Deeper into Modern Warehousing

Rahul Dave(@rahuldave), Univ.Ai

Slowly Changing Dimensions(SCD)

- What happens when a dimension changes? In most bizdev situations like a product sku, or a customer's phone number, this is an infrequent change and the dimension is called a SCD.

- There are 3 kinds of SCDs.

    1. Type 1: Overwrite existing dimension records. History is lost.

    2. Type 2: Keep history. When a record changes, flag it as changed, and create a new row in the dimension table that reflects the current status of the dimension attributes

    3. Type 3: create a new column instead of a new row!

Type 1

Product Key	Product Description	Department	SKU Number (Natural Key)
12345	IntelliKidz 1.0	Education	ABC922-Z

Product Key	Product Description	Department	SKU Number (Natural Key)
12345	IntelliKidz 1.0	Strategy	ABC922-Z

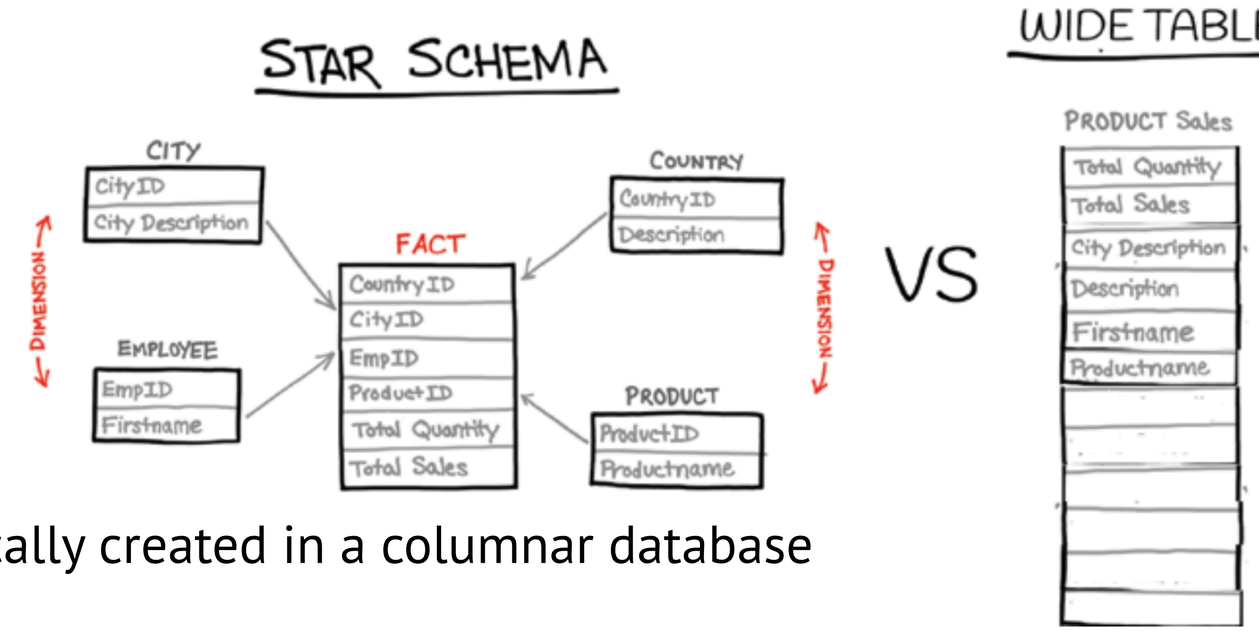
Type 2

Product Key	Product Description	Department	SKU Number (Natural Key)
12345	IntelliKidz 1.0	Education	ABC922-Z
25984	IntelliKidz 1.0	Strategy	ABC922-Z

Type 3

Product Key	Product Description	Department	Prior Department	SKU Number (Natural Key)
12345	IntelliKidz 1.0	Strategy	Education	ABC922-Z

Wide denormalized tables



- a wide table is a collection of many fields, typically created in a columnar database
- field may be single valued or have nested data, although there is great benefit in having strongly typed cells
- there can be one or multiple keys tied to the grain of the data
- wide tables are usually sparse: run-length encoding makes this cheap in a columnar database
- fields can be added over time to update the table
- while one still defines dimension tables as a single source of truth, dimensions can be included in wide tables to eliminate joins and improve performance

Kimball relaxation

- With Kimball you spend time up-front deciding the schema and write ETL tools. At that time, memory was low, storage was expensive, and BI tools ran on OLAP cubes. Modeling was needed,

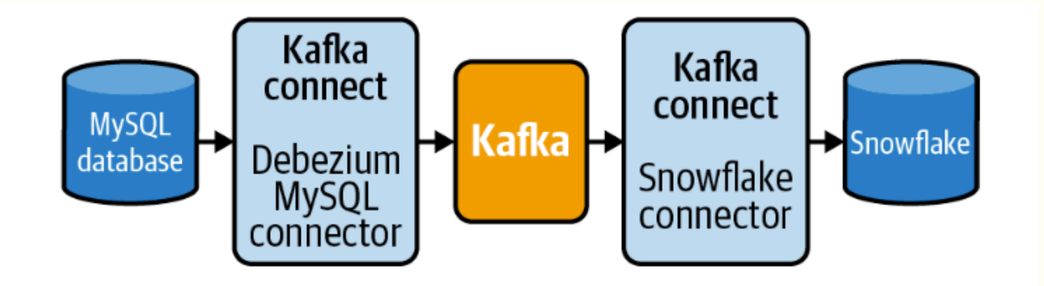
- Now with wide tables in columnar databases, modeling is optional. If we know the questions to answer, we can define modeled fact tables. But we can start with wide fact tables just taken from the source.

- What about snapshot tables which coarse the grain? With columnar tables chewing through millions of rows in seconds, you might not need to coarse grain. You can even spin up multiple servers to handle your query. No need to create snapshots for reports not widely used. Still snapshotting is a best practice, especially for more important, higher throughput queries such as sales per day.

- Accumulating snapshot tables are rare in practice, and need to be designed, so the old way carries on to the day.

CDC Extraction

- You can also extract data via **Change Data Capture** (CDC). This uses the source database's WAL or binlog to capture all events.
- The WAL is typically used to replicate database tables in a cluster without having to resort to a SQL query. Here we can hook into the replication system and copy events.
- As a result we will get ALL the events, including deletes, and must remove these when reconstituting the source database in the warehouse.

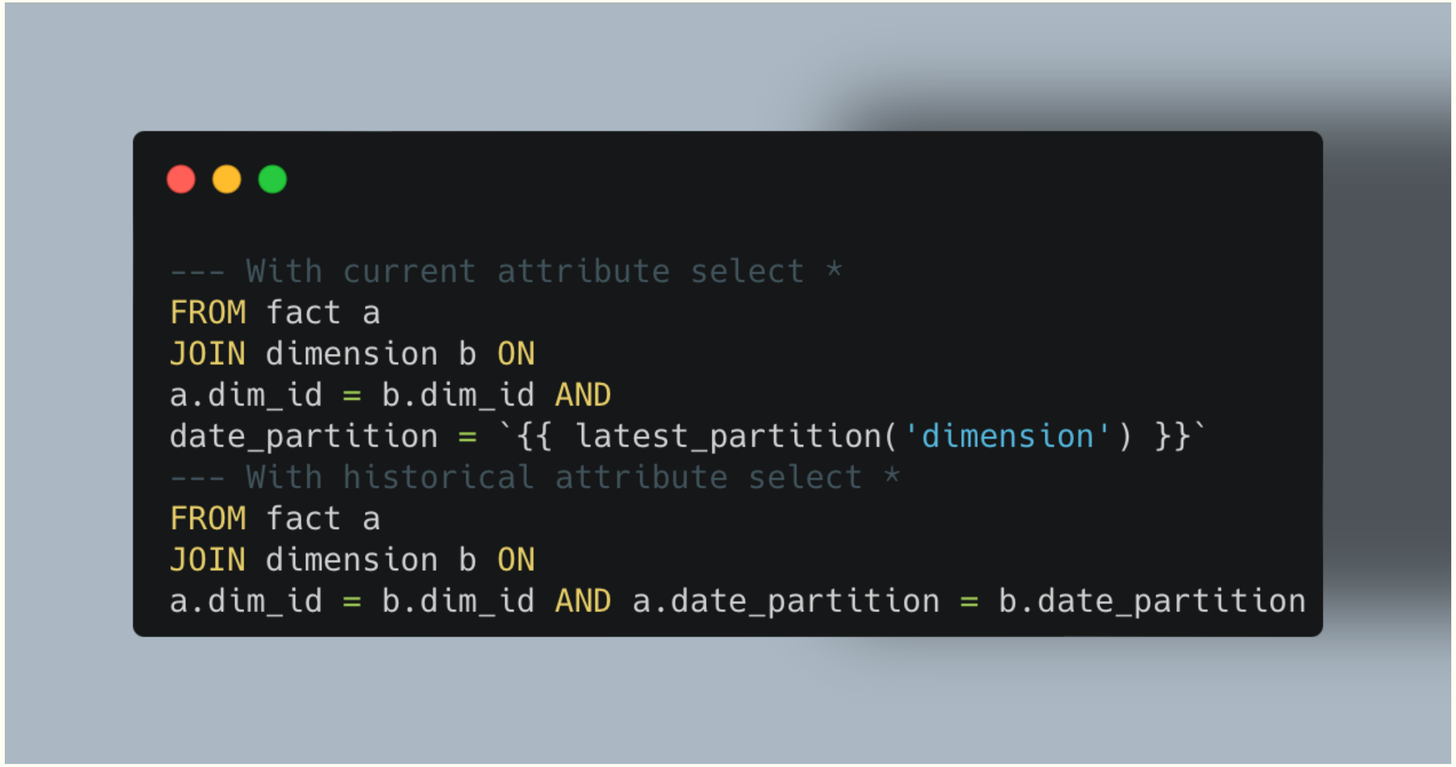


EventType	OrderId	OrderStatus	LastUpdated
insert	1	Backordered	2020-06-01 12:00:00
update	1	Shipped	2020-06-09 12:00:25
delete	1	Shipped	2020-06-10 9:05:12

Type 2 SCD and Partitions

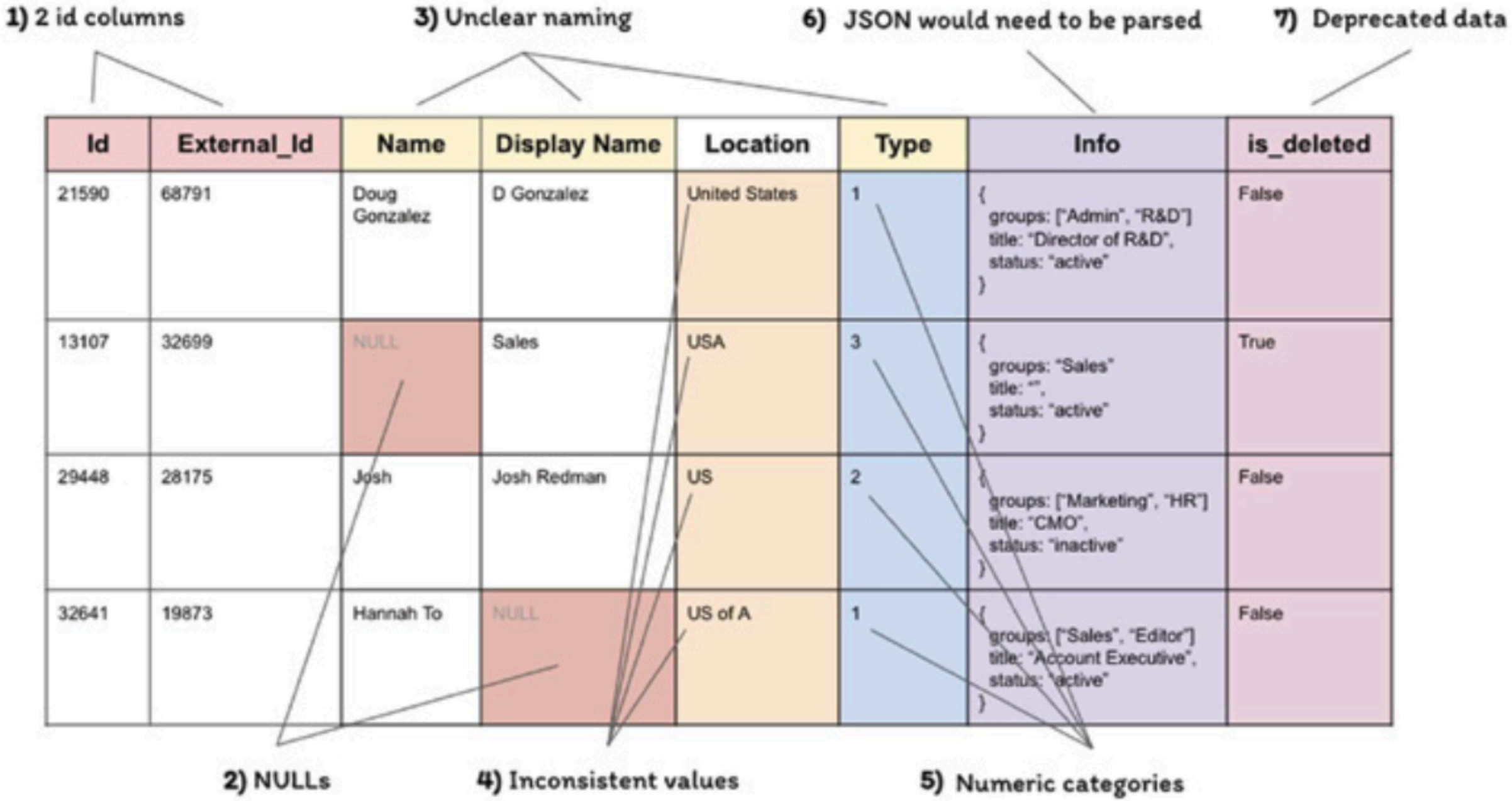
- If dimensions are small, you can create fine-grained **partitions** that contain the whole dimension. This is easier than doing type2 SCDs for history and immediately gives you the current state of the dimension. If the dimension is large, like a customer dimension, you can do incremental partitions (but more work is required to get the current state)

- See https://www.youtube.com/watch?v=4Spo2QRTz1k&t=989s .



Working through a use case

- start with the source table
- this might be fully updated or incrementally updated
- this updating might have happened in the data lake or in the warehouse
- if in a data lake you are using an external sql engine such as spark/duckdb/ presto/trino.


1) 2 id columns
3) Unclear naming
6) JSON would need to be parsed
7) Deprecated data
Id
External_Id
Name
Display Name
Location
Type
Info
is_deleted
21590
68791
Doug Gonzalez
D Gonzalez
United States
1
{ groups: ["Admin", "R&D"] title: "Director of R&D", status: "active" }
False
13107
32699
NULL
Sales
USA
3
{ groups: "Sales" title: "", status: "active" }
True
29448
28175
Josh
Josh Redman
US
2
{ groups: ["Marketing", "HR"] title: "CMO", status: "inactive" }
False
32641
19873
Hannah To
NULL
US of A
1
{ groups: ["Sales", "Editor"] title: "Account Executive", status: "active" }
False
2) NULLs
4) Inconsistent values
5) Numeric categories

t


1) Remove unused column
2) Add consistent column
3) Rename and Standardize
4) Make column names and values descriptive
5) Parse relevant field into new column, remove original column
6) Filter row that was deprecated remove column
id
External_Id
email
name
display_name
country
access_level
Info
is_active
is_deleted
21590
68791
dgonzalez@gmail.com
Doug Gonzalez
D Gonzalez
USA
Can view
{ groups: ["Admin", "R&D"] title: "Director of R&D", status: "active" }
true
False
13107
32699
lisaf@yahoo.com
Sales
USA
Can Admin
{ groups: "Sales" title: "", status: "active" }
true
True
29448
28175
josh@gmail.com
Josh
Josh Redman
USA
Can edit
{ groups: ["Marketing", "HR"] title: "CMO", status: "inactive" }
false
False
32641
19873
hannah@aol.com
Hannah To
BRA
Can view
{ groups: ["Sales", "Editor"] title: "Account Executive", status: "active" }
true
False

```
SELECT id,
  -- external_id, -- unused
  Name,
  Display Name,
  Location,
CASE
          WHEN Type = "1" THEN "Can view"
          WHEN Type = "2" THEN "Can edit"
          WHEN Type = "3" THEN "Can admin"
          END AS access_level,
Info,
  is_deleted
FROM
app.Users
```

```
SELECT id,
  -- external_id, -- unused
  Name,
  Display Name,
  Location,
access.“type” AS access_level, --type --replaced by access_level Info,
is_deleted
FROM
  app.Users
JOIN
  app.access
ON
  Users.type = access.id
```

```
SELECT id,
  -- external_id, -- unused
  Name,
  Display Name,
  CASE
            WHEN Location = "United States" THEN "USA"
            WHEN Location = "US" THEN "USA"
            WHEN Location = "US of A" THEN "USA"
            ELSE Location
END AS country,
access.”type” AS access_level, --type --replaced by access_level CASE
            WHEN Info = "%active" THEN True
            WHEN Info = "%inactive" THEN False
  END AS is_active,
  is_deleted
FROM
  app.Users
JOIN
  app.access
ON
  Users.type = access.id
```

Create a table, view, or materialized view

- These are created IN the warehouse
- The big idea here is after having copied the source tables (source-refreshed) into the data lake (as csv/parquet) or warehouse (in tables) we transform them as needed.
- Some of these transforms, such as de-duplication, stemming, etc) could even be done with Spark in the lake
- Once in the warehouse, all transformations happen there and new tables/views/ materialized views are created

```
CREATE VIEW stg_users AS
SELECT
  Id as id,
  -- External_Id -- unused
  Name as name,
  Display Name as display_name,
  contacts.Email as email,
  CASE
            WHEN Location = "United States" THEN "USA"
            WHEN Location = "US" THEN "USA"
            WHEN Location = "US of A" THEN "USA"
            ELSE Location
  END AS country,
  access.”type” AS access_level,
  --type --replaced by access_level
  CASE
            WHEN Info = "%active" THEN True
            WHEN Info = "%inactive" THEN False
  END AS is_active
FROM
  app.users
JOIN
app.contacts
  ON users.Id = contacts.Id
JOIN
app.access
  ON Users.type = access.id
WHERE
  is_deleted != True
```

Transformations in the Warehouse

- These transformations are “in” the warehouse (or lake to warehouse). This is the *T* in EtLT.
- We start from source tables with a schema and create new tables, views, or more precisely materialized views. A view does have a schema, but it is derived from the underlying tables and the SELECT statement used to define the view.
- Views abstract the underlying table structure, making it easier for users to work with data **without worrying about the underlying schema**. They also enforce consistency across multiple applications by providing a single point of access to specific data elements.
- View encapsulate often used queries, making them easier to reuse and maintain. Materialized views are useful when you have complex and time-consuming queries that involve aggregations, calculations, or joins. By storing the result set, the materialized view can return data much faster than executing the query each time. Materialized views can be set to refresh automatically at specific intervals or upon specific events, ensuring that the data stays up-to-date.
- Materialized views provide a clear separation between the base tables and the aggregated or pre-processed data, making it easier to maintain and understand the data structure.