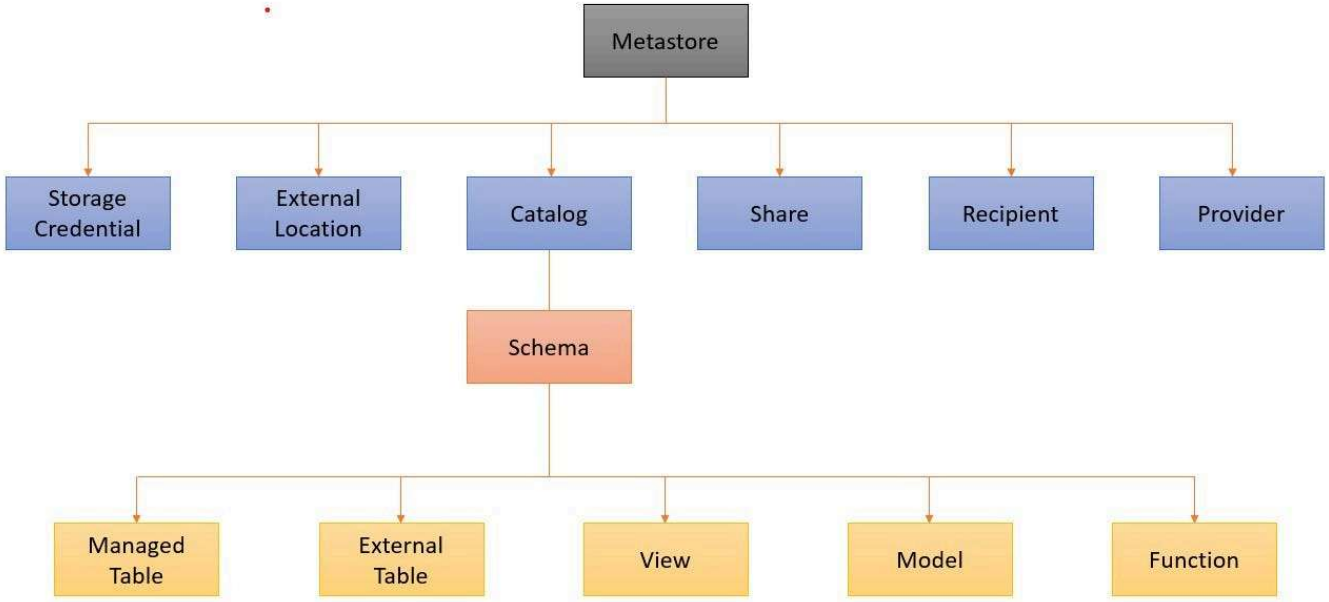
Start coding or generate with AI.



Start coding or generate with AI.

Start coding or generate with AI.





Got it 👍 — you're asking about **Unity Catalog Object Models**. Let me break this down clearly:

Unity Catalog Object Model

Unity Catalog organizes and governs data assets in a **hierarchical object model**. The hierarchy looks like this:

```
Metastore → Catalog → Schema → Table/View/Other Objects
```

🔹 1. **Metastore**

- **Top-level container** for Unity Catalog.
- A **metastore** is tied to a cloud region (e.g., `eastus`, `us-west-2`).
- Holds multiple **catalogs**.
- Each Databricks account can have **one metastore per region**.
- Example: *"Azure East US Metastore"*.

🔹 2. **Catalog**

- A **top-level container for schemas**.
- Think of it like a **database cluster**.
- Used to logically group business domains, e.g.:

    - `sales`
    - `finance`
    - `marketing`
- Example: `sales` catalog contains schemas like `us_east`, `us_west`.

🔹 3. Schema

- A **namespace inside a catalog** (like schemas in SQL databases).
- Contains tables, views, and functions.
- Example:
    - `sales.us_east.orders`
    - Here: `sales` = catalog, `us_east` = schema, `orders` = table.

🔹 4. Data Objects

Inside schemas, you have different **object types**:

- **Tables**
    - Managed or external Delta tables.
    - Example: `sales.us_east.orders`.
- **Views**
    - Logical representations of data (SQL views).
- **Volumes**
    - Storage locations for **unstructured data** (images, PDFs, Parquet files).
- **Functions**
    - User-defined functions (UDFs).
- **Models** *(integration with MLflow)*
    - Machine learning models can also be governed.

🔹 5. Privileges / Permissions

- Access is controlled using **ANSI SQL GRANTs**.
- Example:

```
GRANT SELECT ON TABLE sales.us_east.orders TO user bob;
```

Permissions can be applied at any level:

- Catalog
- Schema
- Table/View/Volume

🖼️ Visual Representation

```
Metastore
 └── Catalog (e.g., sales)
      ├── Schema (e.g., us_east)
      │    ├── Table (orders)
      │    ├── View (top_customers)
      │    └── Volume (images)
      └── Schema (e.g., us_west)
           ├── Table (orders)
           └── Table (returns)
```

✅ **In summary**:

- **Metastore** = root container

- **Catalog** = groups schemas (business domains)
- **Schema** = groups objects (like a DB schema)
- **Objects** = tables, views, volumes, functions, models

Start coding or generate with AI.

✅ Quick Comparison Table

Feature	Managed Table	External Table
Data location	Controlled by Databricks	User specifies storage path
Metadata storage	Metastore	Metastore
Who manages data?	Databricks	You
DROP TABLE effect	Deletes both metadata + data	Deletes only metadata, data remains
Use case	Quick development, temporary data	Production data, shared storage, long-term

---

🔥 **Simple rule of thumb**:

- **Managed = Databricks takes care of everything** (easy, but deleting table deletes data).
- **External = You control where data lives** (safer for production, data persists).

Start coding or generate with AI.

Nice 👏 — you're now diving into **Unity Catalog object management scenarios**. Let's walk through each one step by step with **simple explanations + real-world analogy**.

🔹 Scenario 1: Managed Catalog, Managed Schema, Managed Table

- **Everything is managed by Databricks**.
- Catalog, Schema, and Table all live in Databricks-controlled storage.
- If you delete the table/schema/catalog, the actual **data files also get deleted**.

👉 **Analogy**: You rent a fully furnished flat → cupboards, shelves, and clothes are all provided by the landlord. If you move out, everything is gone.

Example:

```
-- Create managed catalog + schema + table
CREATE CATALOG sales;                -- managed catalog
CREATE SCHEMA sales.us_east;         -- managed schema
CREATE TABLE sales.us_east.orders (  -- managed table
    order_id INT,
    amount DOUBLE
);
```

- DROP TABLE = deletes files.
- DROP SCHEMA = deletes tables + files.
- DROP CATALOG = deletes everything.

🔹 Scenario 2: External Catalog, Managed Schema, Managed Table

- **Catalog storage location is external** (you specify ADLS/S3 path).
- Schema and Table are **managed by Databricks inside that external catalog**.
- If you delete a table, its **data files are deleted**, but only inside that catalog's external location.

👉 **Analogy**: You bring your own house (catalog = external storage), but Databricks builds cupboards (schemas) and puts clothes (tables) inside. If you throw away a cupboard (schema), all clothes inside are also gone.

Example:

```
-- External catalog with ADLS location
CREATE CATALOG sales
  MANAGED LOCATION 'abfss://bronze@9380kp.dfs.core.windows.net/sales/';

-- Managed schema + table inside it
CREATE SCHEMA sales.us_east;
CREATE TABLE sales.us_east.orders (
    order_id INT,
    amount DOUBLE
);
```

- DROP TABLE = deletes files (in catalog location).
- Catalog is external, but table is still **managed** by Databricks.

🔹 Scenario 3: External Catalog, External Schema, Managed Table

- **Catalog and Schema both have external storage locations**.
- Table is **managed** inside the schema's external storage path.
- Dropping the table still deletes the **table files**, but only in the schema's external location.

👉 **Analogy**: You have your own house (catalog = external), inside it you define a separate storage room (schema = external). Databricks puts a cupboard (table) inside that room. If you throw away the cupboard, the clothes (data) inside vanish.

Example:

```
-- External catalog
CREATE CATALOG sales
  MANAGED LOCATION 'abfss://bronze@9380kp.dfs.core.windows.net/sales/';

-- External schema
CREATE SCHEMA sales.us_east
  MANAGED LOCATION 'abfss://bronze@9380kp.dfs.core.windows.net/sales/us_east/';

-- Managed table inside external schema
CREATE TABLE sales.us_east.orders (
    order_id INT,
    amount DOUBLE
);
```

- DROP TABLE = deletes files from schema's location.
- DROP SCHEMA = deletes all tables + files under schema's location.

🔹 Scenario 4: External Catalog, External Schema, External Table

- **Full control by the user**.
- Catalog = external location
- Schema = external location
- Table = external table (user provides path to data files).
- Dropping the table deletes only **metadata — data files remain safe**.

👉 **Analogy**: You bring your own house (catalog), you also bring your own storage room (schema), and you put your own cupboard (table) pointing to your clothes (files). If you throw away the cupboard, your clothes are still there.

Example:

```
-- External catalog
CREATE CATALOG sales
  MANAGED LOCATION 'abfss://bronze@9380kp.dfs.core.windows.net/sales/';

-- External schema
CREATE SCHEMA sales.us_east
  MANAGED LOCATION 'abfss://bronze@9380kp.dfs.core.windows.net/sales/us_east/';
```

```
-- External table with explicit location
CREATE TABLE sales.us_east.orders_ext (
    order_id INT,
    amount DOUBLE
)
USING DELTA
LOCATION 'abfss://bronze@9380kp.dfs.core.windows.net/sales/us_east/orders/';
```

- DROP TABLE = deletes only metadata, data files remain.
- Perfect for **production data pipelines**, where you don't want accidental deletion of files.

✅ Summary Table

Scenario	Catalog	Schema	Table	Data on DROP TABLE
1. Managed–Managed–Managed	Managed	Managed	Managed	Files deleted
2. External–Managed–Managed	External	Managed	Managed	Files deleted (in catalog location)
3. External–External–Managed	External	External	Managed	Files deleted (in schema location)
4. External–External–External	External	External	External	Only metadata deleted (files remain)

🔥 Rule of thumb:

- **Managed Table** = deleting table deletes files.
- **External Table** = deleting table keeps files.
- Whether Catalog/Schema are managed or external only affects **where the files live**.

Do you want me to also **draw a diagram for these 4 scenarios** (like storage hierarchy visuals for Managed vs External)? That could make it super clear for interviews.

Double-click (or enter) to edit

Great question 👍 — this is where people often get confused because **Unity Catalog Volumes** sound similar to **external tables**, but they serve a different purpose. Let's go step by step:

🔹 What is a Volume in Unity Catalog (Databricks)?

- A **Volume** is a **governed storage location for files** inside Unity Catalog.
- It's used to store **unstructured or semi-structured data** (CSV, JSON, images, PDFs, Parquet, etc.).
- Provides a **secure, namespace-based path** that enforces Unity Catalog **permissions** (ACLs).
- Path format:

```
/Volumes/<catalog>/<schema>/<volume_name>/
```

👉 Think of a **volume** as a **folder** managed by Unity Catalog where you can keep any files, but with governance applied.

🔹 Types of Volumes

1. Managed Volume

- Databricks **decides the storage location** (inside catalog or schema's managed location).
- When you drop the volume, both **metadata + files** are deleted.
- Use case: **temporary or intermediate data**, when you don't need to keep files if the volume is dropped.

👉 **Analogy**: A cupboard provided by Databricks in their house. If you throw away the cupboard, all the clothes inside are gone.

Example:

```
-- Create a managed volume
CREATE VOLUME sales.us_east.temp_files;
```

Files live under:

/Volumes/sales/us_east/temp_files/

If you drop the volume:

```
DROP VOLUME sales.us_east.temp_files;
```

➡️ Metadata + files are deleted.

---

2. External Volume

- You **specify the storage path** (ADLS, S3, GCS, etc.).
- Unity Catalog only manages the **metadata**.
- If you drop the volume, only the metadata is deleted, but **files remain** in the external path.
- Use case: **production storage** or **shared data lake files** that must persist beyond Databricks.

👉 **Analogy**: A cupboard in your own house (external storage). If you throw away the cupboard in Databricks, the clothes (files) in your house are still safe.

Example:

```
-- Create an external volume
CREATE EXTERNAL VOLUME sales.us_east.raw_data
LOCATION 'abfss://bronze@9380kp.dfs.core.windows.net/sales/raw_data/';
```

Files live in your external storage (`abfss://...`). If you run:

```
DROP VOLUME sales.us_east.raw_data;
```

➡️ Metadata gone, files remain in ADLS.

🔹 Managed vs External Volumes — Quick Comparison

Feature	Managed Volume	External Volume
Storage location	Decided by Databricks (inside catalog/schema managed location)	User-specified (ADLS/S3/GCS path)
Drop effect	Deletes metadata + files	Deletes metadata only, files remain
Governance	Unity Catalog ACLs	Unity Catalog ACLs
Use cases	Temporary data, scratch space, intermediate results	Production data, shared file storage, persistent data

---

✅ Difference from External Tables

- **External Table**: Provides a **structured SQL view** (rows/columns) on top of existing files (like Delta/Parquet).
- **External Volume**: Just a **governed folder** for raw files — no table schema, no SQL query, just files.

👉 If you need **structured query access** → use **external table**. 👉 If you need **secure file storage** (raw dumps, JSON, images, etc.) → use **volume**.

✅ **In short:**

- **Managed Volume** = Databricks controls storage, files deleted on drop.
- **External Volume** = You control storage path, files remain safe if dropped.

--

```
Start coding or generate with AI.
```

Perfect 👍 — let's break down **Delta Lake** in **simple words with internal structure** so you'll understand how `delta_log + .parquet` work together.

🔹 What is Delta Lake?

Delta Lake is an **open-source storage layer** built on top of **Parquet files**. It adds:

- **ACID transactions** (like a database)
- **Time travel** (query old versions)
- **Schema enforcement & evolution**
- **Data reliability**

👉 In short: **Parquet = just files, Delta = Parquet + transaction log**

🔹 Delta Lake File Structure

Inside a Delta table folder, you'll typically see:

```
/delta_table/
   ├── part-00000-aaaa.snappy.parquet
   ├── part-00001-bbbb.snappy.parquet
   ├── part-00002-cccc.snappy.parquet
   └── _delta_log/
        ├── 00000000000000000000.json
        ├── 00000000000000000001.json
        ├── 00000000000000000002.json
        └── ...
```

🔹 Two Key Parts

1️⃣ Parquet Files

- These hold the **actual data** (rows/columns).
- Example:
  - `part-00000-*.parquet` contains a chunk of the table's rows.
- Files are immutable → once written, they are never changed (only added/removed logically).

👉 Think of **Parquet files as the "books" containing your data.**

2️⃣ _delta_log Folder

- This is the **transaction log** — the **brain of Delta Lake**.
- Contains JSON and checkpoint files:
  - **JSON files** (`00000000000000000001.json`, etc.) = each transaction (insert/update/delete).
  - **Checkpoint files** (`.parquet`) = periodic snapshots of the table state for faster reads.

👉 Think of **_delta_log as the "table of contents" or ledger** that tracks which data files are valid.

🔹 How It Works Together

1. When you **INSERT** new rows:
   - Delta writes new `.parquet` files.
   - Delta adds a new JSON entry in `_delta_log` that points to the new files.
2. When you **DELETE or UPDATE** rows:
   - Delta **doesn't change the old parquet files**.
   - Instead, it **marks them as removed** in `_delta_log` and adds new parquet files with updated data.
3. When you **READ the table**:
   - Delta looks at the **latest transaction log** in `_delta_log`.
   - It figures out which `.parquet` files are valid at that version.
   - It reads only those files → giving you a consistent view of the table.

🔹 Time Travel (Versioning)

Because all changes are tracked in `_delta_log`:

```
-- Query current version
SELECT * FROM delta.`/delta_table/`;

-- Query older version by version number
SELECT * FROM delta.`/delta_table/` VERSION AS OF 2;

-- Query older version by timestamp
SELECT * FROM delta.`/delta_table/` TIMESTAMP AS OF '2025-08-01';
```

👉 You can go "back in time" because old parquet files + logs are still there.

✅ Quick Analogy

- **Parquet files** = actual pages of a book (data).
- **_delta_log** = library ledger recording which pages/editions are valid.
- **Delta Lake** = turns raw Parquet into a "database-like" system with transactions & history.

✅ **In short: Delta Lake = Parquet files (data) + _delta_log (transactions/metadata)** That combo makes your data lake behave like a reliable database.

Would you like me to also show you a **real example of a `_delta_log/00000000000000000001.json` entry** so you see exactly what's inside?

```
Start coding or generate with AI.
```

Got it 👍 — here's the **simple definition of Delta Live Tables (DLT):**

🔹 Delta Live Tables (DLT)

Delta Live Tables is a **framework in Databricks** that helps you **build, manage, and run reliable data pipelines** easily.

Instead of writing complex ETL jobs manually, you just declare:

- **What data you want**
- **How it should be transformed**
- **Quality rules**

...and DLT takes care of:

- Creating the pipelines
- Handling dependencies
- Auto-managing retries, checkpoints, and error handling
- Ensuring data quality with **expectations**

✅ In Simple Words

👉 **Delta Live Tables = Smart ETL pipelines on Delta Lake** It's like saying:

- "Take raw data from source → clean it → store it as a Delta table" and Databricks automatically handles the execution and reliability for you.

🔹 Example

```
CREATE LIVE TABLE clean_orders
AS SELECT *
FROM raw_orders
WHERE amount > 0;
```

- `raw_orders` = raw input

- `clean_orders` = processed output
- DLT automatically makes sure `clean_orders` is always updated whenever `raw_orders` changes.

🔹 Key Benefits

1. **Declarative pipelines** → you define the "what," Databricks handles the "how."
2. **Data quality checks** → define rules like *amount > 0* (bad data is flagged).
3. **Auto lineage tracking** → see where data comes from and goes.
4. **Scalable & reliable** → handles retries, monitoring, orchestration.

✅ **One-line definition for interviews: Delta Live Tables (DLT) is a Databricks framework for building reliable, declarative, and automated ETL pipelines on Delta Lake with built-in data quality and lineage.**

Do you want me to also explain the difference between **Delta Table vs Delta Live Table** (since that's another common interview confusion)?

Start coding or generate with AI.