Status	Done
Due	@November 15, 2024
Project	Data Expert.io
Tags	

Dimensional data modeling [w1d1]

> Week 1 day 1: This week we are focusing all our attention on dimension data modeling

Key concepts this week

1. What is data modeling
2. Dimension data modeling and it's types
3. Knowing Your Data Consumer
4. OLTP ↔ Master data ↔ OLAP
    - OLTP(transactional): Focused on real-time operations and transactions (e.g., banking apps).
    - Master data: the bridge between both
    - OLAP(analytical): Designed for querying and analyzing large datasets for analytical purposes
3. Cumulative Table Design
4. Compactness vs. Usability Tradeoff

5. Temporal Cardinality Explosion
6. Run-Length Encoding Compression Gotchas

What is Data Modeling?

Data modeling is the foundation of any robust data system, defining how data is structured, stored, and accessed. It organizes data into **facts** (measurable events, like sales or transactions) and **dimensions** (descriptive attributes, like product, location, or time) to enable meaningful analysis.

Dimension

A dimension represents an attribute or characteristic of an entity. Think of it as descriptive information about a person, place, or thing. For example:

- A user's birthday (a characteristic of a user).
- A user's favorite food (another characteristic or preference).
- A product's brand or category

Types of dimension:

1. Identifying Dimensions:

These uniquely define or identify an entity. For example, a **user ID or product ID** is a dimension that uniquely identifies a user/product

2. Descriptive Dimensions:

These provide additional details about an entity but don't uniquely identify it. For example, a **products brand**, **user country,** etc

Categories of dimension:

> classified based on how they are changed over time

1. Fixed dimension - remain constant and do not change once recorded. These include attributes like a user's **date of birth** or **sign-up date**, which are inherently static.

2. SCD (Slowly changing dimension) - change over time but not frequently. They capture updates to attributes to maintain historical accuracy or reflect the most current values. For example, a user’s **address** or **job title** might change occasionally, and these updates are tracked to ensure the data remains relevant and accurate for analysis.

Know your customers

While designing data models, keep in mind **“who will use the data”** and **“how they will use it”**

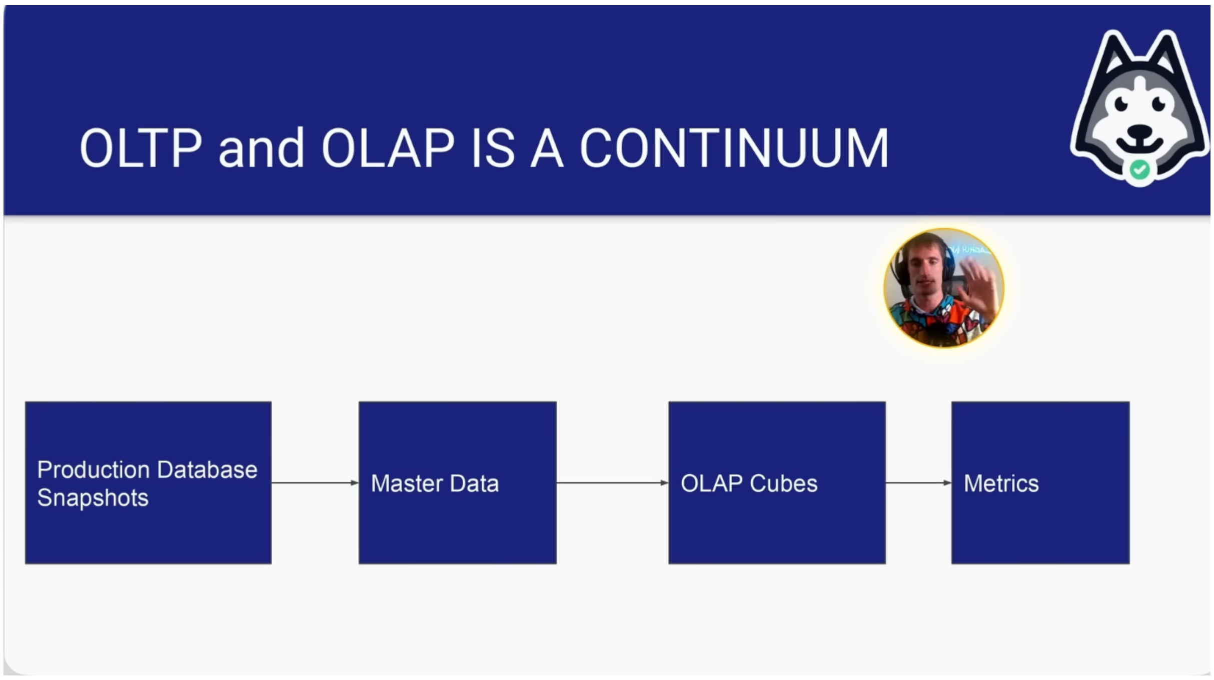
- **Analysts and Data Scientists**
    - Require analytical datasets (OLAP).
    - Prefer easy-to-query data with simpler data types like decimals and strings.
- **Data Engineers**
    - Work with master data involving complex structures (nested, struct, arrays).
    - Handle datasets that are harder to query.
- **ML Engineers**
    - Focus on datasets with identifiers and numerical features.
    - Tailor data to the specific requirements of machine learning models and training processes.
- **Non-Technical Customers**
    - Need data in its simplest form—easy to read, with no querying required.
    - Prefer patterns, annotations, and charts for better understanding.

By aligning your data models with the specific needs of each audience, you can create solutions that are both practical and impactful.

OLAP vs OLTP vs master data

OLTP → mostly for application systems usually in 3NF

OLAP → mostly for analytical purposes



OLTP and OLAP IS A CONTINUUM

Cumulative table design

A cumulative table design is an approach that tracks historical data and changes over time.It

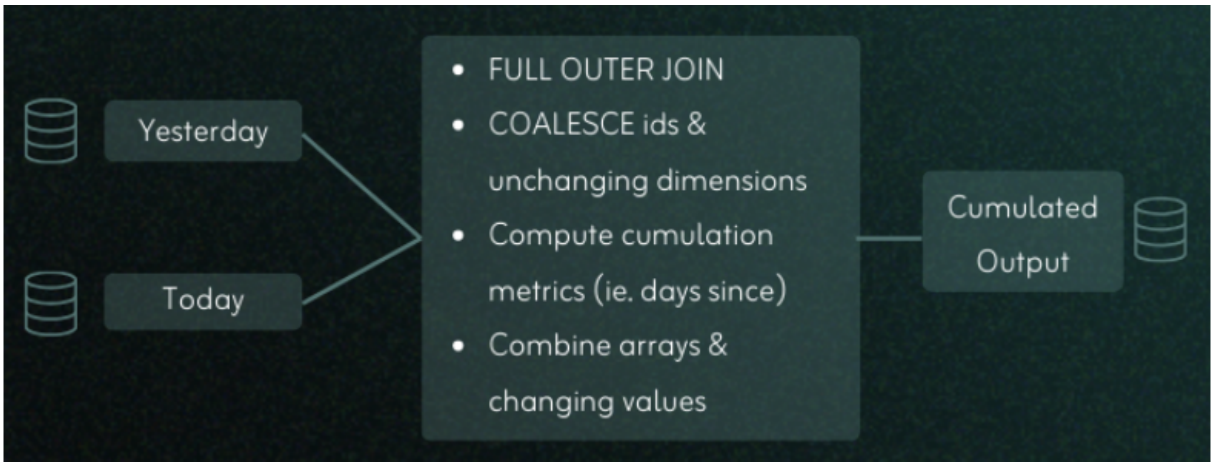
- Maintains a complete history of data by combining yesterday's snapshot with today's updates
    - Uses FULL OUTER JOIN to merge previous and new data, ensuring that:
        - New records are added
        - Existing records are updated
        - Historical records are preserved

Key benefits include:

- Complete historical tracking
- Simplified analysis without losing past data
- Ability to track transitions and changes over time

However, there are some drawbacks:

- Data must be built sequentially
- Difficult to backfill missing historical data
- Can be challenging to handle personal information (PII)


Yesterday
Today
FULL OUTER JOIN
COALESCE ids & unchanging dimensions
Compute cumulation metrics (ie. days since)
Combine arrays & changing values
Cumulated Output

Facebook example for cumulative table design

Facebook uses a **cumulative table** to track all users (dim_all_users) and maintain their history. This table helps understand key user segments and behaviors, such as:

- **New Users**: Users who signed up for the first time.
- **Returning Users**: Users who had been inactive for some time but re-engaged.
- **Churned Users**: Users who stopped engaging or deactivated their accounts.

The cumulative table ensures no data is lost while maintaining a history of users' actions, status changes, and attributes over time.

Key Components of the Design

1. **Data Sources**:
    - **Yesterday's Snapshot**: The state of all users up to a certain date (e.g., active, inactive, new).
    - **Today's Snapshot**: Updates, such as new signups, reactivations, or deactivations from the last 24 hours.

2. **FULL OUTER JOIN**:
    - Combines yesterday's data and today's updates, ensuring that:
    - New users are added.
    - Existing users are updated (e.g., activity or status change).
    - Historical records are retained for inactive or churned users.
3. **COALESCE**:
    - Merges attributes from both snapshots, giving priority to the most recent data.
4. **History Preservation**:
    - By using this approach daily, a cumulative table (dim_all_users) is built that keeps track of the entire user history.

Yesterday's Snapshot (dim_all_users_yesterday):

User ID	Last Login	Status
1	2024-11-14	Active
2	2024-10-01	Inactive
3	NULL	New

Today's Snapshot (dim_all_users_today):

User ID	Last Login	Status
1	2024-11-15	Active
2	2024-11-15	Active
4	2024-11-15	New

FULL OUTER JOIN:

Combining these tables with a FULL OUTER JOIN and COALESCE to merge columns:

User ID	Last Login	Status
1	2024-11-15	Active
2	2024-11-15	Active
3	NULL	New

4	2024-11-15	New

Key Insights from the Cumulative Table

By updating this cumulative table daily, Facebook can derive actionable insights:

1. **New Users**: Users with their first recorded activity (e.g., Status = New): User ID 4 is a new user added today.
2. **Active Users**: Users who logged in or engaged recently (within the last X days). User IDs 1 and 2 are active.
3. **Reactivated Users**: Previously inactive users who logged in again (status transitioned from Inactive to Active). Example: User ID 2 became active after being inactive since 2024-10-01.
4. **Churned Users**: Users who stopped engaging for an extended period: If User ID 3 remains inactive for a set time threshold, they would be flagged as churned.

other use cases:

5. **User Growth**: Total number of new users per day or week.
6. **Churn Analysis**: Identifying churn patterns by tracking users who moved from Active to Inactive. Analyzing time-to-churn (e.g., average inactivity before churn).
7. **Retention Analysis**: Measuring returning users within X days of their last activity.
8. **Behavior Tracking**: Tracking user journey transitions (e.g., New → Active → Inactive → Reactivated).

Strengths and Drawbacks

> strengths of cumulative table design

1. **Historical Analysis Without Shuffle**:

   The cumulative table retains historical data for every entity (e.g., users, customers) in a sequential and non-destructive manner. This means you can analyze how data points (e.g., user activity, purchases) change over time without repeatedly shuffling or reprocessing historical data.

- Example: If you need to know how a user's behavior changed over months, the cumulative table already has the entire timeline, avoiding the need for expensive computations.

2. **Easy "Transition" Analysis**:

   A cumulative table simplifies tracking transitions between states over time, such as: when a user moves from inactive to active or new to existing or when a product changes categories (e.g., from a "trial" product to a "premium" subscription). This is especially useful in cases like state transition modeling, where the focus is on understanding how and when entities change states.

> drawbacks

1. **Can Only Be Backfilled Sequentially**:

   Data in a cumulative table must be built **sequentially**, day by day, as it relies on the difference between snapshots. If historical data is missing or corrupt for a specific period, it can be challenging to backfill accurately.

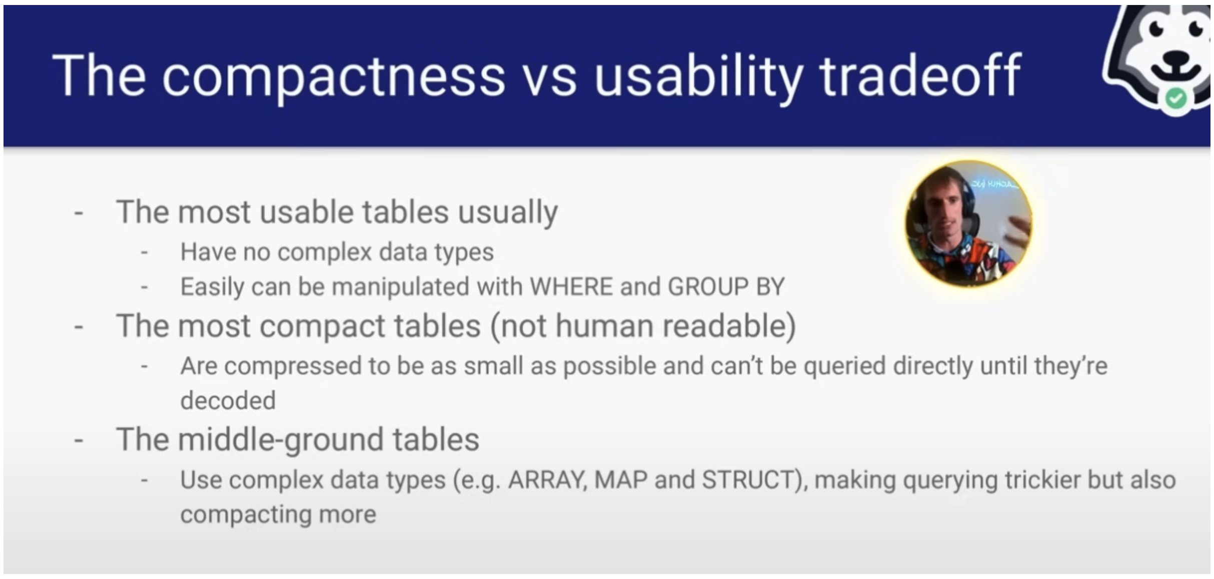
2. **Handling PII Data Can Be a Mess**:

   When managing Personally Identifiable Information (PII) (e.g., names, emails, phone numbers), cumulative tables carry forward inactive or deleted users.

   This creates challenges for compliance with data privacy laws (e.g., GDPR, CCPA), which may require PII to be permanently deleted.

   - Example: If a user deletes their account but their data persists in cumulative tables, ensuring compliance with "right to be forgotten" laws becomes complex.

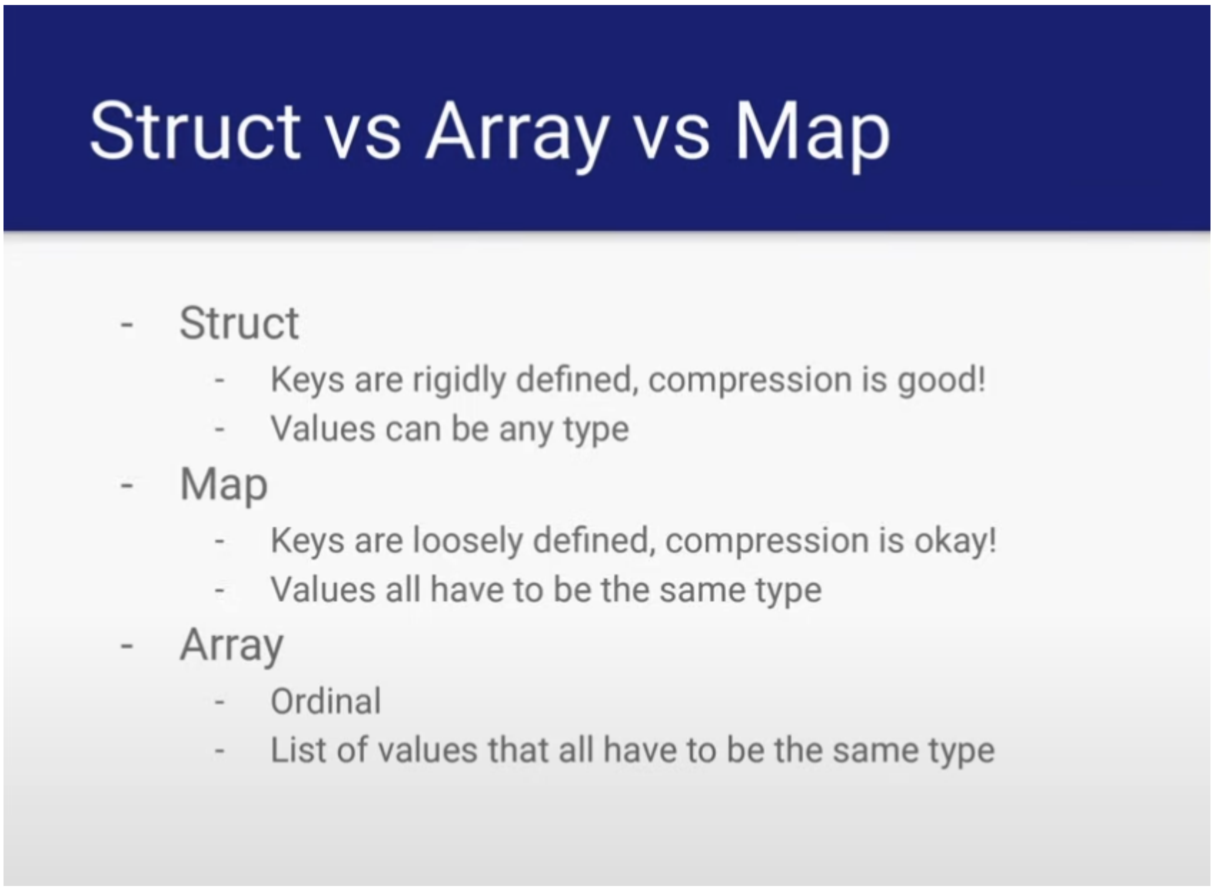
Compactness vs Usability tradeoff



The compactness vs usability tradeoff

- The most usable tables usually
  - Have no complex data types
  - Easily can be manipulated with WHERE and GROUP BY
- The most compact tables (not human readable)
  - Are compressed to be as small as possible and can't be queried directly until they're decoded
- The middle-ground tables
  - Use complex data types (e.g. ARRAY, MAP and STRUCT), making querying trickier but also compacting more

Struct vs Array vs Map



Struct vs Array vs Map

- Struct
  - Keys are rigidly defined, compression is good!
  - Values can be any type
- Map
  - Keys are loosely defined, compression is okay!
  - Values all have to be the same type
- Array
  - Ordinal
  - List of values that all have to be the same type

Struct vs. Array vs. Map: What's the Difference?

1. **Struct**:

   Think of a **Struct** as a fixed blueprint with clearly defined keys (like a form you fill out).

   - **Keys**: Predefined and rigid; every key must exist (e.g., `name`, `age`, `address`).
   - **Values**: Can be of any type (e.g., `name` could be text, `age` could be a number).
   - **Compression**: Very efficient because the structure is consistent across all records.

   Example:

   ```
   {
     "name": "Alice",
     "age": 25,
     "address": "123 Main St"
   }
   ```

2. **Map**:

   A **Map** is like a flexible dictionary or key-value store. The keys are not predefined and can vary.

   - **Keys**: Loosely defined; they don't have to follow a specific structure (e.g., you can add/remove keys).
   - **Values**: All values must be of the **same type** (e.g., all strings or all numbers).
   - **Compression**: Less efficient because the keys are stored for each record.

   Example:

   ```
   {
     "apple": 10,
     "banana": 5,
     "orange": 7
   }
   ```

3. **Array**:

   An **Array** is a simple, ordered list of items. Think of it like a to-do list where each item is in a specific order.

   - **Order Matters**: Items are stored sequentially (position is important).
   - **Values**: All values must be of the **same type** (e.g., all numbers or all strings).
   - **Compression**: Good, as only the values are stored (no keys).
   - **Example**:

     ```
     ["red", "green", "blue"]
     ```

When to Use Each?

- **Struct**: Use when you need **fixed, labeled fields** (e.g., a person's profile with `name`, `age`, etc.).
- **Map**: Use when you need **flexibility** and the number or type of keys varies (e.g., storing product counts by name).
- **Array**: Use when you have an **ordered list of similar items** (e.g., a list of product IDs or colors).

Temporal Cardinality Explosion

When you add a **time dimension** to your dataset—tracking information daily, weekly, or hourly—you dramatically multiply the number of rows you need to store. Instead of having just one record per entity (e.g., a product or listing), you now need one record per entity **per time unit**, which can quickly explode into billions of rows.

Airbnb Example

Airbnb has about **6 million listings**. If you store data at the listing level (one row per listing), you only need 6 million rows. However, if you decide to record **nightly pricing and availability** for a year, you end up with about **2 billion rows**

(6 million listings × 365 days). This is a clear illustration of the "cardinality explosion" from adding a time aspect.

- Keeping one row per listing is compact but makes daily analysis more complex.
- Having one row per listing-night makes analysis easier but greatly increases storage needs.

Modeling Options

To accommodate temporal data, you can use two main approaches:

1. **Listing-Level with an Array**

   Store one row per listing, and include an array of nightly data (e.g., `nightly_prices` = `[100, 105, ...]`). This keeps row counts lower (6 million) but can complicate queries.

   ```
   {
     "listing_id": 1,
     "nightly_prices": [100, 105, 110, ...],
     "availability": [1, 1, 0, ...]  // 1 = available, 0 =
   not available
   }
   ```

2. **Listing-Night Level**

   Store one row per listing per night, which could lead to billions of rows. This is easy to query but demands large amounts of storage.

The Role of Parquet and Sorting

Parquet is a columnar file format that compresses data efficiently—especially when it's sorted by key fields like `listing_id`. If you **sort your data carefully**, Parquet can compress repeated values (e.g., the same listing ID) even if you choose the listing-night level. In many cases, the actual file size won't be drastically different from the array-based approach due to Parquet's compression benefits.

Downsides of Denormalized Temporal Data

When you "explode" temporal data (one row per day per listing), joins and aggregations can become very inefficient. For instance, if you need to join 2 billion listing-night rows with another large table, tools like Spark have to shuffle huge amounts of data, which breaks compression and slows down processing.

- Shuffling moves data across nodes to group it by join keys. This scatter disrupts sorted patterns and eliminates compression gains.

How to Fix It:

1. **Keep Temporal Data Compact**:
    - Store data at the listing level and use an **array of daily values** for pricing and availability.

    ```
    {
      "listing_id": 1,
      "nightly_prices": [100, 105, 110, ...],
      "availability": [1, 1, 0, ...]  // 1 = available, 0 = not available
    }
    ```

2. **Only Denormalize When Needed**:
    - Explode the array into rows only if absolutely necessary for analysis.

Run-Length Encoding Compression Explained

What is Run-Length Encoding (RLE)?

Run-Length Encoding is a compression technique that reduces the size of repetitive data by storing only the value and the number of times it repeats, instead of storing every occurrence.

- **Why is it important?**It's one of the key reasons why **Parquet** (a popular data storage format) is so efficient in big data environments.

How it Works (Example):

In the table shown:

- Columns like `player_name`, `height`, and `college` have **repeated values** for each season.
- Instead of storing `"A.C. Green", "A.C. Green", "A.C. Green", ...` for every season, **RLE** would store:

  ```
  "A.C. Green", repeated 5 times
  "6-9", repeated 5 times
  "Oregon State", repeated 5 times
  ```

Day 1 Lab

https://github.com/DataExpert-io/cumulative-table-design