Databricks Lakehouse Platform

Lakehouse Architecture

1. Describe the relationship between the data lakehouse and the data warehouse. [1]
    - Lakehouse = ACID and Governance of warehouses + flexibility (schema on read) and cost efficiency of lakes
    - Same data for ETL/ELT/processing and for training ML models and BI use cases

2. Identify the improvement in data quality in the data lakehouse over the data lake. [1]
    - ACID + Data governance/auditing
    - Usual features like `CONSTRAINT`
    - New features like `VALIDATE` and Expectations with DLT
    - Delta time travel, for being safe with data changes

3. Compare and contrast silver and gold tables, which workloads will use a bronze table as a source, which workloads will use a gold table as a source. [1]
    - Bronze = raw data. Also known as **Information**
    - Silver = enrich tables via validation and dedup. Can also join columns/tables together for better info.
    - Gold = business level aggregates that were refined from silver. Also known as **Knowledge**

4. Identify elements of the Databricks Platform Architecture, such as what is located in the data plane versus the control plane and what resides in the customer's cloud account. [1]
    - Control plane - backend of Databricks consisting of Notebooks, workspace config, clusters, web GUI, VM management
    - Data plane - where data is processed (i.e. EC2, Azure/GCP VMs) and stored (i.e. S3, Azure Blob), VMs

5. Differentiate between all-purpose clusters and jobs clusters. [1]
    - All purpose clusters - Continuously running, best for dev sessions
    - Job clusters - spins up and runs specifically for a workflow job. Turns off afterwards. Di pwede irestart, and is best for jobs that need high compute (para saglitan lang yung

high cost)

6. Identify how cluster software is versioned using the Databricks Runtime. [1]
    - Runtime is specified for each cluster, and each type will be optimized for different purposes (Standard, ML, Uncategorized)
    - ML packages (Tensorflow, Keras, Pytorch, XGBoost)
    - Photon (optimized for SQL workloads via vectorized query engine)

7. Identify how clusters can be filtered to view those that are accessible by the user. [1]
    - Can set permissions on clusters

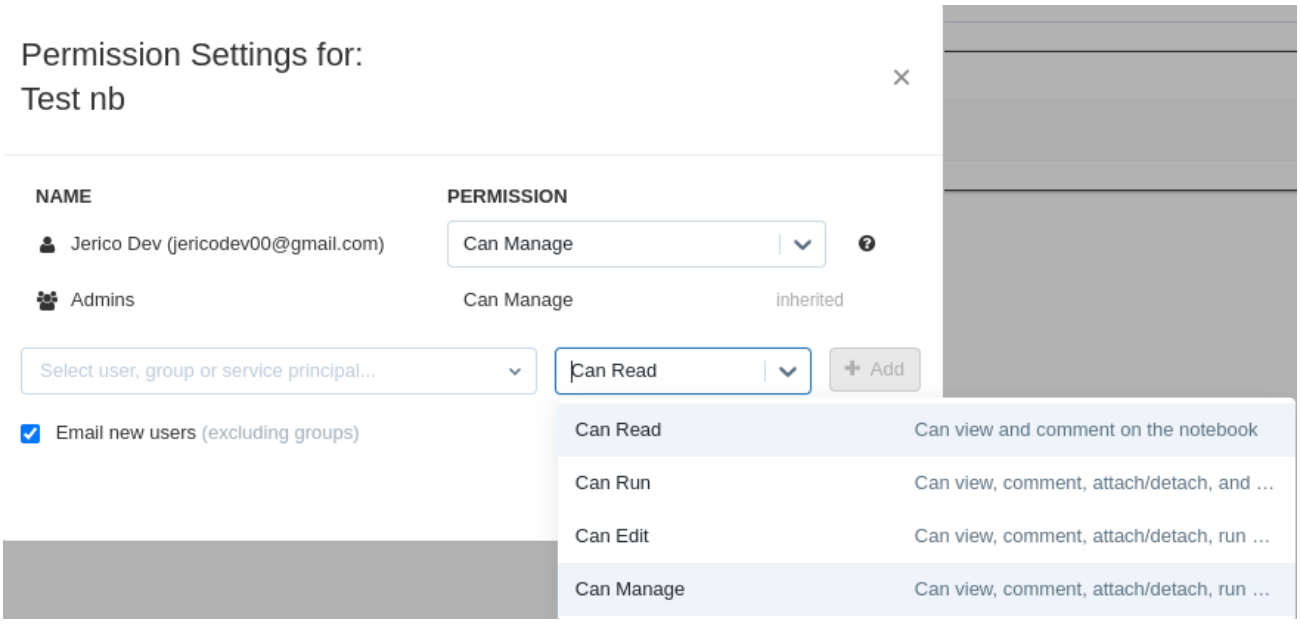
8. Describe how clusters are terminated and the impact of terminating a cluster. [1]
    - Terminating a cluster when a NB is running can impact other users using that cluster

9. Identify a scenario in which restarting the cluster will be useful. [1]
    - To update a long running cluster with the latest images

10. Describe how to use multiple languages within the same notebook. [1]
    - `%sh`, `%md`, `%fs`, `%python`, `%scala`, `%sql` keywords on top of notebook cell

11. Identify how to run one notebook from within another notebook. [1]
    - `%run`, for referencing another NB, can only be used once per cell

12. Identify how notebooks can be shared with others.
    - Share via permissions settings on top right of NB



13. Describe how Databricks Repos enables CI/CD workflows in Databricks. [1, 2]
    - Repos acts similar to Github Desktop, where importing a remote repo can be edited within Databricks
    - Several CI/CD functionality is available by integrating with a tool like Github Actions (Call Repos API to automate)

14. Identify Git operations available via Databricks Repos. [1]
    - Commit, Push, Pull, Merge, Rebase, Reset, and create branches
    - **NOTE**: As of Oct 2023, Rebase and Reset are currently experimental/preview features in the exam context. They are considered as not part of available git operations when asked in the exam.

15. Identify limitations in Databricks Notebooks version control functionality relative to Repos. [1, 2]
    - NBs have version control via Revisions and you can link it with a git repository as well.
    - However NBs cannot be linked to multiple branches, which makes it impractical for developing in parallel

Data Management with Delta Lake

This section on Delta Lake overlaps with ELT With Spark SQL and Python.

1. Identify where Delta Lake provides ACID transactions. [1]
    - Transactions are at the table level, one table at a time
    - Optimistic concurrency control for concurrent transactions
        - BEGIN -> Modify -> Validate -> Commit/Rollback

- Databricks has no BEGIN/END syntax like TSQL. Changes are made in a serial manner (1 at a time ata meaning neto)

2. Identify the benefits of ACID transactions. [1]
   - 'Highest possible data reliability and integrity'

3. Identify whether a transaction is ACID-compliant.
   - Atomic - each txn statement completes or fails ONLY
     - BEGIN/END statements and/or Stored procedures
   - Consistency - data must be predictable before and after txn
     - i.e. row counts consistent when moving rows from one table to another
     - i.e. when moving money from one acc to another, total money must be same
   - Isolation - no other process can change the data/table during a transaction
   - Durability - changes from txn persist, even if servers die (hand in hand with Atomic)

4. Compare and contrast data and metadata.
   - metadata - data about data. used for management, support, and context

```
# Describe statements for showing metadata
DESCRIBE SCHEMA EXTENDED ${schema_name};
DESCRIBE DETAIL <table-name>;
DESCRIBE TABLE EXTENDED <table-name>;
```

5. Compare and contrast managed and external tables. [1]
   - managed tables - made within databricks via DDL
   - external tables - any tables with external data, regardless of where it is stored (dbfs, abfss, adls, s3).
   - when dropping managed, data and metadata is lost. when dropping external, only metadata is lost.

6. Identify a scenario to use an external table. [1]
   - when you need direct access to data outside of Databricks clusters/SQL warehouses (avoid data egress from external source)

```
# Sample syntax
CREATE TABLE <catalog>.<schema>.<table-name>
(
  <column-specification>
```

```
)
LOCATION 's3://<bucket-path>/<table-directory>';
```

7. Create a managed table.

```
CREATE TABLE <catalog-name>.<schema-name>.<table-name>
(
  <column-specification>
);
```

8. Identify the location of a table.

```
# Either command works
DESCRIBE EXTENDED <table-name>
DESCRIBE DETAIL <table-name>
```

9. Inspect the directory structure of Delta Lake files.
   - path contains `/_delta_log/` and `*.snappy.parquet` files which form the delta table
   - delta log contains transactions in the form of `*.crc` and `*.json` files

```
display(dbutils.fs.ls(f"{path}/table"))
```

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

(2) Spark Jobs

Table +

	add	commitInfo
1	{"dataChange": true, "modificationTime": 1695566290429, "path": "part-00000-20716378-fc1d-4822-aacd-7f0e10b83d12-c000.snappy.parquet", "size": 1063, "stats": "{\"numRecords\":1,\"minValues\":{\"id\":2,\"name\":\"Omar\",\"value\":15.2},\"maxValues\":{\"id\":2,\"name\":\"Omar\",\"value\":15.2},\"nullCount\":{\"id\":0,\"name\":0,\"value\":0}}", "tags": {"INSERTION_TIME": "1695566290429000", "MAX_INSERTION_TIME": "1695566290429000", "MIN_INSERTION_TIME": "1695566267608000", "OPTIMIZE_TARGET_SIZE": "268435456"}}	null
2	{"dataChange": true, "modificationTime": 1695566290446, "path": "part-00002-7180bbaa-23a0-4244-8755-065662c0a1ce-c000.snappy.parquet", "size": 1063, "stats": "{\"numRecords\":1,\"minValues\":{\"id\":7,\"name\":\"Blue\",\"value\":7.7},\"maxValues\":{\"id\":7,\"name\":\"Blue\",\"value\":7.7},\"nullCount\":{\"id\":0,\"name\":0,\"value\":0}}", "tags": {"INSERTION_TIME": "1695566290429001", "MAX_INSERTION_TIME": "1695566290429001", "MIN_INSERTION_TIME": "1695566267608000", "OPTIMIZE_TARGET_SIZE": "268435456"}}	null
3	null	{"clusterId": "0913-021744-jb87gxwo", "engineInfo": "Databricks-Runtime, "isolationLevel": "WriteSerializable", "notebook": {"notebookId": "104757795 "operationMetrics": {"executionTimeMs": "5402", "numOutputRows": "2", "nu "0", "numTargetFilesAdded": "2", "numTargetFilesRemoved": "2", "numTarget "numTargetRowsInserted": "1", "numTargetRowsUpdated": "1", "rewriteTime "operationParameters": "[{\"predicate\":\"(type#6319 {\"predicate\":\"(type#6319 = delete)\",\"actionType\":\"delete\"}]", "notMat insert)\",\"actionType\":\"insert\"}]", "predicate": "[\"(id#6327 = id#6316)\"]" "txnId": "ef92fb1a-d905-4123-8c76-9443a67d325c", "userId": "619986985

5 rows | 0.61 seconds runtime — Refreshed 1 minute ago

Command took 0.61 seconds -- by jericodev00@gmail.com at 9/24/2023, 10:43:11 PM on 11.3 n2

10. Identify who has written previous versions of a table.
11. Review a history of table transactions.

`SQL DESCRIBE HISTORY <table-name>`

```
![[Pasted image 20230924224721.png]]
```

12. Roll back a table to a previous version.
13. Identify that a table can be rolled back to a previous version.
14. Query a specific version of a table.

```
# Query what previous version looks like (time travel)
SELECT * FROM students VERSION AS OF 3;

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

15. Identify why Zordering is beneficial to Delta Lake tables.
    - z-ordering = indexing for delta tables (afaik, only works on delta tables). For parquet, you can convert them first to `DELTA` format.

```
OPTIMIZE students
ZORDER BY id
```

16. Identify how vacuum commits deletes.
    - VACUUM deletes old versions of a table (the snappy parquet files)
    - does not delete the delta log, so we can still see the history of the table via `DESCRIBE HISTORY`

```
# By default you cannot delete table versions that are less than 7 days old, we change this for demonstration
SET spark.databricks.delta.retentionDurationCheck.enabled = false;
SET spark.databricks.delta.vacuum.logging.enabled = true;

# DRY RUN first to see which files will be deleted (`*.snappy.parquet files`)
VACUUM students RETAIN 0 HOURS DRY RUN
```

```
VACUUM students RETAIN 0 HOURS
```

17. Identify the kind of files Optimize compacts. [1]
    - small files are compacted and balanced out (combined towards an optimal size, determined by table size)
    - idempotent process

18. Identify CTAS (`CREATE TABLE AS SELECT`) as a solution.

- autoinfer schema from input (only works for sources with well defined schema, i.e. parquet/existing tables)
- does not support `OPTIONS` for csvs
  - you will need to use temp views first with options, then reference the view in CTAS
  - `CREATE OR REPLACE TEMP VIEW.... USING CSV ... OPTIONS...`

19. Create a generated column.
    - column made from another column (lateral referencing)
    - when inserting into table w/ generated column via values, the value needs to be correct/consistent with the generation equation

20. Add a table comment.
    - either in column, or for whole table

```
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")
```

```
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}/ecommerce/raw/users-historical/`;
```

21. Use CREATE OR REPLACE TABLE and INSERT OVERWRITE
    - CRAS options:
      - COMMENT `comment`
      - LOCATION `location`
      - PARTITION BY `column\s`
      - DEEP CLONE - full copy of data and metadata
      - SHALLOW CLONE - metadata only (delta log)
      - CRAS old table still exists via delta time travel

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


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

22. Compare and contrast CREATE OR REPLACE TABLE and INSERT OVERWRITE
    - INSERT OVERWRITE
      - can only work on existing tables
      - only works when schema to insert is correct (matches target table). this is due to delta's schema on write policy
      - can overwrite individual partitions

23. Identify a scenario in which MERGE should be used.
    - when you need to upsert in a single transaction
24. Identify MERGE as a command to deduplicate data upon writing.
    - only INSERT data, WHEN NOT MATCHED
25. Describe the benefits of the MERGE command.
    - custom logic
    - combine update, insert, and delete as single transaction

26. Identify why a COPY INTO statement is not duplicating data in the target table.
    - COPY INTO is idempotent as compared to INSERT INTO, which just appends per use
    - COPY INTO uses a directory/file path, while INSERT INTO uses a query
27. Identify a scenario in which COPY INTO should be used.
    - for incrementally loading data into a table FROM a source that continuously receives data
28. Use COPY INTO to insert data.

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