The **Medallion Architecture**, also known as the **Lakehouse Architecture** or **Multi-Hop Architecture**, is a data design pattern used in data lakes and data lakehouses to logically organize data into multiple layers of refinement.[1] The goal is to progressively improve the quality, structure, and value of data as it moves through the pipeline, from raw ingestion to highly aggregated and easily consumable formats.[2]

It's called "Medallion" because it reflects a progression from less valuable (raw, like unrefined ore) to more valuable (refined, like polished gold), much like a precious metal.[3]

Purpose: Why Use Medallion Architecture?

The Medallion Architecture addresses several challenges in data lake environments:

1. **Data Quality:** Ensures data improves in quality and reliability at each stage, reducing issues in final reports.[4]
2. **Flexibility:** Allows different data consumers to access data at the appropriate level of refinement.[5] Data scientists might use Silver, while business analysts use Gold.[6]
3. **Reproducibility & Auditability:** Provides clear stages, making it easier to re-process data from any point in the pipeline and track its lineage.[7]
4. **Performance:** Optimized data formats and aggregations in later layers lead to faster query performance for analytical workloads.[8]
5. **Cost Efficiency:** Raw data can be stored cheaply, and only necessary subsets are refined, optimizing compute and storage costs.[9]
6. **Separation of Concerns:** Clearly defines responsibilities for different data engineering tasks (ingestion, cleaning, aggregation).[10]

The Three Layers of the Medallion Architecture:

The architecture typically consists of three main layers, often referred to as "zones" or "tiers":[11]

1. Bronze Layer (Raw Zone)[12]

- **Characteristics:**
  - **Immutability:** Stores raw, unaltered data exactly as it was ingested from the source systems.[13]
  - **Append-Only:** Data is typically appended, not updated in place.[14] If a source record changes, a new version is appended.
  - **Schema-on-Read (often):** Data might be semi-structured (JSON, CSV) or even unstructured.[15] Schema is applied when data is read.
  - **High Volume:** Contains all historical data, often very granular.[16]
- **Purpose:**
  - Serves as a complete, unchangeable historical record of all ingested data, providing a single source of truth for raw data.[17]
  - Enables full reprocessing if downstream logic changes or errors are discovered.
  - Acts as a disaster recovery point for source system data.
- **Example:** Raw JSON files from IoT devices, CSV files directly from an SFTP server, full database dumps.[18]

2. Silver Layer (Curated / Refined Zone)[19]

- **Characteristics:**
  - **Cleaned & Conformed:** Data has undergone initial cleaning, parsing, standardization, and de-duplication.[20]
  - **Structured & Typed:** Data is often converted into structured formats (e.g., Parquet, Delta Lake) with defined schemas and data types.
  - **Unified View:** Data from multiple sources might be joined or merged to create a more comprehensive view (e.g., device data joined with device metadata).[21]
  - **Historical (often):** Still maintains a historical record, but of the cleaned and enriched data.
- **Purpose:**

    - Provides a reliable, consistent, and structured view of the data, ready for broader consumption by data analysts, data scientists, and reporting tools.[22]
    - Facilitates exploratory data analysis and feature engineering.
    - Serves as a foundation for building the Gold layer.
- **Example:** Parquet files of cleaned IoT sensor readings with correct timestamps and numeric values, customer order data joined with product details.

3. Gold Layer (Aggregated / Consumption Zone)

- **Characteristics:**
    - **Highly Refined & Aggregated:** Data is transformed into specific data models optimized for business intelligence, reporting, and machine learning.[23]
    - **Business-Oriented:** Data is structured into star schemas, data marts, or specialized aggregates that directly answer business questions.[24]
    - **Performance Optimized:** Designed for rapid query execution, often using columnar formats or specialized databases.
    - **Subset of Data:** Contains only the data relevant for specific analytical use cases, often summarized or aggregated.
- **Purpose:**
    - Serves business users, analysts, and ML engineers directly for reporting, dashboards, and advanced analytics.
    - Provides high-performance access to business-ready data.
    - Encapsulates complex transformations behind simple, queryable tables.[25]
- **Example:** Daily sales aggregates per region, monthly customer churn rates, pre-computed KPIs (like average device uptime per site), tables for specific ML model training.

Benefits of the Medallion Architecture:

- **Improved Data Quality:** Each layer adds value and cleans data, reducing "garbage in, garbage out."[26]
- **Enhanced Data Governance:** Clear separation of responsibilities and data states.
- **Better Performance:** Optimized data structures in Silver and Gold layers accelerate queries.[27]
- **Cost Efficiency:** Raw data stored cheaply; only refined data requires more expensive processing.[28]
- **Self-Service Analytics:** Business users can confidently access Gold-layer data.[29]
- **Flexibility & Agility:** Changes in source systems don't necessarily break downstream consumers, and new use cases can start from appropriate layers.[30]
- **Reproducibility:** Ability to re-process data from any layer in case of errors or evolving requirements.

Implementation Considerations:

The Medallion Architecture is commonly implemented using technologies like:

- **Storage:** Cloud object storage (S3, ADLS, GCS) for the data lake, potentially with HDFS for on-premise or hybrid setups. (Amazon simple storage service, Azure Data Lake Storage Gen2, Google Cloud Storage)
- **File Formats:** Parquet, Avro, ORC for structured and columnar storage.
- **Table Formats:** Delta Lake, Apache Iceberg, Apache Hudi are increasingly used to add transactional capabilities (ACID properties, schema evolution, time travel) to data lake layers, effectively turning the data lake into a "lakehouse."[31]
- **Processing Engines:** Apache Spark (highly versatile for all layers), Apache Flink (for streaming transformations), dbt (for SQL-based transformations, especially Silver to Gold).[32]
- **Databases:** For the Gold layer, a traditional RDBMS like TimescaleDB (for time-series), PostgreSQL, or a data warehouse (Snowflake, BigQuery, Redshift) can be used to serve dashboard queries.

By systematically refining data through these distinct layers, organizations can build robust, scalable, and high-performing data platforms that deliver reliable insights.

==

In data engineering, especially within big data ecosystems like Hadoop and cloud data lakes, **Parquet**, **Avro**, and **ORC** are specialized file formats designed to address the limitations of simpler formats like CSV or JSON when dealing with massive datasets.[1] They offer significant advantages in terms of storage efficiency, query performance, and schema management.[2]

Let's break down each one:

1. Apache Parquet

- **Type: Columnar storage format.** This is the defining characteristic. Instead of storing data row by row (like a spreadsheet or CSV), Parquet stores data column by column.[3] So, all values for Column A are stored together, then all values for Column B, and so on.[4]
- **Key Features & Advantages:**
  - **Efficient Compression:** Since data within a column is typically of the same data type and often has similar values, columnar storage allows for much better compression ratios using various encoding schemes (e.g., Run-Length Encoding, Dictionary Encoding) and compression algorithms (Snappy, Gzip, Zstandard).[5] This significantly reduces storage costs.
  - **Predicate Pushdown/Column Pruning:** When you query only a few columns from a very wide table, the query engine only needs to read the relevant columns from storage, skipping over the data in columns you don't need.[6] This drastically reduces I/O operations and speeds up queries, especially analytical queries.[7]
  - **Optimized for Analytics (OLAP):** Its columnar nature makes it highly efficient for analytical queries (e.g., SUM, AVG, GROUP BY) that often operate on a subset of columns across many rows.[8]
  - **Schema Evolution:** Parquet supports schema evolution, meaning you can add new columns, rename columns, or change data types (within limits) without breaking existing files or requiring a full rewrite of historical data.[9] Metadata about the schema is stored in the file footer.[10]
  - **Splittable:** Parquet files are splittable, allowing them to be processed in parallel across multiple nodes in distributed computing frameworks like Apache Spark and Hadoop MapReduce.[11]
  - **Self-Describing:** Each Parquet file contains metadata about its schema, compression settings, and other details in its footer, making it self-contained.[12]
- **Best Use Case:** Ideal for **read-heavy analytical workloads**, data warehousing, ETL processes, and machine learning pipelines where you often analyze specific attributes across large datasets.[13] Widely adopted in data lakes.

2. Apache Avro

- **Type: Row-oriented data serialization format.** Avro stores data record by record (row by row).
- **Key Features & Advantages:**
  - **Schema-based Serialization:** Avro relies heavily on schemas (defined in JSON) for both writing and reading data. The schema is stored directly with the data (in a file header or a message).
  - **Strong Schema Evolution:** This is Avro's standout feature.[14] Because the schema is embedded or explicitly known by both writer and reader, Avro handles schema changes (like adding new fields with default values, removing fields, or reordering fields) very robustly.[15] Old readers can read new data, and new readers can read old data without issues.[16] This is excellent for long-lived data streams where schemas evolve over time.
  - **Compact Binary Format:** While the schema is JSON (human-readable), the data itself is stored in a compact binary format, making it efficient for storage and fast for serialization/deserialization.[17]
  - **Language Agnostic:** Avro includes APIs for many programming languages, facilitating data exchange between programs written in different languages.
  - **Splittable:** Avro files also include markers that allow them to be split and processed in parallel.[18]
- **Best Use Case:** Ideal for **write-heavy operations**, real-time data streaming, messaging systems, RPC (Remote Procedure Call), and scenarios where robust **schema evolution** is a primary concern (e.g., Kafka topics, log storage, inter-service communication).

3. Apache ORC (Optimized Row Columnar)[19]

- **Type: Columnar storage format.** Like Parquet, ORC organizes data in columns.[20]
- **Key Features & Advantages:**
  - **Optimized for Hive:** ORC was initially developed by Hortonworks specifically to improve the performance of Apache Hive queries and reduce storage footprint in Hadoop.
  - **Superior Compression:** ORC often achieves slightly higher compression ratios than Parquet due to its design, which includes different internal indexing and encoding strategies.
  - **Predicate Pushdown & Indexing:** Similar to Parquet, ORC supports predicate pushdown.[21] It also builds

lightweight internal indexes (within "stripes" of data) that allow query engines to efficiently skip entire sets of rows if they don't contain the queried data.[22] This can significantly speed up queries.
    - **Complex Type Support:** Supports a wide range of complex data types, including structs, lists, maps, and unions.[23]
    - **Self-Describing & Splittable:** Like Parquet, ORC files are self-describing (metadata in the footer) and splittable for parallel processing.[24]
- **Best Use Case:** Excellent for **Hadoop-native batch processing**, especially within Apache Hive and Spark environments, where high compression, efficient query performance (particularly for aggregations), and robust data warehousing capabilities are needed.[25] It often performs very well for mixed read/write workloads in the Hadoop ecosystem.

Comparison Summary:

Feature	**Apache Parquet**	**Apache Avro**	**Apache ORC**
Storage Model	**Columnar** (stores data by column)	**Row-oriented** (stores data by row)	**Columnar** (stores data by column)
Best For	**Analytics, OLAP, Read-heavy workloads**	**Serialization, Streaming, Write-heavy workloads**	**Hadoop/Hive, Batch Processing, Mixed workloads**
Compression	Excellent (high ratios via column-specific encoding)	Moderate (compact binary, but less than columnar)	Excellent (often superior to Parquet)
Query Performance	Very fast (column pruning, predicate pushdown)	Slower for analytical queries (must read entire row)	Very fast (column pruning, robust indexing)
Schema Definition	Self-describing (metadata in file footer)	JSON-defined schema (embedded with data or referenced)	Self-describing (metadata in file footer)
Schema Evolution	Good (add/rename columns, some type changes)	**Excellent** (designed for robust evolution)	Good (add/rename columns, some type changes)
Readability	Not human-readable (binary)	Not human-readable (binary, schema is JSON)	Not human-readable (binary)
Splittable	Yes	Yes	Yes

The choice between Parquet, Avro, and ORC depends heavily on the specific use case and workload characteristics within your data pipeline. Often, you'll find a combination of these formats used across different layers of a data lake, like the Medallion Architecture, to optimize for different stages of data processing and consumption.

==

Delta Lake, Apache Iceberg, and Apache Hudi are open-source **table formats** that bring crucial database-like capabilities, specifically **ACID properties, schema evolution, and time travel**, to data stored in data lakes (typically in formats like Parquet, Avro, or ORC on cloud object storage like S3, ADLS, or GCS, or HDFS).

Traditional data lakes, while excellent for storing massive amounts of raw, diverse data cost-effectively, lacked these transactional guarantees. This made operations like updates, deletes, and concurrent writes complex, error-prone, or very inefficient, often leading to data inconsistencies. These table formats essentially add a **metadata layer and a transaction log** on top of the underlying data files to enable these advanced features.

Here's how they add these capabilities:

1. ACID Properties (Atomicity, Consistency, Isolation, Durability)

These table formats achieve ACID properties primarily through a **transaction log** (or metadata management system) and **optimistic concurrency control**.

- **Atomicity:**
  - **How:** All changes to a table (e.g., adding new data files, marking old files for deletion, updating metadata) are bundled into a single, atomic transaction.
  - **Mechanism:** When a write operation occurs, the new data files are first written to a temporary location. Only after all new data is successfully written and the transaction log entry is prepared is a single, atomic commit operation performed (e.g., updating a pointer in a catalog or writing a new transaction log entry). If any part of the process fails before the final commit, none of the changes are visible, and the table remains in its previous consistent state.
  - **Benefit:** Ensures that operations either complete entirely or are fully rolled back. Prevents partial or corrupted writes.
- **Consistency:**
  - **How:** Transactions transition the table from one valid state to another, adhering to defined rules and constraints.
  - **Mechanism:** The transaction log records the sequence of operations. Each new commit (snapshot) represents a consistent state of the table. These formats also often include features like **schema enforcement** (see below) to prevent writes that would introduce incompatible data, thus maintaining the table's structural consistency. Optimistic concurrency control (see Isolation) ensures that only valid, non-conflicting changes are committed.
  - **Benefit:** Guarantees that data integrity is maintained throughout the transaction lifecycle, preventing illogical or corrupt states.
- **Isolation:**
  - **How:** Concurrent read and write operations on the same table do not interfere with each other. Each transaction operates as if it were the only one happening.
  - **Mechanism:** These formats use **optimistic concurrency control**.
    1. **Read Phase:** A writer first reads the current state of the table (a specific snapshot) to determine which files need to be modified.
    2. **Write Phase:** The writer then creates new data files (for inserts, updates, or deletes) in a temporary location. Existing data files are never modified in place; instead, new versions are written.
    3. **Commit Phase:** The writer attempts to commit its changes by atomically updating the transaction log or metadata. Before committing, it checks if any other writer has committed changes to the same parts of the table *since* its initial read phase.
    4. **Conflict Resolution:** If there's no conflict, the commit succeeds, and a new snapshot is created. If there's a conflict (e.g., another writer modified the same data concurrently), one of the transactions fails (typically with a "concurrent modification exception") and is retried or rolled back.
  - **Benefit:** Guarantees that readers always see a consistent snapshot of the data, and concurrent writers don't corrupt each other's changes.
- **Durability:**
  - **How:** Once a transaction is committed, its changes are permanent and survive system failures (e.g., power outages, node crashes).
  - **Mechanism:** The actual data files (Parquet, ORC, Avro) are written to durable, fault-tolerant object storage (S3, ADLS, GCS) or HDFS, which inherently provides redundancy. The transaction log itself is also stored durably and redundantly in the same underlying storage. Once a commit record is successfully written to the log, the changes are

considered durable.
    - **Benefit:** Ensures that committed data is never lost, even in the face of hardware or software failures.

2. Schema Evolution

Schema evolution is the ability to change a table's schema (e.g., add columns, drop columns, rename columns, change column types) over time without rewriting the entire dataset or breaking existing queries.

- **How:** These formats manage schema versions as part of their metadata. When you make a schema change, a new schema version is recorded in the transaction log.
- **Mechanism:**
    - **Adding Columns:** New columns can be added with default values (or nulls for existing data). Old readers will simply ignore the new columns, while new readers will see the new columns (with nulls for old data).
    - **Renaming Columns:** The format often tracks column IDs or unique identifiers, allowing logical renaming without changing underlying data.
    - **Dropping Columns:** Columns can be logically dropped. Old data files containing the dropped column remain, but newer reads will ignore them.
    - **Type Changes:** Some compatible type changes (e.g., int to long) are typically supported.
    - **Schema Enforcement:** Most also offer "schema enforcement," which by default prevents writes if the incoming data's schema doesn't match the table's current schema, catching errors early. This can be overridden for schema evolution where new columns are intended.
- **Benefit:** Critical for agile data development. Data schemas evolve over time as business requirements change or new data sources are integrated. Without schema evolution, making changes would require costly and time-consuming full table rewrites or complex data migration processes.

3. Time Travel (Data Versioning)

Time travel (or data versioning) allows users to query previous versions of a table's data, or even roll back a table to an earlier state.

- **How:** The transaction log is the core of time travel. Every write operation (insert, update, delete, schema change) creates a new immutable snapshot of the table.
- **Mechanism:** The transaction log maintains a linear history of all table snapshots. Each snapshot points to a specific set of underlying data files that constitute the table's state at that point in time.
    - **Point-in-Time Queries:** Users can specify a timestamp or a version number to query the table "as of" that point in the past. The query engine reads the transaction log to identify the exact set of data files that were part of the table's snapshot at the requested time.
    - **Rollback/Rollforward:** If a bad write or erroneous operation occurs, the table can be reverted to a previous known good state by simply moving the current pointer in the transaction log to an earlier snapshot. This is a metadata operation, not a data copy, making it very fast.
    - **Auditability & Reproducibility:** Provides a full audit trail of all changes to the data, essential for compliance, debugging data pipelines, and reproducing analytical results.
- **Benefit:** Invaluable for data recovery (undoing accidental deletes or bad writes), auditing, debugging data quality issues, and machine learning model training (using consistent historical datasets).

How Delta Lake, Apache Iceberg, and Apache Hudi Implement These:

All three formats employ these core concepts, but their specific implementations and optimizations differ:

- **Delta Lake:**
    - **Core:** Uses a transaction log (_delta_log directory) composed of JSON files (for individual commits) and Parquet files (for checkpoints of the log history).
    - **ACID:** Optimistic concurrency ensures isolation; transaction log ensures atomicity, consistency, and durability.
    - **Time Travel:** Query AS OF VERSION or AS OF TIMESTAMP.
    - **Schema Evolution:** Supports mergeSchema option and autoMerge for flexible schema evolution. Also has schema enforcement by default.
    - **Strengths:** Deeply integrated with Apache Spark, offering high performance for Spark workloads.
- **Apache Iceberg:**

    - **Core:** Uses a metadata-driven approach with a hierarchy of metadata files: metadata files point to manifest lists, which point to manifest files, which in turn list the actual data files (Parquet, ORC, Avro).
    - **ACID:** Achieves atomicity through atomic updates to the table's current metadata file pointer in a catalog. Optimistic concurrency for isolation.
    - **Time Travel:** Stores a history of metadata files (snapshots) allowing queries FOR VERSION AS OF or AT TIMESTAMP.
    - **Schema Evolution:** Robust support for adding, dropping, renaming, and reordering columns without rewriting data.
    - **Hidden Partitioning:** A unique feature where partitioning logic (e.g., days(ts), hour(ts)) is stored in metadata, allowing partition evolution without changing existing queries or data files, making partition management transparent to users.
    - **Strengths:** Designed to be engine-agnostic (works well with Spark, Flink, Trino, Presto, Impala, Dremio), highly flexible partitioning, and strong consistency guarantees.
- **Apache Hudi:**
    - **Core:** Organizes data into a timeline of actions (commits, compactions, cleans) and maintains two file types: base files (Parquet/ORC, representing the latest state) and log files (tracking incremental changes). It supports different table types:
        - **Copy On Write (CoW):** Reads always see the latest committed base file. Updates/deletes cause entire data files to be rewritten. Best for read-heavy workloads.
        - **Merge On Read (MoR):** Base files (columnar) are combined with log files (row-oriented) at read time to get the latest state. Updates are written to log files first, then asynchronously compacted into base files. Better for write-heavy, near real-time ingestion.
    - **ACID:** Achieves atomicity through committing operations to the timeline. Consistency is maintained through the timeline and indexing. Isolation depends on the table type (CoW offers stronger read isolation).
    - **Time Travel:** The timeline allows querying specific instants (timestamps or commit IDs) for historical data.
    - **Schema Evolution:** Supports schema changes with flexible handling of added/removed columns.
    - **Incremental Processing:** Designed for efficient incremental reads and writes, making it suitable for CDC (Change Data Capture) and stream processing.
    - **Strengths:** Originates from Uber's need for highly efficient upserts and incremental processing for very large datasets, making it strong for data ingestion and streaming use cases.

These table formats are foundational to the **Lakehouse Architecture**, bridging the gap between traditional data lakes and data warehouses by bringing transactional reliability and data management capabilities directly to open data formats stored in cloud object storage or HDFS.