**Step 2 : Defining the schema:**

The schema definition of fact tables are like this :-

1. **fact.transactions :**

|  |  |  |
| --- | --- | --- |
| Column Name | Data Type | Description |
| Order\_id | INTEGER | This helps us to identify the individual order. |
| Line\_id | INTEGER | This helps us to identify the individual line |
| type | STRING | Helps to identify the type of order this is. |
| dt | DATETIME | Gives the date |
| Pos\_site\_id | STRING | Helps to give the individual site id. |
| Sku\_id | INTEGER | Gives the individual SKU’s id |
| Fscldt\_id | INTEGER | Gives the id as per the fiscal date |
| Price\_substate\_id | STRING | Gives the substate id |
| Sales\_units | INTEGER | Indicates the units sold |
| Sales\_dollars | INTEGER | Gives the sales in dollars |
| Original\_order\_id | INTEGER | Gives the original order id |
| Original line\_id | INTEGER | Gives the original line id |

1. **fact.averagecosts**

|  |  |  |
| --- | --- | --- |
| Column Name | Data Type | Description |
| Fscldt\_id | INTEGER | Gives the id as per the fiscal date |
| Sku\_id | INTEGER | Gives the individual SKU’s id |
| Average\_unit\_standardcost | INTEGER | Gives the average unit standard cost |
| Average\_unit\_landedcost | INTEGER | Gives the average unit landed cost |

And the similar schema structure for hierarchy or dimension table.

Now we need to load this data to cloud storage like AWS S3.

**Step 3 : Loading data into cloud Storage**

**Using AWS S3 :**

aws s3 cp fact.transactions.dlm.gz s3://yash/raw-data/

aws s3 cp hier.prod.dlm.gz s3://yash/raw-data/

**Step 4 : Load the data into staging table**

**Staging table for fact.transactions**

CREATE TABLE staging.transactions (

order\_id INTEGER NOT NULL, -- Unique order identifier

line\_id INTEGER NOT NULL, -- Unique line identifier within an order

type STRING, -- Type of order

dt DATETIME, -- Order date

pos\_site\_id STRING, -- Point of sale site ID (FK to hier.possite)

sku\_id INTEGER, -- SKU ID (FK to hier.prod)

fscldt\_id INTEGER, -- Fiscal date ID (FK to hier.clnd)

price\_substate\_id STRING, -- Pricing substate ID (FK to hier.pricestate)

sales\_units INTEGER, -- Number of units sold

sales\_dollars INTEGER, -- Sales amount in dollars

original\_order\_id INTEGER, -- Reference to the original order

original\_line\_id INTEGER, -- Reference to the original line

PRIMARY KEY (order\_id, line\_id), -- Composite Primary Key

FOREIGN KEY (pos\_site\_id) REFERENCES staging.possite(pos\_site\_id),

FOREIGN KEY (sku\_id) REFERENCES staging.prod(sku\_id),

FOREIGN KEY (fscldt\_id) REFERENCES staging.clnd(fscldt\_id),

FOREIGN KEY (price\_substate\_id) REFERENCES staging.pricestate(price\_substate\_id)

);

**Staging table for fact.averagecosts**

CREATE TABLE staging.averagecosts (

fscldt\_id INTEGER NOT NULL, -- Fiscal date ID (FK to hier.clnd)

sku\_id INTEGER NOT NULL, -- SKU ID (FK to hier.prod)

average\_unit\_standardcost INTEGER, -- Average unit standard cost

average\_unit\_landedcost INTEGER, -- Average unit landed cost

PRIMARY KEY (fscldt\_id, sku\_id),

FOREIGN KEY (fscldt\_id) REFERENCES staging.clnd(fscldt\_id),

FOREIGN KEY (sku\_id) REFERENCES staging.prod(sku\_id)

);

**Loading the data into tables :-**

COPY INTO staging.transactions

FROM 's3://yash/raw-data/fact.transactions.dlm'

CREDENTIALS 'aws\_role=arn:aws:iam::admin’

DELIMITER '|'

IGNOREHEADER 1;

**Step 5 : Transform and normalize the data**

The goal is to:

1. **Normalize hierarchy tables** (split multi-level hierarchies into separate tables).
2. **Ensure fact tables reference dimension tables** via **Foreign Key constraints**.

**We can normalize the transactions by joining hierarchy tables to replace IDs with meaningful names.**

CREATE TABLE refined.transactions AS

SELECT

t.order\_id,

t.line\_id,

t.type,

t.dt,

p.store\_name AS pos\_site\_name,

pr.product\_name AS sku\_name,

c.year,

c.month,

c.week,

t.sales\_units,

t.sales\_dollars,

t.discount\_dollars

FROM staging.transactions t

LEFT JOIN staging.possite p ON t.pos\_site\_id = p.pos\_site\_id

LEFT JOIN staging.prod pr ON t.sku\_id = pr.sku\_id

LEFT JOIN staging.clnd c ON t.fscldt\_id = c.fscldt\_id;

**Step 6 : Aggregation**

This will create a materialized view

CREATE TABLE refined.mview\_weekly\_sales AS

SELECT

t.pos\_site\_id,

t.sku\_id,

t.fscldt\_id,

t.price\_substate\_id,

t.type,

SUM(t.sales\_units) AS total\_sales\_units,

SUM(t.sales\_dollars) AS total\_sales\_dollars,

SUM(t.discount\_dollars) AS total\_discount\_dollars

FROM staging.transactions t

GROUP BY t.pos\_site\_id, t.sku\_id, t.fscldt\_id, t.price\_substate\_id, t.type;

**Step 7 : Incremental Updates**

**To handle the incremental data loads we’ll use this :-**

MERGE INTO refined.mview\_weekly\_sales AS target

USING (

SELECT

pos\_site\_id,

sku\_id,

fscldt\_id,

price\_substate\_id,

type,

SUM(sales\_units) AS total\_sales\_units,

SUM(sales\_dollars) AS total\_sales\_dollars,

SUM(discount\_dollars) AS total\_discount\_dollars

FROM staging.transactions

GROUP BY pos\_site\_id, sku\_id, fscldt\_id, price\_substate\_id, type

) AS source

ON target.pos\_site\_id = source.pos\_site\_id

AND target.sku\_id = source.sku\_id

AND target.fscldt\_id = source.fscldt\_id

AND target.price\_substate\_id = source.price\_substate\_id

WHEN MATCHED THEN

UPDATE SET

target.total\_sales\_units = target.total\_sales\_units + source.total\_sales\_units,

target.total\_sales\_dollars = target.total\_sales\_dollars + source.total\_sales\_dollars,

target.total\_discount\_dollars = target.total\_discount\_dollars + source.total\_discount\_dollars

WHEN NOT MATCHED THEN

INSERT (pos\_site\_id, sku\_id, fscldt\_id, price\_substate\_id, type, total\_sales\_units, total\_sales\_dollars, total\_discount\_dollars)

VALUES (source.pos\_site\_id, source.sku\_id, source.fscldt\_id, source.price\_substate\_id, source.type, source.total\_sales\_units, source.total\_sales\_dollars, source.total\_discount\_dollars);