DATABRICKS
DATA ENGINEERING ASSOCIATE
PRACTICE QUESTIONS

Created By - Sourav Banerjee

Questions

14.14. What is the syntax for the Spark SQL `UNION`, `MINUS`, and `INTERSECT` set operators?
14.15. What is the syntax for pivot tables?
14.16. What are higher order functions? (`FILTER`, `EXIST`, `TRANSFORM`, `REDUCE`)?
14.17. What is the syntax for `FILTER`?
14.18. What is the syntax for `EXIST`?
14.19. What is the syntax for `TRANSFORM`?
14.20. What is the syntax for `REDUCE`?

15. SQL UDFs and Control Flow

15.1. What is the syntax to define and register SQL UDFs? How do you then apply that function to the data?
15.2. How can you see where the function was registered and basic information about expected inputs and what is returned?
15.3. What are SQL UDFs governed by?
15.4. What permissions must a user have on the function to use a SQL UDF? Describe their scoping.
15.5. What is the syntax used for the evaluation of multiple conditional statements?
15.6. What is the syntax using SQL UDFs for custom control flow within SQL workloads?
15.7. What is the benefit of using SQL UDFs?

16. Python for Databricks SQL & Python Control Flow

16.1. What is the syntax to turn SQL queries into Python strings?
16.2. What is the syntax to execute SQL from a Python cell?
16.3. What function do you call to render a query the way it would appear in a normal SQL notebook?
16.4. What is the syntax to define a function in Python?
16.5. What is the syntax for f-strings?
16.6. How can f-strings be used for SQL queries?
16.7. What is the syntax for `if` / `else` clauses wrapped in a function?
16.8. What are the two methods for casting values to numeric types (int and float)?
16.9. What are `assert` statements and what is the syntax?
16.10. Why do we use `try` / `except` statements and what is the syntax?
16.11. What is the downside of using `try` / `except` statements?
16.12. What is the syntax for `try` / `except` statements where you return an informative error message?
16.13. How do you apply these concepts to execute SQL logic on Databricks, for example to avoid SQL injection attack?

17. Incremental Data Ingestion with Auto Loader

17.1. What is incremental ETL?
17.2. What is the purpose of Auto Loader?
17.3. What are the 4 arguments using Auto Loader with automatic schema inference and evolution?
17.4. How do you begin an Auto Loader stream?
17.5. What is the benefit of Auto Loader compared to structured streaming?
17.6. What keyword indicates that you're using Auto Loader rather than a traditional stream for ingesting?
17.7. What can you do once data has been ingested to Delta Lake with Auto Loader?
17.8. What is the `_rescued_data` column?
17.9. What is the data type encoded by Auto Loader for fields in a text-based file format?
17.10. Historically, what were the two inefficient ways to land new data?
17.11. Is there a delay when records are ingested with an Auto Loader query?
17.12. How do you track the ingestion progress?

18. Reasoning about Incremental Data with Spark Structured Streaming

18.1. What is Spark Structured Streaming?
18.2. What were the traditional approaches to data streams?
18.3. Describe the programming model for Structured Streaming.
18.4. Explain how Structured Streaming ensures end-to-end exactly-once fault-tolerance.

18.5. What is the syntax to read a stream?

18.6. How can you transform streaming data?

18.7. Give an example operation that is not possible when working with streaming data. What methods can you use to circumvent these exceptions?

18.8. How do you persist streaming results?

18.9. What are the 3 most important settings when writing a stream to Delta Lake tables?

18.10. What is the syntax to load data from a streaming temp view back to a DataFrame, and then query the table that we wrote out to?

19. Incremental Multi-Hop in the Lakehouse

19.1. Describe Bronze, Silver, and Gold tables

19.2. Bronze: what additional metadata could you add for enhanced discoverability?

19.3. Can you combine streaming and batch workloads in a unified multi-hop pipeline? What about ACID transactions?

19.4. Describe how you can configure a read on a raw JSON source using Auto Loader with schema inference. What is the `cloudFiles.schemaHints` option?

19.5. What happens with the ACID guarantees that Delta Lake brings to your data when you choose to merge this data with other data sources?

19.6. Describe what happens at the silver level, when we enrich our data.

19.7. Describe what happens at the Gold level.

19.8. What is `.trigger(availableNow=True)` and when is it used?

19.9. What are the important considerations for `complete` output mode with Delta?

19.10. Describe the two options to incrementally process data, either with a triggered option or a continuous option.

20. Using the Delta Live Tables UI

20.1. Describe how Delta Live Tables makes the ETL lifecycle easier.

20.2. Beyond transformations, how can you define your data in your code?

20.3. Describe why large scale ETL is complex when not using DLT.

20.4. How do you create and run a DLT pipeline in the DLT UI?

20.6. How do you explore the DAG?

21. SQL for Delta Live Tables

21.1. What is the syntax to do streaming with SQL for Delta Live tables? What's the keyword that shows you're using Delta Live Tables?

21.2. What is the syntax for declaring a bronze layer table using Auto Loader and DLT?

21.3. What keyword can you use for quality control? How do you reference DLT Tables/Views and streaming tables?

21.4. Declaring gold tables.

21.5. How can you explore the results in the UI?

22. Orchestrating Jobs with Databricks

22.1. What is a Job?

22.2. When scheduling a Job, what are the two options to configure the cluster where the task runs?

22.3. Running a Job and scheduling a Job

22.4. How do you repair an unsuccessful job run?

22.5. How can you view Jobs?

22.6. How can you view runs for a Job and the details of the runs?

22.7. How can you export job run results?

22.8. How do you edit a Job?

22.9. What does Maximum concurrent runs mean?

22.10. How can you set up alerts?

22.11. What is Job access control?

Answers

1. Databricks Architecture & Services

1.1. What are the two main components of the Databricks Architecture?

- The control plane consists of the backend services that Databricks manages in its own cloud account, aligned with the cloud service in use by the customer, AWS, Azure, or GCP. Though the majority of your data does not live there, some elements such as notebook commands & workspace configurations are stored in the control plane, and encrypted at rest. Through the control plane and the associated user interface and APIs that it provides, the customer can launch clusters, start Jobs, get results, and interact with table metadata.
- The data plane is where the data is processed. Following the classic data plane model, all compute

resources in the data plane reside in your own cloud account. The data plane hosts compute resources (clusters), connects to the main data stores back in DBFS, and optionally provides connections to external data sources, either within the same customer cloud account or elsewhere in the internet.

1.2. What are the three different Databricks services?

The Databricks web appplication delivers 3 different services catering to the specific needs of various personas:

- Databricks Data Science and Engineering workspace, also known as the Workspace.
- Databricks SQL provides a simple experience for users who want to run quick, ad hoc queries on the data lake, visualise query results, and create and share dashboards.
- Databricks Machine Learning is an integrated end to end machine learning environment useful for tracking experiments, training models, managing feature development, and serving features and models

1.3. What is a cluster?

- A Databricks cluster is a set of computation resources and configurations on which you run data engineering, data science, and data analytics workloads. These workloads (such as ETL pipelines, streaming analytics, ad hoc analytics, and ML) are run as a set of commands in a notebook or as a Job.
- The clusters live in the data plane with your organisation's cloud account (although cluster management is a function of the control plane).
- A cluster consists of one driver node and zero or more worker nodes.
- Databricks runs one executor per worker node. Therefore the terms executor and worker are used interchangeably in the context of the Databricks architecture. People often think of cluster size in terms of the number of workers, but there are other important factors to consider:
  - Total executor cores (compute): The total number of cores across all executors. This determines the maximum parallelism of a cluster.
  - Total executor memory: The total amount of RAM across all executors. This determines how much data can be stored in memory before spilling it to disk.
  - Executor local storage: The type and amount of local disk storage. Local disk is primarily used in the case of spills during shuffles and caching.
- There’s a balancing act between the number of workers and the size of worker instance types. A cluster with two workers, each with 40 cores and 100 GB of RAM, has the same compute and memory as an eight worker cluster with 10 cores and 25 GB of RAM.
- If you expect many re-reads of the same data, then your workloads may benefit from caching. Consider a storage optimized configuration with Delta Cache.

1.4. What are the two cluster types?

- All purpose clusters analyse data collaboratively using interactive notebooks. You can create all-purpose clusters from the Workspace or through the command line interface, or the REST APIs that Databricks provides. You can terminate and restart an all-purpose cluster. Multiple users can share all-purpose clusters to do collaborative interactive analysis.
- Jobs clusters run automated jobs in an expeditious and robust way. The Databricks Job scheduler

creates job clusters when you run Jobs and terminates them when the associated Job is complete. You cannot restart a job cluster. These properties ensure an isolated execution environment for each and every Job.

1.5. How long does Databricks retain cluster configuration information?

- Databricks retains cluster configuration information for up to 200 all-purpose clusters terminated in the last 30 days.
- Databricks retains cluster configuration information for up to 30 job clusters recently terminated by the job scheduler.
- To keep an all-purpose cluster configuration even after it has been terminated for more than 30 days, an administrator can pin a cluster to the cluster list.
- When you run a job on a New Job Cluster (which is usually recommended), the cluster terminates and is unavailable for restarting when the job is complete. On the other hand, if you schedule a job to run on an Existing All-Purpose Cluster that has been terminated, that cluster will autostart.

1.6. What are the three cluster modes?

Databricks supports three cluster modes: Standard, High Concurrency, and Single Node. Most regular users use Standard or Single Node clusters.

- Standard clusters are ideal for processing large amounts of data with Apache Spark.
- Single Node clusters are intended for jobs that use small amounts of data or non-distributed workloads such as single-node machine learning libraries.
- High Concurrency clusters are ideal for groups of users who need to share resources or run ad-hoc jobs. Administrators usually create High Concurrency clusters. Databricks recommends enabling autoscaling for High Concurrency clusters.

1.7. Cluster size and autoscaling

- When you create a Databricks cluster, you can either provide a fixed number of workers for the cluster or provide a minimum and maximum number of workers for the cluster.
- When you provide a fixed size cluster, Databricks ensures that your cluster has the specified number of workers.
- When you provide a range for the number of workers, Databricks chooses the appropriate number of workers required to run your job. This is referred to as autoscaling. With autoscaling, Databricks dynamically reallocates workers to account for the characteristics of your job. Certain parts of your pipeline may be more computationally demanding than others, and Databricks automatically adds additional workers during these phases of your job (and removes them when they're no longer needed). Autoscaling makes it easier to achieve high cluster utilization, because you don't need to provision the cluster to match a workload. This applies especially to workloads whose requirements change over time (like exploring a dataset during the course of a day), but it can also apply to a one-time shorter workload whose provisioning requirements are unknown. Autoscaling thus offers two advantages:

    - Workloads can run faster compared to a constant-sized under-provisioned cluster.
    - Autoscaling clusters can reduce overall costs compared to a statically-sized cluster.

- Depending on the constant size of the cluster and the workload, autoscaling gives you one or both of these benefits at the same time. The cluster size can go below the minimum number of workers selected when the cloud provider terminates instances. In this case, Databricks continuously retries to re-provision instances in order to maintain the minimum number of workers.

1.8. What is local disk encryption?

- Some instance types you use to run clusters may have locally attached disks. Databricks may store shuffle data or ephemeral data on these locally attached disks. To ensure that all data at rest is encrypted for all storage types, including shuffle data that is stored temporarily on your cluster's local disks, you can enable local disk encryption.
- When local disk encryption is enabled, Databricks generates an encryption key locally that is unique to each cluster node and is used to encrypt all data stored on local disks. The scope of the key is local to each cluster node and is destroyed along with the cluster node itself. During its lifetime, the key resides in memory for encryption and decryption and is stored encrypted on the disk. This feature is currently in
- Public Preview.

1.9. What are the cluster security modes?

If your workspace is enabled for Unity Catalog, you use security mode instead of High Concurrency cluster mode to ensure the integrity of access controls and enforce strong isolation guarantees. High Concurrency cluster mode is not available with Unity Catalog.

- None: No isolation. Does not enforce workspace-local table access control or credential passthrough. Cannot access Unity Catalog data.
- **Single User**: Can be used only by a single user (by default, the user who created the cluster). Other users cannot attach to the cluster. When accessing a view from a cluster with Single User security mode, the view is executed with the user's permissions. Single-user clusters support workloads using Python, Scala, and R. Init scripts, library installation, and DBFS FUSE mounts are supported on single-user clusters. Automated jobs should use single-user clusters.
- **User Isolation**: Can be shared by multiple users. Only SQL workloads are supported. Library installation, init scripts, and DBFS FUSE mounts are disabled to enforce strict isolation among the cluster users.
- Table ACL only (Legacy): Enforces workspace-local table access control, but cannot access Unity Catalog data.
- Passthrough only (Legacy): Enforces workspace-local credential passthrough, but cannot access Unity Catalog data.

The only security modes supported for Unity Catalog workloads are Single User and User Isolation. For more information, see Cluster security mode.

1.10. What is a Pool?

- To reduce cluster start time, you can attach a cluster to a predefined pool of idle instances, for the driver and worker nodes. The cluster is created using instances in the pools. If a pool does not have sufficient idle resources to create the requested driver or worker nodes, the pool expands by allocating new instances from the instance provider. When an attached cluster is terminated, the instances it used are returned to the pools and can be reused by a different cluster.
- If you select a pool for worker nodes but not for the driver node, the driver node inherit the pool from

the worker node configuration.

1.11. What happens when you terminate / restart / delete a cluster?

- **When a cluster terminates (i.e. stops)**:
  - all cloud resources currently in use are deleted. This means that associated VMs and operational memory will be purged, attached volume storage will be deleted, network connections between nodes will be removed. In short, all resources previously associated with the compute environment will be completely removed.
  - Any results that need to be persisted should be saved to a permanent location. You wont lose your code or data files that you saved appropriately.
  - Clusters will also terminate automatically due to inactivity assuming this setting is used.
  - Cluster configuration settings are maintained, and you can then use the restart button to deploy a new set of cloud resources using the same configuration.
- The **Restart** button allows us to manually restart our cluster. This can be useful if we need to completely clear out the cache on the cluster or wish to completely reset our compute environment.
- The **Delete** button will stop our cluster and remove the cluster configuration.
- Changing most settings by clicking on the `Edit` button will require running clusters to be restarted.

---

2. Basic Operations for Databricks Notebooks

2.1. Which magic command do you use to run a notebook from another notebook?

```
%run ../Includes/Classroom-Setup-1.2
```

2.2. What is Databricks utilities and how can you use it to list out directories of files from Python cells?

- Databricks notebooks provide a number of utility commands for configuring and interacting with the environment: dbutils docs. You can use `dbutils.fs.ls()` to list out directories of files from Python cells.

```
display(dbutils.fs.ls("/databricks-datasets"))
```

2.3. What function should you use when you have tabular data returned by a Python cell?

`display()`

---

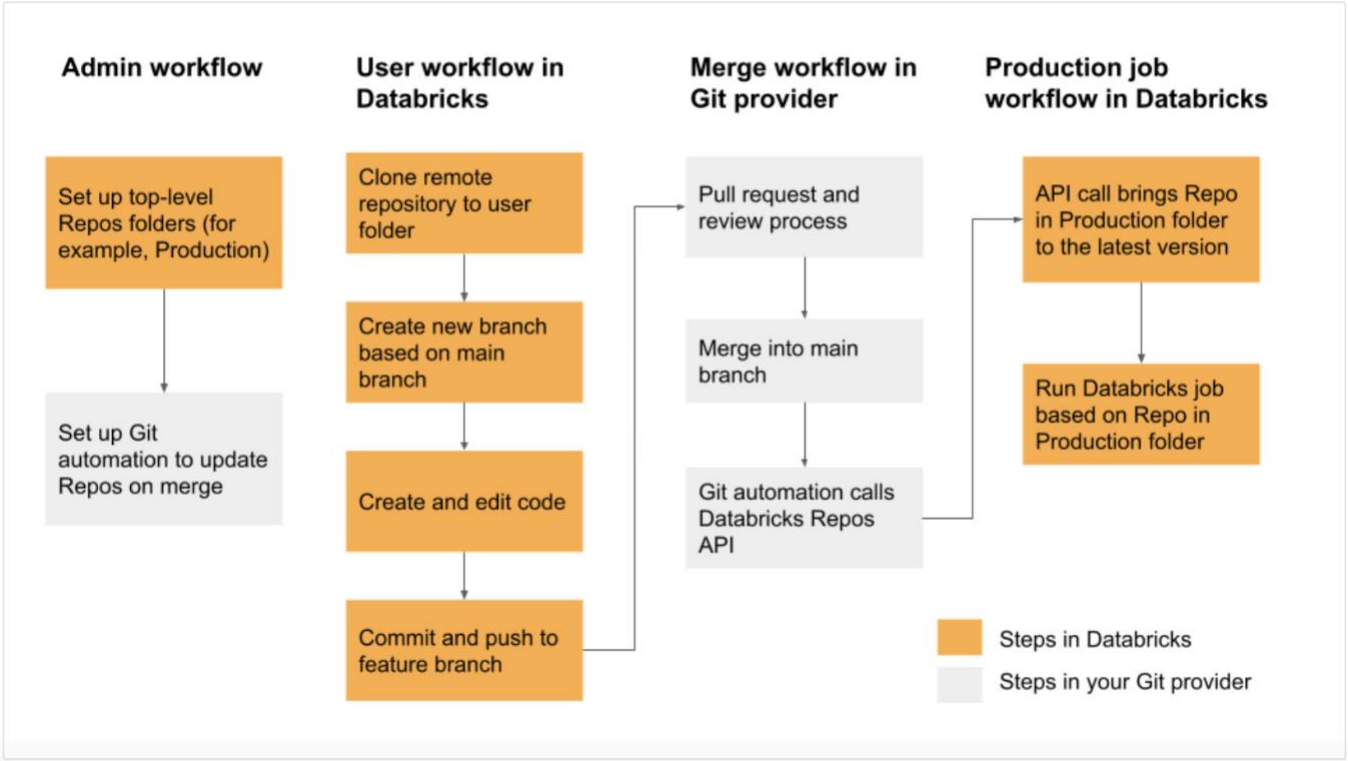
3. Git Versioning with Databricks Repos

3.1. What is Databricks Repos?

- Anywhere there is collaborative code development, there is a need for revision control, particularly when managing the deployment of that code into a production system. To support best practices for data science and engineering code development, Databricks Repos provides repository-level integration with Git providers, allowing you to work in an environment that is backed by revision control using Git.
- You can develop code in a Databricks notebook and sync it with a remote Git repository. Databricks Repos lets you use Git functionality such as cloning a remote repo, managing branches, pushing and pulling changes, and visually comparing differences upon commit.
- Notebooks have some basic revision control built in, but they're fairly basic and do not support the requirements of a full fledged software development life cycle. For example, there is no centralised, immutable history maintained. The history is attached to a single instance of a notebook, meaning that if you export a notebook or create a copy of it, the history's lost. The history can also easily be manipulated or deleted entirely by users. You cannot merge changes or create branches, or named tags. There are no external integration points to support a CI/CD pipeline.
- Databricks supports the following Git service providers: Azure DevOps, GitHub, GitLab, Bitbucket. Databricks currently does not support private Git servers.

3.2. What are the recommended CI/CD best practices to follow when developing with Repos?

- High level overview of industry best practices with respect to revision control and continuous integration and development workflows:



- **Admin workflow - set up top-level folders**

Databricks Repos have user-level folders and non-user top level folders. User-level folders are automatically created when users first clone a remote repository. You can think of Databricks Repos in user folders as “local checkouts” that are individual for each user and where users make changes to their code.

Admins can create non-user top level folders. The most common use case for these top level folders is to create Dev, Staging, and Production folders that contain Databricks Repos for the appropriate versions or branches for development, staging, and production. For example, if your company uses the Main branch for production, the Production folder would contain Repos configured to be at the Main branch.

Typically, permissions on these top-level folders are read-only for all non-admin users within the workspace.

- **Set up Git automation to update Databricks Repos on merge**

To ensure that Databricks Repos are always at the latest version, you can set up Git automation to call the Repos API 2.0. In your Git provider, set up automation that, after every successful merge of a PR into the main branch, calls the Repos API endpoint on the appropriate repo in the Production folder to bring that repo to the latest version. For example, on GitHub this can be achieved with GitHub Actions.

- **Developer workflow**

In your user folder in Databricks Repos, clone your remote repository. A best practice is to create a new feature branch or select a previously created branch for your work, instead of directly committing and pushing changes to the main branch. You can make changes, commit, and push changes in that branch. When you are ready to merge your code, create a pull request and follow the review and merge processes in Git.

Here is an example workflow. Note that this workflow requires that you have already set up your Git integration.

1. Clone your existing Git repository to your Databricks workspace.
2. Use the Repos UI to create a feature branch from the main branch. This example uses a single feature branch *feature-b* for simplicity. You can create and use multiple feature branches to do your work.
3. Make your modifications to Databricks notebooks and files in the Repo.
4. Commit and push your changes to your Git provider.
5. Coworkers can now clone the Git repository into their own user folder.
    1. Working on a new branch, a coworker makes changes to the notebooks and files in the Repo.
    2. The coworker commits and pushes their changes to the Git provider.
6. To merge changes from other branches or rebase the feature branch, you must use the Git command line or an IDE on your local system. Then, in the Repos UI, use the Git dialog to pull changes into the *feature-b* branch in the Databricks Repo.
7. When you are ready to merge your work to the main branch, use your Git provider to create a PR to merge the changes from *feature-b*.
8. In the Repos UI, pull changes to the main branch.

- **Production job workflow**

You can point a job directly to a notebook in a Databricks Repo. When a job kicks off a run, it uses the current version of the code in the repo.

If the automation is setup as described in Admin workflow, every successful merge calls the Repos API to update the repo. As a result, jobs that are configured to run code from a repo always use the latest version available when the job run was created.

---

4. Delta Lake and the Lakehouse

4.1. What is the definition of a Delta Lake?

- Delta Lake is the technology at the heart of the Databricks Lakehouse platform. It is an open source technology that enables building a data lakehouse on top of existing storage systems.
- While Delta Lake was initially developed exclusively by Databricks, it's been open sourced for almost 3 years. As with Apache Spark, Databricks is committed to working with the open source community to continue to develop and expand the functionality of Delta Lake.
- Delta Lake builds upon standard data formats. It is powered primarily by data stored in the Parquet format, one of the most popular open source format for working with Big Data. Additional metadata leverage other open source formats such as JSON.
- Delta Lake is optimized for cloud object storage. While it can be run on a number of different storage mediums, it has been specifically optimized for the behaviour of cloud based object storage. Object storage is cheap, durable, highly available and effectively infinitely scalable.
- Delta Lake is built for scalable metadata handling. A primary objective in designing Delta Lake was to solve the problem of quickly returning point queries in some of the world's largest and most rapidly changing datasets. Rather than locking you into a traditional database system where cost continue to scale as your data increases in size, Delta Lake decouples computing and storage costs and provides optimized performance on data regardless of scale. Decoupling storage and compute.

4.2. How does Delta Lake address the data lake pain points to ensure reliable, ready-to-go data?

- ACID Transactions – Delta Lake adds ACID transactions to data lakes. ACID stands for atomicity, consistency, isolation, and durability, which are a standard set of guarantees most databases are designed around. Since most data lakes have multiple data pipelines that read and write data at the same time, data engineers often spend a significant amount of time to make sure that data remains reliable during these transactions. With ACID transactions, each transaction is handled as having a distinct beginning and end. This means that data in a table is not updated until a transaction successfully completes, and each transaction will either succeed or fail fully.

  These transactional guarantees eliminate many of the motivations for having both a data lake and a data warehouse in an architecture. Appending data is easy, as each new write will create a new version of a data table, and new data won’t be read until the transaction completes. This means that data jobs that fail midway can be disregarded entirely. It also simplifies the process of deleting and updating records - many changes can be applied to the data in a single transaction, eliminating the possibility of incomplete deletes or updates.
- **Schema Management** – Delta Lake gives you the ability to specify and enforce your data schema. It automatically validates that the schema of the data being written is compatible with the schema of the table it is being written into. Columns that are present in the table but not in the data are set to null. If there are extra columns in the data that are not present in the table, this operation throws an exception. This ensures that bad data that could corrupt your system is not written into it. Delta Lake also enables you to make changes to a table’s schema that can be applied automatically.
- **Scalable Metadata Handling** – Big data is very large in size, and its metadata (the information about the files containing the data and the nature of the data) can be very large as well. With Delta Lake, metadata is processed just like regular data - with distributed processing.

- **Unified Batch and Streaming Data** – Delta Lake is designed from the ground up to allow a single system to support both batch and stream processing of data. The transactional guarantees of Delta Lake mean that each micro-batch transaction creates a new version of a table that is instantly available for insights. Many Databricks users use Delta Lake to transform the update frequency of their dashboards and reports from days to minutes while eliminating the need for multiple systems.
- **Data Versioning and Time Travel** – With Delta Lake, the transaction logs used to ensure ACID compliance create an auditable history of every version of the table, indicating which files have changed between versions. This log makes it easy to retain historical versions of the data to fulfill compliance requirements in various industries such as GDPR and CCPA. The transaction logs also include metadata like extended statistics about the files in the table and the data in the files. Databricks uses Spark to scan the transaction logs, applying the same efficient processing to the large amount of metadata associated with millions of files in a data lake.

4.3. Describe how Delta Lake brings ACID transactions to object storage

- Delta Lake brings ACID to cloud-based object storage. The main issues that ACID solves for data engineers are:
  - Difficult to append data: Delta Lake provides guaranteed consistency for the state at the time an append begins, as well as atomic transactions and high durability. As such, appends will not fail due to conflict, even when writing from many data sources simultaneously. This solves the problem of making it difficult to append data for certain data lake operations.
  - Difficult to modify existing data: upserts allow us to apply updates and deletes with simple syntax as a single atomic transaction.
  - Jobs failing mid way: changes won't be committed until a job has succeeded. Jobs will either fail or succeed completely.
  - Real time operations are not easy: Delta Lake allows atomic micro batched transaction processing in near real time through a tight integration with Structured Streaming, meaning that you can use both real time and batch operations in the same set of Delta Lake tables.
  - Costly to keep historical data versions: the transaction logs used to guarantee atomicity, consistency and isolation, allow snapshot queries which easily enables time travel on your Delta Lake tables.

4.4. Is Delta Lake the default for all tables created in Databricks?

- Yes, Delta Lakes is the default for all tables created in Databricks.

4.5. What data objects are in the Databricks Lakehouse?

- The Databricks Lakehouse architecture combines data stored with the Delta Lake protocol in cloud object storage with metadata registered to a metastore. There are five primary objects in the Databricks Lakehouse:
  - **Catalog**: a grouping of databases.
  - **Database** or schema: a grouping of objects in a catalog. Databases contain tables, views, and functions.
  - **Table**: a collection of rows and columns stored as data files in object storage.

    - View: a saved query typically against one or more tables or data sources.
    - Function: saved logic that returns a scalar value or set of rows.

4.6. What is a metastore?

- The metastore contains all of the metadata that defines data objects in the lakehouse. Databricks provides the following metastore options:
    - Unity Catalog: you can create a metastore to store and share metadata across multiple Databricks workspaces. Unity Catalog is managed at the account level.
    - **Hive metastore**: Databricks stores all the metadata for the built-in Hive metastore as a managed service. An instance of the metastore deploys to each cluster and securely accesses metadata from a central repository for each customer workspace.
    - External metastore: you can also bring your own metastore to Databricks.
- Regardless of the metastore used, Databricks stores all data associated with tables in object storage configured by the customer in their cloud account.

4.7. What is a catalog?

- A catalog is the highest abstraction (or coarsest grain) in the Databricks Lakehouse relational model. Every database will be associated with a catalog. Catalogs exist as objects within a metastore. Before the introduction of Unity Catalog, Databricks used a two-tier namespace. Catalogs are the third tier in the Unity Catalog namespacing model:

```
catalog_name.database_name.table_name
```

- The built-in Hive metastore only supports a single catalog, `hive_metastore`.

4.8. What is a Delta Lake table?

- A Databricks table is a collection of structured data. A Delta table stores data as a directory of files on cloud object storage and registers table metadata to the metastore within a catalog and schema.
- As Delta Lake is the default storage provider for tables created in Databricks, all tables created in Databricks are Delta tables, by default.
- Because Delta tables store data in cloud object storage and provide references to data through a metastore, users across an organization can access data using their preferred APIs; on Databricks, this includes SQL, Python, PySpark, Scala, and R.

4.9. How do relational objects work in Delta Live Tables?

- Delta Live Tables uses the concept of a “virtual schema” during logic planning and execution. Delta Live Tables can interact with other databases in your Databricks environment, and Delta Live Tables can publish and persist tables for querying elsewhere by specifying a target database in the pipeline configuration settings.
- All tables created in Delta Live Tables are Delta tables, and can be declared as either managed or unmanaged tables.
- While views can be declared in Delta Live Tables, these should be thought of as temporary views scoped to the pipeline.

- Temporary tables in Delta Live Tables are a unique concept: these tables persist data to storage but do not publish data to the target database.
- Some operations, such as `APPLY CHANGES INTO`, will register both a table and view to the database; the table name will begin with an underscore (`_`) and the view will have the table name declared as the target of the `APPLY CHANGES INTO` operation. The view queries the corresponding hidden table to materialize the results.

---

5. Managing Delta Tables

Note: all Delta Tables will be referred to as tables in this document.

5.1. What is the syntax to create a Delta Table?

```
CREATE TABLE students
  (id INT, name STRING, value DOUBLE);
```

```
CREATE TABLE IF NOT EXISTS students
  (id INT, name STRING, value DOUBLE)
```

5.2. What is the syntax to insert data?

Insert many records in a single transaction:

```
INSERT INTO students
VALUES
  (4, "Ted", 4.7),
  (5, "Tiffany", 5.5),
  (6, "Vini", 6.3)
```

Transactions run as soon as they're executed, and commit as they succeed.

This syntax is also valid but not good practice as each statement is processed as a separate transaction with its own ACID guarantees:

```
INSERT INTO students VALUES (1, "Yve", 1.0);
INSERT INTO students VALUES (2, "Omar", 2.5);
INSERT INTO students VALUES (3, "Elia", 3.3);
```

5.3. Are concurrent reads on Delta Lake tables possible?

Delta Lake guarantees that any read against a table will always return the most recent version of the table, and that you'll never encounter a state of deadlock due to ongoing operations. Table reads can never conflict with other operations, and the newest version of your data is immediately available to all clients that can query your lakehouse. Because all transaction information is stored in cloud object storage alongside your data files, concurrent reads on Delta Lake tables is limited only by the hard limits of object storage on cloud vendors.

```
SELECT * FROM students
```

5.4. What is the syntax to update particular records of a table?

Updating records provides atomic guarantees as well.

```
UPDATE students
SET value = value + 1
WHERE name LIKE "T%"
```

5.5. What is the syntax to delete particular records of a table?

`DELETE` statements are also atomic, so there's no risk of only partially succeeding when removing data from your data lakehouse. `DELETE` will always result in a single transaction, whether it removes one or many records.

```
DELETE FROM students
WHERE value > 6
```

5.6. What is the syntax for merge and what are the benefits of using merge?

Databricks uses the `MERGE` keyword to perform upserts, which allows updates, inserts, and other data manipulations to be run as a single command.

If you write 3 statements, one each to insert, update, and delete records, this would result in 3 separate transactions; if any of these transactions were to fail, it might leave our data in an invalid state. Instead, we combine these actions into a single atomic transaction, applying all 3 types of changes together.

`MERGE` statements must have at least one field to match on, and each `WHEN MATCHED` or `WHEN NOT MATCHED` optional clause can have any number of additional conditional statements.

```
MERGE INTO table_a a
USING table_b b
ON a.col_name=b.col_name
WHEN MATCHED AND b.col = X
  THEN UPDATE SET *
WHEN MATCHED AND a.col = Y
  THEN DELETE
WHEN NOT MATCHED AND b.col = Z
  THEN INSERT *
```

`update*` and `insert*` are used to update/insert all the columns in the target table with matching columns from the source data set. The equivalent Delta Lake APIs are `updateAll()` and `insertAll()`.

- **Deduplication case (python syntax):**

Let's say you want to backfill a `loan_delta` table with historical data on past loans. But some of the historical data may already have been inserted in the table, and you don't want to update those records because they may contain more up-to-date information. You can deduplicate by the `loan_id` while inserting by running the following merge operation with only the `insert` action (since the `update` action is optional):

```
# in python

(deltaTable
  .alias("t")
  .merge(historicalUpdates.alias("s"), "t.loan_id = s.loan_id")
  .whenNotMatchedInsertAll()
  .execute())
```

Refer to the documentation for more details: https://oreil.ly/XBag7

5.7. What is the syntax to delete a table?

Assuming that you have proper permissions on the target table, you can permanently delete data in the lakehouse using a `DROP TABLE` command. In a properly configured lakehouse, users should not be able to delete production tables.

```
DROP TABLE students
```

---

6. Advanced Delta Lake Features

6.1. What is Hive?

Databricks uses a Hive metastore by default to register databases, tables, and views. Apache Hive is a distributed, fault-tolerant data warehouse system that enables analytics at a massive scale, allowing users to read, write, and manage petabytes of data using SQL.

Hive is built on top of Apache Hadoop, which is an open-source framework used to efficiently store and process large datasets. As a result, Hive is closely integrated with Hadoop, and is designed to work quickly on petabytes of data. What makes Hive unique is the ability to query large datasets, leveraging Apache Tez or MapReduce, with a SQL-like interface.

6.2. What are the two commands to see metadata about a table?

Using `DESCRIBE EXTENDED` allows us to see important metadata about our table.

```
DESCRIBE EXTENDED students
```

`DESCRIBE DETAIL` is another command that allows us to explore table metadata.

```
DESCRIBE DETAIL students
```

6.3. What is the syntax to display the Delta Lake files?

A Delta Lake table is actually backed by a collection of files stored in cloud object storage. We can see the files backing our Delta Lake table by using a Databricks Utilities function.

```
%python
display(dbutils.fs.ls(f"{DA.paths.user_db}/students"))
```

`DESCRIBE DETAIL` allows us to see some other details about our Delta table, including the number of files.

```
DESCRIBE DETAIL students
```

6.4. Describe the Delta Lake files, their format and directory structure

Our directory contains a number of Parquet data files and a directory named `_delta_log`.

- Records in Delta Lake tables are stored as data in Parquet files.
- Transactions to Delta Lake tables are recorded in the `_delta_log`. It has all the metadata about what Parquet files are currently valid. Each transaction results in a new `JSON` file being written to the Delta

Lake transaction log.

We can peek inside the `_delta_log` to see more to see the transactions.

```
%python
display(dbutils.fs.ls(f"{DA.paths.user_db}/students/_delta_log"))
```

6.5. What does the query engine do using the transaction logs when we query a Delta Lake table?

Rather than overwriting or immediately deleting files containing changed data, Delta Lake uses the transaction log to indicate whether or not files are valid in a current version of the table. When we query a Delta Lake table, the query engine uses the transaction logs to resolve all the files that are valid in the current version, and ignores all other data files.

You can look at a particular transaction log and see if records were inserted / updated / deleted.

```
%python
display(spark.sql(f"SELECT * FROM
json.`{DA.paths.user_db}/students/_delta_log/00000000000000000007.json`"))
```

In the output, the `add` column contains a list of all the new files written to our table; the `remove` column indicates those files that no longer should be included in our table.

6.6. What commands do you use to compact small files and index tables?

Having a lot of small files is not very efficient, as you need to open them before reading them. Small files can occur for a variety of reasons; e.g. performing a number of operations where only one or several records are inserted.

Using the `OPTIMIZE` command allows you to combine files toward an optimal size (scaled based on the size of the table). It will replace existing data files by combining records and rewriting the results.

When executing `OPTIMIZE`, users can optionally specify one or several fields for `ZORDER` indexing. It speeds up data retrieval when filtering on provided fields by colocating data with similar values within data files. Co-locality is used by Delta Lake data-skipping algorithms to dramatically reduce the amount of data that needs to be read.

For example, if we know that the data analysts in our team query a lot of files by `id`, we can make the process more efficient by looking only at the `ids` they're interested in. It indexes on `id` and clusters the `ids` into separate files, so that we dont have to read every file when querying the data.

```
OPTIMIZE events

OPTIMIZE events WHERE date >= '2017-01-01'

OPTIMIZE events
WHERE date >= current_timestamp() - INTERVAL 1 day
ZORDER BY (eventType)
```

For more information about the `OPTIMIZE` command, see Optimize performance with file management.

6.7. How do you review a history of table transactions?

Because all changes to the Delta Lake table are stored in the transaction log, we can easily review the table history.

```
DESCRIBE HISTORY students
```

The `operationsParameters` column will let you review predicates used for updates, deletes, and merges.

The `operationMetrics` column indicates how many rows and files are added in each operation.

The `version` column designates the state of a table once a given transaction completes.

The `readVersion` column indicates the version of the table an operation executed against.

6.8. How do you query and roll back to previous table version?

- **Query:**

The transaction log provides us with the ability to query previous versions of our table. These time travel queries can be performed by specifying either the integer version or a timestamp.

```
SELECT *
FROM students VERSION AS OF 3
```

Note that we're not recreating a previous state of the table by undoing transactions against our current version; rather, we're just querying all those data files that were indicated as valid as of the specified version.

- **Rollback:**

Suppose that you accidentally delete all of your data (eg by typing `DELETE FROM students`, where we delete all the records in our table). Luckily, we can simply rollback this commit. Note that a `RESTORE` command is recorded as a transaction; you won't be able to completely hide the fact that you accidentally deleted all the records in the table, but you will be able to undo the operation and bring your table back to a desired state.

```
RESTORE TABLE students TO VERSION AS OF 8
```

6.9. What command do you use to clean up stale data files and what are the consequences of using this command?

Imagine that you `optimize` your data. You know that while all your data has been compacted, the data files from previous versions of your table are still being stored. You can remove these files and remove access to previous versions of the table by running `VACUUM` on the table.

While Delta Lake versioning and time travel are great for querying recent versions and rolling back queries, keeping the data files for all versions of large production tables around indefinitely is very expensive (and can lead to compliance issues if PII is present).

Databricks will automatically clean up stale files in Delta Lake tables. If you wish to manually purge old data files, this can be performed with the `VACUUM` operation.

By default, `VACUUM` will prevent you from deleting files less than 7 days old. This is because if you run `VACUUM` on a Delta table, you lose the ability to time travel back to a version older than the specified data retention period.

So to use this command you need to turn off the check to prevent premature deletion of data files, and make sure that logging of `VACUUM` commands is enabled. Finally, you can use the `DRY RUN` option of vacuum to print out all records to be deleted (useful to review files manually before deleting them)

```
SET spark.databricks.delta.retentionDurationCheck.enabled = false;
SET spark.databricks.delta.vacuum.logging.enabled = true;

VACUUM students RETAIN 0 HOURS DRY RUN
```

The cell above modifies some Spark configurations. The first command overrides the retention threshold check to allow us to demonstrate permanent removal of data. The second command sets `spark.databricks.delta.vacuum.logging.enabled` to `true` to ensure that the `VACUUM` operation is recorded in the transaction log.

Note that vacuuming a production table with a short retention can lead to data corruption and/or failure of long-running queries. This is for demonstration purposes only and extreme caution should be used when disabling this setting.

When running `VACUUM` and deleting files, we permanently remove access to versions of the table that require these files to materialize.

Because `VACUUM` can be such a destructive act for important datasets, it's always a good idea to turn the retention duration check back on.

```
SET spark.databricks.delta.retentionDurationCheck.enabled = true
```

The table history will indicate the user that completed the `VACUUM` operation, the number of files deleted, and log that the retention check was disabled during this operation.

Note that because Delta Cache stores copies of files queried in the current session on storage volumes deployed to your currently active cluster, you may still be able to temporarily access previous table versions (though systems should not be designed to expect this behavior). Restarting the cluster will ensure that these cached data files are permanently purged.

6.10. Using Delta Cache

- Using the Delta cache is an excellent way to optimize performance.
- Note: The Delta cache is *not* the same as caching in Apache Spark. One notable difference is that the Delta cache is stored entirely on the local disk, so that memory is not taken away from other operations within Spark. When enabled, the Delta cache automatically creates a copy of a remote file in local storage so that successive reads are significantly sped up.

---

7. Databases and Tables on Databricks

7.1. What is the syntax to create a database with default location (no location specified)?

```
CREATE DATABASE IF NOT EXISTS db_name_default_location;
```

7.2. What is the syntax to create a database with specified location?

```
CREATE DATABASE IF NOT EXISTS db_name_custom_location LOCATION
'path/db_name_custom_location.db';
```

7.3. How do you get metadata information of a database? Where are the databases located (difference between default vs custom location)

This command allows us to find the database location.

```
DESCRIBE DATABASE EXTENDED db_name;
```

Default location is under `dbfs:/user/hive/warehouse/` and the database directory is the name of the database with the `.db` extension, so so it is `dbfs:/user/hive/warehouse/db_name.db`. This is a directory that our database is tied to.

Whereas the location of the database with custom location is in the directory specified after the `LOCATION` keyword.

7.4. What's the best practice when creating databases?

Generally speaking, it's going to be best practice to declare a location for a given database. This ensures that you know exactly where all of the data and tables are going to be stored.

Note that in Databricks, the terms “schema” and “database” are used interchangeably (whereas in many relational systems, a database is a collection of schemas).

7.5. What is the syntax for creating a table in a database with default location and inserting data? What is the syntax for a table in a database with custom location?

- Creating a table in the database with **default location** and inserting data. The schema must be provided because there is no data from which to infer the schema.

```
USE db_name_default_location;

CREATE OR REPLACE TABLE managed_table_in_db_with_default_location (width INT, length
INT, height INT);
INSERT INTO managed_table_in_db_with_default_location
VALUES (3, 2, 1);
SELECT * FROM managed_table_in_db_with_default_location;
```

- Same syntax when creating a table in the database with **custom location** and inserting data. The schema must be provided because there is no data from which to infer the schema.

```
USE db_name_custom_location;

CREATE OR REPLACE TABLE managed_table_in_db_with_custom_location (width INT, length
INT, height INT);
INSERT INTO managed_table_in_db_with_custom_location
VALUES (3, 2, 1);
SELECT * FROM managed_table_in_db_with_custom_location;
```

Python syntax:

```
df.write.saveAsTable("table_name")
```

7.6. Where are managed tables located in a database and how can you find their location?

Databricks manages both the metadata and the data for a managed table. Managed tables are the default when creating a table. The data for a managed table resides in the `LOCATION` of the database it is registered to. This managed relationship between the data location and the database means that in order to move a managed table to a new database, you must rewrite all data to the new location.

You can use this command to find a table location within the database, both for default and custom location.

```
DESCRIBE EXTENDED managed_table_in_db;
```

- **Default location:**

By default, managed tables in a database without the location specified will be created in the `dbfs:/user/hive/warehouse/<database_name>.db/` directory.

Command to display the files:

```
%python
hive_root =  f"dbfs:/user/hive/warehouse"
db_name =    f"db_name_default_location.db"
table_name = f"managed_table_in_db_with_default_location"

tbl_location = f"{hive_root}/{db_name}/{table_name}"
print(tbl_location)

files = dbutils.fs.ls(tbl_location)
display(files)
```

- **Custom location:**

The managed table is created in the path specified with the `LOCATION` keyword during database creation. As such, the data and metadata for the table are persisted in a directory there.

```
%python

table_name = f"managed_table_in_db_with_custom_location"
tbl_location = f"{DA.paths.working_dir}/_custom_location.db/{table_name}"
print(tbl_location)

files = dbutils.fs.ls(tbl_location)
display(files)
```

7.7. What is the syntax to create an external table?

Databricks only manages the metadata for unmanaged (external) tables; when you drop a table, you do not affect the underlying data. Unmanaged tables will always specify a `LOCATION` during table creation; you can either register an existing directory of data files as a table or provide a path when a table is first defined.

Because data and metadata are managed independently, you can rename a table or register it to a new database without needing to move any data. Data engineers often prefer unmanaged tables and the flexibility they provide for production data.

```
USE db_name_default_location;

CREATE OR REPLACE TEMPORARY VIEW temp_delays USING CSV OPTIONS (
  path = '${da.paths.working_dir}/flights/departuredelays.csv',
  header = "true",
  mode = "FAILFAST" -- abort file parsing with a RuntimeException if any malformed
lines are encountered
);
CREATE OR REPLACE TABLE external_table LOCATION 'path/external_table' AS
  SELECT * FROM temp_delays;

SELECT * FROM external_table;
```

Python syntax:

```
df.write.option("path", "/path/to/empty/directory").saveAsTable("table_name")
```

7.8. What happens when you drop tables (difference between a managed and an unmanaged table)?

- **Managed tables:**

Databricks manages both the metadata and the data for a managed table; when you drop a table, you also delete the underlying data. Data analysts and other users that mostly work in SQL may prefer this behavior. Managed tables are the default when creating a table.

For managed tables, when dropping the table, the table's directory and its log and data files will be deleted, only the database directory remains.

```
DROP TABLE managed_table_in_db_with_default_location;
```

```
DROP TABLE managed_table_in_db_with_custom_location;
```

- **Unmanaged (external) tables:**

Databricks only manages the metadata for unmanaged (external) tables; when you drop a table, you do not affect the underlying data. Because data and metadata are managed independently, you can rename a table or register it to a new database without needing to move any data.

For production workloads, we will often want to define those tables as external. This will avoid the potential issue of dropping a production table, and avoid having to do an internal migration if we need to associate these data files with a different database or change a table name at a later point as we're working with this particular table. Data engineers often prefer unmanaged tables and the flexibility they provide for production data.

```
DROP TABLE external_table;
```

After executing the above, the table definition no longer exists in the metastore, but the underlying data remain intact.

Even though this table no longer exists in our database, we still have access to the underlying data files, meaning that we can still interact with these files directly, or we can define a new table using these files.

```
%python
tbl_path = f"{DA.paths.working_dir}/external_table"
files = dbutils.fs.ls(tbl_path)
display(files)
```

7.9. What is the command to drop the database and its underlying tables and views?

Using `cascade`, we will delete all the tables and views associated with a database.

```
DROP DATABASE db_name_default_location CASCADE;
DROP DATABASE db_name_custom_location CASCADE;
```

---

8. Views and CTEs on Databricks

8.1. How can you show a list of tables and views?

```
SHOW TABLES;
```

8.2. What is the difference between Views, Temp Views & Global Temp Views?

View	Persisted as an object in a database. Persisted across multiple sessions, just like a table. You can query views from any part of the Databricks product (permissions allowing). Creating a view does not process or write any data; only the query text (i.e. the logic) is registered to the metastore in the associated database against the source.
Temporary View	Limited scope and persistence and is not registered to a schema or catalog. In notebooks and jobs, temp views are scoped to the notebook or script level. Cannot be referenced outside of the notebook in which they are declared, and will no longer exist when the notebook detaches from the cluster In DB SQL, temp view is scoped to the query level. Multiple statements within the same query can use the temp view, but it cannot be referenced in other queries, even within the same dashboard. A temp view is not persisted across multiple sessions, including for the following scenarios where a new session may be created: restarting a cluster, detaching and reattaching to a cluster, installing a python package which in turn restarts the Python interpreter, or simply opening a new notebook.
Global Temporary View	Global temporary views are scoped to the cluster level and can be shared between notebooks or jobs that share computing resources. They are registered to a separate database, the global temp database (rather than our declared database), so it won't show up in our list of tables associated with our declared database. This database lives as part of the cluster, and as long as the cluster is on, then that global temp view will be available from any Spark session that connects to that cluster, or notebook attached to that cluster. Global temp views are lost when the cluster is restarted. Databricks recommends using views with appropriate table ACLs instead of global temporary views.

8.3. What is the syntax for each?

- Views

```
CREATE VIEW view_delays_abq_lax AS
  SELECT *
  FROM external_table
  WHERE origin = 'ABQ' AND destination = 'LAX';

SELECT * FROM view_delays_abq_lax;
```

- Temp views

```
CREATE TEMPORARY VIEW temp_view_delays_gt_120
AS SELECT * FROM external_table WHERE delay > 120 ORDER BY delay ASC;

SELECT * FROM temp_view_delays_gt_120;
```

- Show tables including views and temp views

```
SHOW TABLES;
```

- Global temp views

```
CREATE GLOBAL TEMPORARY VIEW global_temp_view_dist_gt_1000
AS SELECT * FROM external_table WHERE distance > 1000;

SELECT * FROM global_temp.global_temp_view_dist_gt_1000;
```

Note the `global_temp` database qualifer in the subsequent `SELECT` statement.

- Show tables for global temp views

```
SHOW TABLES IN global_temp;
```

8.4. Do views create underlying files?

No. Creating a view does not process or write any data; only the query text (i.e. the logic) is registered to the metastore in the associated database against the source.

8.5. Where are global temp views created?

They are registered to a separate database, the global temp database (rather than our declared database), so it won't show up in our list of tables associated with our declared database. This database lives as part of the cluster, and as long as the cluster is on, then that global temp view will be available from any Spark session that connects to that cluster, or notebook attached to the cluster.

8.6. What is the syntax to select from global temp views?

```
SELECT * FROM global_temp.name_of_the_global_temp_view;
```

8.7. What are CTEs? What is the syntax?

The CTE only lasts for the duration of the query. It helps making the code more readable.

```
WITH cte_table AS (
  SELECT
    col1,
    col2,
    col3
```

```
  FROM
    external_table
  WHERE
    col1 = X
  GROUP BY col2
)
SELECT
  *
FROM
  cte_table
WHERE
  col1 > X
  AND col2 = Y;
```

8.8. What is the syntax to make multiple column aliases using a CTE?

```
WITH flight_delays(
  total_delay_time,
  origin_airport,
  destination_airport
)AS(
  SELECT
    delay,
    origin,
    destination
  FROM
    external_table
)
SELECT
  *
FROM
  flight_delays
WHERE
  total_delay_time > 120
  AND origin_airport = "ATL"
  AND destination_airport = "DEN";
```

8.9. What is the syntax for defining a CTE in a CTE?

```
WITH lax_bos AS (
  WITH origin_destination (origin_airport, destination_airport) AS (
    SELECT
      origin,
```

```
      destination
    FROM
      external_table
  )
  SELECT
    *
  FROM
    origin_destination
  WHERE
    origin_airport = 'LAX'
    AND destination_airport = 'BOS'
)
SELECT
  count(origin_airport) AS `Total Flights from LAX to BOS`
FROM
  lax_bos;
```

8.10. What is the syntax for defining a CTE in a subquery?

```
SELECT
  max(total_delay) AS `Longest Delay (in minutes)`
FROM
  (
    WITH delayed_flights(total_delay) AS (
      SELECT
        delay
      FROM
        external_table
    )
    SELECT
      *
    FROM
      delayed_flights
  );
```

8.11. What is the syntax for defining a CTE in a subquery expression

```
SELECT
  (
    WITH distinct_origins AS (
      SELECT DISTINCT origin FROM external_table
    )
    SELECT
      count(origin) AS `Number of Distinct Origins`
    FROM
      distinct_origins
  ) AS `Number of Different Origin Airports`;
```

8.12. What is the syntax for defining a CTE in a `CREATE VIEW` statement?

```
CREATE OR REPLACE VIEW BOS_LAX
AS WITH origin_destination(origin_airport, destination_airport)
AS (SELECT origin, destination FROM external_table)
SELECT * FROM origin_destination
WHERE origin_airport = 'BOS' AND destination_airport = 'LAX';

SELECT count(origin_airport) AS `Number of Delayed Flights from BOS to LAX` FROM
BOS_LAX;
```

9. Extracting Data Directly from Files

9.1. How do you query data from a single file?

To query the data contained in a single file, execute the query with the following pattern:

```
SELECT * FROM file_format.`/path/to/file`
```

```
SELECT * FROM json.`${da.paths.datasets}/raw/events-kafka/001.json`
```

9.2. How do you query a directory of files?

Assuming all of the files in a directory have the same format and schema, all files can be queried simultaneously by specifying the directory path rather than an individual file.

```
SELECT * FROM json.`${da.paths.datasets}/raw/events-kafka`
```

9.3. How do you create references to files?

Additional Spark logic can be chained to queries against files. When we create a view from a query against a path, we can reference this view in later queries.

```
CREATE OR REPLACE TEMP VIEW events_temp_view
AS SELECT * FROM json.`${da.paths.datasets}/raw/events-kafka/`;

SELECT * FROM events_temp_view
```

9.4. How do you extract text files as raw strings?

When working with text-based files (which include JSON, CSV, TSV, and TXT formats), you can use the `text` format to load each line of the file as a row with one string column named `value`. This can be useful when data sources are prone to corruption and custom text parsing functions will be used to extract value from text fields.

```
SELECT * FROM text.`${da.paths.datasets}/raw/events-kafka/`
```

9.5. How do you extract the raw bytes and metadata of a file? What is a typical use case for this?

Some workflows may require working with entire files, such as when dealing with images or unstructured data. Using `binaryFile` to query a directory will provide: file metadata alongside the binary representation of the file contents. Specifically, the fields created will indicate the `path`, `modificationTime`, `length`, and `content`.

```
SELECT * FROM binaryFile.`${da.paths.datasets}/raw/events-kafka/`
```

10. Providing Options for External Sources

10.1. Explain why executing a direct query against CSV files rarely returns the desired result.

CSV files are one of the most common file formats, but, unlike JSON or Parquet, it's not a self describing file format. Executing a direct query against these files rarely returns the desired results. When executing a direct query, the header row can be extracted as a table row, all columns can be loaded as a single column, and the column can contain nested data that is being truncated.

```
SELECT * FROM csv.`${da.paths.working_dir}/sales-csv`
```

10.2. Describe the syntax required to extract data from most formats against external sources.

While Spark will extract some self-describing data sources efficiently using default settings, many formats will require declaration of schema or other options. While there are many additional configurations you can set while creating tables against external sources, the syntax below demonstrates the essentials required to extract data from most formats.

```
CREATE TABLE table_identifier (col_name1 col_type1, ...)
USING data_source
OPTIONS (key1 = "val1", key2 = "val2", ...)
LOCATION = path
```

The cell below demonstrates using Spark SQL DDL to create a table against an external CSV source.

```
CREATE TABLE sales_csv
  (order_id LONG, email STRING, transactions_timestamp LONG, total_item_quantity
INTEGER, purchase_revenue_in_usd DOUBLE, unique_items INTEGER, items STRING)
USING CSV
OPTIONS (
  header = "true",
  delimiter = "|"
)
LOCATION "${da.paths.working_dir}/sales-csv"
```

10.3. What happens to the data, metadata and options during table declaration for these external sources?

Note that no data has moved during table declaration. Similar to when we directly queried our files and created a view, we are still just pointing to files stored in an external location.

All the metadata and options passed during table declaration will be persisted to the metastore, ensuring that data in the location will always be read with these options.

10.4. Does the column order matter if additional csv data files are added to the source directory at a later stage?

When working with `CSV` s as a data source, it's important to ensure that column order does not change if additional data files will be added to the source directory. Because the data format does not have strong schema enforcement, Spark will load columns and apply column names and data types in the order specified during table declaration.

10.5. What is the syntax to show all of the metadata associated with the table definition?

Running `DESCRIBE EXTENDED` on a table will show all of the metadata associated with the table definition.

```
DESCRIBE EXTENDED sales_csv
```

Options passed during table declaration are included as `Storage Properties` , (e.g. specifying the pipe delimiter and presence of a header).

10.6. What are the limits of tables with external data sources?

Whenever we're defining tables or queries against external data sources, we cannot expect the performance guarantees associated with Delta Lake and Lakehouse. For example, while Delta Lake tables will guarantee that you always query the most recent version of your source data, tables registered against other data sources may represent older cached versions.

The cell below executes some logic that we can think of as just representing an external system directly updating the files underlying our table.

```
%python
(spark.table("sales_csv")
      .write.mode("append")
      .format("csv")
      .save(f"{DA.paths.working_dir}/sales-csv"))
```

10.7. How can you manually refresh the cache of your data?

At the time you query the data source, Spark automatically caches the underlying data in local storage. This ensures that on subsequent queries, Spark will provide the optimal performance by just querying this local cache.

Our external data source is not configured to tell Spark that it should refresh this data. We can manually refresh the cache of our data by running the `REFRESH TABLE` command. Note that refreshing our table will invalidate our cache, meaning that we'll need to rescan our original data source and pull all data back into memory. For very large datasets, this may take a significant amount of time.

```
REFRESH TABLE sales_csv
```

10.8. What is the syntax to extract data from SQL Databases?

Databricks has a standard JDBC driver for connecting with many flavors of SQL databases.

```
CREATE TABLE
USING JDBC
OPTIONS (
    url = "jdbc:{databaseServerType}://{jdbcHostname}:{jdbcPort}",
    dbtable = "{jdbcDatabase}.table", user = "{jdbcUsername}",

    password = "{jdbcPassword}"
)
```

In the code sample below, we'll connect with SQLite. Note that the backend-configuration of the JDBC server assume you are running this notebook on a single-node cluster. If you are running on a cluster with multiple workers, the client running in the executors will not be able to connect to the driver.

```
DROP TABLE IF EXISTS users_jdbc;

CREATE TABLE users_jdbc
USING JDBC
OPTIONS (
  url = "jdbc:sqlite:/${da.username}_ecommerce.db",
  dbtable = "users"
)
```

You can then query this table as if it was defined locally.

```
SELECT * FROM users_jdbc
```

Looking at the table metadata reveals that we have captured the schema information from the external system. Storage properties (which would include the username and password associated with the connection) are automatically redacted.

```
DESCRIBE EXTENDED users_jdbc
```

While the table is listed as `MANAGED`, listing the contents of the specified location confirms that no data is being persisted locally. Note that some SQL systems such as data warehouses will have custom drivers.

10.9. Explain the two basic approaches that Spark uses to interact with external SQL databases and their limits

Spark will interact with various external databases differently, but generally two approaches can be taken when working in external SQL databases.

You can move the entire source table(s) to Databricks and then executing logic on the currently active cluster. However, this can incur significant overhead because of network transfer latency associated with moving all data over the public internet

You can push down the query to the external SQL database and only transfer the results back to Databricks. However, this can incur significant overhead because the execution of query logic in source systems not optimized for big data queries.

---

11. Creating Delta Tables

11.1. What is a CTAS statement and what is the syntax?

`CREATE TABLE AS SELECT` statements create and populate Delta tables using data retrieved from an input query.

```
CREATE OR REPLACE TABLE sales AS
SELECT * FROM parquet.`${da.paths.datasets}/raw/sales-historical/`;

DESCRIBE EXTENDED sales;
```

11.2. Do CTAS support manual schema declaration?

CTAS statements automatically infer schema information from query results and do not support manual schema declaration. This means that CTAS statements are useful for external data ingestion from sources with well-defined schema, such as Parquet files and tables. CTAS statements also do not support specifying additional file options. This presents significant limitations when trying to ingest data from CSV files.

11.3. What is the syntax to overcome the limitation when trying to ingest data from CSV files?

To correctly ingest this `csv` data to a Delta Lake table, we'll need to use a reference to the files that allows us to specify options. We specify the options to a temporary view, and then use this temp view as the source for a CTAS statement to successfully register the Delta table.

```
CREATE OR REPLACE TEMP VIEW sales_tmp_vw
  (order_id LONG, email STRING, transactions_timestamp LONG, total_item_quantity
INTEGER, purchase_revenue_in_usd DOUBLE, unique_items INTEGER, items STRING)
USING CSV
OPTIONS (
  path = "${da.paths.datasets}/raw/sales-csv",
  header = "true",
  delimiter = "|"
);

CREATE TABLE sales_delta AS
  SELECT * FROM sales_tmp_vw;

SELECT * FROM sales_delta
```

11.4. How do you filter and rename columns from existing tables during table creation?

Simple transformations like changing column names or omitting columns from target tables can be easily accomplished during table creation.

```
CREATE OR REPLACE TABLE purchases AS
SELECT order_id AS id, transaction_timestamp, purchase_revenue_in_usd AS price
FROM sales;

SELECT * FROM purchases
```

Note that we could have accomplished the same goal with a view, as shown below.

```
CREATE OR REPLACE VIEW purchases_vw AS
SELECT order_id AS id, transaction_timestamp, purchase_revenue_in_usd AS price
FROM sales;

SELECT * FROM purchases_vw
```

11.5. What is a generated column and how do you declare schemas with generated columns?

Generated columns are a special type of column whose values are automatically generated based on a user-specified function over other columns in the Delta table.

As noted previously, CTAS statements do not support schema declaration. For example, a timestamp column can be some variant of a Unix timestamp, which may not be the most useful for analysts to derive insights. This is a situation where generated columns would be beneficial. You can also provide a descriptive column comment for the generated column.

In the example below, the `date` column is generated by converting the existing `transaction_timestamp` column to a timestamp, and then a date.

```
CREATE OR REPLACE TABLE purchase_dates (
  id STRING,
  transaction_timestamp STRING,
  price STRING,
  date DATE GENERATED ALWAYS AS (
    cast(cast(transaction_timestamp/1e6 AS TIMESTAMP) AS DATE))
    COMMENT "generated based on `transactions_timestamp` column")
```

Because `date` is a generated column, if we write to `purchase_dates` without providing values for the `date` column, Delta Lake automatically computes them.

The cell below configures a setting to allow for generating columns when using a Delta Lake `MERGE` statement.

```
SET spark.databricks.delta.schema.autoMerge.enabled=true;

MERGE INTO purchase_dates a
USING purchases b
ON a.id = b.id
WHEN NOT MATCHED THEN
  INSERT *
```

All dates will be computed correctly as data is inserted, although neither our source data or insert query specify the values in this field. As with any Delta Lake source, the query automatically reads the most recent snapshot of the table for any query; you never need to run `REFRESH TABLE`.

It's important to note that if a field that would otherwise be generated is included in an insert to a table, this insert will fail if the value provided does not exactly match the value that would be derived by the logic used to define the generated column.

11.6. What are the two types of table constraints and how do you display them?

Because Delta Lake enforces schema on write, Databricks can support standard SQL constraint management clauses to ensure the quality and integrity of data added to a table.

Databricks currently support two types of constraints: `NOT NULL` constraints, and `CHECK` constraints (Generated columns are a special implementation of `check` constraints).

In both cases, you must ensure that no data violating the constraint is already in the table prior to defining the constraint. Once a constraint has been added to a table, data violating the constraint will result in write failure.

Below, we'll add a `CHECK` constraint to the `date` column of our table. Note that `CHECK` constraints look like standard `WHERE` clauses you might use to filter a dataset. We can alter our purchase_dates table and add the constraint `valid_date` that checks whether `date` is greater than the string `'2020-01-01'`.

```
ALTER TABLE purchase_dates ADD CONSTRAINT valid_date CHECK (date > '2020-01-01');
```

Table constraints are shown in the `Table Properties` field (you'll need to scroll down to see it).

```
DESCRIBE EXTENDED purchase_dates
```

11.7. Which built-in Spark SQL commands are useful for file ingestion (for the select clause)?

Our `SELECT` clause leverages two built-in Spark SQL commands useful for file ingestion:

- `current_timestamp()` records the timestamp when the logic is executed;
- `input_file_name()` records the source data file for each record in the table

11.8. What are the three options when creating tables?

The `CREATE TABLE` clause contains several options:

- A `COMMENT` is added to allow for easier discovery of table contents
- A `LOCATION` is specified, which will result in an external (rather than managed) table
- The table is `PARTITIONED BY` a date column; this means that the data from each date will exist within its own directory in the target storage location

```
CREATE OR REPLACE TABLE users_pii
COMMENT "Contains PII"
LOCATION "${da.paths.working_dir}/tmp/users_pii"
PARTITIONED BY (first_touch_date)
AS
  SELECT *,
    cast(cast(user_first_touch_timestamp/1e6 AS TIMESTAMP) AS DATE) first_touch_date,
    current_timestamp() updated,
    input_file_name() source_file
  FROM parquet.`${da.paths.datasets}/raw/users-historical/`;

SELECT * FROM users_pii;
```

The metadata fields added to the table provide useful information to understand when records were inserted and from where. This can be especially helpful if troubleshooting problems in the source data becomes necessary. All of the comments and properties for a given table can be reviewed using `DESCRIBE TABLE EXTENDED`.

```
DESCRIBE EXTENDED users_pii
```

11.9. As a best practice, should you default to partitioned tables for most use cases when working with Delta Lake?

Most Delta Lake tables (especially small-to-medium sized data) will not benefit from partitioning. Because partitioning physically separates data files, this approach can result in a small files problem and prevent file compaction and efficient data skipping.

The benefits observed in Hive or HDFS do not translate to Delta Lake, and you should consult with an experienced Delta Lake architect before partitioning tables. As a best practice, you should default to non-partitioned tables for most use cases when working with Delta Lake.

11.10. What are the two options to copy Delta Lake tables and what are the use cases?

- `DEEP CLONE` fully copies data and metadata from a source table to a target. This copy occurs incrementally, so executing this command again can sync changes from the source to the target location. Because all the data files must be copied over, this can take quite a while for large datasets.

```
CREATE OR REPLACE TABLE purchases_clone
DEEP CLONE purchases
```

- If you wish to create a copy of a table quickly to test out applying changes without the risk of modifying the current table, `SHALLOW CLONE` can be a good option. `SHALLOW CLONE` just copIES the Delta

transaction logs, meaning that the data doesn't move.

```
CREATE OR REPLACE TABLE purchases_shallow_clone
SHALLOW CLONE purchases
```

In either case, data modifications applied to the cloned version of the table will be tracked and stored separately from the source. Cloning is a great way to set up tables for testing SQL code while still in development.

---

12. Writing to Delta Tables

12.1. What are the multiple benefits of overwriting tables instead of deleting and recreating tables?

We can use overwrites to atomically replace all of the data in a table. There are multiple benefits to overwriting tables instead of deleting and recreating tables:

- Overwriting a table is much faster because it doesn't need to list the directory recursively or delete any files;
- The old version of the table still exists and can be easily retrieved using Time Travel;
- It's an atomic operation. Concurrent queries can still read the table while you are deleting the table; Due
- to ACID transaction guarantees, if overwriting the table fails, the table will be in its previous state.

12.2. What are the two easy methods to accomplish complete overwrites?

1. `CREATE OR REPLACE TABLE` (CRAS) statements fully replace the contents of a table each time they execute. The table will be overwritten, contrary to a `CREATE IF NOT EXISTS` statement.

```
CREATE OR REPLACE TABLE events AS
SELECT * FROM parquet.`${da.paths.datasets}/raw/events-historical`
```

2. `INSERT OVERWRITE` provides a nearly identical outcome as above. This cell overwrites data in the sales table using the results of an input query executed directly on parquet files in the `sales-historical` dataset (data in the target table will be replaced by data from the query).

```
INSERT OVERWRITE sales
SELECT * FROM parquet.`${da.paths.datasets}/raw/sales-historical/`
```

12.3. What are the differences between the two?

`INSERT OVERWRITE` provides a nearly identical outcome as above: data in the target table will be replaced by data from the query. However, `INSERT OVERWRITE` can only overwrite an existing table, not create a new one like our CRAS statement; `INSERT OVERWRITE` can overwrite only with new records that match the current table schema (and thus can be a "safer" technique for overwriting an existing table without disrupting downstream consumers). Finally, it can overwrite individual partitions.

A primary difference has to do with how Delta Lake enforces schema on write. Whereas a CRAS statement will allow us to completely redefine the contents of our target table, `INSERT OVERWRITE` will fail if we try to change our schema (unless we provide optional settings).

The table history also records these two operations (`CRAS` statements and `INSERT OVERWRITE`) differently.

12.4. What is the syntax to atomically append new rows to an existing Delta table? Is the command idempotent?

We can use `INSERT INTO` to atomically append new rows to an existing Delta table. This allows for incremental updates to existing tables, which is much more efficient than overwriting each time.

Append new sale records to the `sales` table using `INSERT INTO`:

```
INSERT INTO sales
SELECT * FROM parquet.`${da.paths.datasets}/raw/sales-30m`
```

`INSERT INTO` does not have any built-in guarantees to prevent inserting the same records multiple times. Re-executing the above cell would write the same records to the target table, resulting in duplicate records.

12.5. What is the syntax for the the `MERGE` SQL operation and the benefits of using merge?

You can upsert data from a source table, view, or DataFrame into a target Delta table using the `MERGE` SQL operation. Delta Lake supports inserts, updates and deletes in `MERGE` , and supports extended syntax beyond the SQL standards to facilitate advanced use cases.

```
MERGE INTO target a
USING source b
ON {merge_condition}
WHEN MATCHED THEN {matched_action}
WHEN NOT MATCHED THEN {not_matched_action}
```

The main benefits of `MERGE` are: 1. updates, inserts, and deletes are completed as a single transaction; 2. multiple conditions can be added in addition to matching fields; 3. it provides extensive options for implementing custom logic

Below, we'll only update records if the current row has a `NULL` email and the new row does not. All unmatched records from the new batch will be inserted:

```
MERGE INTO users a
USING users_update b
ON a.user_id = b.user_id
WHEN MATCHED AND a.email IS NULL AND b.email IS NOT NULL THEN
  UPDATE SET email = b.email, updated = b.updated
WHEN NOT MATCHED THEN INSERT *
```

We're merging records from the users update view into the users table, matching records by user id. For each new record in the users update view, we check for rows in the users table with the same user id. If there's a match, and the email field in the users update view is not null, then we'll update the row in the user's dataset using the row in users update. If a new record in users update does not have the same user id as any of the existing records in the users table, this record will be inserted as a new row in the user's table.

12.6. How can you use merge for deduplication?

A common ETL use case is to collect logs or other every-appending datasets into a Delta table through a series of append operations. Many source systems can generate duplicate records. With merge, you can avoid inserting the duplicate records by performing an insert-only merge.

This optimized command uses the same `MERGE` syntax but only provided a `WHEN NOT MATCHED` clause. Below, we use this to confirm that records with the same `user_id` and `event_timestamp` aren't already in the `events` table. This prevents adding records that already exist in the events table.

```
MERGE INTO events a
USING events_update b
ON a.user_id = b.user_id AND a.event_timestamp = b.event_timestamp
WHEN NOT MATCHED AND b.traffic_source = 'email' THEN
  INSERT *
```

12.7. What is the syntax to have an idempotent option to incrementally ingest data from external systems?

`COPY INTO` provides SQL engineers an idempotent option to incrementally ingest data from external systems (idempotence is a property of some operations such that no matter how many times you execute them, you achieve the same result). This operation is potentially much cheaper than full table scans for data that grows predictably.

Note that this operation does have some expectations: data schema should be consistent and duplicate records should try to be excluded or handled downstream.

While we're showing simple execution on a static directory below, the real value is in multiple executions over time picking up new files in the source automatically.

```
COPY INTO sales
FROM "${da.paths.datasets}/raw/sales-30m"
FILEFORMAT = PARQUET
```

This incrementally loads from the directory of the `sales-30-m` dataset into the `sales` table specifying parquet as the file format. This can be part of our ingestion capabilities, or to get your Delta Tables out in another format for someone else.

Recent feature: a `validate` keyword now allows you to check that the data format of your source data is still in line with the data in your target table before you incrementally load files.

12.8. How is `COPY INTO` different than Auto Loader?

`COPY INTO` and Auto Loader are different. Very similar functionality but focused on a SQL analyst doing a batch execution, whereas Auto Loader requires Structured Streaming. They're similar, but different technologies.

---

13. Cleaning Data

13.1. Do COUNT and DISTINCT queries skip or count nulls?

Counts Nulls	Skips Nulls
`COUNT(*)` > special case that counts the total number of rows, including rows that are only `NULL` values	`COUNT(col_name)`
`DISTINCT(*)` > The presence of `NULL` is also taken as a `DISTINCT` record	`COUNT(DISTINCT(*))` > we count distinct values without `NULL` because `NULL` is not something we can count
`DISTINCT(col_name)` > The presence of `NULL` is also taken as a `DISTINCT` record	`COUNT(DISTINCT(col_name))` > we count distinct values without `NULL` because `NULL` is not something we can count

`NULL` is the absence of value, or the lack of value, therefore it is not something we can count.

13.2. What is the syntax to count null values?

```
SELECT * FROM table_name WHERE col_name IS NULL
```

OR

```
SELECT count_if(col_name IS NULL) AS new_col_name FROM table_name
```

Example:

```
SELECT
  count(user_id) AS total_ids,
  count(DISTINCT user_id) AS unique_ids,
  count(email) AS total_emails,
  count(DISTINCT email) AS unique_emails,
  count(updated) AS total_updates,
  count(DISTINCT(updated)) AS unique_updates,
  count(*) AS total_rows,
  count(DISTINCT(*)) AS unique_non_null_rows
FROM users_dirty
```

13.3. What is the syntax to count for distinct values in a table for a specific column?

```
SELECT COUNT(DISTINCT(col_1, col_2)) FROM table_name WHERE col_1 IS NOT NULL
```

13.4. What is the syntax to cast a column to valid timestamp?

```
SELECT datetime(col_name "HH:mm:ss") AS new_col_name FROM table_name
```

```
SELECT CAST(col_name_with_transformation AS timestamp) AS new_col_name
```

13.5. What is the syntax for regex?

```
SELECT regexp_extract(string_to_search , "regex_to_match",
optional_match_portion_to_be_returned) AS email_domain) FROM table_name
```

Example:

```
SELECT *,
  date_format(first_touch, "MMM d, yyyy") AS first_touch_date,
  date_format(first_touch, "HH:mm:ss") AS first_touch_time,
  regexp_extract(email, "(?<=@).+", 0) AS email_domain
FROM (
  SELECT *,
    CAST(user_first_touch_timestamp / 1e6 AS timestamp) AS first_touch
  FROM deduped_users
)
```

14. Advanced SQL Transformations

14.1. What is the syntax to deal with binary-encoded JSON values in a human readable format?

- For binary-encoded JSON values (e.g. Kafka data), you can cast the `key` and `value` as strings to look at these in a human-readable format.

```
CREATE OR REPLACE TEMP VIEW events_strings AS
  SELECT string(key), string(value)
  FROM events_raw;

SELECT * FROM events_strings
```

	key	value
1	UA000000107384208	{"device":"macOS","ecommerce":{},"event_name":"checkout","event_previous_timestamp":1593880801027797,"event_timestamp":1593880822506642,"geo":{"city":"Traverse City","state":"MI"},"items":[{"item_id":"M_STAN_T","item_name":"Standard Twin Mattress","item_revenue_in_usd":595.0,"price_in_usd":595.0,"quantity":1}],"traffic_source":"google","user_first_touch_timestamp":1593879413256859,"user_id":"UA000000107384208"}
2	UA000000107388621	{"device":"Windows","ecommerce":{},"event_name":"email_coupon","event_previous_timestamp":1593880770092554,"event_timestamp":1593880829320848,"geo":{"city":"Hickory","state":"NC"},"items":[{"coupon":"NEWBED10","item_id":"M_STAN_F","item_name":"Standard Full Mattress","item_revenue_in_usd":850.5,"price_in_usd":945.0,"quantity":1}],"traffic_source":"direct","user_first_touch_timestamp":1593879889503719,"user_id":"UA000000107388621"}
3	UA000000107397512	{"device":"Windows","ecommerce":{},"event_name":"main","event_timestamp":1593880824305898,"geo":{"city":"Fargo","state":"ND"},"items":[],"traffic_source":"facebook","user_first_touch_timestamp":1593880824305898,"user_id":"UA000000107397512"}
4	UA000000107369427	{"device":"Android","ecommerce":{},"event_name":"add_item","event_previous_timestamp":1593880753875794,"event_timestamp":1593880826675403,"geo":{"city":"Chicago","state":"IL"},"items":[{"item_id":"M_STAN_T","item_name":"Standard Twin Mattress","item_revenue_in_usd":595.0,"price_in_usd":595.0,"quantity":1}],"traffic_source":"facebook","user_first_touch_timestamp":1593877772975990,"user_id":"UA000000107369427"}

14.2. What is the Spark SQL functionality to directly interact with JSON data stored as strings?

- Spark SQL has built-in functionality to directly interact with JSON data stored as strings. We can use the `:` syntax to traverse nested data structures.

	key	value
1	UA000000107384208	{"device":"macOS","ecommerce":{},"event_name":"checkout","event_previous_timestamp":1593880801027797,"event_timestamp":1593880822506642,"geo":{"city":"Traverse City","state":"MI"},"items":[{"item_id":"M_STAN_T","item_name":"Standard Twin Mattress","item_revenue_in_usd":595.0,"price_in_usd":595.0,"quantity":1}],"traffic_source":"google","user_first_touch_timestamp":1593879413256859,"user_id":"UA000000107384208"}
2	UA000000107388621	{"device":"Windows","ecommerce":{},"event_name":"email_coupon","event_previous_timestamp":1593880770092554,"event_timestamp":1593880829320848,"geo":{"city":"Hickory","state":"NC"},"items":[{"coupon":"NEWBED10","item_id":"M_STAN_F","item_name":"Standard Full Mattress","item_revenue_in_usd":850.5,"price_in_usd":945.0,"quantity":1}],"traffic_source":"direct","user_first_touch_timestamp":1593879889503719,"user_id":"UA000000107388621"}
3	UA000000107397512	{"device":"Windows","ecommerce":{},"event_name":"main","event_timestamp":1593880824305898,"geo":{"city":"Fargo","state":"ND"},"items":[],"traffic_source":"facebook","user_first_touch_timestamp":1593880824305898,"user_id":"UA000000107397512"}
4	UA000000107369427	{"device":"Android","ecommerce":{},"event_name":"add_item","event_previous_timestamp":1593880753875794,"event_timestamp":1593880826675403,"geo":{"city":"Chicago","state":"IL"},"items":[{"item_id":"M_STAN_T","item_name":"Standard Twin Mattress","item_revenue_in_usd":595.0,"price_in_usd":595.0,"quantity":1}],"traffic_source":"facebook","user_first_touch_timestamp":1593877772975990,"user_id":"UA000000107369427"}

```
SELECT value:device, value:geo:city
FROM events_strings
```

	device	city
1	macOS	Traverse City
2	Windows	Hickory
3	Windows	Fargo
4	Android	Chicago
5	macOS	Fargo
6	Android	Tuttle

Truncated results, showing first 1000 rows.
Click to re-execute with maximum result limits.

Command took 0.60 seconds -- by lara.rachidi@databricks.com at 04/06/2022, 15:34:08 on data-engineering-training

14.3. What are struct types? What is the syntax to parse JSON objects into struct types with Spark SQL?

- Spark SQL also has the ability to parse JSON objects into struct types (a native Spark type with nested attributes) by using a `from_json` function. However, this `from_json` function requires a schema. To derive the schema of our the data, you can take a row example with no null fields, and use Spark SQL's `schema_of_json` function.
- In the example below, we copy and paste an example JSON row to the function and chain it into the

`from_json` function to cast our `value` field to a struct type.

Syntax:

```
CREATE OR REPLACE TEMP VIEW parsed_events AS
  SELECT from_json(value, schema_of_json('{insert_example_schema_here}')) AS json
  FROM events_strings;

SELECT * FROM parsed_events
```

Syntax applied to example:

```
CREATE OR REPLACE TEMP VIEW parsed_events AS
  SELECT from_json(value, schema_of_json('{"device":"Linux","ecommerce":
{"purchase_revenue_in_usd":1075.5,"total_item_quantity":1,"unique_items":1},"event_name
":"finalize","event_previous_timestamp":1593879231210816,"event_timestamp":159387933577
9563,"geo":{"city":"Houston","state":"TX"},"items":
[{"coupon":"NEWBED10","item_id":"M_STAN_K","item_name":"Standard King
Mattress","item_revenue_in_usd":1075.5,"price_in_usd":1195.0,"quantity":1}],"traffic_so
urce":"email","user_first_touch_timestamp":1593454417513109,"user_id":"UA00000010611617
6"}')) AS json
  FROM events_strings;

SELECT * FROM parsed_events
```

Table | Data Profile

	json
1	{"device": "macOS", "ecommerce": {"purchase_revenue_in_usd": null, "total_item_quantity": null, "unique_items": null}, "event_name": "checkout", "event_previous_timestamp": 1593880801027797, "event_timestamp": 1593880822506642, "geo": {"city": "Traverse City", "state": "MI"}, "items": [{"coupon": null, "item_id": "M_STAN_T", "item_name": "Standard Twin Mattress", "item_revenue_in_usd": 595, "price_in_usd": 595, "quantity": 1}], "traffic_source": "google", "user_first_touch_timestamp": 1593879413256859, "user_id": "UA000000107384208"}
2	{"device": "Windows", "ecommerce": {"purchase_revenue_in_usd": null, "total_item_quantity": null, "unique_items": null}, "event_name": "email_coupon", "event_previous_timestamp": 1593880770092554, "event_timestamp": 1593880829320848, "geo": {"city": "Hickory", "state": "NC"}, "items": [{"coupon": "NEWBED10", "item_id": "M_STAN_F", "item_name": "Standard Full Mattress", "item_revenue_in_usd": 850.5, "price_in_usd": 945, "quantity": 1}], "traffic_source": "direct", "user_first_touch_timestamp": 1593879889503719, "user_id": "UA000000107388621"}
3	{"device": "Windows", "ecommerce": {"purchase_revenue_in_usd": null, "total_item_quantity": null, "unique_items": null}, "event_name": "main", "event_previous_timestamp": null, "event_timestamp": 1593880824305898, "geo": {"city": "Fargo", "state": "ND"}, "items": [], "traffic_source": "facebook", "user_first_touch_timestamp": 1593880824305898, "user_id": "UA000000107397512"}

This is now a struct field. We have a temporary view `parsed_events` with a column we named `json`. The values in each record were correctly parsed and stored in a struct with the nested values.

14.4. Once a JSON string is unpacked to a struct type, what is the syntax to flatten the fields into columns? What is the syntax to interact with the subfields in a struct type?

- Once a JSON string is unpacked to a struct type, Spark supports `*` (star) unpacking to flatten fields into columns.

```
CREATE OR REPLACE TEMP VIEW new_events_final AS
  SELECT json.*
  FROM parsed_events;

SELECT * FROM new_events_final
```

Table | Data Profile

	device	ecommerce	event_name	event_previous_timestamp	event_timestamp	geo
1	macOS	{"purchase_revenue_in_usd": null, "total_item_quantity": null, "unique_items": null}	checkout	1593880801027797	1593880822506642	{"city": "Travers
2	Windows	{"purchase_revenue_in_usd": null, "total_item_quantity": null, "unique_items": null}	email_coupon	1593880770092554	1593880829320848	{"city": "Hickory
3	Windows	{"purchase_revenue_in_usd": null, "total_item_quantity": null, "unique_items": null}	main	null	1593880824305898	{"city": "Fargo",
4	Android	{"purchase_revenue_in_usd": null, "total_item_quantity": null, "unique_items": null}	add_item	1593880753875794	1593880826675403	{"city": "Chicag
5	macOS	{"purchase_revenue_in_usd": null, "total_item_quantity": null, "unique_items": null}	checkout	1593880741311315	1593880830140019	{"city": "Fargo",
6	Android	{"purchase_revenue_in_usd": 1195, "total_item_quantity": 1, "unique_items": 1}	finalize	1593876666372094	1593880830038130	{"city": "Tuttle",

Truncated results, showing first 1000 rows.
Click to re-execute with maximum result limits.

Command took 0.54 seconds -- by lara.rachidi@databricks.com at 04/06/2022, 15:36:02 on data-engineering-training

14.5. What is the syntax to deal with nested struct types?

- Spark SQL has robust syntax for working with complex and nested data types.
- Looking at the fields in the **`events`** table, we see that the `ecommerce` field is a struct that contains a double and 2 longs.

```
DESCRIBE events
```

Table | Data Profile

	col_name	data_type	comment
1	device	string	
2	ecommerce	struct<purchase_revenue_in_usd:double,total_item_quantity:bigint,unique_items:bigint>	
3	event_name	string	
4	event_previous_timestamp	bigint	
5	event_timestamp	bigint	
6	geo	struct<city:string,state:string>	
7	items	array<struct<coupon:string,item_id:string,item_name:string,item_revenue_in_usd:double,price_in_usd:double,quantity:bigint>>	
8	traffic_source	string	
9	user_first_touch_timestamp	bigint	
10	user_id	string	
11			
12	# Partitioning		
13	Not partitioned		

We can interact with the subfields in this field using standard `.` syntax similar to how we might traverse nested data in JSON.

Let's select a subfield `purchase_revenue_in_usd` of the `ecommerce` column. This returns a new column with the values for the subfield extracted from the ecommerce column.

```
SELECT ecommerce.purchase_revenue_in_usd
FROM events
WHERE ecommerce.purchase_revenue_in_usd IS NOT NULL
```

Table | Data Profile

	purchase_revenue_in_usd
1	1795
2	1045
3	535.5
4	1095
5	940.5
6	1995

Truncated results, showing first 1000 rows.
Click to re-execute with maximum result limits.

Command took 0.86 seconds -- by lara.rachidi@databricks.com at 04/06/2022, 15:38:54 on data-engineering-training

14.6. What is the syntax for exploding arrays of structs?

The `items` field in the `events` table is an array of structs. Spark SQL has a number of functions specifically to deal with arrays. The `explode` function lets us put each element in an array on its own row.

```
SELECT user_id, event_timestamp, event_name, explode(items) AS item
FROM events
```

Table | Data Profile

	user_id	event_timestamp	event_name	item
1	UA000000106466918	1593595620017592	add_item	{"coupon": null, "item_id": "M_STAN_Q", "item_name": "Standard Queen Mattress", "item_revenue_in_usd": 1045, "price_in_usd": 1045, "quantity": 1}
2	UA000000106541016	1593617822519726	guest	{"coupon": null, "item_id": "M_STAN_T", "item_name": "Standard Twin Mattress", "item_revenue_in_usd": 595, "price_in_usd": 595, "quantity": 1}
3	UA000000106537762	1593623248104571	finalize	{"coupon": null, "item_id": "M_PREM_Q", "item_name": "Premium Queen Mattress", "item_revenue_in_usd": 1795, "price_in_usd": 1795, "quantity": 1}
4	UA000000106544798	1593619763160248	guest	{"coupon": null, "item_id": "M_STAN_Q", "item_name": "Standard Queen Mattress", "item_revenue_in_usd": 1045, "price_in_usd": 1045, "quantity": 1}
5	UA000000106527304	1593733998028702	cart	{"coupon": "NEWBED10", "item_id": "M_PREM_T", "item_name": "Premium Twin Mattress", "item_revenue_in_usd": 985.5, "price_in_usd": 1095, "quantity": 1}
6	UA000000106506669	1593817553389549	cart	{"coupon": "NEWBED10", "item_id": "M_STAN_T", "item_name": "Standard Twin Mattress", "item_revenue_in_usd": 535.5, "price_in_usd": 595, "quantity": 1}

Truncated results, showing first 1000 rows.
Click to re-execute with maximum result limits.

14.7. What is the syntax to collect arrays?

- The `collect_set` function can collect unique values for a field, including fields within arrays.
- The `flatten` function allows multiple arrays to be combined into a single array.
- The `array_distinct` function removes duplicate elements from an array.

We combine these queries to create a simple table that shows the unique collection of actions and the items in a user's cart.

```
SELECT user_id,
  collect_set(event_name) AS event_history,
  array_distinct(flatten(collect_set(items.item_id))) AS cart_history
FROM events
GROUP BY user_id
```

Table | Data Profile

	user_id	event_history	cart_history
1	UA000000102368951	["checkout", "cart", "add_item", "cc_info", "finalize", "register", "shipping_info", "mattresses"]	["M_PREM_K", "M_STAN_F"]
2	UA000000102377152	["mattresses", "cart", "cc_info", "finalize", "register", "shipping_info", "checkout"]	["M_STAN_F"]
3	UA000000102388362	["finalize"]	["M_STAN_F", "P_DOWN_S"]
4	UA000000102392998	["mattresses"]	[]
5	UA000000102435405	["checkout", "cart", "add_item", "cc_info", "guest", "finalize", "shipping_info", "mattresses"]	["M_STAN_T"]
6	UA000000102462564	["mattresses", "login", "add_item", "cart", "cc_info", "finalize", "shipping_info", "checkout"]	["M_STAN_T", "P_FOAM_S", "M_STAN_Q"]

Truncated results, showing first 1000 rows.
Click to re-execute with maximum result limits.

Command took 3.05 seconds -- by lara.rachidi@databricks.com at 04/06/2022, 15:39:55 on data-engineering-training

14.8. What is the syntax for an `INNER JOIN`?

By default, the join type is `INNER`. That means the results will contain the intersection of the two sets, and any rows that are not in both sets will not appear.

The SQL `JOIN` clause is used to combine records from two or more tables in a database. A `JOIN` is a means for combining fields from two tables by using values common to each.

Here we chain a join with a lookup table to an **`explode`** operation to grab the standard printed item name.

```
CREATE OR REPLACE VIEW sales_enriched AS
SELECT *
FROM (
  SELECT *, explode(items) AS item
  FROM sales) a
INNER JOIN item_lookup b
ON a.item.item_id = b.item_id;

SELECT * FROM sales_enriched
```

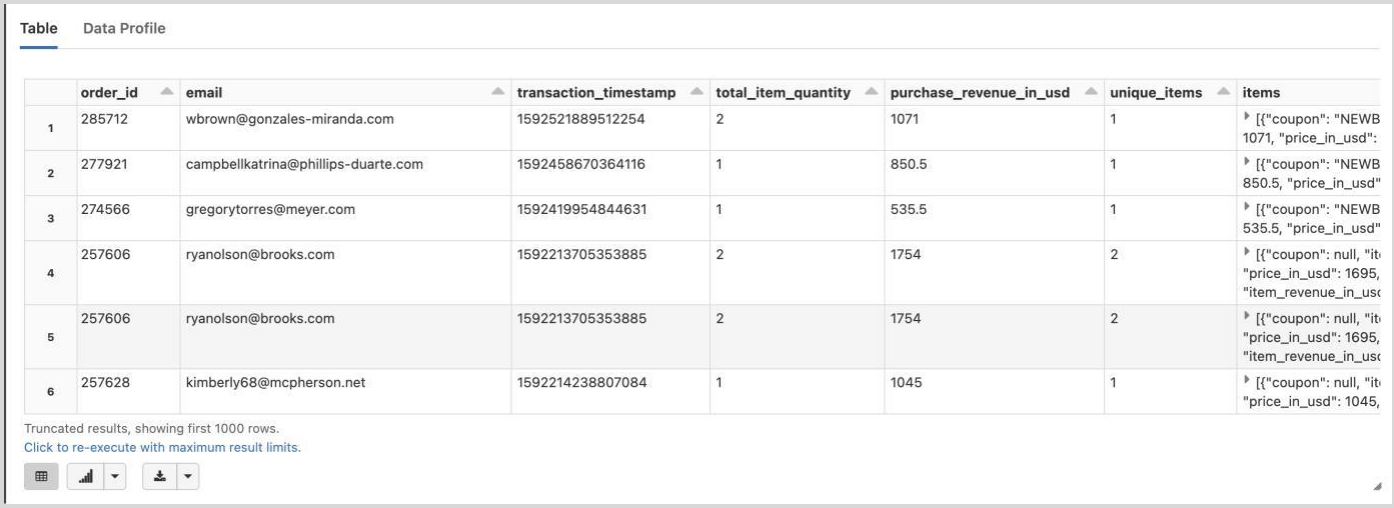


Table | Data Profile

	order_id	email	transaction_timestamp	total_item_quantity	purchase_revenue_in_usd	unique_items	items
1	285712	wbrown@gonzales-miranda.com	1592521889512254	2	1071	1	[{"coupon": "NEWB 1071, "price_in_usd":
2	277921	campbellkatrina@phillips-duarte.com	1592458670364116	1	850.5	1	[{"coupon": "NEWB 850.5, "price_in_usd"
3	274566	gregorytorres@meyer.com	1592419954844631	1	535.5	1	[{"coupon": "NEWB 535.5, "price_in_usd"
4	257606	ryanolson@brooks.com	1592213705353885	2	1754	2	[{"coupon": null, "it "price_in_usd": 1695, "item_revenue_in_usc
5	257606	ryanolson@brooks.com	1592213705353885	2	1754	2	[{"coupon": null, "it "price_in_usd": 1695, "item_revenue_in_usc
6	257628	kimberly68@mcpherson.net	1592214238807084	1	1045	1	[{"coupon": null, "it "price_in_usd": 1045,

Truncated results, showing first 1000 rows.
Click to re-execute with maximum result limits.

14.9. What is the syntax for an outer join?

The joined table retains each row—even if no other matching row exists. Outer joins subdivide further into left outer joins, right outer joins, and full outer joins, depending on which table's rows are retained: left, right, or both.

Conceptually, a full outer join combines the effect of applying both left and right outer joins. Where rows in the FULL OUTER JOINed tables do not match, the result set will have NULL values for every column of the table that lacks a matching row. For those rows that do match, a single row will be produced in the result set (containing columns populated from both tables).

For example, this allows us to see each employee who is in a department and each department that has an employee, but also see each employee who is not part of a department and each department which doesn't have an employee.

Example of a full outer join (the `OUTER` keyword is optional):

```
SELECT *
FROM employee FULL OUTER JOIN department
  ON employee.DepartmentID = department.DepartmentID;
```

14.10. What is the syntax for a left/right join?

The result of a left outer join (or simply left join) for tables A and B always contains all rows of the "left" table (A), even if the join-condition does not find any matching row in the "right" table (B). This means that if the `ON` clause matches 0 (zero) rows in B (for a given row in A), the join will still return a row in the result (for that row)—but with NULL in each column from B. A left outer join returns all the values from an inner join plus all values in the left table that do not match to the right table, including rows with NULL (empty) values in the link column.

Example of a left outer join (the `OUTER` keyword is optional):

```
SELECT *
FROM employee
LEFT OUTER JOIN department ON employee.DepartmentID = department.DepartmentID;
```

A right outer join (or right join) closely resembles a left outer join, except with the treatment of the tables reversed. Every row from the "right" table (B) will appear in the joined table at least once. If no matching row from the "left" table (A) exists, NULL will appear in columns from A for those rows that have no match in B.

A right outer join returns all the values from the right table and matched values from the left table (NULL in the case of no matching join predicate). For example, this allows us to find each employee and his or her department, but still show departments that have no employees.

Below is an example of a right outer join (the `OUTER` keyword is optional):

```
SELECT *
FROM employee RIGHT OUTER JOIN department
  ON employee.DepartmentID = department.DepartmentID;
```

14.11. What is the syntax for an anti-join?

Anti-join between two tables returns rows from the first table where no matches are found in the second table. It is opposite of a semi-join.

Below is an example of a left anti-join. It is the exact same as a left join except for the WHERE clause. This is what differentiates it from a typical left join.

The query below is finding all customers that did not have a matching `cse_id` in the `customer_success_engineer` table. By setting the `cse_id` column in the example above to null, it is finding all rows in the left table that did not have a matching record (a null value) in the table on the right.

```
SELECT
  *
FROM customers a
LEFT JOIN customer_success_engineer b
  ON a.assigned_cse_id = b.cse_id
WHERE TRUE
  AND b.cse_id IS NULL
```

14.12. What is the syntax for a cross-join?

The `CROSS JOIN` is used to generate a paired combination of each row of the first table with each row of the second table. This join type is also known as cartesian join.

Suppose we are having tea and we want to have a list of all combinations of available tea and cake.

tea
Green tea
Peppermint tea
English Breakfast

cake
Carrot cake
Brownie
Tarte tatin

A `CROSS JOIN` will create all paired combinations of the rows of the tables that will be joined.

cake	tea
Carrot cake	Green tea
Brownie	Green tea
Tarte tatin	Green tea
Carrot cake	Peppermint tea
Brownie	Peppermint tea
Tarte tatin	Peppermint tea
Carrot cake	English breakfast
Brownie	English breakfast
Tarte tatin	English breakfast

Cross join syntax:

```
SELECT ColumnName_1,
       ColumnName_2,
       ColumnName N
FROM [Table_1]
     CROSS JOIN [Table_2]
```

```
SELECT * FROM tea
CROSS JOIN cake
```

Below is an alternative syntax for cross-join that does not include the `CROSS JOIN` keyword; we will place the tables that will be joined after the `FROM` clause and separated with a comma.

```
SELECT ColumnName_1,
       ColumnName_2,
       ColumnName_N
FROM [Table_1],[Table_2]
```

14.13. What is the syntax for a semi-join?

Semi join is used to return one copy of rows from a table where at least one match is found in the values with the second table. `EXISTS` is used instead of a `JOIN` keyword. The main advantage of this kind of join query is that it makes the queries run faster.

A semi join returns rows that match an EXISTS subquery without duplicating rows from the left side of the predicate when multiple rows on the right side satisfy the criteria of the subquery.

```
SELECT columns
FROM table_1
WHERE EXISTS (
SELECT values
FROM table_2
WHERE table_2.column = table_1.column);
```

Example:

`employee` table

employee_id	Employee_name	Employee_age
1	Inès	28
2	Ghassan	26
3	Camille	25
4	Cécile	35

`client` table

client_id	client_name	client_age
10	Leïla	20
11	Claire	28
12	Thomas	25
13	Rébecca	30

```
SELECT employee.employee_id, employee.employee_name
FROM employee
WHERE EXISTS (
SELECT 28
FROM client
WHERE client.client_age = employee.employee_age);
```

Output

client_id	client_name	client_age
1	Inès	28

After the joining, the selected fields of the rows of the `employee` table satisfying the equality condition will be displayed as a result, but this equality condition is valid only for those rows in the `client` table that does have the value of `client_age` as 28.

14.14. What is the syntax for the Spark SQL `UNION`, `MINUS`, and `INTERSECT` set operators?

`UNION` returns the collection of two queries. The query below returns the same results as if we inserted our `new_events_final` into the `events` table.

```
SELECT * FROM events
UNION
SELECT * FROM new_events_final
```

The SQL `UNION` operator is different from join as it combines the result of two or more `SELECT` statements. Each `SELECT` statement within the `UNION` must have the same number of columns. The columns must also have similar data types. Also, the columns in each `SELECT` statement must be in the same order.

`INTERSECT` returns all rows found in both relations.

```
SELECT * FROM events
INTERSECT
SELECT * FROM new_events_final
```

`MINUS` returns all the rows found in one dataset but not the other;

14.15. What is the syntax for pivot tables?

A pivot tbale allows you to transform rows into columns and group by any data field. The `PIVOT` clause is used for data perspective. You can get the aggregated values based on specific column values, which will be turned to multiple columns used in `SELECT` clause. The `PIVOT` clause can be specified after the table name or subquery.

`SELECT * FROM ()`: The `SELECT` statement inside the parentheses is the input for this table.

`PIVOT`: The first argument in the clause is an aggregate function and the column to be aggregated. Then, we specify the pivot column in the `FOR` subclause. Finally, the `IN` operator contains the pivot column values.

Here we use `PIVOT` to create a new `transactions` table that flattens out the information contained in the `sales` table. This flattened data format can be useful for dashboarding, but also useful for applying machine learning algorithms for inference or prediction.

```
CREATE OR REPLACE TABLE transactions AS
```

```
SELECT * FROM (
  SELECT
    email,
    order_id,
    transaction_timestamp,
    total_item_quantity,
    purchase_revenue_in_usd,
    unique_items,
    item.item_id AS item_id,
    item.quantity AS quantity
  FROM sales_enriched
) PIVOT (
  sum(quantity) FOR item_id in (
    'P_FOAM_K',
    'M_STAN_Q',
    'P_FOAM_S',
    'M_PREM_Q',
    'M_STAN_F',
    'M_STAN_T',
    'M_PREM_K',
    'M_PREM_F',
    'M_STAN_K',
    'M_PREM_T',
    'P_DOWN_S',
    'P_DOWN_K'
  )
);

SELECT * FROM transactions
```

	email	order_id	transaction_timestamp	total_item_quantity	purchase_revenue_in_usd	unique_items	P_FOAM_K	M_
1	ewalker@jordan-johnson.com	280515	1592493134477009	1	1195	1	null	nul
2	james27@hotmail.com	413849	1593468897891582	1	59	1	null	nul
3	cassie17@medina-anderson.com	445523	1593713036444518	2	2074	2	1	nul
4	angela76@yahoo.com	291471	1592582917701512	1	945	1	null	nul
5	gromero25@hotmail.com	458773	1593807106272431	2	2054	2	null	nul
6	brayjose@carter.com	455266	1593791115255038	1	1195	1	null	nul

Truncated results, showing first 1000 rows.

	N_Q	P_FOAM_S	M_PREM_Q	M_STAN_F	M_STAN_T	M_PREM_K	M_PREM_F	M_STAN_K	M_PREM_T	P_DOWN_S	P_DOWN_K
1		null	null	null	null	null	null	1	null	null	null
2		1	null	null	null	null	null	null	null	null	null
3		null	null	null	null	1	null	null	null	null	null
4		null	null	1	null	null	null	null	null	null	null
5		1	null	null	null	1	null	null	null	null	null
6		null	null	null	null	null	null	1	null	null	null

Truncated results, showing first 1000 rows.

14.16. What are higher order functions? (`FILTER`, `EXIST`, `TRANSFORM`, `REDUCE`)?

Higher order functions in Spark SQL allow you to work directly with complex data types. When working with hierarchical data, records are frequently stored as array or map type objects. Higher-order functions allow you to transform data while preserving the original structure. Higher order functions include:

- `FILTER` filters an array using the given lambda function.
- `EXIST` tests whether a statement is true for one or more elements in an array.
- `TRANSFORM` uses the given lambda function to transform all elements in an array.
- `REDUCE` is more advanced than transform. It takes two lambda functions to reduce the elements of an array to a single value by merging the elements into a buffer, and then apply a finishing function on the final buffer.

14.17. What is the syntax for `FILTER` ?

In this example, we want to remove items that are not king-sized from all records in our `items` column. We can use the `FILTER` function to create a new column that excludes that value from each array.

```
FILTER (items, i -> i.item_id LIKE "%K") AS king_items
```

In the statement above:

- `FILTER` : the name of the higher-order function
- `items` : the name of our input array
- `i` : the name of the iterator variable. You choose this name and then use it in the lambda function. It iterates over the array, cycling each value into the function one at a time.
- `->` : Indicates the start of a function
- `i.item_id LIKE "%K"` : This is the function. Each value is checked to see if it ends with the capital letter K. If it is, it gets filtered into the new column, `king_items`

```
-- filter for sales of only king sized items
SELECT
  order_id,
  items,
  FILTER (items, i -> i.item_id LIKE "%K") AS king_items
FROM sales
```

You may write a filter that produces a lot of empty arrays in the created column. When that happens, it can be useful to use a `WHERE` clause to show only non-empty array values in the returned column.

In this example, we accomplish that by using a subquery. They are useful for performing an operation in multiple steps. In this case, we're using it to create the named column that we will use with a `WHERE` clause.

```
CREATE OR REPLACE TEMP VIEW king_size_sales AS

SELECT order_id, king_items
FROM (
  SELECT
    order_id,
    FILTER (items, i -> i.item_id LIKE "%K") AS king_items
  FROM sales)
WHERE size(king_items) > 0;

SELECT * FROM king_size_sales
```

14.18. What is the syntax for `EXIST` ?

Exists tests whether a statement is true for one or more elements in an array. Let's say we want to flag all blog posts with `"Company Blog"` in the categories field. I can use the `EXISTS` function to mark which entries include that category.

Let's dissect this line of code to better understand the function:

```
EXISTS (categories, c -> c = "Company Blog") companyFlag
```

- **`EXISTS`** : the name of the higher-order function
- **`categories`** : the name of our input array
- **`c`** : the name of the iterator variable. You choose this name and then use it in the lambda function. It iterates over the array, cycling each value into the function one at a time. Note that we're using the same kind as references as in the previous command, but we name the iterator with a single letter **`->`** : Indicates the start of a function **`c = "Engineering Blog"`** : This is the function. Each value is checked to see if it is the same as the value `"Company Blog"` . If it is, it gets flagged into the new column, `companyFlag`

14.19. What is the syntax for `TRANSFORM` ?

Built-in functions are designed to operate on a single, simple data type within a cell; they cannot process array values. `TRANSFORM` can be particularly useful when you want to apply an existing function to each element in an array.

Compute the total revenue from king-sized items per order:

```
TRANSFORM(king_items, k -> CAST(k.item_revenue_in_usd \* 100 AS INT)) AS item_revenues
```

In the statement above, for each value in the input array, we extract the item's revenue value, multiply it by 100, and cast the result to integer. Note that we're using the same kind as references as in the previous command, but we name the iterator with a new variable, `k` .

```
-- get total revenue from king items per order
CREATE OR REPLACE TEMP VIEW king_item_revenues AS

SELECT
  order_id,
  king_items,
  TRANSFORM (
    king_items,
    k -> CAST(k.item_revenue_in_usd * 100 AS INT)
  ) AS item_revenues
FROM king_size_sales;

SELECT * FROM king_item_revenues
```

Table | Data Profile

	order_id	king_items	item_revenues
1	289539	[{"coupon": "NEWBED10", "item_id": "M_STAN_K", "item_name": "Standard King Mattress", "item_revenue_in_usd": 1075.5, "price_in_usd": 1195, "quantity": 1}]	[107550]
2	258127	[{"coupon": null, "item_id": "M_STAN_K", "item_name": "Standard King Mattress", "item_revenue_in_usd": 1195, "price_in_usd": 1195, "quantity": 1}]	[119500]
3	258653	[{"coupon": null, "item_id": "M_STAN_K", "item_name": "Standard King Mattress", "item_revenue_in_usd": 1195, "price_in_usd": 1195, "quantity": 1}]	[119500]
4	288311	[{"coupon": "NEWBED10", "item_id": "P_FOAM_K", "item_name": "King Foam Pillow", "item_revenue_in_usd": 71.10000000000001, "price_in_usd": 79, "quantity": 1}]	[7110]
5	277614	[{"coupon": "NEWBED10", "item_id": "M_PREM_K", "item_name": "Premium King Mattress", "item_revenue_in_usd": 1795.5, "price_in_usd": 1995, "quantity": 1}]	[179550]
6	259947	[{"coupon": null, "item_id": "M_STAN_K", "item_name": "Standard King Mattress", "item_revenue_in_usd": 1195, "price_in_usd": 1195, "quantity": 1}]	[119500]

Truncated results, showing first 1000 rows.
Click to re-execute with maximum result limits.

Command took 0.46 seconds -- by lara.rachidi@databricks.com at 04/06/2022, 15:44:36 on data-engineering-training

Another example:

```
TRANSFORM(categories, cat -> LOWER(cat)) lwrCategories
```

- **`TRANSFORM`**: the name of the higher-order function
- **`categories`**: the name of our input array
- **`cat`**: the name of the iterator variable. You choose this name and then use it in the lambda function. It iterates over the array, cycling each value into the function one at a time.
- **`->`**: Indicates the start of a function
- **`LOWER(cat)`**: This is the function. For each value in the input array, the built-in function `LOWER()` is applied to transform the word to lowercase.

14.20. What is the syntax for `REDUCE`?

`REDUCE` is more advanced than `TRANSFORM`; it takes two lambda functions. You can use it to reduce the elements of an array to a single value by merging the elements into a buffer, and applying a finishing function on the final buffer.

We will use the reduce function to find an average value, by day, for CO2 readings. Take a closer look at the individual pieces of the `REDUCE` function by reviewing the list below.

```
REDUCE(co2_level, 0, (c, acc) -> c + acc, acc ->(acc div size(co2_level)))
```

- **`co2_level`** is the input array
- `0` is the starting point for the buffer. Remember, we have to hold a temporary buffer value each time a new value is added to from the array; we start at zero in this case to get an accurate sum of the values in the list.
- **`(c, acc)`** is the list of arguments we'll use for this function. It may be helpful to think of `acc` as the buffer value and `c` as the value that gets added to the buffer.
- **`c + acc`** is the buffer function. As the function iterates over the list, it holds the total (`acc`) and adds the next value in the list (`c`).
- **`acc div size(co2_level)`** is the finishing function. Once we have the sum of all numbers in the array, we divide by the number of elements to find the average.

---

15. SQL UDFs and Control Flow

15.1. What is the syntax to define and register SQL UDFs? How do you then apply that function to the data?

A SQL UDF applies a recipe to a particular text and returns a result.

```
CREATE OR REPLACE FUNCTION function_name(param TYPE)
RETURNS type_to_be_returned RETURN function_itself
```

Let's apply a function to a temp view called `foods` that has a column called `food` and values corresponding to various types of food:

```
CREATE OR REPLACE TEMPORARY VIEW foods(food) AS VALUES
("beef"),
("beans"),
("potatoes"),
("bread");
```

Create the function:

```
CREATE OR REPLACE FUNCTION yelling(text STRING)
RETURNS STRING
RETURN concat(upper(text), "!!!")
```

Apply the function to the data in the temp view:

```
SELECT yelling(food) FROM foods
```

15.2. How can you see where the function was registered and basic information about expected inputs and what is returned?

```
DESCRIBE FUNCTION EXTENDED yelling
```

15.3. What are SQL UDFs governed by?

SQL UDFs exist as objects in the metastore and are governed by the same Access Control Lists (ACLs) as databases, tables, or views.

15.4. What permissions must a user have on the function to use a SQL UDF? Describe their scoping.

The user must have `USAGE` and `SELECT` permissions on the function to use it.

SQL UDFs will persist between execution environments (which can include notebooks, DBSQL queries, and jobs).

15.5. What is the syntax used for the evaluation of multiple conditional statements?

```
SELECT *,
  CASE
    WHEN food = "beans" THEN "I love beans"
    WHEN food = "potatoes" THEN "My favorite vegetable is potatoes"
    WHEN food <> "beef" THEN concat("Do you have any good recipes for ", food ,"?")
    ELSE concat("I don't eat ", food)
  END
FROM foods
```

15.6. What is the syntax using SQL UDFs for custom control flow within SQL workloads?

Define the function with a custom control flow.

```
CREATE FUNCTION foods_i_like(food STRING)
RETURNS STRING
RETURN CASE
  WHEN food = "beans" THEN "I love beans"
  WHEN food = "potatoes" THEN "My favorite vegetable is potatoes"
  WHEN food <> "beef" THEN concat("Do you have any good recipes for ", food ,"?")
  ELSE concat("I don't eat ", food)
END;
```

```
SELECT foods_i_like(food) FROM foods
```

15.7. What is the benefit of using SQL UDFs?

Especially for enterprises that might be migrating users from systems with many defined procedures or custom-defined formulas, SQL UDFs can allow a handful of users to define the complex logic needed for common reporting and analytic queries.

---

16. Python for Databricks SQL & Python Control Flow

16.1. What is the syntax to turn SQL queries into Python strings?

```
print("""
SELECT *
FROM table_name
""")
```

16.2. What is the syntax to execute SQL from a Python cell?

```
spark.sql("SELECT * FROM table_name")
```

16.3. What function do you call to render a query the way it would appear in a normal SQL notebook?

```
display(spark.sql("SELECT * FROM table_name"))
```

16.4. What is the syntax to define a function in Python?

```
def return_new_string(string_arg):
    return "The string passed to this function was " + string_arg
```

16.5. What is the syntax for f-strings?

```
f"I can substitute {my_string} here"
```

You can insert the result of a function into an f-string:

```
f"I can substitute functions like {return_new_string('foobar')} here"
```

16.6. How can f-strings be used for SQL queries?

```
table_name = "users"
filter_clause = "WHERE state = 'CA'"

query = f"""
SELECT *
FROM {table_name}
{filter_clause}
"""

print(query)
```

16.7. What is the syntax for `if` / `else` clauses wrapped in a function?

```
def foods_i_like(food):
    if food == "beans":
        print(f"I love {food}")
    elif food == "potatoes":
        print(f"My favorite vegetable is {food}")
    elif food != "beef":
        print(f"Do you have any good recipes for {food}?")
    else:
        print(f"I don't eat {food}")
```

16.8. What are the two methods for casting values to numeric types (int and float)?

The two methods to cast values to numeric types are `int()` and `float()`, e.g. `int("2")`.

16.9. What are `assert` statements and what is the syntax?

`assert` statements allow us to run simple tests of Python code. If an `assert` statement evaluates to true, nothing happens. If it evaluates to false, an error is raised.

Example asserting that the number `2` is an integer:

```
assert type(2) == int
```

16.10. Why do we use `try` / `except` statements and what is the syntax?

Errors will stop the execution of a notebook script; all cells after an error will be skipped when a notebook is scheduled as a production job. If we enclose code that might throw an error in a `try` statement, we can define alternate logic when an error is encountered. So `try` / `except` provides robust error handling. When a non-numeric string is passed, an informative message is printed out.

```
def try_int(num_string):
    try:
        int(num_string)
        result = f"{num_string} is a number."
    except:
        result = f"{num_string} is not a number!"

    print(result)
```

16.11. What is the downside of using `try` / `except` statements?

When using `try` / `except` statements, an error will not be raised when an error occurs. Implementing logic that suppresses errors can lead to logic silently failing.

16.12. What is the syntax for `try` / `except` statements where you return an informative error message?

```
def three_times(number):
    try:
        return int(number) * 3
    except ValueError as e:
        print(f"You passed the string variable '{number}'.\n")
        print(f"Try passing an integer instead.")
        return None
```

As implemented, this logic would only be useful for interactive execution of this logic. The message isn't currently being logged anywhere, and the code will not return the data in the desired format; human intervention would be required to act upon the printed message.

16.13. How do you apply these concepts to execute SQL logic on Databricks, for example to avoid SQL injection attack?

Using a simple `if` clause with a function allows us to execute arbitrary SQL queries, optionally displaying the results, and always returning the resultant DataFrame.

```
def simple_query_function(query, preview=True):
    query_result = spark.sql(query)
    if preview:
        display(query_result)
    return query_result
```

```
result = simple_query_function(query)
```

Below, we execute a different query and set preview to `False`, as the purpose of the query is to create a temp view rather than return a preview of data.

```
new_query = "CREATE OR REPLACE TEMP VIEW id_name_tmp_vw AS SELECT id, name FROM
demo_tmp_vw"
simple_query_function(new_query, preview=False)
```

Suppose we want to protect our company from malicious SQL, like the query below.

```
injection_query = "SELECT * FROM demo_tmp_vw; DROP DATABASE prod_db CASCADE; SELECT *
FROM demo_tmp_vw"
```

We can use the `find()` method to test for multiple SQL statements by looking for a semicolon. If it's not found, it will return `-1`. With that knowledge, we can define a simple search for a semicolon in the query string and raise a custom error message if it was found (not `-1`).

```
def injection_check(query):
    semicolon_index = query.find(";")
    if semicolon_index >= 0:
        raise ValueError(f"Query contains semi-colon at index
{semicolon_index}\nBlocking execution to avoid SQL injection attack")
```

Always be wary of allowing untrusted users to pass text that will be passed to SQL queries. Note that only one query can be executed using `spark.sql()`, so text with a semi-colon will always throw an error. If we add this method to our earlier query function, we now have a more robust function that will assess each query for potential threats before execution. We will see normal performance with a safe query, and prevent execution when when bad logic is run.

```
def secure_query_function(query, preview=True):
    injection_check(query)
    query_result = spark.sql(query)
    if preview:
        display(query_result)
    return query_result
```

---

17. Incremental Data Ingestion with Auto Loader

17.1. What is incremental ETL?

- A full load corresponds to the entire data dump that takes place the first time a data source is loaded into the warehouse.
- An incremental load is when the delta between target and source data is dumped at regular intervals. The last extract date is stored so that only records added after this date are loaded.
- Incremental ETL is important since it allows us to deal solely with new data that has been encountered since the last ingestion. Reliably processing only the new data reduces redundant processing and helps enterprises reliably scale data pipelines.

17.2. What is the purpose of Auto Loader?

- Historically, ingesting files from a data lake into a database has been a complicated process. Building a continuous, cost effective, maintainable, and scalable data transformation and ingestion system is not trivial.
- Databricks Auto Loader provides an easy-to-use mechanism for incrementally and efficiently processing new data files as they arrive in cloud file storage. It incrementally processes files from a directory in cloud object storage into a Delta Lake table. This optimized solution provides a way for data teams to load raw data from cloud object stores at lower costs and latencies.
- It automatically configures and listens to a notification service for new files and can scale up to millions of files per second. It also takes care of common issues such as schema inference and schema evolution. It allows you to continuously ingest data into Delta Lake.
- Due to the benefits and scalability that Auto Loader delivers, Databricks recommends its use as general best practice when ingesting data from cloud object storage.

17.3. What are the 4 arguments using Auto Loader with automatic schema inference and evolution?

- Auto Loader can automatically detect the introduction of new columns to your data and restart so you don’t have to manage the tracking and handling of schema changes yourself. When using Auto Loader with automatic schema inference and evolution, the 4 arguments shown below should allow ingestion of most datasets:

argument	what it is	how it's used
`data_source`	The directory of the source data	Auto Loader will detect new files as they arrive in this location and queue them for ingestion; passed to the `.load()` method
`source_format`	The format of the source data	While the format for all Auto Loader queries will be `cloudFiles`, the format of the source data should always be specified for the `cloudFiles.format` option
`table_name`	The name of the target table	Spark Structured Streaming supports writing directly to Delta Lake tables by passing a table name as a string to the `.table()` method. Note that you can either append to an existing table or create a new table
`checkpoint_directory`	The location for storing metadata about the stream	This argument is passed to the `checkpointLocation` and `cloudFiles.schemaLocation` options. Checkpoints keep track of streaming progress, while the schema location tracks updates to the fields in the source dataset

17.4. How do you begin an Auto Loader stream?

- We define a function and some path variables in a setup script to begin an Auto Loader stream. We want to configure and execute a query to process `JSON` files from the location specified by `source_path` into a table named `target_table`.

```
def autoload_to_table(data_source, source_format, table_name, checkpoint_directory):
    query = (spark.readStream
                  .format("cloudFiles")
                  .option("cloudFiles.format", source_format)
                  .option("cloudFiles.schemaLocation", checkpoint_directory)
                  .load(data_source)
                  .writeStream
                  .option("checkpointLocation", checkpoint_directory)
                  .option("mergeSchema", "true")
                  .table(table_name))
    return query
```

- Here, we're reading from a source directory of JSON files.

```
query = autoload_to_table(data_source = f"{DA.paths.working_dir}/tracker",
                          source_format = "json",
                          table_name = "target_table",
                          checkpoint_directory = f"
{DA.paths.checkpoints}/target_table")
```

- Because Auto Loader uses Spark Structured Streaming to load data incrementally, the code above doesn't appear to finish executing. We can think of this as a continuously active query. This means that as soon as new data arrives in our data source, it will be processed through our logic and loaded into

our target table. The great thing about Auto Loader is that when new data comes in, it will pick it up and process it automatically. You dont need to have a tool like Airflow checking in every hour or so.

17.5. What is the benefit of Auto Loader compared to structured streaming?

- With `cloudFiles.schemaLocation`, Auto Loader will infer schema wheareas traditional structured streaming will not (it has to be defined in traditional structured streaming).
- Auto Loader will scan the first gigabytes of data and infer the schema for you. It will save it in the `cloudFiles.schemaLocation` directory and version it. When it encounters a new schema, and if you have your cluster set to `auto restart`, Auto Loader will gracefully fail, update the schema and start again.

17.6. What keyword indicates that you're using Auto Loader rather than a traditional stream for ingesting?

`cloudFiles` is the keyword indicating that you're using Auto Loader rather than a traditional stream for ingesting.

17.7. What can you do once data has been ingested to Delta Lake with Auto Loader?

- Once data has been ingested to Delta Lake with Auto Loader, users can interact with it the same way they would any table.

```
%sql
SELECT * FROM target_table
```

17.8. What is the `_rescued_data` column?

The `_rescued_data` column is added by Auto Loader automatically to capture any data that might be malformed and not fit into the table otherwise. If you have specified the schema for any of the fields in your dataset and you encounter records that are not valid for that specified schema, those invalid records will end up in your `_rescued_data` column.

Rather than failing the job, or dropping records, you will automatically quarantine that data in a separate column which will allow you to do programmatic or manual review of that data and see if you can fix those records and insert those back into your base dataset.

17.9. What is the data type encoded by Auto Loader for fields in a text-based file format?

Because JSON is a text-based format, Auto Loader will encode all fields as `STRING` type. This is the safest and most permissive type, ensuring that the least amount of data is dropped or ignored at ingestion due to type mismatch.

17.10. Historically, what were the two inefficient ways to land new data?

Historically, many systems have been configured to either reprocess all records in a source directory to calculate current results or require data engineers to implement custom logic to identify new data that's arrived since the last time a table was updated.

An Auto Loader query automatically detects and processes records from the source directory into the target table.

17.11. Is there a delay when records are ingested with an Auto Loader query?

There is a slight delay as records are ingested, but an Auto Loader query executing with default streaming configuration should update results in near real time.

17.12. How do you track the ingestion progress?

- The query below shows the table history. A new table version should be indicated for each `STREAMING UPDATE`. These update events coincide with new batches of data arriving at the source.

```
%sql
DESCRIBE HISTORY target_table
```

- Each streaming update corresponds to a new batch of files being added to that source directory and ingested. We can see the number of rows being ingested with each batch.
- From a lakehouse perspective, it makes data ingestion very easy. We don't have to use Airflow to orchestrate or use any additional code to process what has or has not already been processed. It's all handled automatically by Auto Loader.

---

18. Reasoning about Incremental Data with Spark Structured Streaming

18.1. What is Spark Structured Streaming?

- Spark Structured Streaming extends the functionality of Apache Spark to allow for simplified configuration and bookkeeping when processing incremental datasets. While incremental processing is not absolutely necessary to work successfully in the data lakehouse, our experience helping some of the world's largest companies derive insights from the world's largest datasets has led to the conclusion that many workloads can benefit substantially from an incremental processing approach. Many of the core features at the heart of Databricks have been optimized specifically to handle these ever-growing datasets.

- The magic behind Spark Structured Streaming is that it allows users to interact with ever-growing data sources as if they were just a static table of records, by treating infinite data as a table. New data in the data stream translates into new rows appended to an unbounded table. Structured Streaming lets us define a query against the data source and automatically detect new records and propagate them through previously defined logic. Spark Structured Streaming is optimised on Databricks to integrate closely with Delta Lake and Auto Loader.
- Example situations where you have an ever growing dataset:
    - data scientists need access to frequently updated records in an operational database;
    - credit card transactions need to be compared to past customer behavior to identify and flag fraud;
    - a multi-national retailer seeks to serve custom product recommendations using purchase history;
    - log files from distributed systems need to be analayzed to detect and respond to instabilities; clickstream data from millions of online shoppers needs to be leveraged for A/B testing of UX.
- New data in a data stream might correspond to:
    - a new JSON log file landing in cloud storage;
    - updates to a database captured in a CDC feed (change data capture);
    - events queued in a pub/sub messaging feed;
    - a CSV file of sales closed the previous day.

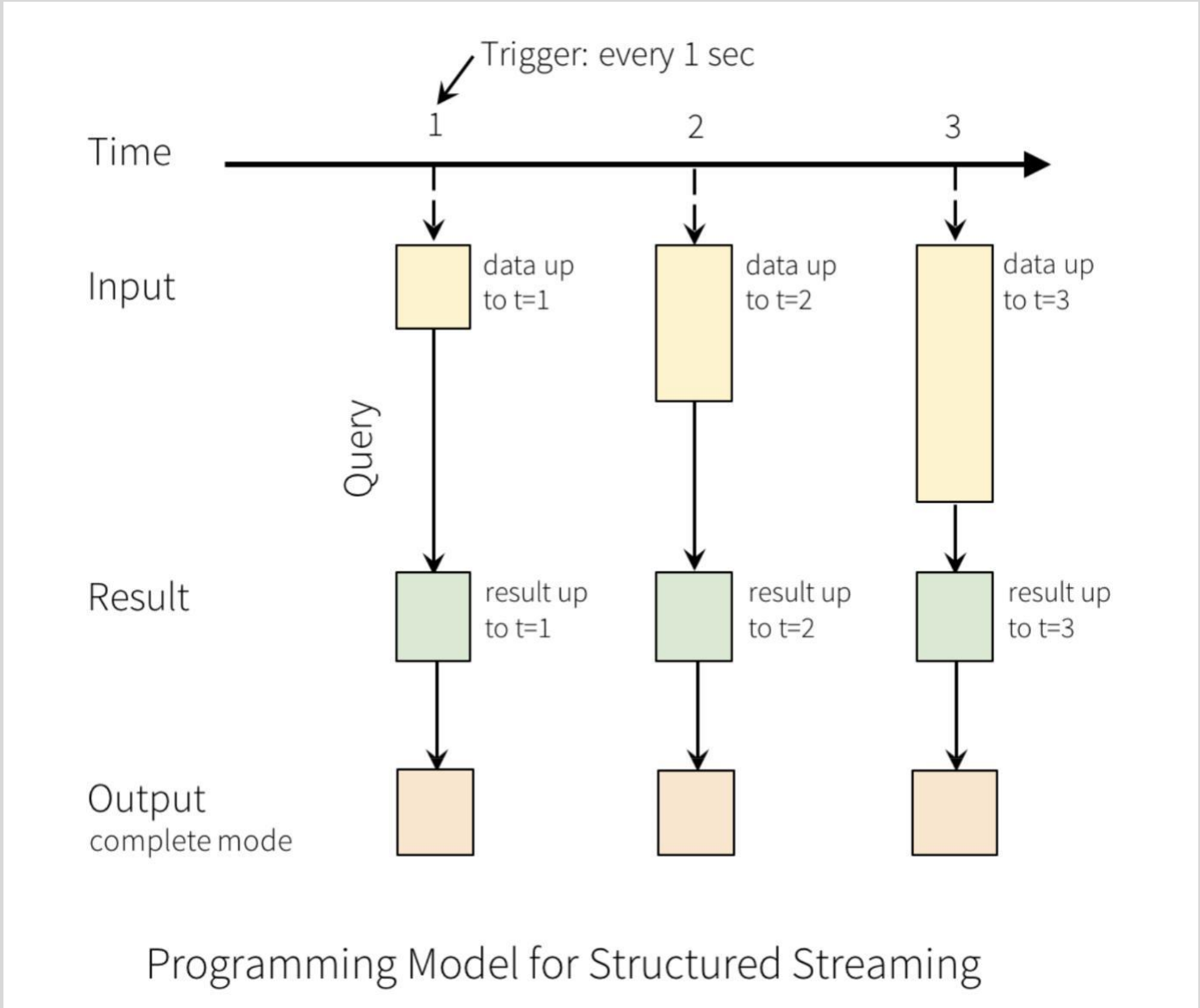
18.2. What were the traditional approaches to data streams?

- Many organisations have traditionally taken an approach of reprocessing the entire source dataset each time they want to update their results, or have their data engineers write custom logic to only capture those files or records that have been added since the last time an update was run.

18.3. Describe the programming model for Structured Streaming.

- The developer defines an input table by configuring a streaming read against a source. The syntax for doing this is similar to working with static data (the input table is that infinite table we were thinking about).
- A query is defined against the input table. Both the DataFrames API and Spark SQL can be used to easily define transformations and actions against the input table. The query is going to define the transformation logic and define where that input source is going to go.
- This logical query on the input table generates the results table. The results table contains the incremental state information of the stream. That results table can be thought as this kind of temporary table that is stored in the memory on our system before our output table is updated. The output of a
- streaming pipeline will persist updates to the results table by writing to an external sink. Generally, a sink will be a durable system such as files or a pub/sub messaging bus (generally a Delta Lake table). It is the destination we are going to write all those results to (sink is like a compatible target for a stream).

- New rows are appended to the input table for each trigger interval (i.e., how frequently you're looking for new data, data up to trigger 1 (t=1), t=2 or t=3). These new rows are essentially analogous to micro-batch transactions and will be automatically propagated through the results table to the sink.



Programming Model for Structured Streaming

18.4. Explain how Structured Streaming ensures end-to-end exactly-once fault-tolerance.

- Structured Streaming ensures end-to-end (from source, execution engine, sink), exactly-once (every record will appear just once, there won't be duplicates and it will always arrive to your sink) semantics under any failure condition (fault tolerance).
- Structured Streaming sources, sinks, and the underlying execution engine work together to track the progress of stream processing. If a failure occurs, the streaming engine attempts to restart and/or reprocess the data.
- The two conditions for the underlying streaming mechanism to work are:
  - **Replayable approach**: Structured Streaming uses checkpointing and write ahead logs to record the offset range of data being processed during each trigger interval (a unique ID allows you to pick up where you left off). This means that in order for this to work, we need to define a `checkpoint`. `checkpoint` is providing Spark Structured Streaming with a location to store the progress of previous runs of your stream. This approach only works if the streaming source is replayable; replayable sources include cloud-based object storage and pub/sub messaging services.
  - **Idempotent sinks**: The streaming sinks are designed to be idempotent - that is, multiple writes of the same data (as identified by the offset) do not result in duplicates being written to the sink.

18.5. What is the syntax to read a stream?

- The `spark.readStream()` method returns a `DataStreamReader` used to configure and query the stream. The code uses the PySpark API to incrementally read a Delta Lake table named `bronze` and register a streaming temp view named `streaming_tmp_vw`.

```
(spark.readStream
    .table("bronze")
    .createOrReplaceTempView("streaming_tmp_vw"))
```

- When we execute a query on a streaming temporary view, the results of the query will continuously be updated as new data arrives in the source. Think of a query executed against a streaming temp view as an always-on incremental query. It's important to shut down those streams before moving on. A continuously running stream will keep an interactive cluster alive.

```
%sql
SELECT * FROM streaming_tmp_vw
```

18.6. How can you transform streaming data?

- We can execute most transformation against streaming temp views the same way we would with static data. Because we are querying a streaming temp view, this becomes a streaming query that executes indefinitely, rather than completing after retrieving a single set of results.
- For streaming queries like this, Databricks Notebooks include interactive dashboards that allow users to monitor streaming performance. Note that none of these records are being persisted anywhere at this point. This is just in memory.

```
%sql
SELECT device_id, count(device_id) AS total_recordings
FROM streaming_tmp_vw
GROUP BY device_id
```

18.7. Give an example operation that is not possible when working with streaming data. What methods can you use to circumvent these exceptions?

- Most operations on a streaming DataFrame are identical to a static DataFrame, but there are some exceptions to this. The model of the data can be considered as a constantly appending table. Sorting is one of a handful of operations that is either too complex or logically not possible to do when working with streaming data.
- Advanced streaming methods like windowing and watermarking can be used to add additional functionality to incremental workloads.

18.8. How do you persist streaming results?

- To persist streaming results, we need to write that stream out to a sink.
- We start by creating a temp view. Recall that a temp view is just a set of instructions. In this case, we're capturing our previous aggregation in a temp view, i.e. our `group by` statement based on our `device_id` and the number of recordings we've seen. This doesnt trigger a stream because it's just a set of instructions. It's not until we try to return this temp view or write it out to a location that that stream will trigger.

- Defining a temp view from a streaming read, and then defining another temp view against that streaming temp view to apply your logic is a pattern that you can use to leverage SQL in order to do incremental data processing with Databricks.
- You'll need to use the PySpark Structured Streaming APIs for the data stream writer in the next step, but the logic in between can be completed with SQL, meaning that it's easy to take logic that has been written by SQL-only analysts or engineers and inject that into your streaming or incremental workloads without needing to do a full refactor of that code base.

```
%sql
CREATE OR REPLACE TEMP VIEW device_counts_tmp_vw AS (
  SELECT device_id, COUNT(device_id) AS total_recordings
  FROM streaming_tmp_vw
  GROUP BY device_id
)
```

18.9. What are the 3 most important settings when writing a stream to Delta Lake tables?

To persist the results of a streaming query, we must write them out to durable storage. The `DataFrame.writeStream` method returns a `DataStreamWriter` used to configure the output. When writing to Delta Lake tables, the three most important settings are:

- **Checkpointing** with `checkpointLocation`
  - Databricks creates checkpoints by storing the current state of your streaming job to cloud storage (so a checkpoint location is a place in cloud object storage where we can store that stream progress in). Checkpointing combines with write ahead logs to allow a terminated stream to be restarted and continue from where it left off. Spark takes care of the bookkeeping for us (which files are new, what has changed since the last time we ran our job, etc.)
  - Checkpoints cannot be shared between separate streams. A checkpoint is required for every streaming write to ensure. Each stream that we write will need to have its own unique checkpoint that is tied to that stream.
- **Output Modes**, similar to static/batch workloads.
  - `.outputMode("append")`: This is the default. Only newly appended rows are incrementally appended to the target table with each batch
  - `.outputMode("complete")`: The Results Table is recalculated each time a write is triggered; the target table is overwritten with each batch
- **Trigger Intervals**, specifying when the system should process the next set of data.
  - Unspecified: This is the default. This is equivalent to using `processingTime="500ms"`. By default, Spark will automatically detect and process all data in the source that has been added since the last trigger.
  - Fixed interval micro-batches with `.trigger(processingTime="2 minutes")`: the query will be executed in micro-batches and kicked off at the user-specified intervals
  - Triggered micro-batch with `.trigger(once=True)`: the query will execute a single micro-batch to process all the available data and then stop on its own
  - Triggered micro-batch with `.trigger(availableNow=True)`: this is a more recent trigger type, where the query will execute multiple micro-batches to process all the available data and then stop on its own

    Triggered micro-batch is really similar to batch, you still need an orchestrator to start this stream. This is a great option if you dont want this stream to keep running for cost reasons, but you want the benefits of the end to end exactly once fault tolerance guarantees.

18.10. What is the syntax to load data from a streaming temp view back to a DataFrame, and then query the table that we wrote out to?

- Now pulling it all together. We pass our streaming temp view back to our data stream writer by using `spark.table()`. Note that if it's not a streaming temp view, we'll not be able to use the writeStream method - it will automatically pick up the streaming or static nature of the temporary view. We provide the necessary options and write this out to a table named `device_counts`.

```
(spark.table("device_counts_tmp_vw")
    .writeStream
    .option("checkpointLocation", f"{DA.paths.checkpoints}/silver")
    .outputMode("complete")
    .trigger(availableNow=True)
    .table("device_counts")
    .awaitTermination() # This optional method blocks execution of the next cell until the incremental batch write has succeeded
)
```

- When we execute this, we dont have it continuously running. It executes as if it was a batch operation. We can change our trigger method to change this query from a triggered incremental batch to an always-on query triggered every 4 seconds. We can use the same checkpoint to make it an always on query. This logic will start from the point where the previous query left off.

```
query = (spark.table("device_counts_tmp_vw")
              .writeStream
              .option("checkpointLocation", f"{DA.paths.checkpoints}/silver")
              .outputMode("complete")
              .trigger(processingTime='4 seconds')
              .table("device_counts"))
```

- When we query the `device_counts` table that we wrote out to, we treat that as a static table. It is being updated by an incremental or streaming query but the table itself will give us static results. Because we are now querying a table (not a streaming DataFrame), the following will not be a streaming query.

```
%sql
SELECT *
FROM device_counts
```

19. Incremental Multi-Hop in the Lakehouse

19.1. Describe Bronze, Silver, and Gold tables

- In a medallion architecture, the data is going to be further validated and enriched as it moves from left to right. On the left, we have various sources where data might be coming from.
- Bronze tables contain raw data ingested from various sources (JSON files, RDBMS data, IoT data, to name a few examples). Bronze makes sure that data is appended incrementally and grows over time. We're interested in retaining the full unprocessed history of each dataset in an efficient storage format which will provide us with the ability to recreate any state of a given data system.
- Silver tables provide a more refined view of our data. We can join fields from various bronze tables to enrich streaming records, or update account statuses based on recent activity. The silver layer might contain many pipelines and silver tables. Various different views for a given dataset. The goal is that this silver layer becomes that validated single source of truth for our data. This is the dream that the data lake could have been: Correct schema, deduplicated records, but no aggregations for our business users yet.

- Gold: highly refined and aggregated data. Data thas has been transformed to knowledge. Updates to these tables will be completed as part of regularly scheduled production workloads, which helps control costs and allows SLAs for data freshness to be established. Gold tables provide business level aggregates often used for reporting and dashboarding. This would include aggregations such as daily active website users, weekly sales per store, or gross revenue per quarter by department. The end outputs are actionable insights, dashboards and reports of business metrics. Gold tables will often be stored in a separate storage container to help avoid cloud limits on data requests. In general, because aggregations, joins and filtering are being handled before data is written to the golden layer, query performance on data in the gold tables should be exceptional.

19.2. Bronze: what additional metadata could you add for enhanced discoverability?

Additional metadata might be added to our data upon ingestion for enhanced discoverability as well as description of the state of the source dataset or some optimised performance in our downstream application. Examples: source file names that are being ingested, recording of the time where that data was originally processed.

19.3. Can you combine streaming and batch workloads in a unified multi-hop pipeline? What about ACID transactions?

Delta Lake allows users to easily combine streaming and batch workloads in a unified multi-hop pipeline. Each stage of the pipeline represents a state of our data valuable to driving core use cases within the business.

Because all data and metadata lives in object storage in the cloud, multiple users and applications can access data in near-real time, allowing analysts to access the freshest data as it's being processed.

Each stage can be configured as a batch or streaming job, and ACID transactions ensure that we succeed or fail completely.

19.4. Describe how you can configure a read on a raw JSON source using Auto Loader with schema inference. What is the `cloudFiles.schemaHints` option?

- We configure a read on a raw JSON source using Auto Loader with schema inference. For a JSON data source, Auto Loader will default to inferring each column as a string. You can specify the data type for a column using the `cloudFiles.schemaHints` option. Specifying improper types for a field will result in null values.

```
(spark.readStream
    .format("cloudFiles")
    .option("cloudFiles.format", "json")
    .option("cloudFiles.schemaHints", "time DOUBLE")
    .option("cloudFiles.schemaLocation", f"{DA.paths.checkpoints}/bronze")
    .load(DA.paths.data_landing_location)
    .createOrReplaceTempView("recordings_raw_temp"))
```

- We can enrich our raw data with additional metadata describing the source file and the time it was ingested. This additional metadata can be ignored during downstream processing while providing useful information for troubleshooting errors if corrupt data is encountered.

```
%sql
CREATE OR REPLACE TEMPORARY VIEW recordings_bronze_temp AS (
  SELECT *, current_timestamp() receipt_time, input_file_name() source_file
  FROM recordings_raw_temp
)
```

- The code below passes our enriched raw data back to PySpark API to process an incremental write to a Delta Lake table. When new data arrives, the changes are immediately detected by this streaming query.

```
(spark.table("recordings_bronze_temp")
      .writeStream
      .format("delta")
      .option("checkpointLocation", f"{DA.paths.checkpoints}/bronze")
      .outputMode("append")
      .table("bronze"))
```

- We are then loading a static CSV file to add patient data to our recordings. In production, we could use Databricks' Auto Loader feature to keep an up-to-date view of this data in our Delta Lake.

```
(spark.read
      .format("csv")
      .schema("mrn STRING, name STRING")
      .option("header", True)
      .load(f"{DA.paths.data_source}/patient/patient_info.csv")
      .createOrReplaceTempView("pii"))
```

```
%sql
SELECT * FROM pii
```

19.5. What happens with the ACID guarantees that Delta Lake brings to your data when you choose to merge this data with other data sources?

The ACID guarantees that Delta Lake brings to your data are managed at the table level, ensuring that only fully successfully commits are reflected in your tables. If you choose to merge these data with other data sources, be aware of how those sources version data and what sort of consistency guarantees they have.

19.6. Describe what happens at the silver level, when we enrich our data.

As a second hop in our silver level, we enrich and check our data. We join the recordings data with the PII to add patient names, the time for the recordings we parse the time for the recordings to the format `'yyyy-MM-dd HH:mm:ss'` to be human-readable, and we perform a quality check by excluding heart rates that are `<= 0`.

```
(spark.readStream
  .table("bronze")
  .createOrReplaceTempView("bronze_tmp"))
```

```
%sql
CREATE OR REPLACE TEMPORARY VIEW recordings_w_pii AS (
  SELECT device_id, a.mrn, b.name, cast(from_unixtime(time, 'yyyy-MM-dd HH:mm:ss') AS
timestamp) time, heartrate
  FROM bronze_tmp a
  INNER JOIN pii b
  ON a.mrn = b.mrn
  WHERE heartrate > 0)
```

```
(spark.table("recordings_w_pii")
      .writeStream
      .format("delta")
      .option("checkpointLocation", f"{DA.paths.checkpoints}/recordings_enriched")
      .outputMode("append")
      .table("recordings_enriched"))
```

```
%sql
SELECT COUNT(*) FROM recordings_enriched
```

19.7. Describe what happens at the Gold level.

We read a stream of data from `recordings_enriched` and write another stream to create an aggregate gold table of daily averages for each patient.

```
(spark.readStream
  .table("recordings_enriched")
  .createOrReplaceTempView("recordings_enriched_temp"))
```

```
%sql
CREATE OR REPLACE TEMP VIEW patient_avg AS (
  SELECT mrn, name, mean(heartrate) avg_heartrate, date_trunc("DD", time) date
  FROM recordings_enriched_temp
  GROUP BY mrn, name, date_trunc("DD", time))
```

19.8. What is `.trigger(availableNow=True)` and when is it used?

- Using `.trigger(availableNow=True)` provides us the ability to continue to use the strengths of structured streaming while trigger this job one-time to process all available data in micro-batches. We see our stream initialises, and once it completes, it's going to shut down as per the trigger once. As a reminder, strengths of structured streaming are exactly once end-to-end fault tolerant processing, and automatic detection of changes in upstream data sources.
- If we know the approximate rate at which our data grows, we can appropriately size the cluster we schedule for this job to ensure fast, cost-effective processing. The customer will be able to evaluate how much updating this final aggregate view of their data costs and make informed decisions about how frequently this operation needs to be run.
- Downstream processes subscribing to this table do not need to re-run any expensive aggregations. Files just need to be de-serialised and then queries based on included fields can quickly be pushed down against this already-aggregated source.

```
(spark.table("patient_avg")
      .writeStream
      .format("delta")
      .outputMode("complete")
      .option("checkpointLocation", f"{DA.paths.checkpoints}/daily_avg")
      .trigger(availableNow=True) # you want the benefits of streaming but as a single
batch
      .table("daily_patient_avg"))
```

19.9. What are the important considerations for `complete` output mode with Delta?

- When using `complete` output mode, we rewrite the entire state of our table each time our logic runs. While this is ideal for calculating aggregates, we cannot read a stream from this directory, as Structured Streaming assumes data is only being appended. Streaming always expects data to be appending. As soon as you do a complete output mode, that target table cannot be seen as a source for a future stream.

- Use cases for complete mode: when you're writing from your silver to your gold. You want to aggregate over all the data that's available. Or when building a dashboard, we're interested in those aggregations over a period of time. It's like a point in time snapshot. (It won't let you do aggregations in append, because of this concept of infinite data, e.g. what's the average of infinity?)
- Then you also have update. Update is similar to merge in Delta Lake, but not quite as powerful. If you combine Delta Lake and Structured Streaming, you have to set structured streaming mode to update, and do your merge. That's how it knows not to get rid of the whole dataset, but rather update specific records.
- The gold Delta table we have just registered will perform a static read of the current state of the data each time we run the following query.

```
%sql
SELECT * FROM daily_patient_avg
```

- The above table includes all days for all users. If the predicates for our ad hoc queries match the data encoded here, we can push down our predicates to files at the source and very quickly generate more limited aggregate views.

```
%sql
SELECT *
FROM daily_patient_avg
WHERE date BETWEEN "2020-01-17" AND "2020-01-31"
```

19.10. Describe the two options to incrementally process data, either with a triggered option or a continuous option.

When landing additional files in our source directory, we'll be able to see these process through the first 3 tables in our Delta Lake, but we will need to re-run our final query to update our `daily_patient_avg` table, since this query uses the `trigger available now` syntax. The `trigger once` logic defined against the silver table is only going to be executed as a batch when we choose to execute it, meaning that it needs to be manually triggered.

We have the ability to incrementally process data either with a triggered option where we're doing a batch incremental operation, or a continuous option where we have an always on incremental stream.

20. Using the Delta Live Tables UI

20.1. Describe how Delta Live Tables makes the ETL lifecycle easier.

- Delta Live Tables (DLT) makes it easy to build and manage reliable data pipelines that deliver high-quality data on Delta Lake. DLT helps data engineering teams simplify ETL development and management with declarative pipeline development, automatic data testing, and deep visibility for monitoring and recovery.
- By just adding `LIVE` to your SQL queries, DLT will begin to automatically take care of all of your operational, governance and quality challenges. With the ability to mix Python with SQL, users get powerful extensions to SQL to implement advanced transformations and embed AI models as part of the pipelines.
- DLT provides deep visibility into pipeline operations with detailed logging and tools to visually track operational stats and quality metrics. With this capability, data teams can understand the performance and status of each table in the pipeline. Data engineers can see which pipelines have run successfully or failed, and can reduce downtime with automatic error handling and easy refresh.
- DLT takes the queries that you write to transform your data and instead of just executing them against a database, DLT deeply understands those queries and analyzes them to understand the data flow between them. Once DLT understands the data flow, lineage information is captured and can be used to keep data fresh and pipelines operating smoothly.
- Because DLT understands the data flow and lineage, and because this lineage is expressed in an environment-independent way, different copies of data (i.e. development, production, staging) are isolated and can be updated using a single code base. The same set of query definitions can be run on any of those datasets.
- The ability to track data lineage is hugely beneficial for improving change management and reducing development errors, but most importantly, it provides users the visibility into the sources used for analytics – increasing trust and confidence in the insights derived from the data.

20.2. Beyond transformations, how can you define your data in your code?

Your data should be a single source of truth for what is going on inside your business. Beyond just the transformations, there are 3 things that should be included in the code that defines your data:

- **Quality Expectations**: With declarative quality expectations, DLT allows users to specify what makes bad data bad and how bad data should be addressed with tunable severity.
- **Documentation with Transformation**: DLT enables users to document where the data comes from, what it's used for and how it was transformed. This documentation is stored along with the transformations, guaranteeing that this information is always fresh and up to date.
- **Table Attributes**: Attributes of a table (e.g. "contains PII") along with quality and operational information about table execution is automatically captured in the Event Log. This information can be used to understand how data flows through an organization and meet regulatory requirements.

20.3. Describe why large scale ETL is complex when not using DLT.

- With declarative pipeline development, improved data reliability and cloud-scale production operations, DLT makes the ETL lifecycle easier and enables data teams to build and leverage their own data pipelines to get to insights faster, ultimately reducing the load on data engineers.
- Large scale ETL is complex when not using DLT:
  - **Complex pipeline development**: hard to build and maintain table dependencies; difficult to switch between batch and stream processing
  - **Data quality and governance**: difficult to monitor and enforce data quality; impossible to trace data lineage
  - **Difficult pipeline operations**: poor observability at granular, data level; error handling and recovery is laborious

20.4. How do you create and run a DLT pipeline in the DLT UI?

- To create and configure a pipeline, click the `Jobs` button on the sidebar and select the `Delta Live Tables` tab.
- Triggered pipelines run once and then shut down until the next manual or scheduled update. (this corresponds to this `triggered=once`). Continuous pipelines run continuously, ingesting new data as it arrives. Choose the mode based on latency and cost requirements.
- If you specify a value for `Target`, tables are published to the specified database. Without a `Target` specification, we would need to query the table based on its underlying location in DBFS (relative to the Storage Location).
- `Enable autoscaling`, `Min Workers` and `Max Workers` control the worker configuration for the underlying cluster processing the pipeline. Notice the DBU estimate provided, similar to that provided when configuring interactive clusters.
- With a pipeline created, you will now run the pipeline.
- You can run the pipeline in development mode. Development mode accelerates the development

lifecycle by reusing the cluster (as opposed to creating a new cluster for each run) and disabling retries so that you can readily identify and fix errors. Refer to the documentation for more information on this feature.

20.6. How do you explore the DAG?

- As the pipeline completes, the execution flow is graphed. Selecting the tables reviews the details. If a flow has data expectations declared, those metrics are tracked in the `Data Quality` section.

---

21. SQL for Delta Live Tables

21.1. What is the syntax to do streaming with SQL for Delta Live tables? What's the keyword that shows you're using Delta Live Tables?

- You can use SQL to declare Delta Live Tables implementing a simple multi-hop architecture. At its simplest, you can think of DLT SQL as a slight modification to traditional CTAS statements. DLT tables and views will always be preceded by the `LIVE` keyword.
- For each query, the live keyword automatically captures the dependencies between datasets defined in the pipeline and uses this information to determine the execution order. A pipeline is a graph that links together the datasets that have been defined by SQL or Python.

21.2. What is the syntax for declaring a bronze layer table using Auto Loader and DLT?

- `sales_orders_raw` ingests JSON data incrementally from the example dataset found in `/databricks-datasets/retail-org/sales_orders/` .
- Incremental processing via Auto Loader (which uses the same processing model as Structured Streaming), requires the addition of the `STREAMING` keyword in the declaration. The `cloud_files()` method enables Auto Loader to be used natively with SQL. This method takes the following positional parameters: the source location, the source data format, and an arbitrarily sized array of optional reader options. In this case, we set `cloudFiles.inferColumnTypes` to `true` . The comment provides additional metadata that would be visible to anyone exploring the data catalog.

```
CREATE OR REFRESH STREAMING LIVE TABLE sales_orders_raw
COMMENT "The raw sales orders, ingested from /databricks-datasets."
AS SELECT * FROM cloud_files("/databricks-datasets/retail-org/sales_orders/", "json",
map("cloudFiles.inferColumnTypes", "true"))
```

- `customers` presents `CSV` customer data found in `/databricks-datasets/retail-org/customers/`. This table will soon be used in a join operation to look up customer data based on sales records.

```
CREATE OR REFRESH STREAMING LIVE TABLE customers
COMMENT "The customers buying finished products, ingested from /databricks-datasets."
AS SELECT * FROM cloud_files("/databricks-datasets/retail-org/customers/", "csv");
```

21.3. What keyword can you use for quality control? How do you reference DLT Tables/Views and streaming tables?

- Now we declare tables implementing the silver layer. At this level we apply operations like data cleansing and enrichment. Our first silver table enriches the sales transaction data with customer information in addition to implementing quality control by rejecting records with a null order number. The `CONSTRAINT` keyword introduces quality control. Similar in function to a traditional `WHERE` clause, `CONSTRAINT` integrates with DLT, enabling it to collect metrics on constraint violations. Constraints provide an optional `ON VIOLATION` clause, specifying an action to take on records that violate the constraint. The three modes currently supported by DLT include: `FAIL UPDATE` (pipeline failure when constraint is violated), `DROP ROW` (discard records that violate constraints), or `OMITTED` (records violating constraints will be included, but violations will be reported in metrics). The DLT UI, will show you a pie chart of how much is on violation, and how much isn't if you have a live table with a constraint.

- References to other DLT tables and views will always include the `LIVE.` prefix. A target database name will automatically be substituted at runtime, allowing for easily migration of pipelines between DEV/QA/PROD environments.
- References to streaming DLT tables use the `STREAM()`, supplying the table name as an argument.

```
CREATE OR REFRESH STREAMING LIVE TABLE sales_orders_cleaned(
  CONSTRAINT valid_order_number EXPECT (order_number IS NOT NULL) ON VIOLATION DROP ROW
)
COMMENT "The cleaned sales orders with valid order_number(s) and partitioned by order_datetime."
AS
  SELECT f.customer_id, f.customer_name, f.number_of_line_items,
         timestamp(from_unixtime((cast(f.order_datetime as long)))) as order_datetime,
         date(from_unixtime((cast(f.order_datetime as long)))) as order_date,
         f.order_number, f.ordered_products, c.state, c.city, c.lon, c.lat,
c.units_purchased, c.loyalty_segment
  FROM STREAM(LIVE.sales_orders_raw) f
  LEFT JOIN LIVE.customers c
    ON c.customer_id = f.customer_id
    AND c.customer_name = f.customer_name
```

21.4. Declaring gold tables.

- In this case, we declare a table delivering a collection of sales order data based in a specific region. In aggregating, the report generates counts and totals of orders by date and customer. This is really easy SQL syntax: we don't have to define checkpoints in here, that's all managed for us by Delta Live Tables.

```
CREATE OR REFRESH LIVE TABLE sales_order_in_la
COMMENT "Sales orders in LA." AS

  SELECT city, order_date, customer_id, customer_name, ordered_products_explode.curr,
         sum(ordered_products_explode.price) as sales,
         sum(ordered_products_explode.qty) as quantity,
         count(ordered_products_explode.id) as product_count
  FROM (SELECT city, order_date, customer_id, customer_name, explode(ordered_products)
as ordered_products_explode
        FROM LIVE.sales_orders_cleaned
        WHERE city = 'Los Angeles')
  GROUP BY order_date, city, customer_id, customer_name, ordered_products_explode.curr
```

21.5. How can you explore the results in the UI?

- Explore the DAG representing the entities involved in the pipeline and the relationships between them. You can click on each to view a summary, which includes: Run status; Metadata summary; Schema; Data quality metrics
- In the storage location, you can find an `autoloader` directory, a `checkpoints` directory, a `system` directory (which captures events associated with the pipeline. These event logs are stored as a Delta table, which you can query), and a `tables` directory which lists the tables we created.

---

22. Orchestrating Jobs with Databricks

22.1. What is a Job?

- A job is a way to run non-interactive code in a Databricks cluster. For example, you can run an extract, transform, and load (ETL) workload interactively or on a schedule. You can also run jobs interactively in the notebook UI.
- Your job can consist of a single task or can be a large, multi-task workflow with complex dependencies. Databricks manages the task orchestration, cluster management, monitoring, and error reporting for all of your jobs. You can run your jobs immediately or periodically through an easy-to-use scheduling system.

- You can implement a task in a JAR, a Databricks notebook, a Delta Live Tables pipeline, or an application written in Scala, Java, or Python. You control the execution order of tasks by specifying dependencies between the tasks. You can configure tasks to run in sequence or parallel.
- For example, you can have a workflow that does the following:
  - Ingests raw clickstream data and performs processing to sessionize the records.
  - Ingests order data and joins it with the sessionized clickstream data to create a prepared data set for analysis.
  - Extracts features from the prepared data.
  - Performs tasks in parallel to persist the features and train a machine learning model.

22.2. When scheduling a Job, what are the two options to configure the cluster where the task runs?

- You can select either New Job Cluster or Existing All-Purpose Clusters.

22.3. Running a Job and scheduling a Job

You can run the job immediately.

To define a schedule for the job, you can set the Schedule Type to `Scheduled`, specifying the period, starting time, and time zone. You can optionally select the `Show Cron Syntax` checkbox.

Note that Databricks enforces a minimum interval of 10 seconds between subsequent runs triggered by the schedule of a job regardless of the seconds configuration in the cron expression. You can choose a time zone that observes daylight saving time or UTC.

The job scheduler is not intended for low latency jobs. Due to network or cloud issues, job runs may occasionally be delayed up to several minutes. In these situations, scheduled jobs will run immediately upon service availability.

22.4. How do you repair an unsuccessful job run?

You can repair failed or canceled multi-task jobs by running only the subset of unsuccessful tasks and any dependent tasks. Because successful tasks and any tasks that depend on them are not re-run, this feature reduces the time and resources required to recover from unsuccessful job runs.

You can change job or task settings before repairing the job run. Unsuccessful tasks are re-run with the current job and task settings. For example, if you change the path to a notebook or a cluster setting, the task is re-run with the updated notebook or cluster settings. You can view the history of all task runs on the Task run details page.

22.5. How can you view Jobs?

You can filter jobs in the Jobs list by using keywords, selecting only the jobs you own, selecting all jobs you have permissions to access (access to this filter requires that Jobs access control is enabled), or by using tags. To search for a tag created with only a key, type the key into the search box. To search for a tag created with a key and value, you can search by the key, the value, or both the key and value. For example, for a tag with the key `department` and the value `finance`, you can search for `department` or `finance` to find matching jobs. To search by both the key and value, enter the key and value separated by a colon; for example, `department:finance`.

22.6. How can you view runs for a Job and the details of the runs?

When clicking a job name, the Runs tab appears with a table of active runs and completed runs. To switch to a matrix view, click Matrix. The matrix view shows a history of runs for the job, including each job task.

The `Job Runs` row of the matrix displays the total duration of the run and the state of the run. To view details of the run, including the start time, duration, and status, hover over the bar in the Job Runs row.

Each cell in the Tasks row represents a task and the corresponding status of the task. To view details of each task, including the start time, duration, cluster, and status, hover over the cell for that task.

The job run and task run bars are color-coded to indicate the status of the run. Successful runs are green, unsuccessful runs are red, and skipped runs are pink. The height of the individual job run and task run bars provides a visual indication of the run duration.

Databricks maintains a history of your job runs for up to 60 days. If you need to preserve job runs, Databricks recommends that you export results before they expire.

The job run details page contains job output and links to logs, including information about the success or failure of each task in the job run.

22.7. How can you export job run results?

You can export notebook run results and job run logs for all job types. For notebook job runs, you can export a rendered notebook that can later be imported into your Databricks workspace.

You can also export the logs for your job run. You can set up your job to automatically deliver logs to DBFS or S3 through the Job API.

22.8. How do you edit a Job?

You can change the schedule, cluster configuration, alerts, maximum number of concurrent runs, and add or change tags. If job access control is enabled, you can also edit job permissions.

To add labels or key:value attributes to your job, you can add tags when you edit the job. You can use tags to filter jobs in the Jobs list; for example, you can use a `department` tag to filter all jobs that belong to a specific department. Tags also propagate to job clusters created when a job is run, allowing you to use tags with your existing cluster monitoring.

22.9. What does Maximum concurrent runs mean?

The maximum number of parallel runs for this job. Databricks skips the run if the job has already reached its maximum number of active runs when attempting to start a new run. Set this value higher than the default of 1 to perform multiple runs of the same job concurrently. This is useful, for example, if you trigger your job on a frequent schedule and want to allow consecutive runs to overlap with each other, or you want to trigger multiple runs that differ by their input parameters.

22.10. How can you set up alerts?

You can add one or more email addresses to notify when runs of this job begin, complete, or fail.

22.11. What is Job access control?

Job access control enables job owners and administrators to grant fine-grained permissions on their jobs. Job owners can choose which other users or groups can view the results of the job. Owners can also choose who can manage their job runs (Run now and Cancel run permissions). See Jobs access control for details.

22.12. How do you edit tasks?

You can define the order of execution of tasks in a job using the `Depends on` drop-down. You can set this field to one or more tasks in the job.

Configuring task dependencies creates a Directed Acyclic Graph (DAG) of task execution, a common way of representing execution order in job schedulers. Databricks runs upstream tasks before running downstream tasks, running as many of them in parallel as possible.

22.13. What are the individual task configuration options?

Individual tasks have the following configuration options: Cluster, Dependent libraries, Task parameter variables, Timeout, Retries.

To configure the cluster where a task runs, click the Cluster drop-down. You can edit a shared job cluster, but you cannot delete a shared cluster if it is still used by other tasks.

Dependent libraries will be installed on the cluster before the task runs. You must set all task dependencies to ensure they are installed before the run starts. Follow the recommendations in Library dependencies for specifying dependencies.

You can pass templated variables into a job task as part of the task's parameters. These variables are replaced with the appropriate values when the job task runs. You can use task parameter values to pass the context about a job run, such as the run ID or the job's start time.

When a job runs, the task parameter variable surrounded by double curly braces is replaced and appended to an optional string value included as part of the value. For example, to pass a parameter named `MyJobId` with a value of `my-job-6` for any run of job ID 6, add the following task parameter:

```
{
  "MyJobID": "my-job-{{job_id}}"
}
```

Timeout corresponds to the maximum completion time for a job. If the job does not complete in this time, Databricks sets its status to "Timed Out".

Retries is a policy that determines when and how many times failed runs are retried. To set the retries for the task, click Advanced options and select Edit Retry Policy.

22.14. What are the recommendations for cluster configuration for specific job types?

Cluster configuration is important when you operationalize a job. The following provides general guidance on choosing and configuring job clusters, followed by recommendations for specific job types.

- **First, make sure to use shared job clusters.**
  - To optimize resource usage with jobs that orchestrate multiple tasks, use shared job clusters. A shared job cluster allows multiple tasks in the same job run to reuse the cluster. You can use a single job cluster to run all tasks that are part of the job, or multiple job clusters optimized for specific workloads.
  - To use a shared job cluster: First, select `New Job Clusters` when you create a task and complete the cluster configuration. Then, select the new cluster when adding a task to the job. Any cluster you configure when you select `New Job Clusters` is available to any task in the job.
  - A shared job cluster is scoped to a single job run, and cannot be used by other jobs or runs of the same job.
  - Libraries cannot be declared in a shared job cluster configuration. You must add dependent libraries in task settings.
- **Second, choose the correct cluster type for your job.**
  - `New Job Clusters` are dedicated clusters for a job or task run. A shared job cluster is created and started when the first task using the cluster starts, and terminates after the last task using the cluster completes.
  - The cluster is not terminated when idle but terminates only after all tasks using it have completed. If a shared job cluster fails or is terminated before all tasks have finished, a new cluster is created.

A cluster scoped to a single task is created and started when the task starts, and terminates when the task completes. In production, Databricks recommends using new shared or task scoped clusters so that each job or task runs in a fully isolated environment.

- When you run a task on a new cluster, the task is treated as a data engineering (task) workload, subject to the task workload pricing. When you run a task on an existing all-purpose cluster, the task is treated as a data analytics (all-purpose) workload, subject to all-purpose workload pricing. When selecting your all-purpose cluster, you will get a warning about how this will be billed as all-purpose compute. Production jobs should always be scheduled against new job clusters appropriately sized for the workload, as this is billed at a much lower rate.
- If you select a terminated existing cluster and the job owner has `Can Restart` permission, Databricks starts the cluster when the job is scheduled to run.
- Existing all-purpose clusters work best for tasks such as updating dashboards at regular intervals.

22.15. What is new with Jobs?

- Until now, each task had its own cluster to accommodate for the different types of workloads. While this flexibility allows for fine-grained configuration, it can also introduce a time and cost overhead for cluster startup or underutilization during parallel tasks.
- In order to maintain this flexibility, but further improve utilization, we now have cluster reuse. By sharing job clusters over multiple tasks customers can reduce the time a job takes, reduce costs by eliminating overhead and increase cluster utilization with parallel tasks. We have the ability to schedule multiple tasks as part of a job, allowing Databricks Jobs to fully handle orchestration for most production workloads.

- When defining a task, customers will have the option to either configure a new cluster or choose an existing one. With cluster reuse, your list of existing clusters will now contain clusters defined in other tasks in the job.
- When multiple tasks share a job cluster, the cluster will be initialized when the first relevant task is starting. This cluster will stay on until the last task using this cluster is finished. This way there is no additional startup time after the cluster initialization, leading to a time/cost reduction while using the job clusters which are still isolated from other workloads.
- The cluster is not terminated when idle. It terminates only after all tasks using it have completed.
- To decrease new job cluster start time, create a pool and configure the job's cluster to use the pool.

22.16. Notebook job tips

- Total notebook cell output (the combined output of all notebook cells) is subject to a 20MB size limit. Additionally, individual cell output is subject to an 8MB size limit. If total cell output exceeds 20MB in size, or if the output of an individual cell is larger than 8MB, the run is canceled and marked as failed.
- If you need help finding cells near or beyond the limit, run the notebook against an all-purpose cluster and use this notebook autosave technique.

23. Navigating Databricks SQL and Attaching to Warehouses

SQL warehouse is a compute resource that lets you run SQL commands on data objects within Databricks SQL.

Note: Databricks have released a naming change for Databricks SQL that replaces the term "endpoint" with "warehouse". No functional change is intended; this is just a naming change.

23.1. How do you visualise dashboards and insights from your query results?

Navigate to Databricks SQL, make sure a SQL Warehouse is on and accessible, then go to the home page in Databricks SQL, and locate the Sample dashboards and click `Visit gallery`. Click `Import` next to the Retail Revenue & Supply Chain option.

You can discover insights from your query results with a wide variety of rich visualizations.

Databricks SQL allows you to organize visualizations into dashboards with an intuitive drag-and-drop interface.

You can then share your dashboards with others, both within and outside your organization, without the need to grant viewers direct access to the underlying data.

You can configure dashboards to automatically refresh, as well as to alert viewers to meaningful changes in the data.

23.2. How do you update a DBSQL dashboard?

Use the sidebar navigator to find the Dashboards. Click on your dashboard, and gover over the plot; three vertical dots should appear. Click on these. This allows you to select `View Query`.

You can then review the SQL code used to populate this plot. Note that 3 tier namespacing is used to identify the source table; this is a preview of new functionality to be supported by Unity Catalog. You can click `Run` in the top right of the screen to preview the results of the query.

You can review and edit the visualisation. You can also click on the `Add Visualization` button to the right of the visualization name, and configure the visualisation as you see fit. You can then select `Add to Dashboard` from the menu.

23.3. How do you create a new query?

Use the sidebar to navigate to Queries, and create one. Make sure you are connected to a warehouse. In the Schema Browser, click on the current metastore and select `samples`. Select the `tpch` database, and click on the `partsupp` table to get a preview of the schema. While hovering over the `partsupp` table name, click the >> button to insert the table name into your query text.

You can then write your first query and save it by giving it a name. You can also add the query to your dashboard, and navigate back to your dashboard to view this change.

You can always change the organization of visualizations, e.g. by dragging and resizing visualizations.

23.4. How can you set a SQL query refresh schedule?

Locate the `Refresh Schedule` field at the bottom right of the SQL query editor box; click the blue `Never`. Use the drop down to change to Refresh every `1 minute`. For `Ends`, click the `On` radio button, and select tomorrow's date.

23.5. How can you review and refresh your dashboard?

Use the left side bar to navigate to `Dashboards`. Click the blue `Refresh` button to update your dashboard. Click the `Schedule` button to review dashboard scheduling options. Note that scheduling a dashboard to update will execute all queries associated with that dashboard.

23.6. How can you share your dashboard?

Click the blue `Share` button. Select `All Users` from the top field. Choose `Can Run` from the right field and click `Add`. Change the `Credentials` to `Run as viewer`.

Note: At present, in this demo no other users should have any permissions to run your dashboard, as they have not been granted permissions to the underlying databases and tables using Table ACLs. If you wish other users to be able to trigger updates to your dashboard, you will either need to grant them permissions to `Run as owner` or add permissions for the tables referenced in your queries.

23.7. How can you set up an alert for your dashboard?

Use the left side bar to navigate to `Alerts`. Click `Create Alert` in the top right. Click the field at the top left of the screen to give the alert a name. Select your query. You can configure the `Trigger when` options, e.g. with `Value column : total_records`; `Condition : >`; and `Threshold: 15`. For `Refresh`, select `Never`. Click `Create Alert`. On the next screen, click the blue `Refresh` in the top right to evaluate the alert.

23.8. How can you review alert destination options?

From the preview of your alert, click the blue `Add` button to the right of `Destinations` on the right side of the screen. At the bottom of the window that pops up, locate and click the blue text in the message `Create new destinations in Alert Destinations`.

23.9. Can you only use the UI when working with DB SQL?

While we're using the Databricks SQL UI in this demo, SQL Warehouses integrate with a number of other tools to allow external query execution, as well as having full API support for executing arbitrary queries programmatically.

---

24. Introducing Unity Catalog

Unity Catalog in a nutshell: visibility into where data came from, who created it and when, how it has been modified over time, how it's being used, and more.

24.1. List the four key functional areas for data governance.

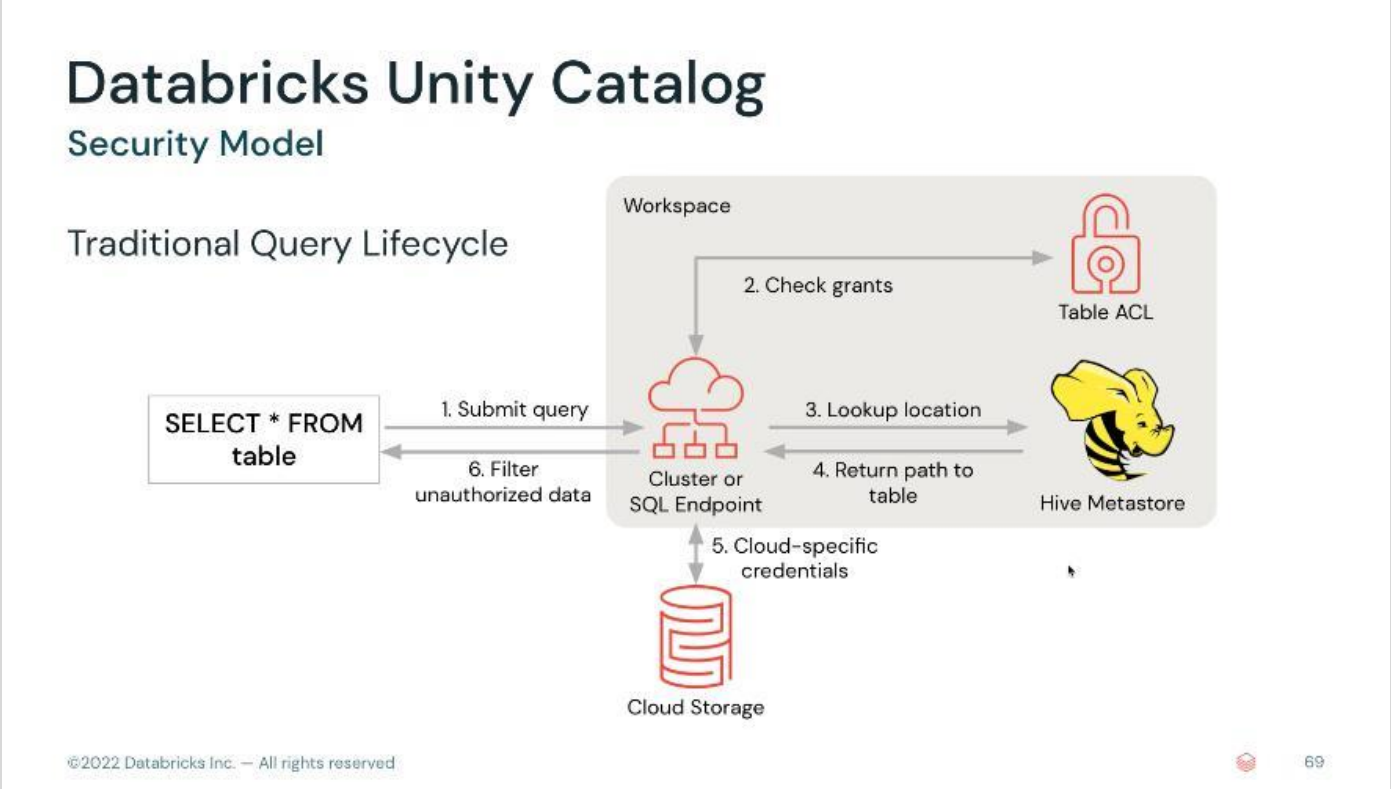
- Data Access Control: Who has access to what?
- Data Access Audit: Understand who accessed what and when? What did they do? Compliance aspect
- Data Lineage: Which data objects feed downstream data objects - if you make a change to an upstream table, how does that affect downstream and vice versa
- Data Discovery: Important to find your data and see what actually exists.

24.2. Explain how Unity Catalog simplifies this with one tool to cover all of these areas.

- With Unity Catalog, we're seeking to cover all of these areas with one tool.
- Traditionally, governance has been a challenge on data lakes. This is primarily due to the file formats that exist on objects stores. Governance is traditionally tied to the specific cloud providers data governance, and tied to files. This introduces a lot of complexity. For example, you can lock down access at the file level, but it doesnt allow you to do anything more granular, e.g. row based access controls, or columns.
- Also, if you need to update your file structures, e.g. for performance reasons, you'll need to update your data governance model as well. Conversely, if you update your data governance model, you'll need to update your file format as well, which can involve rewriting files, or changing the structure of the underlying files.
- If you have a multi cloud infrastructure, you're going to need to set permissions on various data sources for each of those clouds.
- We seek to simplify this with Unity Catalog, by giving secure access to our various personas in a simple manner.
- With UC, we now have this additional catalog qualifier (also note that schema and database are synonymous terms in Databricks). Catalog is a collection of databases.

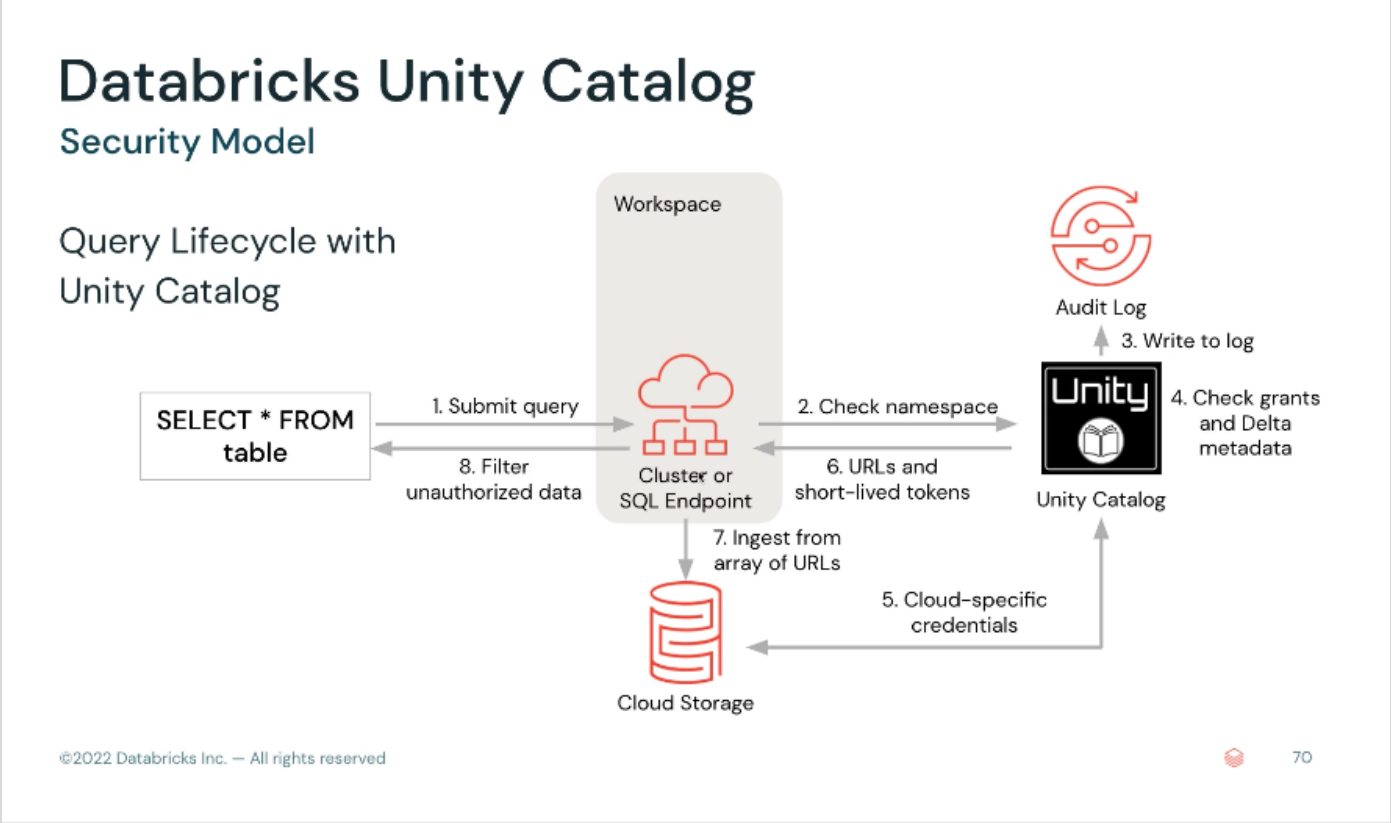
24.3. Walk through a traditional query lifecycle, and how it changes when using Unity Catalog. Highlight the differences and why this makes a query lifecycle much simpler for data consumers.

- Traditionally, a query would be submitted to a cluster / SQL Warehouse. The cluster/SQL Warehouse would go and check the table ACL from the hive metastore to ensure that the query has proper access. If it did, it would query the hive metastore again to find the location of the files that are being queried. Those locations paths would then be returned to the cluster/SQL Warehouse. Then the cluster, using pre-existing IAM role, or cloud specific alternative would then go out to query the cloud storage directly and return the data. And ultimately, return the query results back to the user. Notice what's taking place within this one single, specific workspace (grey area). The problem is that replicating this control model across different workspaces is simply not possible.



- With Unity Catalog, the query goes to the cluster/SQL warehouse. The cluster will go out and check the Unity Catalog namespace. UC will write to the log that this query was submitted. This helps with audibility and compliance (trace for every query that is submitted). UC will then check the grants to make sure that the security is valid. At that point, the UC (with its own pre-configured IAM role) will go out to the cloud storage directly and return pre-configured short-lived URLs and associated tokens, which it will then return to the cluster or SQL warehouse. The cluster will then use those URLs to go out to the cloud storage with the tokens, to get the proper access, return the data, and then return the full query results back to the user. Notice now what's happening in the grey area representing the workspace, by contrast to the old way of doing this.
- Unity Catalog introduces a change. The amount of workspace specific infrastructure is reduced. Unity Catalog exists outside of the workspace at the account level. This means that multiple, different workspaces can rely on that same Unity Catalog. All of the grants are going to be managed in Unity Catalog. Our clusters or warehouses will be able to leverage Unity Catalog regardless of the workspace

that they're in, confirm the credentials for the individual user that is submitting a query, and then grant those permissions to cloud object storage to return the data so that the query can be materialized.



25. Managing Permissions for Databases, Tables, and Views

25.1. What is the data explorer, how do you access it and what does it allow you to do?

The data explorer allows users and admins to navigate databases, tables, and views; explore data schema, metadata, and history; set and modify permissions of relational entities.

25.2. What are the default permissions for users and admins in DBSQL?

By default, admins will have the ability to view all objects registered to the metastore and will be able to control permissions for other users in the workspace.

Users will default to having no permissions on anything registered to the metastore, other than objects that they create in DBSQL; note that before users can create any databases, tables, or views, they must have `create` and `usage` privileges specifically granted to them.

Generally, permissions will be set using Groups that have been configured by an administrator, often by importing organizational structures from SCIM integration with a different identity provider.

Access Control Lists (ACLs) are used to control permissions.

25.3. List the 6 objects for which Databricks allows you to configure permissions.

Databricks allows you to configure permissions for the following objects:

Object	Scope
CATALOG	controls access to the entire data catalog.
DATABASE	controls access to a database.
TABLE	controls access to a managed or external table.
VIEW	controls access to SQL views.
FUNCTION	controls access to a named function.
ANY FILE	controls access to the underlying filesystem. Users granted access to ANY FILE can bypass the restrictions put on the catalog, databases, tables, and views by reading from the file system directly.

25.4. For each object owner, describe what they can grant privileges for.

Databricks admins and object owners can grant privileges according to the following rules:

Role	Can grant access privileges for
Databricks administrator	All objects in the catalog and the underlying filesystem.
Catalog owner	All objects in the catalog.
Database owner	All objects in the database.
Table owner	Only the table (similar options for views and functions).

25.5. Describe all the privileges that can be configured in Data Explorer.

The following privileges can be configured in Data Explorer:

Privilege	Ability
ALL PRIVILEGES	gives all privileges (is translated into all the below privileges).
SELECT	gives read access to an object.
MODIFY	gives ability to add, delete, and modify data to or from an object.
READ_METADATA	gives ability to view an object and its metadata.
USAGE	does not give any abilities, but is an additional requirement to perform any action on a database object.
CREATE	gives ability to create an object (for example, a table in a database).

25.6. Can an owner be set as an individual or a group, or both?

An owner can be set as an individual OR a group. For most implementations, having one or several small groups of trusted power users as owners will limit admin access to important datasets while ensuring that a single user does not create a choke point in productivity.

25.7. What is the command to generate a new database and grant permissions to all users in the DBSQL query editor?

To enable the ability to create databases and tables in the default catalog using Databricks SQL, a workspace admin can run the following command in the DBSQL query editor:

```
GRANT usage, create ON CATALOG hive_metastore TO users
```

To confirm this has run successfully, they can execute the following query:

```
SHOW GRANT ON CATALOG hive_metastore
```
