

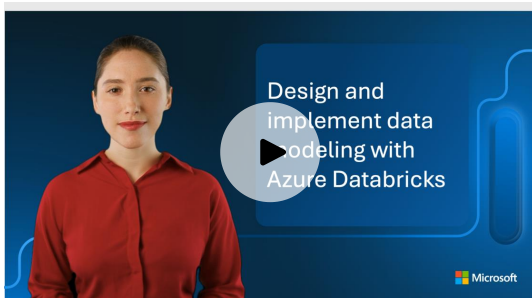
Design and implement data modeling with Azure Databricks

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/1-introduction>

Introduction

Completed



Your organization processes millions of transactions daily from multiple source systems, and stakeholders need reliable analytics that reflect both current operations and historical trends. The data engineering team faces critical decisions: How should data flow from source systems into the **lakehouse**? Which table formats provide the right balance of performance and compatibility? How do you maintain historical accuracy while keeping queries fast?

Data modeling in Azure Databricks with **Unity Catalog** addresses these challenges through deliberate design choices that affect every downstream consumer. You select **ingestion patterns** that match your latency requirements. You choose table formats like **Delta Lake** or **Apache Iceberg** based on transaction needs and cross-platform compatibility. You design **partitioning schemes** and **clustering strategies** that align with how analysts query the data.

These decisions compound over time. A table partitioned incorrectly today creates performance problems that grow with data volume. A **dimension table** without proper change tracking loses historical context. Thoughtful upfront design creates a foundation that scales efficiently and answers questions you haven't yet anticipated.

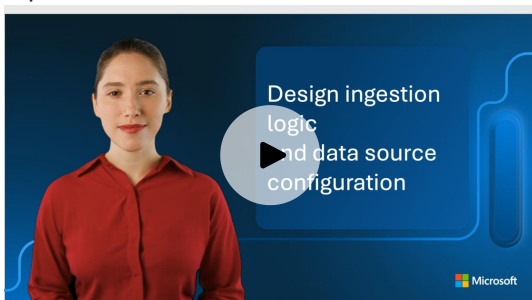
This module guides you through key data modeling decisions for Azure Databricks environments. You'll learn to design ingestion logic, select the right tools for each data source type, and implement storage strategies that optimize query performance. You'll explore **slowly changing dimensions** for historical accuracy and understand when **managed tables** provide advantages over **external tables**.

2. Design ingestion logic and data source configuration

<https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/2-design-data-ingestion-logic>

Design ingestion logic and data source configuration

Completed



Designing ingestion logic requires you to make critical decisions about how data moves from source systems into Azure Databricks. These decisions affect data freshness, processing efficiency, and system reliability. Before you select tools or write code, you need to understand the characteristics of your data sources and define extraction strategies that align with your business requirements.

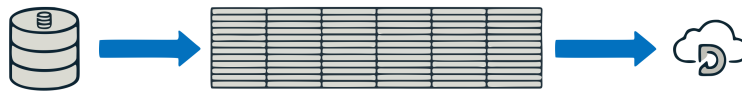
In this unit, you explore the key design considerations for data ingestion logic and data source configuration in Azure Databricks. The focus here is on conceptual logic and design patterns—specific tool selection and implementation details are covered in later units.

Understand extraction types

Extraction type determines how you capture data from source systems and impacts both the **timeliness** and **efficiency** of your data pipeline. Azure Databricks supports three primary extraction patterns, each suited to different scenarios.

Full extraction

Full extraction reads the entire dataset from the source system during each ingestion run.



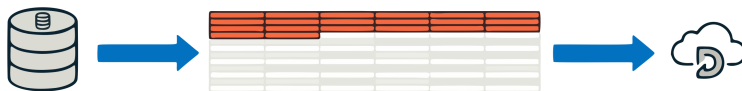
This approach works well when:

- Source systems don't support change tracking
- Data volumes are small enough to reprocess efficiently
- You need to rebuild destination tables from scratch periodically
- Data relationships are complex and incremental logic would be error-prone

Full extraction becomes impractical as data volumes grow. Reprocessing millions of records when only a few thousand changed wastes compute resources and delays downstream consumers.

Incremental extraction

Incremental extraction processes only new or changed records since the last ingestion run. The engine tracks what data has been processed and queries only for updates.



This pattern requires:

- A reliable **change indicator** in the source data (timestamp, sequence number, or version column)
- **State management** to track the last processed position
- Logic to handle **late-arriving** or **out-of-order** records

Incremental extraction significantly improves efficiency for large datasets. Instead of reading millions of rows, you might process only thousands of changes per run.

Streaming extraction

Streaming extraction provides **near real-time** data capture through continuous processing. Rather than running on a schedule, streaming jobs remain active and process records as they arrive.



Consider streaming when:

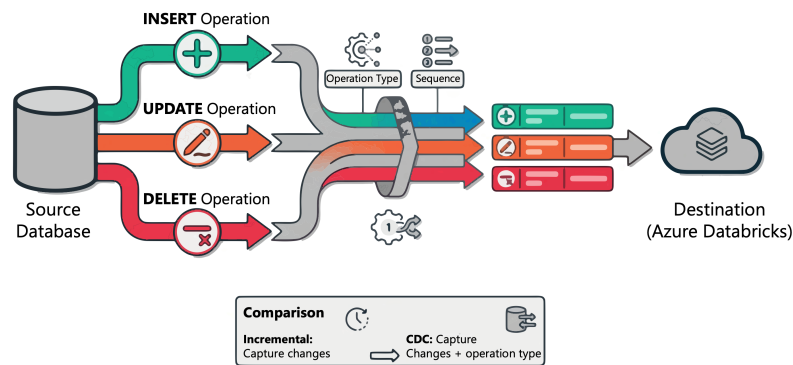
- Business processes require data freshness measured in seconds or minutes
- Source systems publish events to message buses or change feeds

- You need to react to events as they occur

Streaming extraction requires always-on infrastructure but delivers the lowest latency for data availability.

Design for change data capture

Change data capture (CDC) is a specialized extraction pattern that captures individual row-level changes, including inserts, updates, and deletes. CDC differs from simple incremental extraction because it preserves the operation type for each change.



When designing for CDC, consider how your source system produces change records:

- **Database transaction logs:** Databases like SQL Server can expose row-level changes using its transaction log–based CDC feature. Downstream systems such as Azure Databricks can then read these changes and process them as part of the ingestion pipeline.
- **Change data feeds:** Delta tables and some databases like Azure Cosmos DB provide built-in change tracking that surfaces row changes with metadata.
- **Periodic snapshots:** Some systems don't support continuous CDC, requiring you to compare snapshots to determine changes.

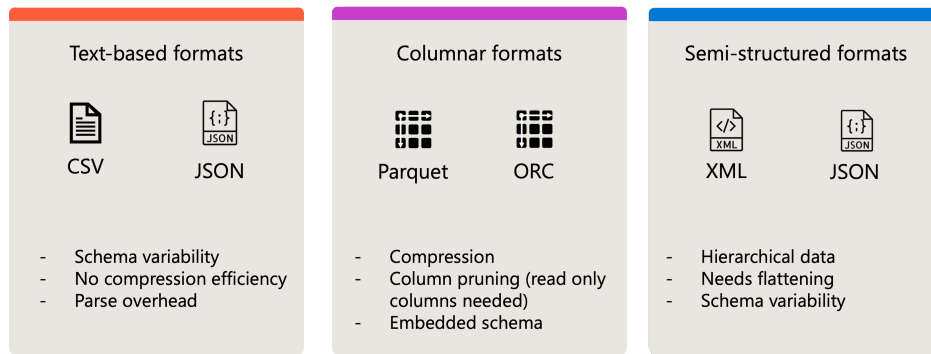
CDC enables use cases that simple incremental extraction can't support. You can accurately replicate deletes, maintain audit trails, and process updates that modify key columns.

Note

CDC requires sequencing information to handle out-of-order events correctly. Your source data needs a column (such as a timestamp or sequence number) that establishes the correct order of changes.

Identify source file type characteristics

When data arrives as files rather than from transactional systems, the file format influences your ingestion design. Each format has distinct characteristics that affect processing.



Text-based formats

CSV and **JSON** files are common source formats because they're human-readable and widely supported. However, they present challenges for efficient processing:

- **Schema variability:** Files might have inconsistent column orders, missing fields, or data type variations
- **No compression efficiency:** Text formats typically consume more storage than binary alternatives
- **Parse overhead:** Every value requires text-to-type conversion during reads

Use text-based formats when source systems can only produce these formats or when data volumes are small enough that efficiency concerns don't apply.

Columnar formats

Parquet and **ORC** files store data in columnar format, enabling efficient analytical queries. These formats offer:

- **Compression:** Columnar storage achieves better compression ratios by grouping similar values
- **Column pruning:** Queries read only the columns they need, reducing I/O
- **Embedded schema:** The file contains metadata describing column names and types

Design your ingestion to favor columnar source formats when you have control over how upstream systems export data.

Semi-structured formats

XML and nested JSON files represent hierarchical data that doesn't fit a flat tabular structure. When ingesting semi-structured data:

- Plan for schema extraction or definition.
- Determine whether to flatten the structure or preserve nesting.

- Account for varying document structures within the same source.

Data source connection considerations

Data source configuration defines how Azure Databricks connects to and authenticates with source systems. Your design must address connection security, credential management, and access patterns.

Cloud object storage sources

Cloud storage (Azure Data Lake Storage, Amazon S3, or Google Cloud Storage) serves as the most common landing zone for ingestion. When configuring storage connections:

- Use Unity Catalog external locations to govern access to storage paths.
- Configure appropriate authentication methods (managed identity, service principals, or storage credentials).
- Design folder structures that support efficient incremental processing.

Folder structure significantly impacts how efficiently Azure Databricks can identify and process new files. Consider these patterns:

Date-based partitioning: Organize files by date hierarchy to enable partition pruning during incremental reads:

```
/data/sales/year=2025/month=11/day=28/  
/data/sales/year=2025/month=11/day=29/
```

This structure allows queries to scan only relevant date partitions rather than listing all files.

Source-based organization: When ingesting from multiple sources, separate data by source system to simplify processing logic:

```
/landing/erp/orders/  
/landing/crm/customers/  
/landing/web/clickstream/
```

Processing state folders: Use separate folders to track file processing status:

```
/incoming/      # New files arrive here  
/processing/    # Files currently being processed  
/archive/       # Successfully processed files  
/failed/        # Files that failed processing
```

When designing folder structures, consider:

- **File discovery overhead:** Flat structures with thousands of files slow down listing operations. Partitioned structures reduce the files Azure Databricks must enumerate.

- **Auto Loader compatibility:** Azure Databricks **Auto Loader** efficiently tracks new files in cloud storage. Structure folders to align with how Auto Loader discovers and checkpoints files.
- **Partition column extraction:** Use Hive-style partitioning (`key=value`) so Azure Databricks can automatically extract partition values as columns without parsing file contents.

Database sources

Connecting to transactional databases requires different considerations than cloud storage:

- **Network connectivity** between Azure Databricks and the database server.
- Authentication credentials stored securely (using **secrets** rather than hardcoded values).
- **Connection pooling** and **query pushdown** capabilities.
- Impact on source system performance during extraction.

Streaming sources

Message buses and event streams (such as Azure Event Hubs or other common messaging platforms) require:

- Broker connection details and authentication.
- **Consumer group** configuration for parallel processing.
- **Offset management** to track consumption position.
- **Schema registry** integration for message format validation.

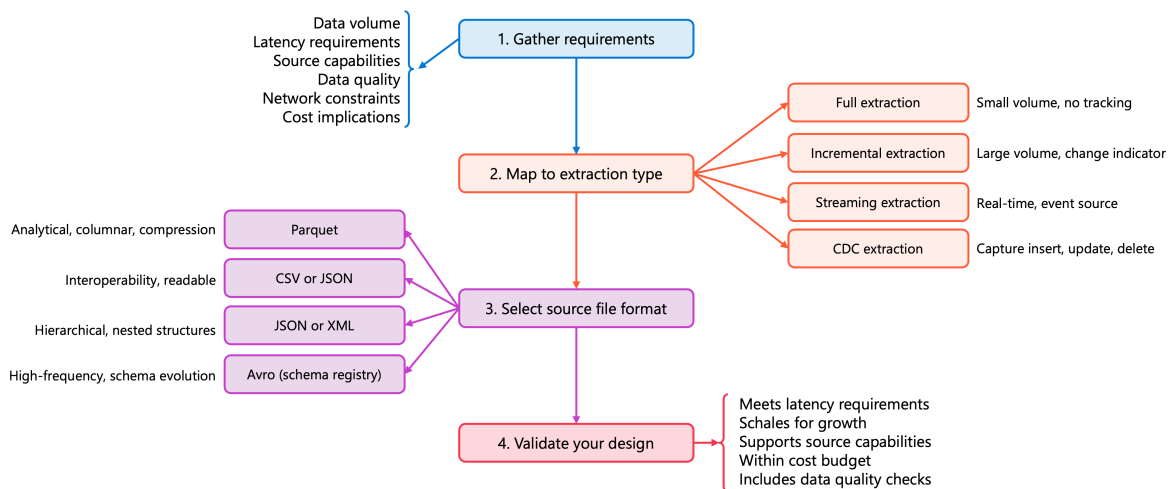
SaaS application sources

Enterprise applications like Salesforce, ServiceNow, or SAP expose data through APIs with unique characteristics:

- API rate limits that constrain extraction throughput.
- Pagination strategies for large result sets.
- Authentication flows (OAuth, API keys, or certificate-based).
- Available change tracking or webhook capabilities.

Apply a requirements-driven design framework

Effective ingestion design starts with analyzing requirements before selecting extraction types or file formats. Use the following repeatable framework to guide your design decisions.



Step 1: Gather requirements

Before making any technical decisions, answer these key questions:

Requirement Category	Questions to Answer
Data volume	How much data exists in the source? How much changes between runs? What's the growth rate?
Latency requirements	How fresh must data be for downstream consumers? Are there SLAs to meet?
Source capabilities	Does the source support change tracking, CDC, or only full extraction? What file formats can it produce?
Data quality	How reliable is the source data? What validation or cleansing is needed?
Network constraints	What bandwidth exists between source and Azure Databricks? Are there firewall or VPN requirements?
Cost implications	What are the compute, storage, and data transfer costs of different approaches?

Step 2: Map requirements to extraction type

Use your requirements analysis to select the appropriate extraction pattern:

If your requirements indicate...	Then choose...
Small data volume (< 100K rows), no change tracking available, simple rebuild acceptable	Full extraction
Large data volume, reliable change indicator exists (timestamp/sequence), hourly or daily freshness sufficient	Incremental extraction
Real-time or near real-time freshness required, source publishes events, streaming infrastructure available	Streaming extraction

If your requirements indicate...	Then choose...
Need to capture deletes and operation types, audit trail required, source supports transaction log capture	CDC extraction

Step 3: Select source file format

When you control the source format or need to choose a landing zone format, match the format to your needs:

If your scenario involves...	Then prefer...
Large analytical datasets, columnar queries, need compression	Parquet
Interoperability with many systems, human readability needed	CSV or JSON
Hierarchical or nested data structures	JSON or XML
High-frequency streaming with schema evolution	Avro with schema registry

Step 4: Validate your design

Before implementing, verify your design addresses these concerns:

- **The extraction type meets latency requirements.** If not, downstream consumers receive stale data, potentially causing incorrect business decisions or SLA violations.
- **The approach scales to handle projected data growth.** If not, pipelines that work today may fail or become prohibitively slow as data volumes increase.
- **Source system capabilities support your chosen pattern.** If not, the pipeline fails at runtime or requires costly workarounds that add complexity.
- **Cost estimates are within budget constraints.** If not, the solution may be technically sound but financially unsustainable for production use.
- **Data quality checks are planned for the pipeline.** If not, invalid or corrupted data propagates to downstream systems, causing errors or incorrect analytics.

Apply the framework: A practical example

An e-commerce platform needs to ingest order data. Orders are stored in a PostgreSQL database, and the business requires hourly reporting.

Step 1 - Gather requirements:

- **Volume:** 50,000 new orders daily, 2 million total rows, 15% annual growth.
- **Latency:** One-hour freshness acceptable for reporting.
- **Source:** PostgreSQL supports timestamp-based queries but no native CDC.
- **Quality:** Source data is validated at entry; minimal cleansing needed.

- **Network:** Direct connectivity via Azure Private Link.
- **Cost:** Minimize compute usage; storage costs are secondary.

Step 2 - Map to extraction type:

Large volume with reliable timestamp column and hourly latency requirement → **Incremental extraction.**

Why not the others?

- **Full extraction:** Reprocessing 2 million rows hourly wastes compute when only ~2,000 orders change per hour.
- **Streaming:** One-hour latency doesn't justify the cost of always-on streaming infrastructure.
- **CDC:** PostgreSQL doesn't expose native transaction log CDC, and the use case doesn't require tracking deletes or operation types.

Step 3 - Select file format:

Data lands in Azure Data Lake Storage for processing → **Parquet** for efficient columnar storage.
Date-partitioned layout for incremental discovery →
`/landing/orders/year=2025/month=11/day=28/`.

Why not the others?

- **CSV/JSON:** Larger file sizes, no compression benefits, and slower query performance for analytical workloads.
- **Avro:** Better suited for streaming with schema evolution; batch ingestion doesn't need its streaming-oriented features.
- **XML:** Adds parsing complexity without benefits for structured tabular order data.

Step 4 - Validate design:

- ✓ Hourly incremental meets one-hour latency requirement.
- ✓ Processing only changes scales efficiently with growth.
- ✓ PostgreSQL `order_updated_at` column enables incremental queries.
- ✓ Processing thousands vs. millions of rows reduces compute costs.

Final decision: Incremental extraction using the `order_updated_at` timestamp column, scheduled hourly, landing data as Parquet files in Azure Data Lake Storage.

Tip

Document your design decisions and the rationale behind them. When requirements change or issues arise, this documentation helps you understand why the current approach was chosen and what tradeoffs were made. Consider creating a design decision record for each major pipeline.

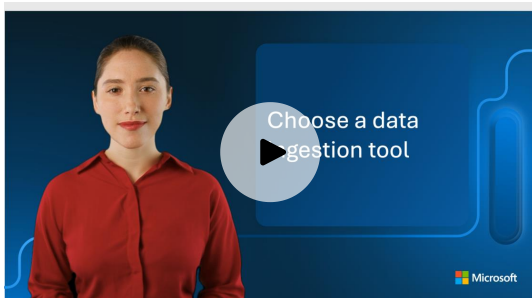
Building on this requirements-driven foundation, you're ready to evaluate the specific tools and loading methods that implement your ingestion design.

3. Choose a data ingestion tool

<https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/3-choose-data-ingestion-tool>

Choose a data ingestion tool

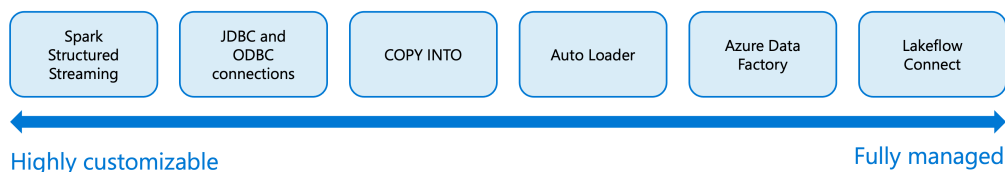
Completed



When you design data pipelines in Azure Databricks, selecting the right ingestion tool directly impacts pipeline reliability, maintenance effort, and long-term costs. Each tool addresses different scenarios, data sources, and operational requirements. Understanding these differences helps you make informed decisions that align with your organization's needs.

Understand the ingestion tool landscape

Azure Databricks provides multiple ingestion options that span a spectrum from fully managed services to highly customizable frameworks. The tools you choose depend on factors such as your data source type, latency requirements, and how much control you need over the ingestion process.



At the most managed end, **Lakeflow Connect** offers turnkey connectors for enterprise applications and databases. These connectors handle authentication, change tracking, error recovery, and schema evolution automatically. At the customizable end, **Spark Structured Streaming** gives you complete control over how data flows through your pipeline, though it requires more development and maintenance effort.

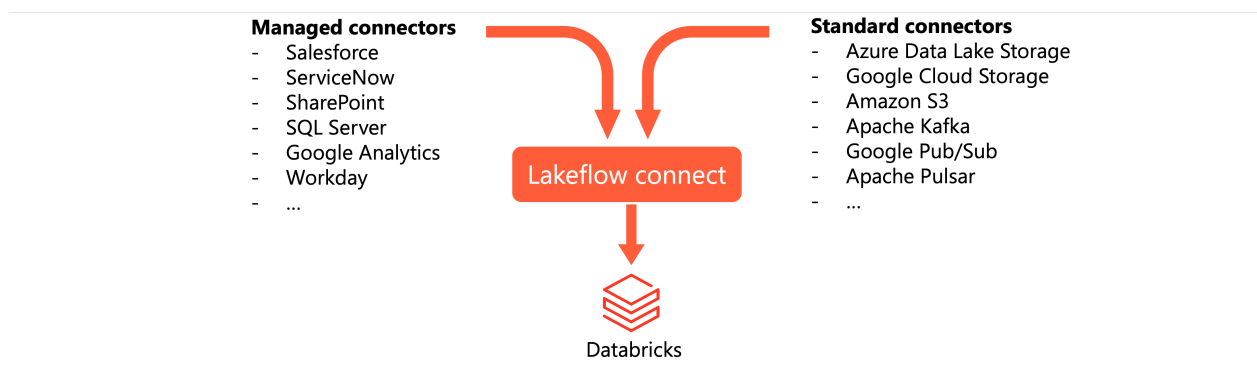
Between these extremes, you find tools like **Auto Loader** for cloud storage files, **COPY INTO** for SQL-based batch loading, and notebook-based approaches using **JDBC/ODBC** connections. Additionally, **Azure Data Factory** serves as an orchestration layer that can coordinate data movement into Azure Databricks from the broader Azure ecosystem.

Lakeflow Connect

Lakeflow Connect represents the modern approach to data ingestion in Azure Databricks. It offers both fully managed connectors and standard connectors that integrate with Lakeflow Spark Declarative Pipelines for a streamlined experience.

Managed connectors provide out-of-the-box integration with popular enterprise applications and databases. Supported sources include Salesforce, ServiceNow, SharePoint, SQL Server, Google Analytics, and Workday. These connectors handle the complexities of source-specific authentication, incremental data capture, API rate limiting, and automated retries. They run on serverless compute and write to streaming tables governed by Unity Catalog.

Standard connectors extend Lakeflow Connect to cloud object storage and message buses. You can ingest data from Amazon S3, Azure Data Lake Storage, or Google Cloud Storage using Auto Loader. For real-time data, standard connectors support Apache Kafka, Google Pub/Sub, and Apache Pulsar.



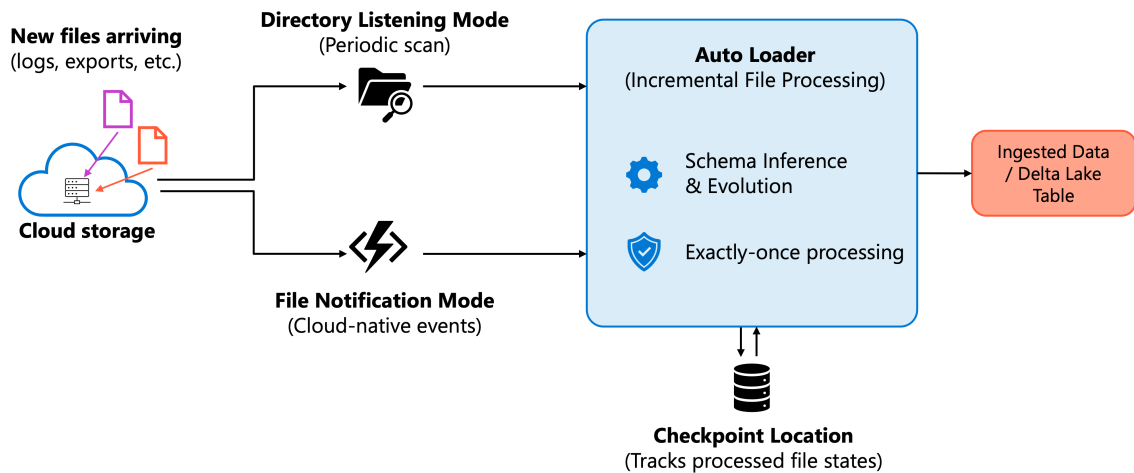
Consider Lakeflow Connect when you need reliable, low-maintenance ingestion from supported sources. The managed connectors significantly reduce development time and ongoing maintenance compared to custom solutions. However, if your data source isn't supported or you need highly specialized ingestion logic, you might need to explore other options.

Lakeflow Connect supports both **scheduled** ingestion (hourly, daily, weekly) and **on-demand** triggers, making it suitable for regular data refresh cycles and ad hoc loading scenarios.

Auto Loader

Auto Loader provides incremental file processing for data arriving in cloud storage. Rather than scanning entire directories to find new files, Auto Loader efficiently tracks which files have been processed and ingests only new arrivals.

Auto Loader works by maintaining state in a checkpoint location that tracks processed files. When new files land in your cloud storage, Auto Loader discovers them through either directory listing or file notification mode. Directory listing periodically scans for new files, while file notification mode uses cloud-native events for immediate detection.



Key capabilities of Auto Loader include:

- **Schema inference and evolution:** Automatically detects the schema of incoming files and handles schema changes gracefully.
- **Exactly-once processing:** Guarantees each file is processed once, even across failures and restarts.
- **Scalability:** Efficiently handles billions of files for backfills or migrations.
- **Format support:** Processes JSON, CSV, Parquet, Avro, ORC, XML, text, and binary files.

Auto Loader fits scenarios where data arrives as files in cloud storage—common patterns include log files, exports from upstream systems, and data landing zones. You can use Auto Loader with Lakeflow Spark Declarative Pipelines for a managed experience or with Structured Streaming for more control.

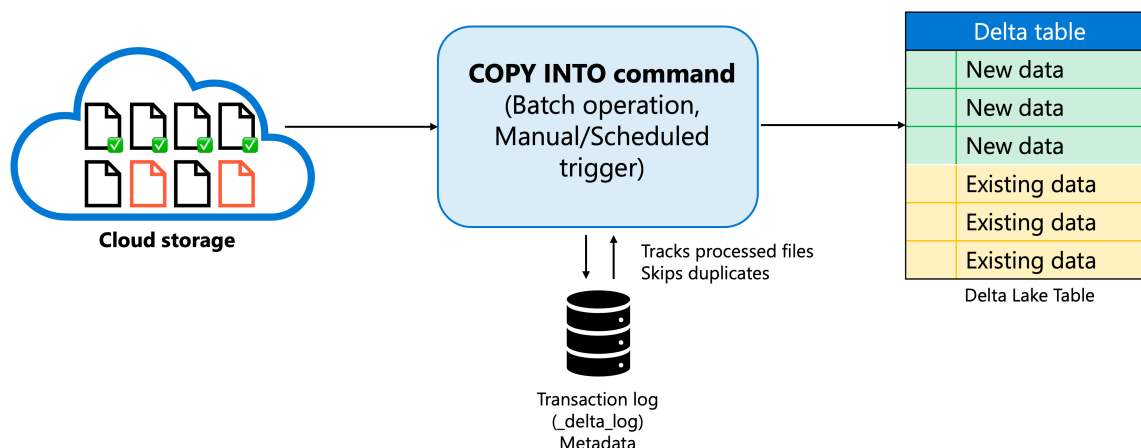
COPY INTO

The `COPY INTO` SQL command loads data from cloud storage into Delta tables. Unlike Auto Loader, which runs continuously as a stream, `COPY INTO` executes as a batch operation that you schedule or trigger manually.

`COPY INTO` tracks which files it has already processed, making the operation idempotent. Running the same command multiple times doesn't duplicate data because previously loaded files are skipped. This characteristic makes `COPY INTO` useful for scheduled batch jobs where you want predictable, repeatable behavior.

Note

The tracking metadata is stored as part of the Delta table's transaction log (the `_delta_log` directory)



Common scenarios for `COPY INTO` include one-time data migrations, scheduled batch loads from landing zones, and SQL-first workflows where you prefer declarative syntax over streaming APIs.

Choose `COPY INTO` over Auto Loader when you're performing a one-time batch load of a large dataset. `COPY INTO` provides a simpler configuration model without requiring streaming infrastructure, making it ideal for loading hundreds of gigabytes or even terabytes of data in a single operation. Auto Loader's continuous processing model and checkpoint management add unnecessary complexity when you only need to load data once.

However, for ongoing incremental ingestion at scale, Auto Loader provides better performance and more robust file discovery.

Spark Structured Streaming

Spark Structured Streaming provides the foundation for real-time data processing in Azure Databricks. It enables continuous ingestion from various streaming sources with exactly-once processing guarantees.

Streaming sources supported include:

- **Apache Kafka and Azure Event Hubs:** For event-driven architectures and message queue processing.
- **Delta Lake change feeds:** For propagating changes between Delta tables.
- **Cloud storage via Auto Loader:** For file-based streaming ingestion.
- **Socket and custom sources:** For specialized ingestion scenarios.

Choose Structured Streaming when you need fine-grained control over streaming behavior, custom watermarking or windowing logic, or when working with sources not supported by higher-level abstractions.

JDBC and ODBC connections

Notebook-based ingestion using JDBC or ODBC remains a practical approach for extracting data from relational databases. You write code that connects directly to source databases, executes queries, and loads results into Delta tables.

This approach supports virtually any database with a JDBC or ODBC driver: SQL Server, Oracle, PostgreSQL, MySQL, and many others. You implement full or incremental loads by writing appropriate queries, often using watermark columns to track extraction progress.

While Lakeflow Connect managed connectors are replacing JDBC/ODBC for many scenarios, notebook-based connections remain valuable when:

- Your database isn't supported by Lakeflow Connect managed connectors.
- You need custom extraction logic that managed connectors can't accommodate.
- You're maintaining existing pipelines that use this pattern.
- You require specific query pushdown optimizations for large tables.

When using JDBC/ODBC, store credentials securely using Databricks secrets rather than hardcoding them in notebooks. Consider the performance impact on source systems and implement appropriate throttling or scheduling to avoid overwhelming transactional databases.

For on-premises databases, additional network infrastructure is required—either Azure ExpressRoute/VPN for direct JDBC connectivity from Databricks, or Azure Data Factory with a Self-Hosted Integration Runtime to land data in cloud storage first.

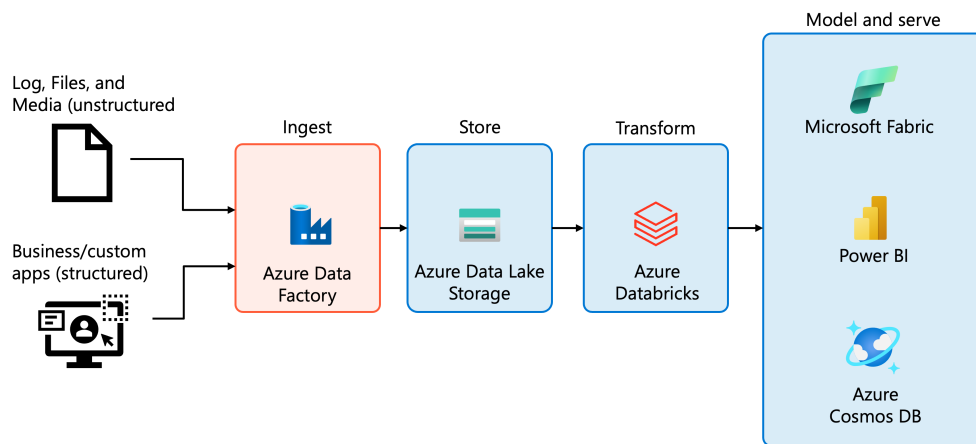
Azure Data Factory

Azure Data Factory (ADF) serves as an orchestration and data movement platform within the Azure ecosystem. It excels at coordinating workflows that span multiple services and landing data from diverse sources into cloud storage where Azure Databricks can process it.

ADF integrates extensively with Microsoft services like Dynamics 365, Azure SQL Database, and Microsoft Fabric. When your data originates from these sources, ADF provides straightforward connectivity and can land data in Azure Data Lake Storage for subsequent Databricks processing.

Understanding ADF's appropriate role is important. Use ADF for:

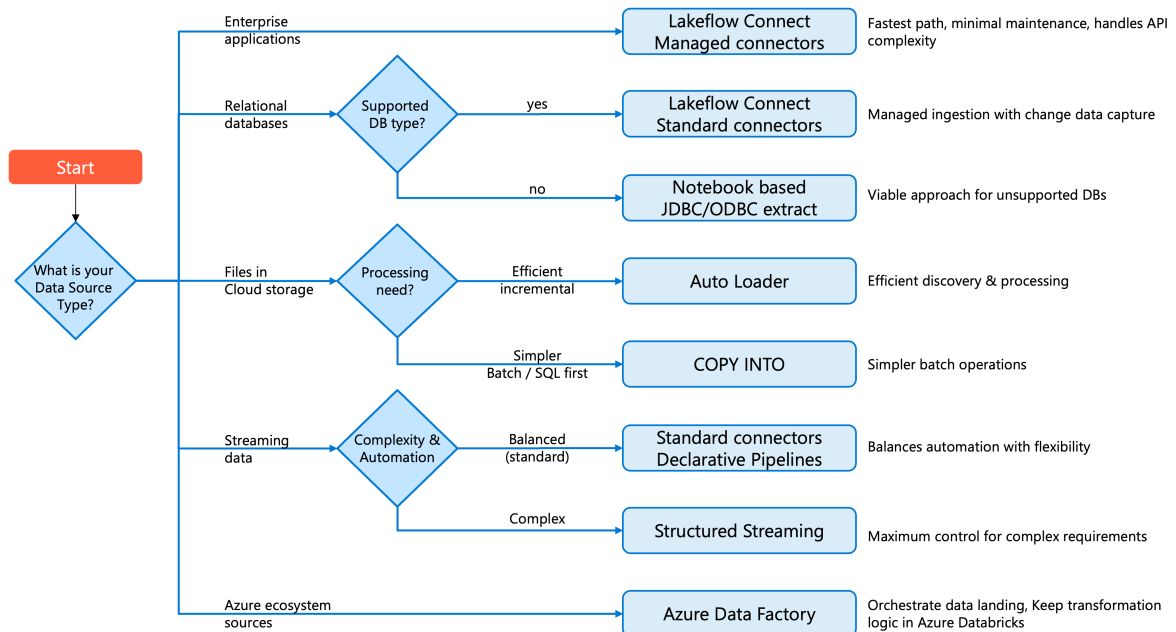
- **Orchestration:** Coordinating workflows that involve multiple Azure services.
- **Initial data landing:** Moving data from Azure sources to cloud storage.
- **Enterprise integration:** When company policy mandates ADF for data movement.



Avoid using ADF for core ETL transformations inside Databricks. Running transformations through ADF activities introduces unnecessary complexity and performs less efficiently than Databricks-native processing. Let ADF handle data movement into your lakehouse, then use Azure Databricks tools for transformation logic.

Apply a decision framework

Selecting an ingestion tool becomes clearer when you evaluate your scenario against key decision criteria.



Start by identifying your data source type, then consider operational requirements and organizational context.

For **enterprise applications** (Salesforce, ServiceNow, SharePoint), Lakeflow Connect managed connectors offer the fastest path to production with minimal maintenance. These connectors handle the complexity of API integration, rate limiting, and change tracking.

For **relational databases** with supported types, Lakeflow Connect database connectors provide managed ingestion with change data capture. For unsupported databases, notebook-based JDBC/ODBC extraction remains a viable approach.

For **files in cloud storage**, Auto Loader provides efficient incremental discovery and processing. Use COPY INTO for simpler batch scenarios or when you prefer SQL-first workflows.

For **streaming data** from message buses (Kafka, Event Hubs, Pub/Sub), standard connectors with Lakeflow Spark Declarative Pipelines balance automation with flexibility. For complex streaming requirements, Structured Streaming offers maximum control.

For **Azure ecosystem sources** requiring coordination with other services, Azure Data Factory can orchestrate data landing into your lakehouse. Keep transformation logic within Azure Databricks rather than in ADF data flows.

Tip

Start with the most managed option that meets your requirements. Managed connectors reduce development time and maintenance burden. Only move to more customizable approaches when specific requirements can't be met by higher-level abstractions.

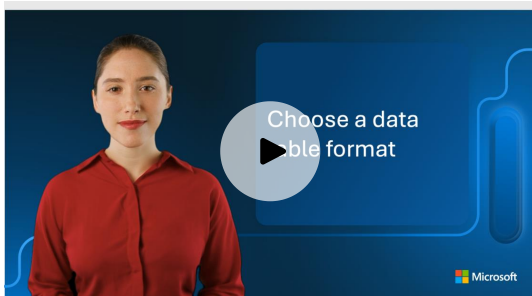
Your choice also depends on team skills and organizational standards. Document your decisions and the reasoning behind them to help future team members understand the pipeline architecture.

4. Choose a data table format

<https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/4-choose-data-table-format>

Choose a data table format

Completed

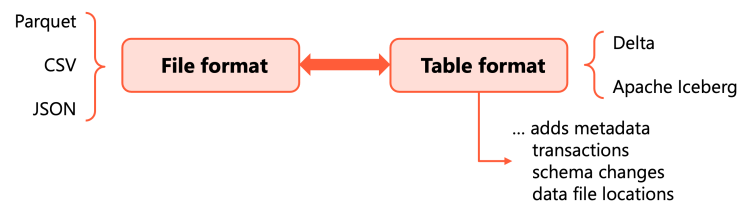


Data engineers face a critical decision early in every data pipeline: which table format should store your data? This choice affects query performance, transaction reliability, interoperability with other platforms, and long-term maintenance of your lakehouse architecture.

Azure Databricks supports multiple storage formats, each designed for different scenarios. Understanding their differences helps you select the right format for your specific requirements.

Understand file formats versus table formats

Before comparing options, it's important to distinguish between file formats and table formats. A **file format** like Parquet, CSV, or JSON defines how data is physically stored on disk. A **table format** like Delta Lake or Apache Iceberg builds on top of file formats by adding a metadata layer that tracks transactions, schema changes, and data file locations.



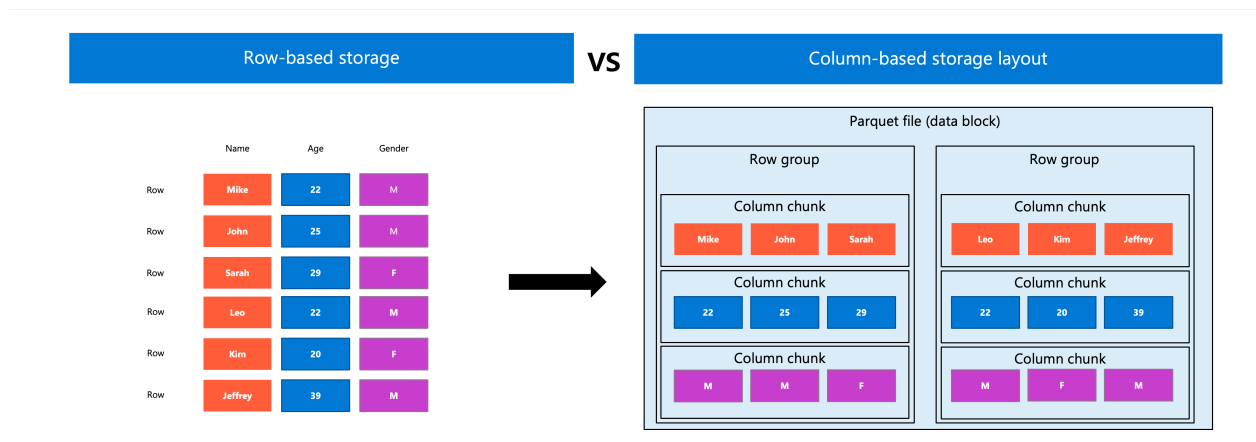
Both Delta Lake and Apache Iceberg use Parquet as their underlying file format. The key difference lies in the metadata layer that enables advanced features like ACID transactions, time travel, and schema evolution.

How Parquet stores data differently

Parquet uses **columnar storage** rather than row-based storage. This fundamental difference significantly impacts analytical query performance.

In row-based formats like CSV, data is organized by record. Reading a single column requires scanning every row in the file. In columnar formats like Parquet, data for each column is stored together. When a query needs only specific columns, Parquet reads just those columns and skips the rest entirely.

Parquet organizes data into **row groups**, each containing **column chunks**. Every column chunk stores statistics including minimum and maximum values for that data segment. When you execute a query with a filter condition, the query engine checks these statistics first. If the filter value falls outside the min/max range of a row group, the engine skips reading that entire section.



This columnar approach provides two optimization techniques that dramatically improve analytics performance:

- **Column pruning:** The engine reads only the columns your query references, reducing I/O substantially.
- **Predicate pushdown:** Filter conditions are evaluated against column statistics before reading data, skipping irrelevant row groups entirely.

For analytical workloads that typically access a subset of columns across many rows, Parquet substantially outperforms row-based formats.

Delta Lake as the default format

Delta Lake is the default storage format for all tables in Azure Databricks. When you create a table without specifying a format, Azure Databricks automatically uses Delta Lake.

Delta Lake extends Parquet files with a **transaction log** that records every change to your table. This log enables several capabilities that Parquet alone doesn't provide:

- **ACID transactions:** Multiple concurrent readers and writers can access the same table without data corruption. Each transaction either completes entirely or has no effect.
- **Time travel:** Query previous versions of your table using timestamps or version numbers. Roll back changes if needed.
- **Schema enforcement and evolution:** The table validates incoming data against its defined schema, preventing corrupt or incompatible data from entering your lakehouse. When requirements change, you can add new columns, modify data types, or rename columns without rewriting existing data.
- **Unified batch and streaming:** Use the same table as both a batch source and a streaming source or sink.

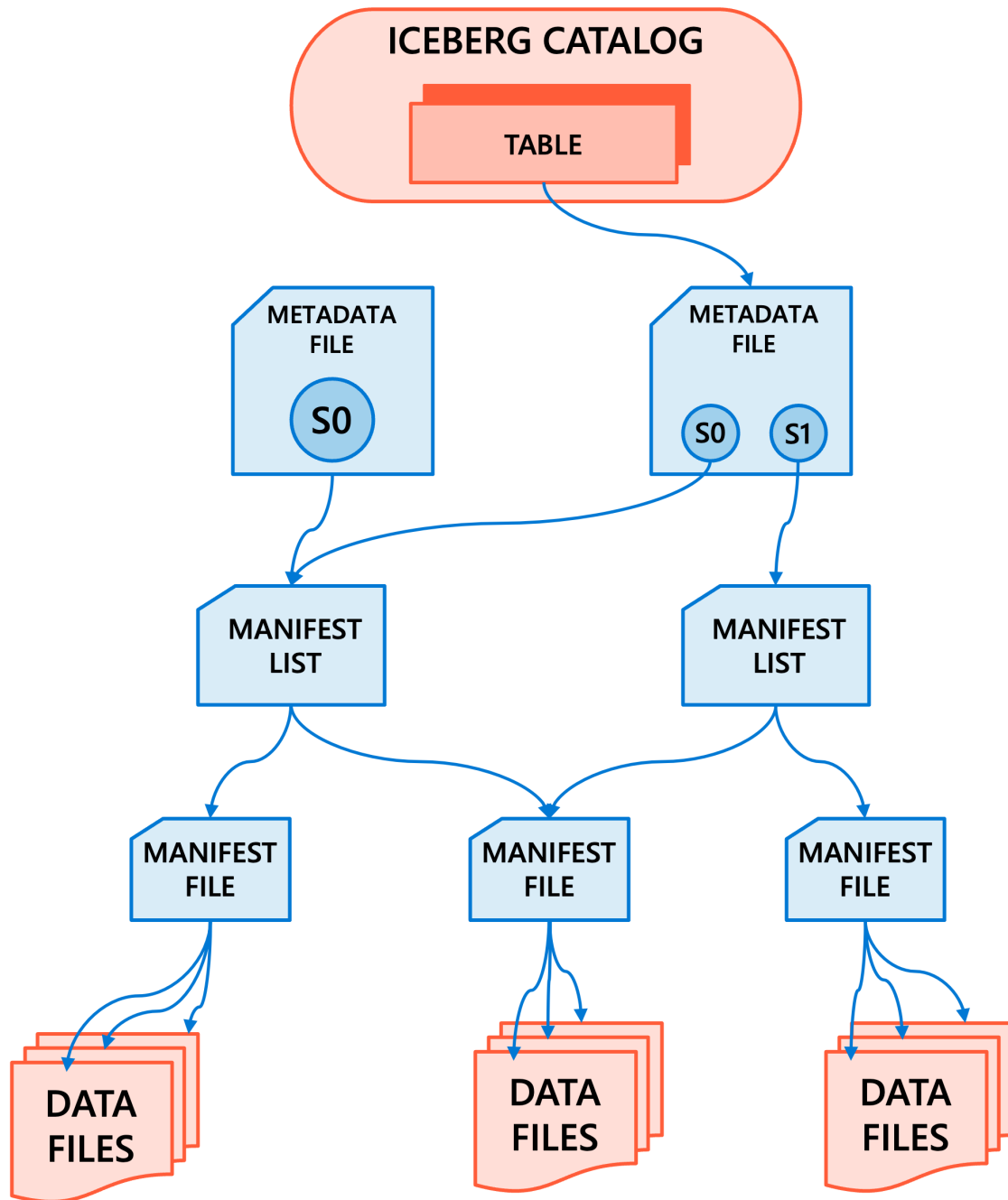
Delta Lake integrates deeply with the Azure Databricks platform. Features like **liquid clustering**, **predictive optimization**, and **change data feed** rely on Delta Lake's transaction log.

For most scenarios in Azure Databricks, Delta Lake is the recommended choice. It provides the performance benefits of Parquet with transactional guarantees that production data systems require.

Apache Iceberg for cross-platform scenarios

Apache Iceberg is an alternative open table format that Azure Databricks supports for managed and foreign tables. Like Delta Lake, Iceberg provides ACID transactions, schema evolution, and time travel capabilities built on top of Parquet files.

Iceberg's hierarchical and well-defined logical structure makes it highly flexible, supporting scalability and advanced optimizations. It supports manifest pruning, file-level statistics, hidden partitioning and partition evolution.



The primary reason to choose Iceberg is **interoperability**. Iceberg is widely adopted across the data ecosystem, including platforms like Snowflake, Amazon Athena, Trino, Apache Spark, and Apache Flink. If your organization runs workloads across multiple platforms that need to read and write the same tables, Iceberg provides a common format all platforms understand.

Unity Catalog can manage Iceberg tables directly, allowing Azure Databricks to serve as an Iceberg catalog for external engines. You can also read Iceberg tables managed by other catalogs like AWS Glue or Snowflake Horizon Catalog through foreign table connections.

Consider Iceberg when:

- Your data needs to be accessed by multiple processing engines outside Azure Databricks.
- Your organization has standardized on Iceberg for cross-platform data sharing.
- You're integrating with systems that don't support Delta Lake.

For workloads that run primarily within Azure Databricks, Delta Lake remains the recommended format due to its deeper platform integration and optimization features.

When row-based formats apply

CSV and JSON remain valid choices for specific scenarios, even though they're less efficient for large-scale analytics.

Use CSV or JSON when:

- You're receiving data from external systems in these formats as an initial landing zone.
- You need human-readable data for debugging or quick inspection.
- You're integrating with systems that only support text-based formats.
- Data volumes are small enough that performance differences are negligible.

For persistent storage of analytical data, convert incoming CSV or JSON files to **Delta Lake tables** during ingestion. This conversion captures the efficiency gains of columnar storage while adding transactional guarantees.

Apply decision criteria

Selecting a table format depends on your specific requirements. Start with Delta Lake as your default choice, then evaluate alternatives only when specific constraints require them.

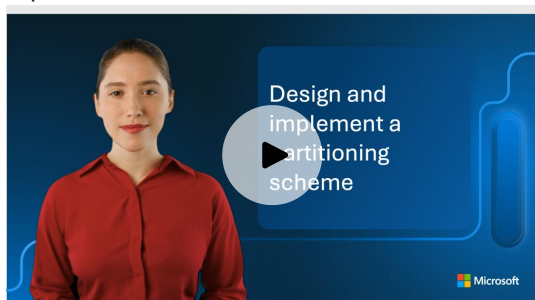
Consideration	Delta Lake	Apache Iceberg	CSV/JSON
Analytics performance	Excellent	Excellent	Poor
ACID transactions	Yes	Yes	No
Time travel	Yes	Yes	No
Azure Databricks integration	Deep	Supported	Basic
Cross-platform compatibility	Growing	Wide	Universal
Recommended for production	Yes	Yes (when interop required)	No (landing zone only)

Your choice also depends on organizational standards. If your architecture team has established conventions for table formats, align with those decisions to maintain consistency across your data platform.

5. Design and implement a data partitioning scheme

Design and implement a data partitioning scheme

Completed



Data partitioning directly affects how efficiently your queries run and how much compute resources you consume. When you partition a table, you divide data into discrete segments based on column values, allowing the query engine to read only relevant portions of your dataset. This approach, called **partition pruning**, can dramatically reduce data scans when filters align with partition boundaries.

Consider a retail company processing millions of daily sales transactions. Analysts frequently query data by date ranges to generate weekly reports, compare month-over-month trends, and analyze seasonal patterns. Without partitioning, every query scans the entire transaction history. With proper partitioning by date, queries for a specific week touch only a fraction of the stored files.

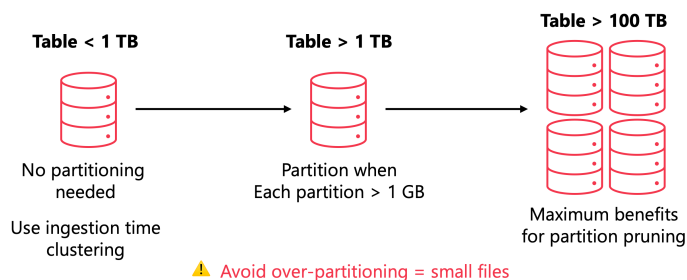
Determine when partitioning applies

Databricks recommends that you don't partition tables containing less than **1 TB** of data. For smaller tables, the overhead of managing partition metadata and the potential for creating many small files outweighs any query performance benefits. Unpartitioned Delta tables in recent Databricks Runtime versions automatically benefit from **ingestion time clustering**, which provides similar query advantages without explicit configuration.

When your tables exceed **1 TB**, partitioning becomes a valuable optimization technique. At this scale, the benefits of reading only relevant data segments significantly offset the costs of partition management. Partitioning also enables parallel processing, as Spark can distribute different partitions across cluster nodes for concurrent execution.

Important

Each partition should contain at least **1 GB** of data. This guideline helps you avoid over-partitioning, which creates excessive small files and degrades query performance.



For very large tables in the hundreds of terabytes range, partitioning delivers even greater benefits. At this scale, partition pruning dramatically reduces the amount of data the query engine needs to process, improving both performance and cost efficiency.

Choose effective partition keys

Partition key selection requires you to understand your data characteristics and query patterns. Effective partition keys share common traits that maximize performance benefits while avoiding common pitfalls.

Low cardinality columns work best

Choose columns with a **limited number of distinct values**. Date columns, geographic regions, product categories, and status fields make excellent partition keys because they create a manageable number of partitions. A column with thousands or millions of unique values, like customer ID or transaction ID, leads to over-partitioning and poor performance.

For example, partitioning a sales table by `sale_date` at the daily level might create 365 partitions per year. If each day contains sufficient data volume, this granularity works well. However, partitioning by `sale_timestamp` at the second level would create millions of partitions, each containing minimal data.

Align with query patterns

Your partition key should match the columns that users filter on most frequently. If analysts consistently query sales data by date ranges, partitioning by date enables efficient pruning. If queries primarily filter by region, consider region as your partition key instead.

When multiple filter patterns exist, prioritize the most common or most performance-critical queries. You can define one partitioning scheme per table, so choose columns that benefit the majority of your workload. For significantly different query patterns, consider creating separate optimized **materialized views** or **summary tables** for specific use cases.

Consider data growth over time

Select a partition scheme that scales appropriately as your data grows. Partitioning by year works well initially but might create partitions that are too large over time. Conversely, partitioning by hour might work for high-volume data but creates too many small partitions for moderate volumes.

Evaluate your expected data growth rate when choosing partition granularity. A balance between partition size and the number of partitions helps maintain consistent query performance as your dataset expands.

Also consider data distribution across partitions. Severely uneven partition sizes can create processing bottlenecks, as nodes handling larger partitions take longer to complete while others remain idle.

Implement partitions in Azure Databricks

You define partitions when creating a table using the `PARTITIONED BY` clause in SQL or the `partitionBy` method in PySpark. Once you establish partitions, Azure Databricks automatically routes incoming data to the appropriate partition directories based on column values.

Create a partitioned table with SQL

The following example creates a sales transactions table partitioned by the transaction date:

```
CREATE TABLE sales.transactions (  
  transaction_id STRING,  
  customer_id STRING,  
  product_id STRING,  
  quantity INT,  
  unit_price DECIMAL(10,2),  
  total_amount DECIMAL(12,2),  
  transaction_date DATE  
)  
PARTITIONED BY (transaction_date);
```

When you query this table with a date filter, the query engine reads only the partitions that match your filter criteria:

```
SELECT product_id, SUM(total_amount) AS revenue  
FROM sales.transactions  
WHERE transaction_date BETWEEN '2024-01-01' AND '2024-01-31'  
GROUP BY product_id;
```

Create a partitioned table with PySpark

When using PySpark, specify partitioning during the write operation:

```
df.write.format("delta") \
    .partitionBy("transaction_date") \
    .saveAsTable("sales.transactions")
```

For subsequent writes, maintain consistency by including the same partition specification:

```
new_transactions_df.write.format("delta") \
    .mode("append") \
    .partitionBy("transaction_date") \
    .saveAsTable("sales.transactions")
```

Use generated columns for partition flexibility

Generated columns provide a way to derive partition values from existing columns without storing redundant data. This approach is useful when your source data contains timestamps but you want to partition by date:

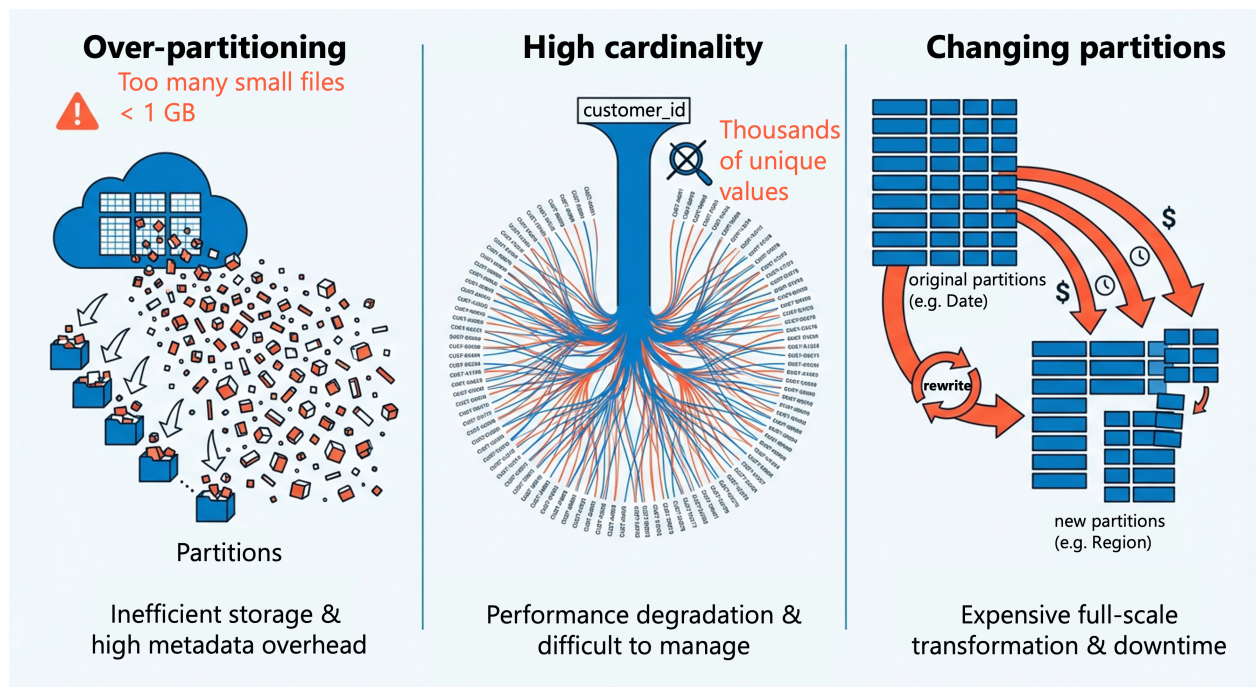
```
CREATE TABLE sales.transactions (
  transaction_id STRING,
  customer_id STRING,
  product_id STRING,
  quantity INT,
  unit_price DECIMAL(10,2),
  total_amount DECIMAL(12,2),
  transaction_timestamp TIMESTAMP,
  transaction_date DATE GENERATED ALWAYS AS (CAST(transaction_timestamp AS DATE))
)
PARTITIONED BY (transaction_date);
```

Tip

With this configuration, Delta Lake automatically generates partition filters when you query using the base timestamp column. A query filtering on `transaction_timestamp` benefits from partition pruning without requiring explicit date filters.

Avoid common partitioning mistakes

Understanding what to avoid is as important as knowing what to do. Several common mistakes can significantly degrade performance and complicate data management.



Over-partitioning creates small file problems

Creating too many partitions, each containing insufficient data, leads to **small file problems**. When partitions contain files smaller than the target file size, query performance suffers from excessive metadata operations and reduced read efficiency. Aim for partitions with at least **1 GB** of data.

Signs of over-partitioning include slow query performance despite partition pruning, a large number of small files in partition directories, and long table optimization times. If you observe these symptoms, consider choosing a coarser partition granularity.

Partitioning on high-cardinality columns fails

Columns with many unique values make poor partition keys. Partitioning by customer ID, transaction ID, or other **high-cardinality columns** creates millions of partitions that degrade rather than improve performance. Reserve high-cardinality column optimization for other techniques that are designed for this purpose.

Changing partitions requires rewriting data

Partition columns are part of the table structure. Changing your partitioning strategy requires **rewriting the entire table** to a new location with different partition boundaries. This operation consumes significant compute resources and time for large tables.

Plan your partitioning strategy carefully before loading data at scale. Consider future query patterns and data growth to avoid costly restructuring later.

Document partitioning decisions

Record your partition key choices and the reasoning behind them. This documentation helps team members understand query optimization opportunities and prevents accidental changes that could impact performance. Include information about expected partition sizes, query patterns that benefit from the scheme, and any constraints that influenced your decisions.

Consider liquid clustering as an alternative

When you encounter partitioning challenges—especially with **high-cardinality columns** like timestamps or when query patterns are **uncertain**—liquid clustering provides a modern alternative. Unlike traditional partitioning, liquid clustering offers:

- Flexibility to adjust optimization strategies without rewriting data
- Automatic maintenance as tables grow
- Reduced partition management overhead

Liquid clustering is the **recommended approach** for new tables where partitioning constraints would limit performance or create excessive maintenance burden. It's particularly effective for tables that exceed 1 TB and require filtering on columns that would create too many partitions using traditional approaches.

Note

Liquid clustering is not compatible with partitioned tables. Choose one approach based on your requirements when designing new tables.

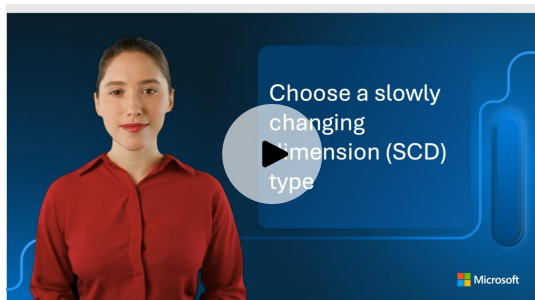
For implementation details, syntax, and optimization strategies, refer to the **Design and implement a clustering strategy** unit later in this module.

6. Choose a slowly changing dimension (SCD) type

<https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/6-choose-slowly-changing-dimension-type>

Choose a slowly changing dimension (SCD) type

Completed



Dimension tables in a data warehouse capture descriptive attributes about business entities like customers, products, or employees. These attributes change over time, and how you handle those changes directly affects your ability to perform accurate historical analysis. Slowly changing dimensions (SCDs) provide frameworks for managing these changes, and choosing the right SCD type is a critical design decision.

As a data engineer, you're expected to understand SCD types and implement them when directed. While data architects often make the initial design decisions, you need sufficient knowledge to evaluate trade-offs, recommend approaches, and execute implementations correctly. This unit helps you understand the characteristics of each SCD type so you can select the appropriate approach for your business requirements.

Understand dimension changes

Before selecting an SCD type, consider what happens when source data changes. Imagine a customer named Hank Zoeng moves from Seattle to Portland. Your customer dimension table currently shows Seattle as his city. You have three fundamental options:

- **Overwrite** the existing value with Portland, losing all record that he was ever in Seattle.
- **Preserve** the Seattle record and create a new version showing Portland, maintaining the complete history.
- **Track** a limited history by adding a "previous city" column alongside the current value.

Each approach serves different analytical needs. Sales reports that show "current customer location" need different handling than analyses tracking "customer migration patterns" or "sales by customer's region at time of purchase."

SCD Type 0

SCD Type 0 represents the simplest approach: never update dimension attributes after the initial insert. The original values remain permanently, regardless of source system changes.

Customer_ID	Customer_Name	Birth_Date	Registration_Date	Country
1001	Alice Smith	1988-04-12	2015-06-01	USA
1002	Bob Johnson	1979-11-03	2016-09-15	Canada



Customer_ID	Attribute Changed	Old Value	New Value	Action Taken
1001	Country	USA	UK	✗ No update — original value preserved

This type works well for:

- **True constants:** Birth dates, original registration dates, or founding dates that genuinely never change.
- **Audit requirements:** When regulations require preserving the original value exactly as first recorded.
- **Reference data:** Lookup values that represent permanent classifications.

Type 0 is rarely used in isolation because most dimension attributes do change over time. However, individual columns within a dimension table might use Type 0 treatment while other columns use different SCD types.

SCD Type 1

SCD Type 1 overwrites the existing dimension record with new values. When Hank moves from Seattle to Portland, you update his row directly. The dimension table always reflects the current state of the source system.

Before update				After update			
ID	Name	City	Email	ID	Name	City	Email
2001	Hank Miller	Seattle	hank.miller@example.com	2001	Hank Miller	Portland	hank.miller@example.com

Metric	Before Update	After Update (Type 1)
Total Purchases While in City	Seattle: \$10,000	Portland: \$10,000
Historical Accuracy Preserved?	✓ Yes (Seattle)	✗ No (value shifts to Portland)

Type 1 is appropriate when:

- **Historical values don't matter:** Correcting data entry errors, updating email addresses, or refreshing phone numbers.
- **Analysis focuses on current state:** Reports that answer "where do our customers live today?" rather than "where did they live when they made purchases?"
- **Simplicity is prioritized:** Type 1 requires no additional columns or versioning logic.

Important

Type 1 changes affect historical aggregations. If Hank purchased \$10,000 of products while in Seattle, those sales now appear attributed to Portland after the update. This behavior might misrepresent historical performance by region.

Consider using Type 1 for supplementary attributes that don't affect analytical accuracy, such as:

- Contact information (email, phone, address corrections).
- Descriptive text fields (product descriptions, customer notes).
- Error corrections in any attribute.

SCD Type 2

SCD Type 2 preserves complete history by inserting new rows for each version of a dimension member. When Hank moves to Portland, the existing Seattle row remains and a new Portland row is created. Both rows share the same natural key (customer ID) but have different surrogate keys.

Note

A natural key is also referred to as a **business key** or **source system key**. It represents the unique identifier from the original data source, such as customer ID, employee number, or product code.

Type 2 dimensions require additional tracking attributes:

Attribute	Purpose
Surrogate key	Unique identifier for each version
Validity start date	When this version became effective
Validity end date	When this version expired (often 12/31/9999 for current)
Current flag	Boolean indicating the active version

The following table shows how Hank Zoeng's customer record appears after moving from Seattle to Portland:

SK	CustomerID	Name	City	ValidFrom	ValidTo	Current
1001	C-555	Hank Zoeng	Seattle	2020-01-15	2024-03-01	FALSE
1002	C-555	Hank Zoeng	Portland	2024-03-01	9999-12-31	TRUE

Type 2 is appropriate when:

- **Historical accuracy matters:** Analyzing sales by the customer's location at the time of purchase.
- **Audit trails are required:** Tracking employee assignments, territory changes, or organizational restructuring over time.
- **Trend analysis is needed:** Understanding how dimension attributes evolved.

Type 2 increases complexity and storage requirements. Each version creates a new row, and fact tables reference specific versions through surrogate keys. Query complexity increases because analysts must understand the versioning model.

Tip

Reserve Type 2 for attributes where historical perspective affects business decisions. Don't apply Type 2 to every changing attribute—the overhead isn't justified for supplementary data like phone numbers.

SCD Type 3

SCD Type 3 tracks limited history by adding columns for previous values. Instead of creating new rows, you add a "previous city" column alongside "current city." This approach tracks only the most recent change, not the complete history.

The following table shows how Hank Zoeng's customer record appears after moving from Seattle to Portland:

CustomerID	Name	CurrentCity	PreviousCity
C-555	Hank Zoeng	Portland	Seattle

Type 3 is appropriate when:

- **Only the previous value matters:** Comparing current versus prior organizational structure.
- **Storage constraints exist:** When Type 2 versioning would create too many rows.
- **Analysis compares before and after:** Product reformulations, price tier changes, or policy updates.

Type 3 is the least common SCD type because:

- It captures only one prior value (or a fixed number of prior values).
- Adding columns for each tracked attribute can widen tables significantly.
- Semantic models can find Type 3 structures difficult to use.

Most scenarios that seem to need Type 3 are better served by Type 2, which provides complete history without schema changes for each attribute.

Compare SCD types

The following table summarizes when to select each SCD type:

Consideration	Type 0	Type 1	Type 2	Type 3
History preserved	Original only	None	Complete	Limited
Storage impact	Minimal	Minimal	High	Moderate
Query complexity	Simple	Simple	Complex	Moderate
Common use cases	Constants, audit requirements	Error corrections, contact updates	Sales attribution, territory tracking	Before/after comparisons
Fact table relationship	Natural key	Natural key	Surrogate key	Natural key

Apply mixed SCD strategies

Production dimension tables often combine multiple SCD types. A customer dimension might handle attributes differently based on their analytical significance:

- **Type 0:** Original registration date (never changes).
- **Type 1:** Email address, phone number (current state is sufficient).
- **Type 2:** Assigned sales region (historical accuracy required for sales attribution).

This hybrid approach balances storage efficiency with analytical requirements. When designing dimensions, evaluate each attribute individually to determine the appropriate change handling strategy.

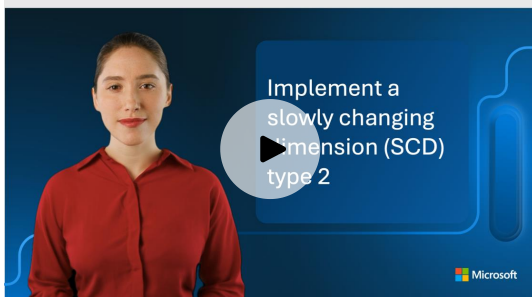
Understanding when to apply each SCD type enables you to implement dimensions that accurately support your organization's analytical requirements. The implementation details for loading and maintaining SCD tables in Azure Databricks are covered in later units.

7. Implement a slowly changing dimension (SCD) type 2

<https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/7-implement-slowly-changing-dimension-type-2>

Implement a slowly changing dimension (SCD) type 2

Completed



After selecting SCD Type 2 for a dimension, you need to implement the table structure and change capture logic in Azure Databricks. This unit focuses on creating SCD Type 2 tables and implementing the MERGE patterns that maintain version history as source data changes.

Create the SCD Type 2 table

The following SQL creates a customer dimension table with SCD Type 2 tracking columns:

```
CREATE TABLE sales.customer (  
  customer_sk BIGINT GENERATED ALWAYS AS IDENTITY,  
  customer_id STRING NOT NULL,  
  customer_name STRING,  
  email STRING,  
  city STRING,  
  region STRING,  
  valid_from TIMESTAMP NOT NULL,  
  valid_to TIMESTAMP NOT NULL,  
  is_current BOOLEAN  
)  
USING DELTA  
TBLPROPERTIES (  
  delta.enableChangeDataFeed = true  
);
```

The `GENERATED ALWAYS AS IDENTITY` clause creates an auto-incrementing surrogate key for each new row. Enabling the change data feed allows downstream processes to capture incremental changes efficiently.

Implement change capture with MERGE

The `MERGE` statement provides an efficient way to implement SCD Type 2 logic when processing updates from source systems. The statement handles inserts, updates, and the versioning logic in a single transaction.

The following pattern closes the current version and inserts the new version when changes occur:

```
MERGE INTO sales.customer AS target  
USING (
```

```

SELECT
    source.customer_id,
    source.customer_name,
    source.email,
    source.city,
    source.region,
    current_timestamp() AS valid_from,
    CAST('9999-12-31' AS TIMESTAMP) AS valid_to,
    true AS is_current
FROM staging.customers AS source
) AS updates
ON target.customer_id = updates.customer_id AND target.is_current = true
WHEN MATCHED AND (
    target.customer_name <> updates.customer_name OR
    target.email <> updates.email OR
    target.city <> updates.city OR
    target.region <> updates.region
) THEN UPDATE SET
    target.valid_to = current_timestamp(),
    target.is_current = false
WHEN NOT MATCHED THEN INSERT (
    customer_id, customer_name, email, city, region,
    valid_from, valid_to, is_current
) VALUES (
    updates.customer_id, updates.customer_name, updates.email,
    updates.city, updates.region, updates.valid_from,
    updates.valid_to, updates.is_current
);

-- Insert new versions for updated records
INSERT INTO sales.customer
SELECT
    s.customer_id,
    s.customer_name,
    s.email,
    s.city,
    s.region,
    current_timestamp() AS valid_from,
    CAST('9999-12-31' AS TIMESTAMP) AS valid_to,
    true AS is_current
FROM staging.customers s
JOIN sales.customer h
    ON s.customer_id = h.customer_id
    AND h.valid_to = current_timestamp()
    AND h.is_current = false;

```

Tip

Consider using Lakeflow Spark Declarative Pipelines with the `AUTO CDC` API for automated SCD Type 2 processing. This approach handles out-of-order records and simplifies SCD Type 2 table maintenance. See the Azure Databricks documentation for change data capture pipelines.

Query historical data

Once you've implemented an SCD Type 2 table, you can query data as it existed at any point in time. The query pattern depends on your analytical needs.

Point-in-time queries

To find the state of data at a specific moment, filter on the validity period:

```
SELECT customer_id, customer_name, city
FROM sales.customer
WHERE valid_from <= '2023-06-15 12:00:00'
    AND valid_to > '2023-06-15 12:00:00';
```

This query returns exactly one row per customer—the version that was valid at the specified timestamp.

Track record history

To view the complete history of changes for a specific entity:

```
SELECT customer_name, city, valid_from, valid_to
FROM sales.customer
WHERE customer_id = 'C-555'
ORDER BY valid_from;
```

This query reveals all versions of customer C-555, showing how their attributes changed over time.

Use Delta Lake time travel

Delta Lake provides built-in time travel capabilities that complement explicit SCD Type 2 table designs. You can query previous table versions using the `TIMESTAMP AS OF` or `VERSION AS OF` syntax:

```
-- Query table state from 7 days ago
SELECT * FROM sales.customer
TIMESTAMP AS OF '2024-01-15';

-- Query a specific table version
SELECT * FROM sales.customer
VERSION AS OF 42;
```

Important

Delta Lake time travel has a default retention of 7 days. For longer historical analysis, use explicit SCD columns (ValidFrom, ValidTo) rather than relying solely on time travel. Configure

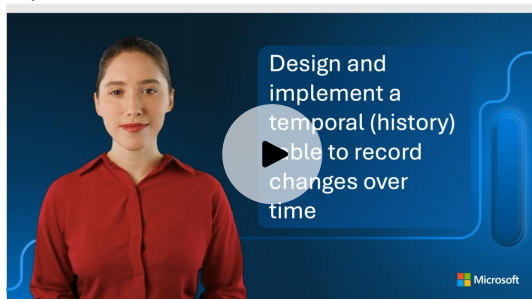
`delta.logRetentionDuration` and `delta.deletedFileRetentionDuration` if you need extended time travel access.

8. Design and implement a temporal (history) table to record changes over time

<https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/8-design-temporal-history-table>

Design and implement a temporal (history) table to record changes over time

Completed



Data pipelines often require tracking not just current values, but every change that occurs over time. Temporal tables provide a mechanism for recording row-level changes automatically, enabling point-in-time queries and complete audit trails. Unlike slowly changing dimensions that focus on business data history, temporal tables capture technical history for operational traceability and debugging.

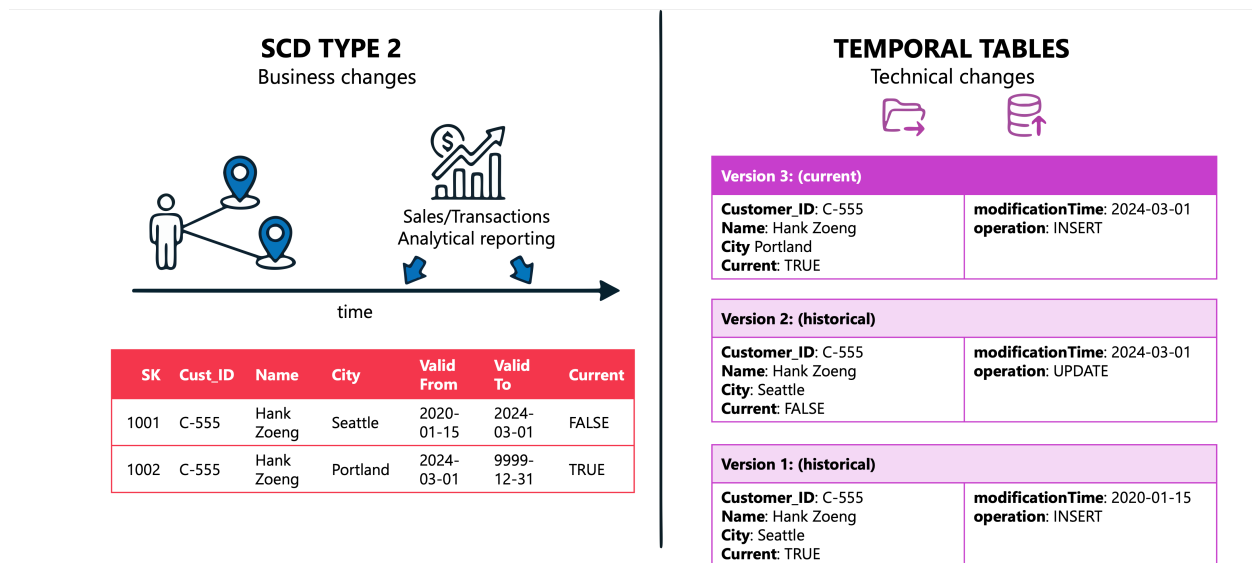
Understand temporal tables versus SCD Type 2

Temporal tables and SCD Type 2 dimensions serve different purposes, though both track historical changes. Understanding when to use each approach helps you design appropriate solutions.

SCD Type 2 tracks changes to business entities over time for analytical reporting. When a customer moves from Seattle to Portland, you want to attribute historical sales to the customer's location at the time of each transaction. SCD Type 2 maintains business-meaningful versions with explicit start and end dates.

Temporal (history) tables capture technical changes for operational needs. When troubleshooting why data loaded incorrectly, you need to see exactly what values existed at each point in time, which

files were processed, and when changes occurred. Temporal tables provide this forensic capability automatically.



Consider using temporal tables when you need to:

- Debug data ingestion issues by examining table state at specific timestamps
- Audit all changes for compliance or regulatory requirements
- Reproduce analyses or reports from a specific point in time
- Track operational metadata like load times and source file information

Enable change data feed for row-level tracking

Delta Lake's change data feed captures every row-level change automatically. When enabled, the runtime records change events with metadata indicating whether each row was inserted, deleted, or updated. Change data feed is disabled by default, so you must explicitly enable it on tables where you need this capability.

To enable change data feed on a new table:

```
CREATE TABLE bronze.sales_transactions (
  transaction_id STRING,
  customer_id STRING,
  amount DECIMAL(10,2),
  transaction_date TIMESTAMP,
  source_file STRING,
  load_timestamp TIMESTAMP
)
USING DELTA
TBLPROPERTIES (delta.enableChangeDataFeed = true);
```

For existing tables, enable change data feed with an `ALTER TABLE` statement:

```
ALTER TABLE bronze.sales_transactions
SET TBLPROPERTIES (delta.enableChangeDataFeed = true);
```

Important

Change data feed only records changes that occur after you enable it. Past changes to a table aren't captured. Enable this feature before loading production data.

Query table history

Once change data feed is enabled, you can query the change history to see exactly what modifications occurred. The change data feed adds three metadata columns to your data:

Column	Description
<code>_change_type</code>	The type of change: <code>insert</code> , <code>update_preimage</code> , <code>update_postimage</code> , or <code>delete</code>
<code>_commit_version</code>	The Delta log version containing the change
<code>_commit_timestamp</code>	The timestamp when the change was committed

Query changes within a specific time range:

```
SELECT
  transaction_id,
  amount,
  _change_type,
  _commit_version,
  _commit_timestamp
FROM table_changes('bronze.sales_transactions', '2026-01-01', '2026-01-31')
WHERE _change_type IN ('insert', 'update_postimage')
ORDER BY _commit_timestamp;
```

This query helps identify when specific records were modified, which is valuable for debugging data quality issues.

Use time travel for point-in-time queries

Delta Lake maintains a transaction log that enables querying data as it existed at any point in history. This capability supports debugging scenarios where you need to compare current data with a previous state.

Query data at a specific timestamp:

```
SELECT * FROM bronze.sales_transactions
```

```
TIMESTAMP AS OF '2026-01-15 14:30:00';
```

Query data at a specific version:

```
SELECT * FROM bronze.sales_transactions  
VERSION AS OF 42;
```

To view the complete history of operations performed on a table:

```
DESCRIBE HISTORY bronze.sales_transactions;
```

The history output includes the operation type, timestamp, user, and detailed metrics for each change.

Note

Time travel availability depends on retention settings. By default, Delta Lake retains data files for 7 days and log files for 30 days. Configure `delta.deletedFileRetentionDuration` and `delta.logRetentionDuration` for longer retention periods.

Persist change history for long-term auditing

Because Delta Lake's built-in time travel has retention limits, you should persist change data to a separate history table for long-term audit requirements. This approach creates a permanent record of all changes.

Use Structured Streaming to continuously capture changes:

```
(spark.readStream  
  .option("readChangeFeed", "true")  
  .option("ignoreDeletes", "false")  
  .table("bronze.sales_transactions")  
  .writeStream  
  .option("checkpointLocation", "/checkpoints/sales_history")  
  .trigger(availableNow=True)  
  .toTable("audit.sales_transactions_history")  
)
```

The `readChangeFeed` option instructs Spark to read the change data feed, which returns row-level change events including inserts, updates, and deletes. The `ignoreDeletes` option controls whether delete operations are included in the stream. Setting it to `false` ensures that delete records are captured in the history table, providing a complete audit trail of all data changes.

The history table contains every change event with full metadata, enabling queries across the entire lifetime of your data—not just the retention window.

Tip

Schedule the history capture process to run after each data load. Using `trigger(availableNow=True)` processes all available changes as a batch, which works well with scheduled workflows.

Configure retention for compliance requirements

For temporal tables supporting compliance or regulatory needs, configure retention settings to match your requirements:

```
ALTER TABLE bronze.sales_transactions
SET TBLPROPERTIES (
    delta.logRetentionDuration = 'interval 90 days',
    delta.deletedFileRetentionDuration = 'interval 90 days'
);
```

Both properties must be set together. The `logRetentionDuration` controls how long history metadata is retained, while `deletedFileRetentionDuration` controls how long deleted data files are preserved before `VACUUM` removes them.

Longer retention periods increase storage costs but provide extended time travel access and change data feed availability for auditing.

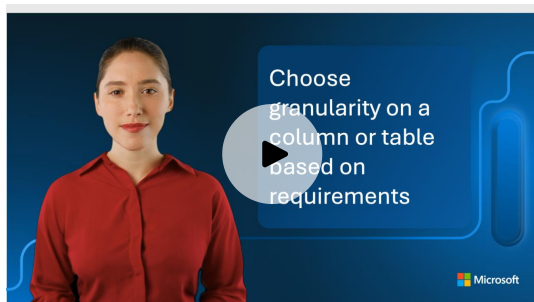
With temporal tables configured properly, you can trace every change to your data from ingestion through transformation, supporting both operational debugging and regulatory compliance requirements.

9. Choose granularity on a column or table based on requirements

<https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/9-choose-granularity>

Choose granularity on a column or table based on requirements

Completed



Granularity is one of the most critical design decisions you make when building a data model. It defines the **level of detail** captured in your tables and directly affects **storage costs**, **query performance**, and the **analytical questions** you can answer. As a data engineer, understanding granularity helps you design tables that meet business requirements while maintaining efficient operations.

This unit explains what granularity means in the context of dimensional modeling and how to evaluate different granularity options based on your organization's requirements.

Understand what granularity represents

Granularity, sometimes called "**grain**," describes the level of detail that each row in a table represents. In a **fact table**, the grain determines what business event or measurement each row captures. In a **dimension table**, the grain determines how entities are represented and versioned.

Consider an IoT sensor data scenario. You could design your fact table at different levels of granularity:



- **Millisecond-level granularity:** Each row represents a single sensor reading captured at the millisecond level. This captures the most detailed information, including precise timestamps, sensor values, and device identifiers for each measurement.

- **Second aggregated granularity:** Each row represents the average, minimum, or maximum sensor value for a device per second. Individual readings are summarized into per-second statistics.
- **Minute aggregated granularity:** Each row represents aggregated sensor statistics for a device during an entire minute. This provides the least detail but requires the least storage.

The grain you choose affects everything downstream. Once you **lose detail through aggregation**, you **can't recover it**. If your minute-aggregated table shows that sensor X had an average temperature of 75°C during a specific minute, you can't determine which milliseconds had temperature spikes or identify the exact moment an anomaly occurred.

Evaluate factors that influence granularity decisions

Choosing the appropriate granularity requires balancing multiple competing concerns. The right choice depends on your specific business context and technical constraints.

Business requirements drive the grain

The analytical questions your organization needs to answer should be the **primary factor** in determining granularity. Ask yourself: what's the **most detailed level of analysis** required?

If operations engineers need to detect equipment anomalies at the millisecond level, per-minute aggregates won't suffice. If management only needs hourly utilization summaries by production line, storing every sensor reading wastes resources.

Consider these scenarios:

Business requirement	Minimum granularity needed
Detect equipment vibration anomalies in real-time	Millisecond event level
Analyze temperature trends for predictive maintenance	Second aggregate level
Monitor hourly energy consumption per device	Minute aggregate level
Report daily production metrics by facility	Hourly aggregate level

Query patterns shape performance

The way users query your data influences the optimal granularity. When queries typically filter and aggregate to higher levels, storing overly fine-grained data creates **unnecessary processing overhead**. Each query must scan and aggregate more rows than necessary.

At the same time, storing data at too coarse a granularity prevents users from drilling down to answer detailed questions. Finding the balance requires understanding both current and anticipated query patterns.

Storage and compute costs accumulate

Finer granularity means **more rows**, which increases **storage costs**. It also increases **compute costs** because queries must process more data. In Azure Databricks, these costs compound over time as historical data accumulates.

A manufacturing facility with thousands of sensors capturing readings at millisecond intervals might generate billions of rows daily at the finest granularity. Aggregating to per-second or per-minute statistics could reduce row counts by 99% or more, translating directly to lower storage and query costs.

Compliance and audit requirements matter

Regulatory requirements sometimes mandate specific granularity levels. Financial services organizations often must retain **transaction-level detail** for audit purposes, regardless of storage costs. Healthcare organizations might need to track individual patient interactions for compliance reasons.

When compliance drives your granularity decision, document the requirements clearly. This justification helps when stakeholders question why you're storing more detail than analytical needs alone would require.

Apply granularity to fact tables

Fact tables benefit most from careful granularity decisions because they typically contain the **largest volumes of data**. The grain of a fact table is determined by the combination of its **dimension keys**.

A sales fact table with dimension keys for date, product, customer, and store has a grain of "one row per product per customer per store per day." Every unique combination of these dimension values creates a distinct row. Adding a time dimension key at the hour level would change the grain to "one row per product per customer per store per hour," significantly increasing row counts.

Date_Key	Hour_Key	Product_Key	Customer_Key	Store_Key	Quantity_Sold	Sales_Amount
20260101	09	101	5001	10	1	20.00
20260101	14	101	5001	10	1	20.00
20260101	11	102	5001	10	1	25.00
20260101	16	101	5002	10	3	60.00

Important

Declare your fact table grain explicitly before adding any columns. The grain statement serves as a contract that guides all subsequent design decisions. For example: "This fact table contains one row per sales order line item."

When designing fact tables, consider whether you need:

- **Atomic grain:** The most detailed level captured in the source system. Choosing atomic grain provides **maximum analytical flexibility** because you can always aggregate up, but never

disaggregate down.

- **Aggregate grain:** Pre-summarized data at a higher level than the source. Aggregate fact tables **improve query performance** for common analytical patterns but limit the questions users can answer.

Many organizations maintain both atomic and aggregate fact tables, using aggregate tables as performance optimization while preserving atomic detail for ad-hoc analysis. In Azure Databricks, **materialized views** can automatically maintain these aggregate tables, refreshing them as the underlying atomic data changes.

Apply granularity to dimension tables

Dimension tables also have a grain, though it works differently than fact table grain. The dimension grain determines how you represent and version your business entities.

For a customer dimension, you might choose:

- **One row per customer:** The current state of each customer, updated in place when attributes change (SCD Type 1).
- **One row per customer per version:** A new row for each significant change, preserving historical states (SCD Type 2).
- **One row per customer per time period:** Periodic snapshots capturing customer attributes at regular intervals.

The dimension grain you choose must **align with your fact table grain**. If your fact table tracks individual transactions with surrogate keys to customer dimension, your customer dimension grain must support that relationship. A transaction referencing customer version 5 must be able to find that specific version in the dimension table.

Balance trade-offs when selecting granularity

No single granularity choice is universally correct. Each decision involves trade-offs that you must evaluate against your specific requirements.

Consideration	Fine granularity	Coarse granularity
Analytical flexibility	High - can answer detailed questions	Low - limited to aggregate analysis
Storage costs	Higher - more rows to store	Lower - fewer rows
Query performance	Slower for aggregate queries	Faster for common patterns
Historical analysis	Supports point-in-time analysis	May lose historical detail
Compliance support	Better for audit requirements	May not meet regulations

When you're uncertain, **starting with finer granularity** is often safer. You can always create aggregate tables later for performance optimization, but you **can't recover detail** that was never

captured. However, this guidance assumes you have the budget and infrastructure to support finer-grained storage initially.

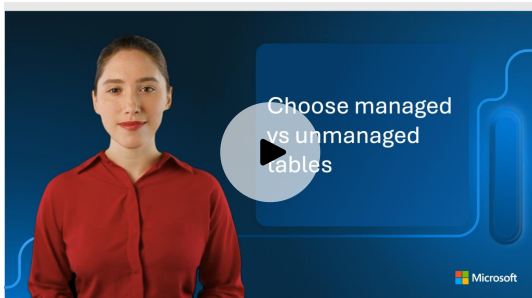
Understanding how to choose appropriate granularity enables you to design data models that support your organization's analytical needs while managing costs effectively. The implementation details for building and loading tables at your chosen granularity in Azure Databricks are covered in later units.

10. Choose managed vs unmanaged tables

<https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/10-choose-managed-vs-unmanaged-tables>

Choose managed vs unmanaged tables

Completed



When you create tables in Azure Databricks, you face a fundamental decision that affects data lifecycle management, governance, and operational flexibility. Understanding the differences between **managed tables** and **external tables** (sometimes called unmanaged tables) helps you choose the right approach for each scenario in your data platform.

In this unit, you learn how to distinguish between managed and external tables, and explore the factors that guide your choice.

Understand managed tables

Managed tables are fully controlled by Unity Catalog. This means Unity Catalog manages both the table metadata *and* the underlying data files in cloud storage. When you create a managed table without specifying a storage location, Azure Databricks automatically stores the data in a managed storage location associated with the containing schema, catalog, or metastore.

Because Unity Catalog has full control over managed tables, it can automatically optimize them for performance and cost. Several features are available exclusively for managed tables:

- **Predictive optimization** runs maintenance operations like compaction and vacuuming automatically
- **Automatic liquid clustering** intelligently selects and updates clustering keys based on query patterns
- **Metadata caching** provides faster query performance through in-memory caching
- **Automatic file cleanup** removes data files 8 days after you drop a table

Managed tables always use Delta Lake or Apache Iceberg format, ensuring ACID transactions and advanced data management capabilities.

Understand external tables

External tables (also called unmanaged tables) take a different approach. Unity Catalog manages only the table metadata, while you maintain control over the data files and their storage location. When you create an external table, you explicitly specify where the data resides using a `LOCATION` clause that points to an external location registered in Unity Catalog.

This separation means that when you drop an external table, Unity Catalog removes the metadata but leaves the data files intact in your storage location. You remain responsible for managing those files directly through your cloud provider.

External tables support a broader range of file formats beyond Delta Lake and Iceberg, including CSV, JSON, Avro, Parquet, ORC, and text files.

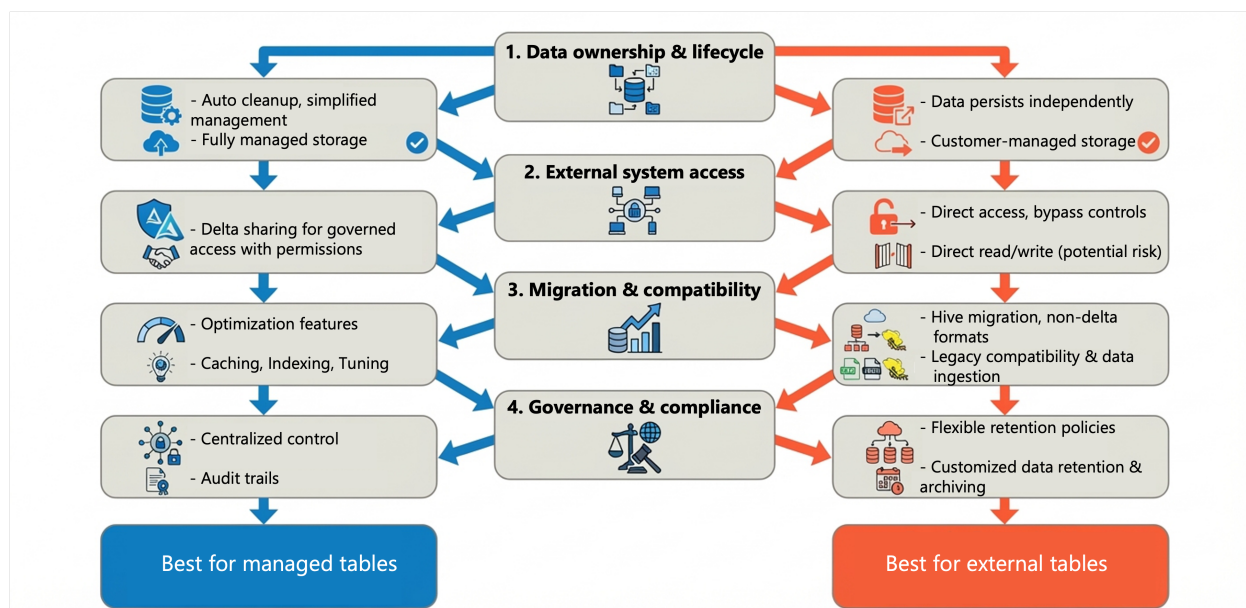
Compare key characteristics

The following table summarizes the key differences between managed and external tables:

Characteristic	Managed tables	External tables
Data storage location	Unity Catalog managed location	User-specified external location
Metadata management	Unity Catalog	Unity Catalog
Data file management	Unity Catalog	User managed
Supported formats	Delta Lake, Apache Iceberg	Delta, CSV, JSON, Avro, Parquet, ORC, text
Automatic optimization	Yes (predictive optimization)	No
Behavior on DROP TABLE	Files deleted after 8 days	Files remain in storage
UNDROP TABLE support	Yes (within 8 days)	No

Evaluate decision criteria

Choosing between managed and external tables depends on your specific requirements. Consider the following factors when making your decision.



Data ownership and lifecycle

If Azure Databricks should fully control the data lifecycle, including automatic cleanup when tables are dropped, use a managed table. This approach simplifies data management and reduces storage costs over time.

If the data must persist independently of the Azure Databricks workspace, or if you need to retain data after removing the table definition, use an external table. The data files remain available for other purposes even after the table is dropped.

External system access

When other tools or systems outside Azure Databricks need to read or write the data directly, external tables provide the necessary flexibility. However, consider that direct external access bypasses Unity Catalog access controls and auditing.

For sharing data across regions, cloud providers, or with external partners, **Delta Sharing** offers a governed approach that maintains security controls even with managed tables.

Migration and compatibility

External tables provide a practical path when migrating from Hive metastore to Unity Catalog. You can register existing data without moving files, allowing a quick upgrade. After migration, you can convert external tables to managed tables to take advantage of optimization features.

If you need to support non-Delta and non-Iceberg formats, external tables are your only option within Unity Catalog.

Governance and compliance

Consider your organization's data retention and compliance requirements. Managed tables offer stronger governance through centralized control, while external tables give you flexibility to manage data according to specific regulatory requirements or storage policies defined outside Azure Databricks.

Important

Avoid registering the same table as an external table in multiple metastores. This practice can lead to consistency issues because changes in one metastore don't automatically propagate to others.

Apply best practices

Databricks recommends managed tables for most use cases because they provide the best combination of governance, performance, and operational simplicity. Use external tables when your requirements specifically call for them.

When you do use external tables:

- Limit external access to reads when possible, with all writes happening through Azure Databricks
- Consider creating one external location per schema for clearer organization
- Plan to migrate external tables to managed tables when their original use case no longer applies

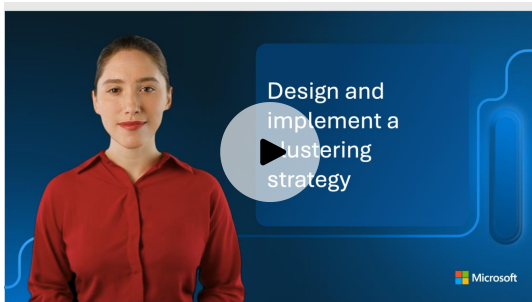
With this understanding of managed and external tables, you can make informed decisions that balance governance requirements, operational needs, and performance considerations for your data platform.

11. Design and implement a clustering strategy

<https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/11-design-clustering-strategy>

Design and implement a clustering strategy

Completed



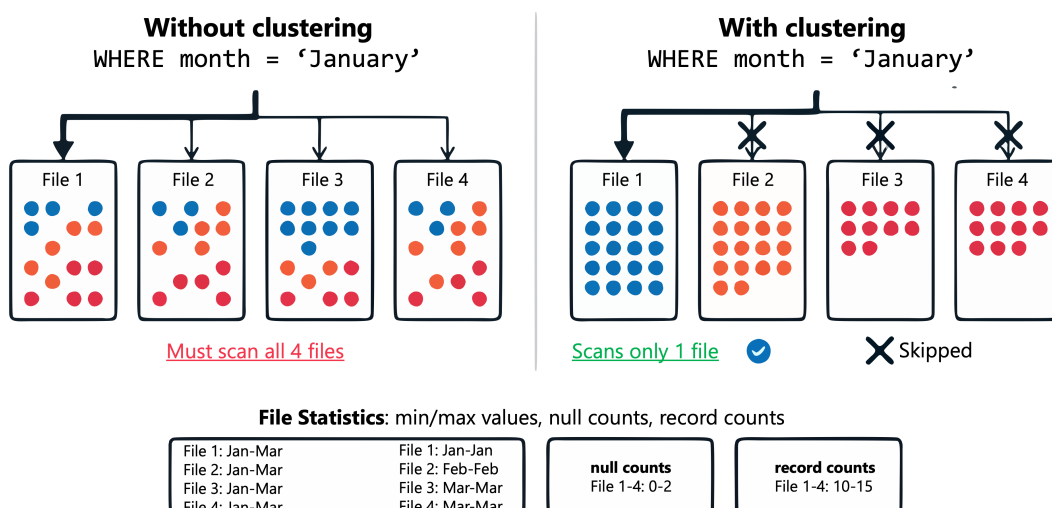
Optimizing how data is physically organized on disk directly affects **query performance** and **storage efficiency**. When you design a clustering strategy for your Delta Lake tables, you determine how the query engine locates and retrieves relevant data. The right approach can dramatically reduce the amount of data scanned during queries, lowering both response times and compute costs.

Azure Databricks provides several complementary techniques for data organization. Understanding when to apply **liquid clustering**, **Z-ordering**, or **deletion vectors** helps you make decisions that balance performance, maintenance effort, and compatibility with your existing tables.

Understand data skipping fundamentals

Before examining specific clustering techniques, it's important to understand the underlying mechanism they all leverage: **data skipping**. Delta Lake automatically collects statistics for each data file, including minimum and maximum values, null counts, and record counts for each column. When you execute a query with filter conditions, the query engine checks these statistics first and skips files that can't possibly contain matching records.

For example, if you query sales data for January 2026, and a particular file's statistics show it contains only records from March 2026, the engine skips that file entirely without reading its contents. This optimization happens automatically, but its effectiveness depends on how well your data is organized.



Data skipping becomes powerful when **related records are stored together** in the same files. If records for each month are scattered randomly across all files, the engine can't skip any files because

each file contains records from every month. Clustering techniques exist to solve exactly this problem.

Implement liquid clustering for new tables

Liquid clustering is the **recommended approach** for organizing data in new Delta Lake tables. Unlike traditional partitioning, liquid clustering allows you to **change your clustering strategy without rewriting existing data**. This flexibility is valuable when query patterns evolve or when you're uncertain which columns will be filtered most frequently.

To enable liquid clustering, use the **CLUSTER BY** clause when creating a table:

```
CREATE TABLE sales.transactions (  
  transaction_id STRING,  
  customer_id STRING,  
  product_category STRING,  
  region STRING,  
  transaction_date DATE,  
  amount DECIMAL(12,2)  
)  
CLUSTER BY (region, transaction_date);
```

You can specify **up to four clustering keys**. Choose columns that appear frequently in your query filters. The order of clustering keys matters less than selecting the right columns, as liquid clustering optimizes for all specified keys simultaneously.

After creating a clustered table, run the **OPTIMIZE** command to apply clustering to your data:

```
OPTIMIZE sales.transactions;
```

Liquid clustering is **incremental**. Each **OPTIMIZE** run clusters only the data that needs reorganization, making regular maintenance efficient. Databricks recommends scheduling **OPTIMIZE** jobs every one or two hours for tables with frequent inserts or updates.

Tip

For Unity Catalog managed tables, enable **predictive optimization** to automate **OPTIMIZE** operations. Azure Databricks analyzes your workload patterns and runs maintenance operations automatically.

Change clustering keys when needed

One significant advantage of liquid clustering over partitioning is the ability to modify your clustering strategy. If your query patterns change, update the clustering keys with an **ALTER TABLE** statement:

```
ALTER TABLE sales.transactions  
CLUSTER BY (customer_id, transaction_date);
```

Subsequent `OPTIMIZE` operations use the new clustering approach. Previously written data isn't automatically rewritten. To apply new clustering to all existing data, run:

```
OPTIMIZE sales.transactions FULL;
```

Enable automatic clustering key selection

Azure Databricks can intelligently select clustering keys based on your actual query patterns. Automatic liquid clustering analyzes workload history and chooses columns that provide the greatest data skipping benefit:

```
CREATE TABLE sales.orders CLUSTER BY AUTO;
```

You can enable automatic clustering on existing tables:

```
ALTER TABLE sales.transactions CLUSTER BY AUTO;
```

Automatic liquid clustering requires predictive optimization to be enabled. Azure Databricks periodically evaluates whether different clustering keys would improve performance and adjusts the strategy accordingly.

Migrate from Z-ordering to liquid clustering

Z-ordering is a legacy technique that organizes data to colocate related information in the same files. While liquid clustering is recommended for new tables, you'll encounter existing tables that use Z-ordering and need to understand both approaches.

Z-ordering uses the `OPTIMIZE` command with a `ZORDER BY` clause:

```
OPTIMIZE sales.legacy_transactions  
ZORDER BY (region, transaction_date);
```

Z-ordering differs from liquid clustering in several important ways:

Aspect	Liquid Clustering	Z-Ordering
Key flexibility	Change keys without data rewrite	Changing keys requires full rewrite
Write performance	Clustering on write for eligible operations	Applied only during OPTIMIZE
Concurrent writes	Row-level concurrency supported	Transaction-level concurrency
Compatibility	Requires Databricks Runtime 13.3 LTS or above	Supported in all runtimes

When you inherit a Z-ordered table, you have two options. Continue using Z-ordering for consistency, or migrate to liquid clustering for improved flexibility. To migrate, first enable liquid clustering on the existing unpartitioned table:

```
ALTER TABLE sales.legacy_transactions
CLUSTER BY (region, transaction_date);
```

Then run a full optimization to reorganize all existing data:

```
OPTIMIZE sales.legacy_transactions FULL;
```

Important

Liquid clustering is not compatible with partitioning. For partitioned tables, evaluate whether the benefits of liquid clustering justify migrating to an unpartitioned table structure.

Manage deletion vectors for efficient updates

Deletion vectors address a different aspect of data organization: how Delta Lake handles modified or deleted records. By default, when you update or delete a single row in a Parquet file, Delta Lake must **rewrite the entire file**. For tables with frequent modifications, this creates significant overhead.

With deletion vectors enabled, Delta Lake **marks modified rows as deleted** in a separate metadata file rather than rewriting the Parquet file immediately. Subsequent reads apply these deletions on the fly, combining the original file with the deletion vector to return accurate results.

Deletion vectors are enabled by default for new tables created with Databricks Runtime 14.1 or above. You can enable them explicitly on existing tables:

```
ALTER TABLE sales.transactions
SET TBLPROPERTIES ('delta.enableDeletionVectors' = true);
```

Deletion vectors are applied physically when the data files are rewritten during:

- An `OPTIMIZE` command
- Auto-compaction triggers
- A `REORG TABLE ... APPLY (PURGE)` operation

For compliance scenarios where you need to ensure deleted data is **physically removed**, run a `REORG` operation followed by `VACUUM`:

```
REORG TABLE sales.transactions APPLY (PURGE);
VACUUM sales.transactions;
```

Deletion vectors also enable **row-level concurrency**, allowing multiple concurrent transactions to modify different rows in the same table without conflicts. This capability is particularly valuable for tables with high update volumes.

12. Exercise - Design and Implement Data Modeling with Azure Databricks

<https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/12-exercise-design-implement-data-modeling-unity-catalog>

Exercise - Design and Implement Data Modeling with Azure Databricks

Completed

Now it's your chance to design and implement data modeling with Azure Databricks. In this lab, you design and implement a Delta Lake data model in Unity Catalog for a retail banking scenario, building a customer dimension with SCD Type 2 history tracking and a transaction fact table with liquid clustering. You apply Change Data Feed to build a queryable FCA compliance audit trail and use Delta Lake time travel to inspect and restore previous table versions.

Note

To complete this lab, you need an [Azure subscription](#) in which you have administrative access.

Launch the exercise and follow the instructions.

[Launch Exercise](#)

13. Knowledge check

<https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/13-knowledge-check>

Knowledge check

Completed

14. Summary

<https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/14-summary>

Summary

Completed

You've learned how to design and implement data modeling strategies that form the foundation of a well-architected Azure Databricks lakehouse. Starting with ingestion logic design, you explored how to match extraction types to source system capabilities and latency requirements. You evaluated ingestion tools—from Lakeflow Connect for managed connectors to Auto Loader for file-based streaming—and selected the right tool for each scenario.

Table format selection between Delta Lake and Apache Iceberg depends on your requirements: Delta Lake for deep Azure Databricks integration and Apache Iceberg when cross-platform compatibility is essential. You designed partitioning schemes that enable partition pruning for tables exceeding 1 TB. Slowly changing dimension types give you the tools to maintain historical accuracy where business requirements demand it.

Granularity decisions establish analytical flexibility. Starting with atomic grain preserves maximum detail, while aggregate tables optimize common query patterns. Liquid clustering organizes data for efficient data skipping, and managed tables simplify operations through predictive optimization.

Apply these concepts by evaluating your current data pipelines against the decision frameworks presented. Start with one table that would benefit from improved clustering, measure performance before and after, and expand your optimizations based on results.