- By convention, Early tables must represent data in raw form, and validation & enrichment must be done at later stages.

This ensures that no data is dropped even if data does not match the standards.

CREATE TABLE _ AS SELECT :-

- CREATE TABLE <table_name> AS SELECT statements creates & populates Delta tables using data retrieved from an input query.

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

DESCRIBE EXTENDED sales;
```

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.

We can see how this would present significant limitations when trying to ingest data from CSV files.

- In case of CSV files, CTAS statements cannot render each data values, and gives each line as a single records' attribute value.

≥ For extracting data from CSV, registering external table, by defining schemas, and other required options, is the best approach :-

```
CREATE TABLE IF NOT EXISTS sales_delta2
  (
    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}/ecommerce/raw/sales-csv",
  header = "true",
  delimiter = "|"
);

SELECT * FROM sales_delta
```

≥ Renaming, Dropping, Adding Columns to a Table, to form New Table :-

≥ CTAS statements can be used to accomplish this :-

test_tbl

user_id	name	email

```
CREATE TABLE tbl2 AS
SELECT user_id AS id, name FROM test_tbl;
```

⇒ New table will contain subset of existing table, that too with a renamed row.

Declare Schema with Generated Columns

As noted previously, CTAS statements do not support schema declaration. We note above that the timestamp column appears to be some variant of a Unix timestamp, which may not be the most useful for our analysts to derive insights. This is a situation where generated columns would be beneficial.

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 (introduced in DBR 8.3).

The code below demonstrates creating a new table while:

1. Specifying column names and types
2. Adding a generated column to calculate the date
3. Providing a descriptive column comment for the generated column

```
CREATE OR REPLACE TABLE purchase_dates (
  id STRING,
  transaction_timestamp STRING,
  price STRING,
  date DATE GENERATED ALWAYS AS (
    cast( cast(transaction_timestamp/1000000 AS TIMESTAMP) AS DATE )
    )
    COMMENT "generated based on `transactions_timestamp` column"
)
```

due to this (Generated Column), when writing to this table, if we do not pass value to this column, its value will be automatically generated. [Just like default value, where the value is calculated using provided logic]

Generated Columns are a special implementation of CHECK constraint.

Add a Table Constraint

Generated columns are a special implementation of check constraints.

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
- `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.

```
nd 27

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

nd 28
```

Table constraints are shown in the `TBLPROPERTIES` field.

```
nd 29

1  DESCRIBE EXTENDED purchase_dates
```

Enrich Tables with Additional Options and Metadata, in CTAS statements

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

We also include logic to create a new date column derived from timestamp data in the source.

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

NOTE: Partitioning is shown here primarily to demonstrate syntax and impact. 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.

```
CREATE OR REPLACE TABLE users_pii
COMMENT "Contains PII"
LOCATION "${da.paths.working_dir}/tmp/users_pii"
PARTITIONED BY (first_touch_date)
AS
  SELECT *,
    cast(cast(user_first_touch_timestamp/1000000 AS TIMESTAMP) AS DATE) first_touch_date,
    current_timestamp() AS updated,
    input_file_name() AS source_file
  FROM parquet.`${da.paths.datasets}/ecommerce/raw/users-historical/`;
```

path in which the newly created table will get stored.

→ parquet file records will be placed in seperate folders (inside table folder), where each folder will be named on the basis of this partition column

⇒ All comments and properties for a table can be viewed using DESCRIBE TABLE EXTENDED

Cloning Delta Lake Tables :-

- Delta lake supports 2 types of cloning :-

DEEP CLONE

- Fully copies data and metadata from a source table to a target table.
- This copy occurs incrementally so executing the command again syncs the changes.

```
CREATE OR REPLACE TABLE
<new_cloned_table_name>
DEEP CLONE <original_table>
```

DEEP CLONE → Command to deep clone

SHALLOW CLONE

- Only copies the Delta transaction logs, i.e, the data does not moves (actual data is not copied.)

```
CREATE OR REPLACE TABLE
<new_cloned_table_name>
SHALLOW CLONE <original_table>
```

SHALLOW CLONE → Command to shallow clone

No use of AS keyword while CLONING.

- Cloning is great way for testing SQL codes, as cloned table transactions are stored and recorded seperately.