# Chapter #9:

# Logical Design

→ Logical design is a set of steps to create the structure of a data mart.

→ it starts with a conceptual design outlining the application domain.

→ The logical design the defines the data structures that will represent the data mart according to the chosen logical model.

→ Also optimizes performance by fine-tuning these structures.

## Steps in transforming Conceptual schema into logical schema:

1. Translating fact schemata into logical schemata:

This involves converting the fact schemata into star, snowflake, and constellation schemata.

Materializing views:

→ This involves creating materialized views, which are pre-computed results of queries that can be stored and used to improve query performance.

Fragmenting fact tables:

→ This involves breaking down fact tables into smaller, more manageable pieces, either vertically or horizontally.

## Form Fact Schemata to form Star Schemata:

→ Fact schemata can be directly translated into star schemata.

→ However this translation is not fully automatic and requires careful design considerations.

→ A relational implementation of a fact schema generally involves a star schema with a fact table containing all measures and descriptive attributes linked to the fact.

## Descriptive Attributes:

↳ Descriptive attributes are pieces of information that cannot be used for aggregation but are still useful.

↳ if descriptive attribute is directly linked to a fact, it should be included in fact table.

↳ if linked with dimensional attribute, it must be included in dimensional table.

## Cross- Dimensional attributes:

↳ Cross dimensional attributes define a many-to-many relationship between two or more dimensional attributes belonging to different hierarchies.

↳ Create a new table to represent this relationship.

↳ Include the cross- dimensional attribute and dimensional attributes as the PK.

↳ Using surrogate key can be a valid solution.

## Shared Hierarchies:

↳ Sometimes a fact schema has repeating hierarchies or parts of hierarchies.

↳ In this case, avoid creating multiple dimension tables with similar data.

↳ There are two ways for dealing with shared hierarchies:

### Total sharing:

↳ Two hierarchies have exactly the same attributes with different meanings.

↳ Need to create only one dimension table for both hierarchies.

↳ The fact table will have two foreign keys referencing the same dimension table.

### Partial Sharing:

↳ Two hierarchies share some of their attributes.

↳ Can be done by introducing redundancy by including shared attributes in both dimension tables or can be by creating a separate shared table for both common attributes.

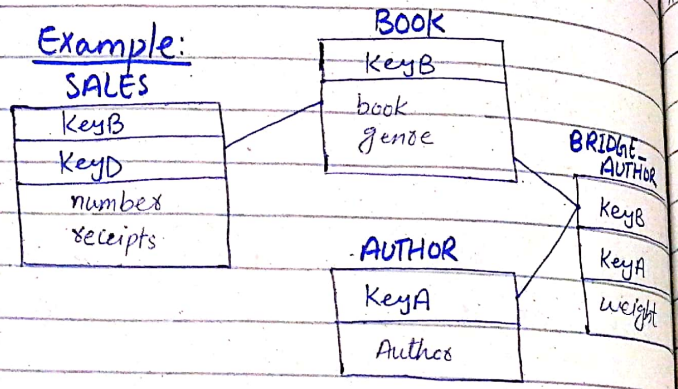
## Multiple Axes:

→ Sometimes hierarchies might show many-to-many relationships between entities.

→ One way to model this r/s by using a bridge table.

→ A "bridge table" contains a combination of the attributes linked by the multiple Axe.

**Example:**




→ Using the bridge table, Two types of queries can be performed:
- Weighted Query
- Impact Query

→ Sometimes, push down the multiple Axe to create a new dimension. This is done when the end attribute of the multiple axe has child attributes that form a hierarchy.

| Push-down | vs | Bridge-Table |
|---|---|---|
| ○ Introduces redundancy in the fact tables.<br>○ Fact table lines are repeated as many times. | | ○ More concise fact tables.<br>○ No redundancy in the fact tables.<br>○ Requires a bridge table. |

## Handling Optional Axes:

⇒ Sometimes, hierarchy might have incomplete solutions.

**NULL values:**

↳ Use NULL values to represent missing values in optional hierarchies.

**Fake values:**

↳ Consider using fake values to represent missing data more explicitly.

## Handling incomplete Hierarchies:

⇒ Can be handle by:
- Upward balancing
- Downward Balancing
- Downupward Balancing

Balancing by enclusion:

- Replacing missing values with a generic signpost like "other".

Downward Balancing:

- Replace missing values with the value of the preceding attribute in the hierarchy.

Upward Balancing:

- Replace missing values with the value of the following attribute in the hierarchy.

## Modeling Recursive Hierarchies:

⇒ Recursive hierarchies are hierarchies where the number of levels can vary.

⇒ This makes them challenging to model in a traditional data warehouse.

⇒ There are two main ways to model these:

- Self-referencing FK
- Navigation Table.

## Degenerate Dimensions:

↳ A degenerate dimension is a hierarchy that consists of only one attribute.

↳ These can be handles:

- By creating a separate dimension table.
- exclude the degenerate dimensions directly in the fact tables.

## Junk Dimensions:

↳ A junk dimension is a dimension table that contains a set of degenerate dimensions.

↳ This can help reduce the number of FK in the fact table.

↳ Improve query performance

↳ Better data Granularity

## Additivity issues:

↳ When creating a DWH, it is important to consider how data will be aggregated.

↳ Different operators have different properties.

* Aggregation operators:
  - SUM, AVG, MIN, MAX
* Distributive operators:
  - SUM & COUNT
* Algebraic operators:
  - MIN / MAX & AVG
* Holistic operators:
  - PERCENTILE & RANK, MODE & MEDIAN etc.

## Snowflake Schemas

⇒ A snowflake schema involves normalizing dimension tables to reduce redundancy.

⇒ When snowflaking might be beneficial
  - Share Hierarchies
  - High Cardinality Ratios
  - Dynamic Hierarchies
  - Readability

# View Materialization

→ Data warehouse store large amount of data.

→ Smaller, more specific views of data are created

→ View materialization: is the process of selecting which of these views to make permanent.

→ Data in these views is pre-calculated and stored, so, it can access it quickly without having to recalculate it.

**Why is it important?**
- Faster query performance for specific questions
- Reduce load on data warehouse

**How to choose views to materialize?**

→ There are two main factors to consider:

Workload evaluation:

↳ Analyze how frequently certain views are used.

View maintenance cost:

↳ Materialized views need to be updated regularly to reflect changes in the underlying data.

# Constraints on View Materialization

1. System Constraints:
   - Disk space:
     ↳ The amount of disk space available limits how many views can materialize.
     ↳ Need to estimate how much space each view will take up.
   - Update Time:
     ↳ Updating materialized views takes time and impact other operations on the data warehouse.
     ↳ Need to consider how often views can be updated and how long it will take.
2. User Constraints:
   - Query Response Time:
     ↳ Users expect quick answers to their queries.
     ↳ This limits the complexity of the materialized views.
   - Data Freshness:
     ↳ Users wants the data in the views up-to-date. Views should be updated fre

# Challenges and Solutions related to view Materialization:

Problem:
- View materialization is a complex problem because there are many possible views that could be created, and each of has its own costs and benefits.
- Need to balance the need for fast query performance with the limitation of system resources.

Solution:
1. Transforming Constraints:
   Simplify the problem by converting update time constraints into space constraints.
2. Multidimensional Lattice:
   - This is a data structure that represents all the possible ways to group and aggregate data in a fact schema.
   - It helps visualize the relationships b/w different views.
3. Candidate views:
   These are the views that are most useful for answering specific queries.

- They are selected based on their ability to answer multiple queries and their cost.

4. Materialization Algorithm:
   ↳ This algorithm helps to identify the best views to materialize.
   ↳ it considers the workload and the available resources.

## View VS Materialized V

| View | Materialized |
|---|---|
| o Does not require space. | o Require space to store permanent. |
| o Provide security by restricting access to data. | o Provide snapshot of view. |
| o set of stored query. | ⇒ INSERT, DELETE, UPDATE are not allowed in materialized view |
| o looks like a table but not table. | |

# View Fragmentation

↳ View fragmentation is a technique used in data warehouse to improve system performance.
↳ It involves breaking down a large table into smaller, more manageable fragments.
↳ This can be done in two ways:
- Horizontal
- Vertical

**Why is useful?**
- Improved query performance
- Optimized storage
- Parallel processing

## Vertical View Fragmentation
↳ multicubing

⇒ This involves dividing a table into multiple parts, each containing a subset of columns but all the rows.

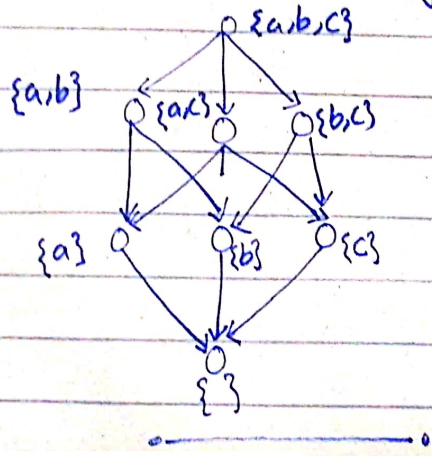
Characteristics:
- Consistency
- Completeness
- Non-redundancy

Reasons for vertical fragmentations
- Optimized workload cost
- Saved Space
- Reduced key replication
- Generalization

## Horizontal Fragmentation

⇒ This involves dividing a table into a subset of rows but all the columns.
⇒ This helps to optimize storage and query performance.
⇒ It deals with individual cubes.
⇒ It does not require additional cost or disk space.

### View materialization Diagram




# Chapter #10

# Data-Staging Design

→ Data staging design is about planning how to load from operational systems into data marts.
→ if the reconciled layer is available, the population process is divided into two phase:
- from the operational sources to the reconciled database.
- From the reconciled database to the data mart.

## Populating Reconciled Databases:

→ This involve fill the reconciled database with data from different sources.

### Reconciled database:

↳ it is like staging area where data from different sources is cleansed, transformed and prepared before it's moved to the final destination.

## How does it work?

**Staging area:**

⇒ A temporary area where "raw" data is stored before being cleansed and transformed.

⇒ it can take up a lot of space. So, it's important to plan for it size.

⇒ These tasks should be associated with the data cleansing procedures, which help eliminate the inconsistencies due to errors and failures.

**Extraction:**

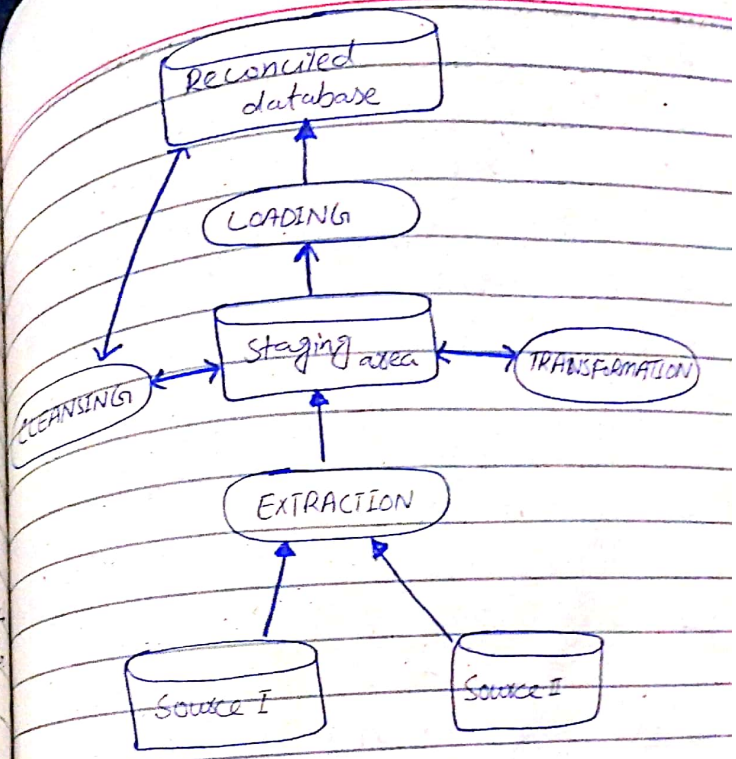
- Tasks aimed at collecting data from sources.

**Transformation:**

- Tasks aimed at adjusting the format of data from source schemata to reconciled schemata.

**Loading:**

- Tasks aimed at entering the transformed data into the reconciled database and updating the already existing data.




## Data extraction:

→Data is collected from different sources.

→These are three different types of data:

**Transient data:**

- only the current snapshot is kept.

**Semi- Periodic data:**

- A limited number of past versions are kept.

Temporal data:
- All changes are tracked for a defined period.

## Data extraction methods

1. Static extraction:
   - Scans all data in the source database.
   - Its good for initial setup but can be slow for large datasets.
2. Incremental extraction:
   - only extracts changed data. It is more efficient for frequent updates.

## Immediate Extraction Techniques:

⇒ These techniques record changes to data as soon as they happen in the operational database.

1. Application-Assisted Extraction:
   - Creates a set of functions in the operational applications to store data changes to the staging area.
   - it can be used for legacy systems that don't support triggers, logs, & timestamps.

Pros:
- can capture changes in real-time and highly flexible.

2. Triggers-based extraction:
   - This technique uses triggers in the database to automatically capture changes to data.
   - A trigger is like a rule that is activated when a specific event occurs, such as when a record is inserted, updated or deleted.
   - when a trigger is activated, it stores the modified data in a special file or table, which can then be retrieved later for processing.

3. Log-based extraction:
   - This technique uses the database log files to capture changes to data.
   - Log files are created by the database to record all changes made to data.
   - By analyzing the log files, identify changes that have been made & extract the relevant data.

Timestamp-based extraction:

⇒ This technique relies on timestamps to identify which data has been modified since the last extraction.

How it works?

Timestamping:

- Each record in the database has a timestamp associated with it, indicating when it was last updated.

Extraction:

- The extraction process compares the timestamps of records to the last extraction time.

Data selection:

- Only records with timestamps newer than the last extraction time are extracted.

5. File Comparison:

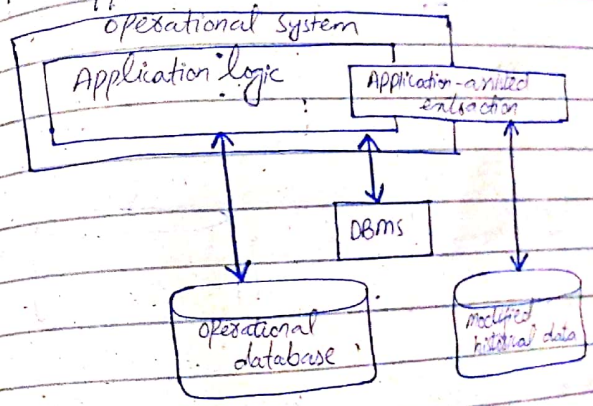
- Compare file versions to identify changes.
- This is delayed extraction technique.
- This requires that, after each extraction, the extracted data be kept in the staging area until the next extraction.

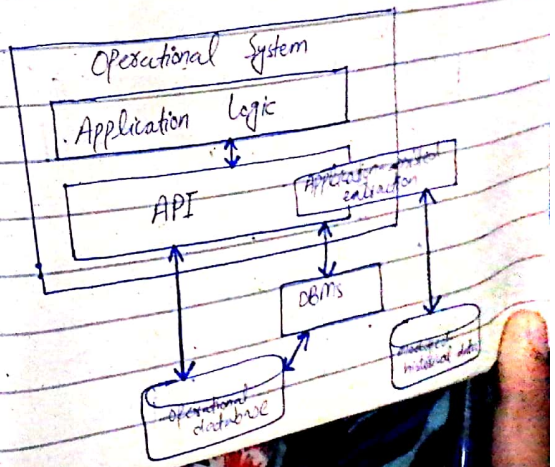
enable its comparison to the new data extracted.

- The technique is normally used when operational data storage is file based or when DBMS does not support triggers or logs.

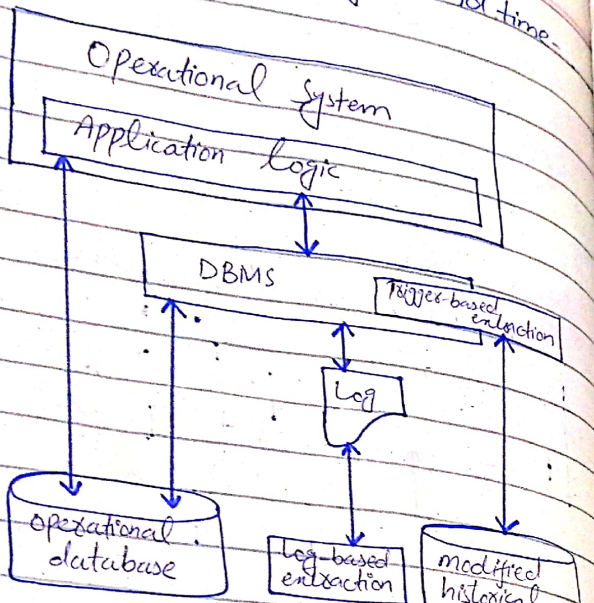
Diagrams:

Application-assisted extraction

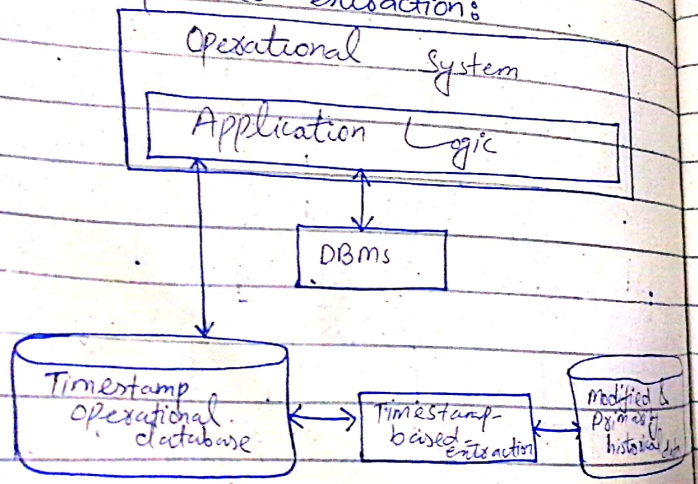


Extraction based on triggers and time-stamp:




Timestamp based extraction:




## Transforming Data:

Data is transformed to fit the target data warehouse or data mart schema.

Common types of transformation are as follows:

1. Conversion:

This involves changing the data type, format or measurement unit of an attribute.

ex:

- Converting dates from one format to another, converting currency from one format to another.
- Write name of a company in different ways.
  - 'Coca-Cola', 'Coca-Cola Co' & so on.

2. Enrichment:

- This involves adding new information to the data by combining existing attributes or by calculating new ones.

example:

- Calculating customers total purchase amount.

Separation/ Concatination:

- This transformation involves separating data that is stored together in the source system but needs to kept separate in the reconciled database.
- This often done through normalization & denormalization process.

# Loading data

⇒ This is the final step in the data staging design process.

⇒ Data that has been extracted and transformed is loaded from the staging area into the reconciled database.

⇒ This type of loading process depends on the extraction technique used and the nature of the data being loaded (temporal or non-temporal).

# Cleansing data & Various Techniques

Data cleansing, also known as data cleaning or data scrubbing, is the process of detecting and correcting errors and inconsistencies in a dataset. It ensures that data is accurate, consistent and ready for analysis.

Why important?

- Data quality
- Data integrity
- Data consistency

Common data cleansing Tasks:

- Identifying and correcting errors
- Standardizing data
- Removing duplicates
- Validating data
- Enriching data

Note:

- while data cleansing can be seen as a part of data transformation, it's often treated separately due to its significant impact on data quality & the complexity of the tasks involved.

## Common causes of data inconsistency

⇒ Some common causes are:

1. Typing Mistakes:
   - Simple errors during data entry like typos.
   - Implementing data validation checks can help prevent these errors.
2. Inconsistency blw attribute value & description:
   - Discrepancies between related data elements.
   - occurs when data is entered into the wrong attribute due to changes in data requirements.
   - for example, entering a mobile number in a landline phone number field.
3. Inconsistency blw correlated attributes:
   - When values in related attributes are inconsistent.
   - for instance, a person's city being "miami" and their state being "california".
   - Data validation rules & cross-checking can help identify and correct such inconsistencies.

missing information:

- Users may leave fields blank due to lack of time or understanding of their importance.

## Mitigating data inconsistency

- Enforce strict data entry rules.
- Data validation checks
- Data cleansing
- Data quality management.

## Data Cleansing Techniques

⇒ There are various techniques for data cleansing:

### 1. Dictionary-based Techniques:

⇒ Compares data values to a predefined dictionary or lookup table to check for validity.

Use cases:

- Identifying & correcting typos
- Standardizing data formats
- Ensuring consistency blw related attributes.

Example: Checking if a city name is valid by comparing it to a list of valid cities.

## 2. Approximate Merging:

⇒ Matches records from different sources that might have slight variations in data.

Usecases:

- Merging data from different systems with overlapping data.
- Identify duplicate records with minor variations.

Example:

- Matching customers records from two databases based on similar names & addresses, even if the exact information might not match perfectly.

## 3. Ad-hoc Techniques:

⇒ Ad-hoc techniques are used when specific business rules or domain-specific checks are required that cannot be handled by standard tools.

These techniques involve writing custom queries or scripts to ensure data integrity and consistency.

Ex:

Financial data:

Verifying that the equation Profit = receipts - expenses holds true for all records.

Capital = assets + credits - debits.

- functional dependency check
- outliers detection

## Data cleansing Tools:

- Open source tools:
  - Python (Pandas, Numpy), R
- Commercial tools:
  - Tableauprep, Alteryx, Trifacta

(Continue this chapter on Register 3)

# Populating Dimension Tables

⇒ Dimension tables are the first tables to be populated in a data warehouse.

⇒ This is because they form the basis of the star schema structure, where fact tables reference dimension tables to provide context.

## key steps to populating:

1. Identifying data to load:
   - Determine which attributes from the reconciled database are relevant to the dimension table.
2. Replacing keys:
   - Assign surrogate key to new tuples in the dimension table.
   - Keep a mapping between the surrogate key and the primary key of the reconciled database tuple. This helps track changes & updates.

## Handling different Hierarchy Types:

Static hierarchies:
- Only new tuples can be added, existing tuples cannot be modified.

Type 1 Hierarchies:
- Overwrite existing tuples with new values, effectively losing historical data.

Type 2 Hierarchies:
- Creating new tuples for each change, preserving historical data.

Type 3 Hierarchies:
- Store a limited number of versions of each attribute.

Fully-Logged Hierarchies:
- Log all changes to dimension table tuples, including historical data.

## Incremental vs Static Population:

Incremental:
- Updates only the changed data which is more efficient for large datasets.

Static:
- Repopulates the entire dimension table, which can be simpler but less efficient.

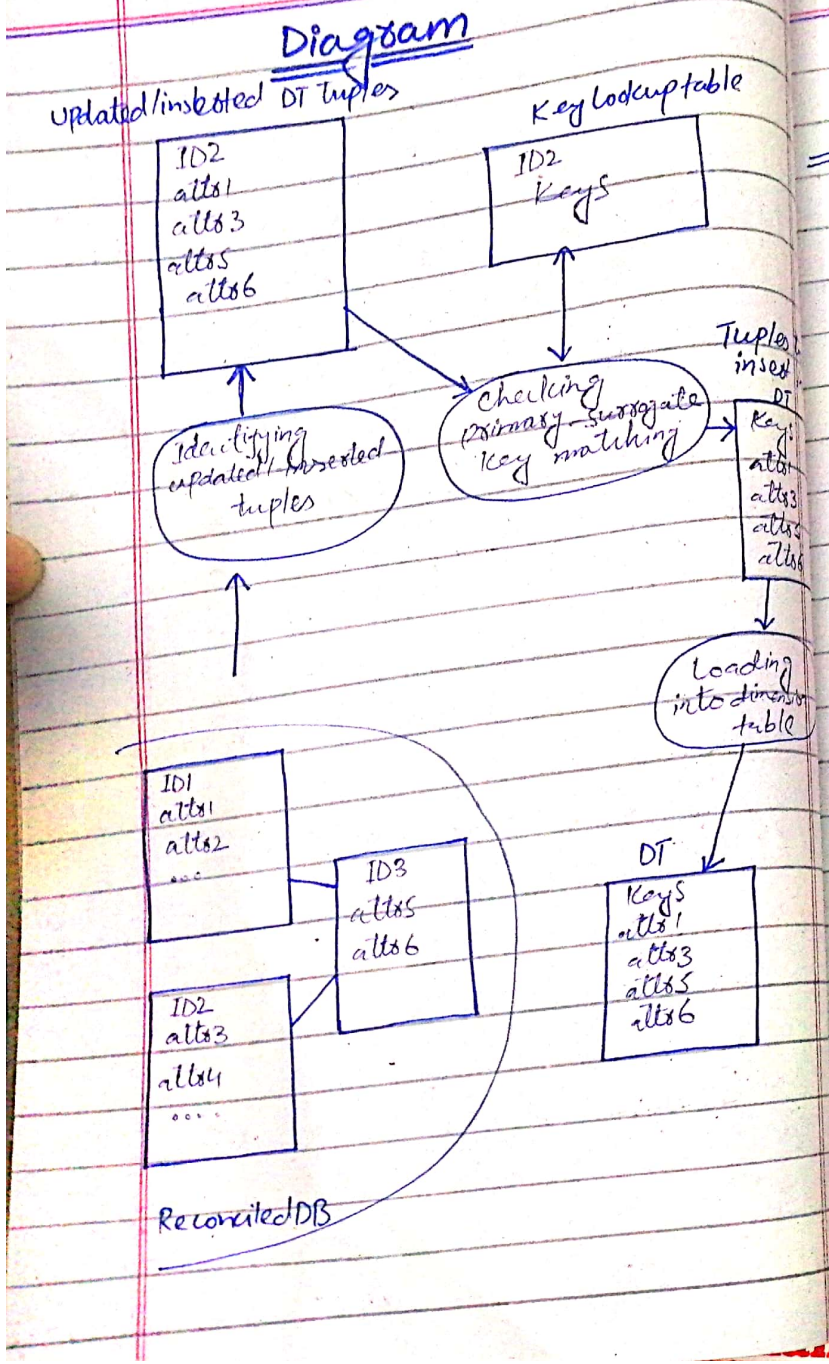
## Challenges in populating Dimension Tables:

- Complex data structures
- Data quality issues
- Performance optimization

## Solutions to the challenge:

- Temporary duplication of dimension table in the staging area.
- Comparing to source data.

## Diagram




# Populating Fact tables

⇒ Populating fact tables is similar to populating dimension tables. But there are some differences.

### Order of operations:

- Should always populate dimension tables before populating fact tables. This ensures that the foreign keys in the fact tables reference valid entries in the dimension tables.

### Key lookup tables:

- Use key lookup tables to map primary key from the reconciled database to surrogate keys in the dimension tables. This helps maintain data consistency and integrity.

### Fact table measures:

- Determine the measures that will be stored in the fact

table. These measures should be derived from attributes in the reconciled schema.

Data changes:
- Fact tables can be updated to reflect changes in the reconciled date.

Change tracking:
- Use flags or timestamps to track changes to fact table tuples.

Efficiency:
- Use incremental updates to avoid unnecessary processing & improve performance.

Data Security:
- Protect sensitive data by implementing appropriate security measures.

Data quality:
- Ensure quality of data in fact table.

Semantic considerations:
- Considers the semantics of different types of changes.
- This will help to determine how to handle updates and maintain data consistency.

Note: Diagram is same like populating dimension table.

Except its has more than one lookup tables.

## Populating Materialized Views:

Materialized views are pre-computed results of SQL queries that are stored in the database.

They can significantly improve query performance, especially for complex &

frequently executed queries.

## How they populated?

**Initial Population:**
- The materialized view is created by executing the underlying SQL query.
- The results of the query are stored in the materialized view.

**Incremental updates:**
- To keep the materialized view up-to-date, incremental updates are applied.

**Challenges:**
- Complexity
- Performance due to frequent updates
- Data Consistency

**Strategies for efficient population**
- Optimized query execution
- Incremental updates
- Parallel Processing
- Caching

SALES

| KeyS | KeyD | KeyP | quantity | receipts |
|---|---|---|---|---|
| 1 | 1 | 1 | 170 | 85 |
| 2 | 2 | 2 | 320 | 160 |
| 3 | 1 | 3 | 412 | 200 |

→ new tuple

STORE

| KeyS | store | city | state |
|---|---|---|---|
| 1 | COOP1 | Columbus | ohio |
| 2 | COOP2 | Austin | Texas |
| 3 | COOP3 | Dayton | ohio |

PRODUCT

| KeyP | Product | type | category | brand |
|---|---|---|---|---|
| 1 | Skim Milk | Dairy Product | Food | Skim |
| 2 | Choko | Sweets | Food | nutella |
| 3 | Yum Yogurt | Dairy Product | Food | Yum |

DATE

| KeyD | date | month | year |
|---|---|---|---|
| 1 | 2/9/2009 | 9/2009 | 2009 |
| 2 | 3/10/2009 | 10/2009 | 2009 |

SALES'

| KeyS | KeyD | KeyT | quantity | receipts |
|---|---|---|---|---|
| 1 | 1 | 1 | 582 | 285 ... |
| 2 | 2 | 2 | 320 | 160 ... |

→ modified tuple

# Chapter # 11

# Indexes for the Data WAREHOUSE.

⇒ Indexes are way to organize data.

⇒ They allow to access specific data quickly without going though all the data.

⇒ They enhance the performance.

⇒ Index key stand for the set of attributes upon which the index was built.

⇒ it help to access relations tuples quickly.

## B+ Tree indexes

⇒ This is a data structure, fancy way of accessing data organizing

- B-Trees are like trees with branches and leaves.
- Each branch points to a specific piece of data, and the leaves contain the actual data.

**How B-tree works?**

⇒ when finding something in a B-Tree, it starts at the top (root) and follows the branches until it reaches the right leaf. This is faster than searching though every piece of data one-by-one.

**B-Tree Properties:**

**Order:**

- The order of B-Tree determines how many children

**Leaves:**

- All the data is stored in the leaves of the B-Tree.

**Pointers:**

- Pointers connect the nodes of the B-tree allowing for efficient navigation

## Searching in B-Trees

**Sequential search:**

- Start at one leaf and following the pointers to the next leaves.

**Interval search:**

- Specific value can search by following the branches of B+tree.

**Note:**

⇒ B+-trees evolved from B-trees (Bayer and McCreight, 1972)

**Types:**

⇒ B+-trees can be either:

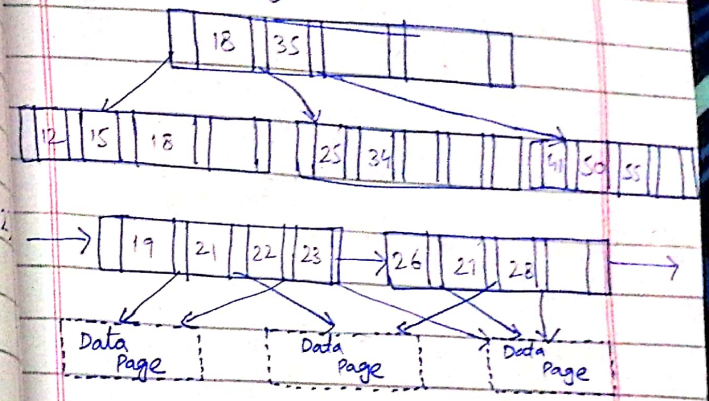
- Primary
- Secondary

**Primary index:**

- This is built on the primary key of a table, which is a unique identifier for each row. Each leaf in a primary index points to exactly one row.

**Secondary index:**

- This is built on any other attribute of a table. Each leaf in a secondary index can point to multiple rows that have the same value for that attribute.




## Bitmap indexes

- A bitmap index like a matrix with rows and columns.
- Each row represents a tuple, and each column represents a distinct value of an attribute

## How bitmap indexes work?

- An in the matrix indicate that the tuple in that row has the attribute value associated with that column.

## Advantages of Bitmap indexes:

- Efficient for filtering:
  ⇒ Bitmap indexes are very efficient for filtering data based on specific attribute values. (Use Boolean operations)
- Space:- efficient
  ⇒ They can be more space-efficient than other types of indexes, especially for columns with low cardinality.

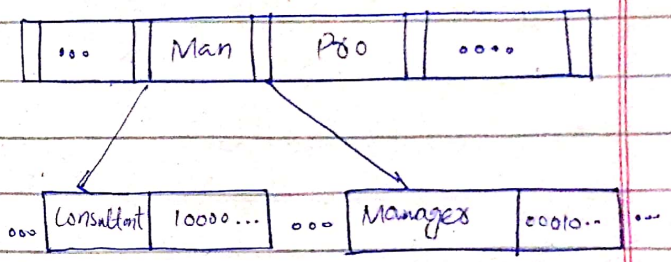
## Optimization:

Organizing Bitmaps as sets of vectors:

By organizing bitmaps as sets of vectors, you can reduce the amount of data that need to be loaded from disk when searching for specific values.

## Structure:

Structure of Bitmap indexes are identical to B+-trees. Except that bitmap index tree leaves contain bit vectors in the place of RID lists.




## B-Tree vs Bitmap

| | |
|---|---|
| Structure: | |
| • Tree like structure with nodes & pointers | • Matrix of bits representing attribute values. |
| Data storage: | |
| • Stores data in nodes. | • Stores bitmaps |

| | representing data presence/absence |
|---|---|
| **Query Types:** | |
| ○ Efficient for exact match and range queries | ○ Efficient for filters based on multiple conditions. |
| **Scalability:** | |
| ○ Scales well with increasing data volume | ○ Scales well for columns with low cardinality. |
| **Space efficiency:** | |
| ○ Generally more space-efficient | ○ Can be or can not be. |
| **Complexity:** | |
| ○ Complex to implement | ○ Simples to implement. |
| **Best use cases:** | |
| ○ General purposes, especially for transactional databases | DWH & analytical applications, especially for filtering & aggregation. |

# Advanced Bitmap indexes:

## Bit-sliced indexes:-

- a type of advanced bitmap index used to organize data in database.
- A bit-sliced index is a matrix with rows and columns.
- Each column of index (slice) is stored separately.

**Pros:**

- Efficient for numeric attributes, because they efficiently calculate aggregate values.
- Scalability
- Complex queries

**Note:**

- Standard bitmap indexes linearly get larger as the number of distinct key values increases.
- Bit-sliced indexes show a logarithmic growth.

## Bitmap-encoded indexes

- Same like matrix will rows and columns.
- But bitmap-encoded indexes can be used for both numeric and non-numeric attributes, making them versatile.

# Projection Indexes

## Structure:

- A projection index as a list of values for a specific attribute.
- This list is ordered in the same way as the rows in the table.

## Advantages:

- Efficient for retrieving attribute values.
- Reduced disk Access.

## Limitations:

- Efficient for fixed length int values.
- Require additional storage space.

# Join indexes

## Structure:

⇒ A list of pairs of tuple IDs.

⇒ Each pair represents two tuples from different tables that are related to each other

## Advantages:

- Efficient for joins:-
- Reduced disk access

## Limitations:

- Space overhead
- Maintenance overhead.

# Star-Join indexes

a list of tuples, each containing a set of Row IDs

from different tables.
- Efficient scan without scanning the entire tables.

**Advantages:**
- Efficient for joining multiple tables
- Reduced disk access.

## Spatial indexes

→ A type of data structure used to organize and query spatial data in databases.

→ Spatial indexes are designed to optimize the storage and retrieval of spatial data.

→ They organize spatial data in a way that allows for efficient searching and queries.

**Common Techniques:**
- R-Tree
- Z-order curve
- Quadtree

## Joining Algorithms:

Joining algorithms are techniques used to combine rows from two or more tables based on a related column.

Some common join algorithms:

**Nested Loop Join:**
- Best for small datasets.
- Simple but inefficient -
- iterates over each row of the outer table and compares it to every row of the inner table.

**Hash join:**
- well suited for equi-joins.
- Efficient for large datasets.
- Creates a hash table for one relation & probes the other relation against it.

## Sort - Merge join:

- Sorts both relations on the join attribute.

⇒ merges the sorted relations to produce the result.

⇒ Efficient for large sorted relations.

## Hybrid Hash join:

⇒ Combine hash join & sort-merge join.

⇒ Divides the larger relations into partitions and processes them in memory or on disk.

## Choosing the right algorithm:

⇒ The choice of joining algorithm depends on various factors:

- Table Sizes
- Data distribution
- Memory availability
- Index availability

---

# Chapter #12:-
# Physical Design

⇒ Physical design is the final phase it is where actually implements the decisions like taking the blueprint and building the actual structure.

Why its important?

⇒ This is not fixed, as the data mart is used, need to make changes to optimize its performance and meet new user needs.

The best approache

⇒ Logical design (Planning the structure) and physical design (building it) work best together.

Challenges:

⇒ Some tasks like creating indexes or materializing views, can improve performance but are complex.

DBMS dependencies:

⇒ The specific DBMS use has a big impact on the physical design choices.

⇒ Different dbms has different features and capabilities.

## Optimizers

What they are?

⇒ are crucial part of physical database design.

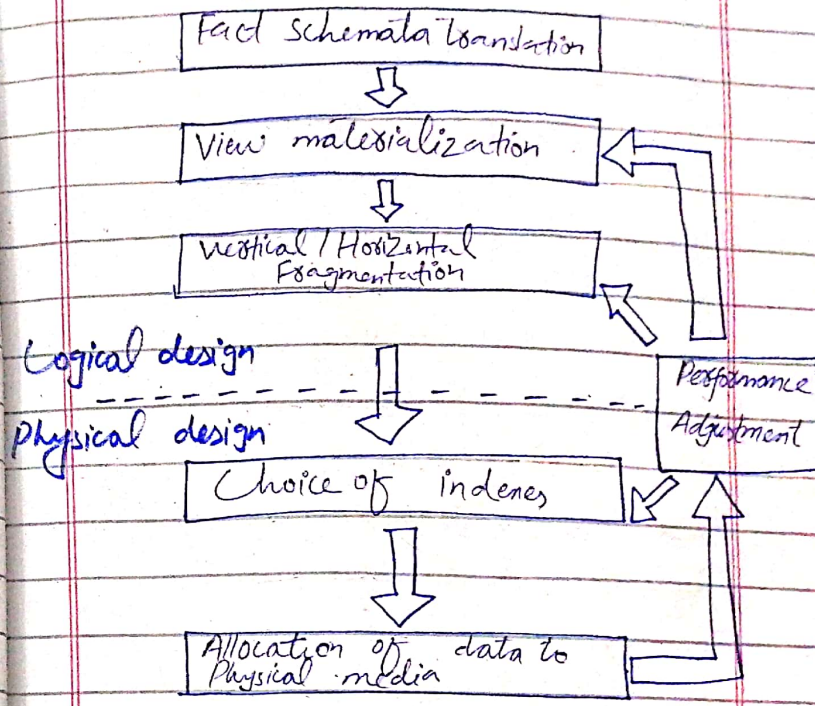
They are like smart planners that figure out the most efficient way to execute a query.

How they work?

⇒ when a query run, the optimizer analyzes it and creates a query execution plan. This plan is step-by-step guide for database to follow, specifying which tables to access,

⇒ which indexes to use, and the order of operations.

## RIs between Physical & Logical design:



## importance of in-depth knowledge of DBMS for effective DWH:

Internal expertise:

⇒ Companies should aim to build internal expertise in DBMSs, as it's a valuable asset for data warehousing projects.

External consultants:

⇒ If using external consultants, it is essential to ensure they have the specific knowledge & skills needed for DWH.

# Types of Optimizers

1. Rule-Based optimizers:

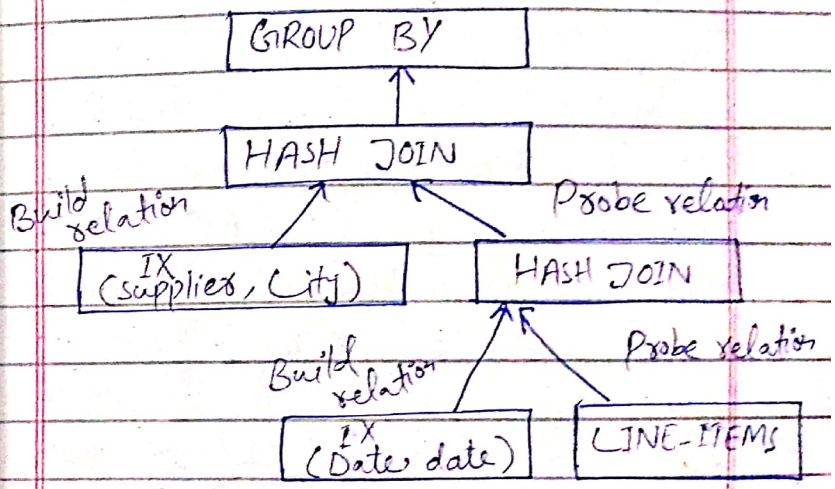
→ These optimizers rely on a set of rules to determine how to execute a query.

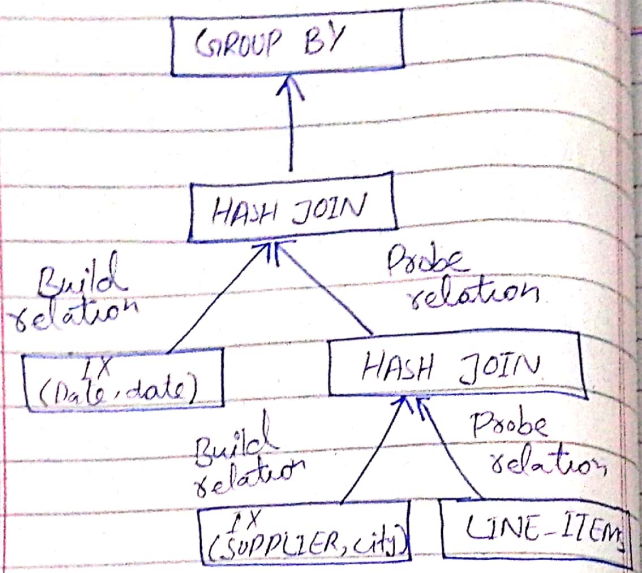
→ These rules consider the DB structure, query structure & available indexes.

Limitation:

⇒ They don't take into account specific information about the data, like how many rows are in a table or how data is distributed. This can lead to suboptimal query execution plans.

## Two different ways to execute a query:



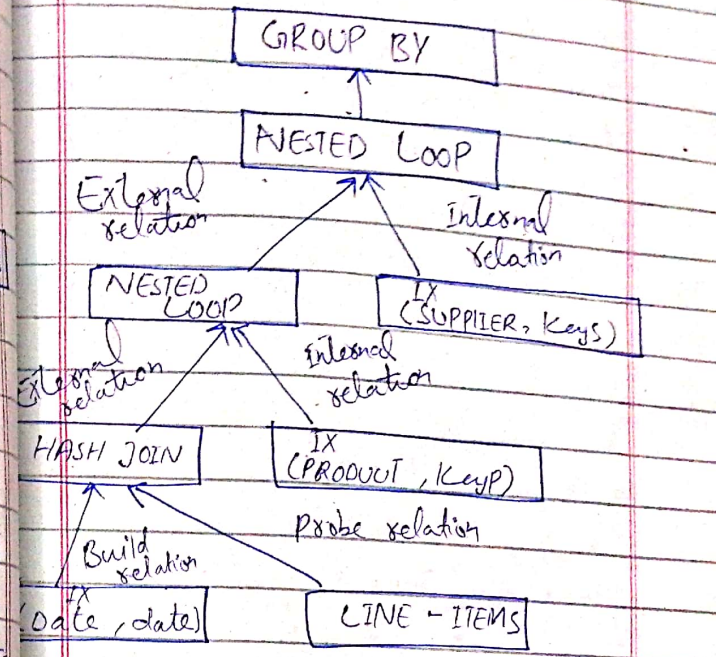



## Query:

```
⇒ SELECT SUM (FT.quantity,
  DT1.Product, DT2.state, DT3.date)
  FROM LINE_ITEM AS FT,
    PRODUCT AS DT1, SUPPLIER as DT2
  , DATE As DT3
  WHERE FT.KeyP = DT1.KeyP AND
        FT.Keys = DT2.Keys AND
        FT.KeyD = DT3.KeyD AND
        DT3.date = '1/1/2008'
  GROUP BY DT1.Product, DT2.state, DT3.date
```

## Example:

## Execution Plan



## 2. Cost-Based Optimizers:

**Def:**

⇒ Cost-based optimizers uses a statistical information about the data to choose the most efficient way to execute a query.

**How they works?**

⇒ They analyze the query and the available indexes, and then estimate the cost of different ways to execute the query. The cost is usually measured in terms of the number of disk I/O operations required.

**Statistical information:**

⇒ Statistical information could be includes the number of rows in each table, the number of distinct values for each attribute & the distribution of values for each attribute.

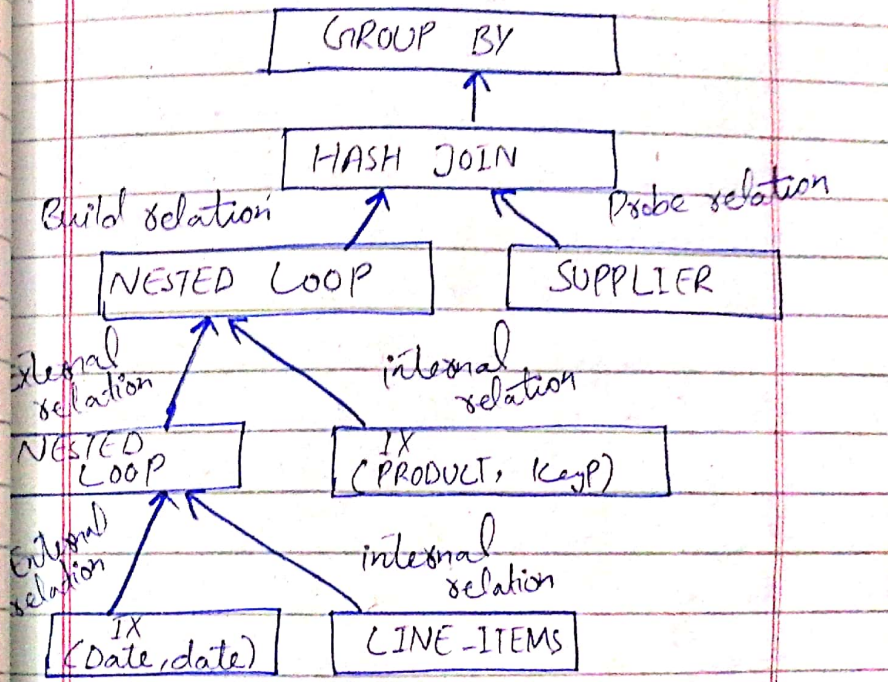
**Pros:**

- More accurate
- Improved Performance.

**Cons:**

- Dependency on statistical information
- Complexity

**Diagram with example:**



## 3. Histograms:

⇒ are a ways to summarize the distribution of values in a table. They divide the range of values into intervals and count the number of values that fall into each interval.

Types:

- Equi-width
- Equi-Height

Accuracy:

⇒ Equi-height are more accurate

⇒ But more difficult to calculate & maintain

Histogram-based estimation:

⇒ Histograms can be used to estimate the selectivity of query by calculate the sum of the frequencies of the intervals that fall within the range of the condition

## Index Selection

What is it?

⇒ Index selection is the process of choosing which indexes to create on data most tables to improve query performance.

⇒ Indexes are like shortcuts that help the DB find specific data more quickly.

Challenges:

⇒ Choosing the right indexes can be difficult because there are many different types of indexes, and each index has its own advantages and disadvantages.

⇒ Creating too many indexes can slow down data updates and increase storage costs.

Two-Phase approach to index selection:

⇒ Identify useful indexes.
⇒ Select the best subset

Factors to consider when selecting index:
- Data Distribution
- Query Patterns
- Space constraints

## ★ Indexing Dimension Tables:

Purpose:
- Dimension tables are used to filter the data in fact tables.
- Indexing dimension tables can improve the performance of queries that involve filtering on dimension attributes.

Types of indexes:
⇒ Two types of indexes that can be used for DTs.
1. RID-list indexes
2. Bitmap indexes.

Choosing the right type of index:
⇒ Depends on the following factors:
- Cardinality
- Query Selectivity
- DBMS features

## ★ Indexing Fact Tables:

Purpose:
⇒ Fact tables store detailed data and they are often very large.
⇒ Indexing fact tables can improve performance of query.

Primary indexes:
⇒ Fact tables typically have a primary key, which is a combination of foreign keys from dimensio tables.
⇒ However, primary indexes are not useful for primary indexes.

Join indexes:

⇒ To improve join performance it is often necessary to create additional indexes on join tables.

⇒ These indexes are called join indexes, and they are typically created on pairs of columns that are frequently joined.

## Additional Physical Design Elements:

⇒ The Physical design of data mart does not end with defining its indexes.

⇒ Also need to define the set of parameters that determine the Physical schemata of data mart.

⇒ The disk space used by a database is normally divided into:

- Tablespaces
- Data files
- Data blocks

### Tablespaces:

- A tablespace is a logical container for database objects such as tables and indexes.
- it is a way to organize and manage disk space.

### Data file:

- Also called segment
- A data file belong to only one tablespace.
- More than on data files can be associated with one tablespace.
- These are Physical files that store the actual data within a tablespace.

### Data Blocks:

- These are the smallest units of data that can be read or written to disk.

## Splitting a DB into Tablespaces

⇒ Although all the database can be theoretically stored to a single tablespace.

⇒ Suggested that separating different types of data into different tablespaces, can improve performance and queries.

### Fault Tolerance:

⇒ if a data file in one tablespace becomes corrupted,

⇒ it is less likely to affect other tablespaces.

⇒ This can improve the overall reliability of the database.

| TS | Description |
|---|---|
| SYSTEM | For data dictionary |
| DATA1...n | For data |
| INDEX1...m | For indexes |
| RBS | for standard rollback |
| TEMP | for temporary data |
| TOOLS1...n | for stored procedures & triggers. |

## ★ Allocating Data files:

### Parallel Processing

⇒ Technique that allows a database system to use multiple processors to execute query simultaneously.

### Inter-query Parallelism:

⇒ Technique that minimize conflicts between multiple queries that are running simultaneously.

⇒ To achieve this, different queries are stored on different disks. So, that queries can access the data independently.

2. Intra-query Parallelism:

⇒ Technique used to improve the speed of individual queries.

⇒ To achieve this, the data required by a query is split into smaller chunks, and each chunk is processed by a separate processor.

## Fault tolerance:

⇒ Ability of a system to continue operating even if one of its component fails.

⇒ This is important for DWH.

⇒ These are several techniques used to improve the fault tolerance of DWH:

- RAID
- Redundancy
- Backup & Recovery

1) Redundancy:

⇒ Creating multiple copies of data and storing them in different disks.

⇒ If one disk fails, the data can be retrieved from the other disk.

2) RAID:

⇒ Redundant Array of independent disks

⇒ Technology that combines multiple disks into a single logical unit.

⇒ RAID can improve performance and fault tolerance.

⇒ Types of RAID configuration:

- RAID 1 (mirroring)
- RAID 0+1 (stripe+mirror)
- RAID 5

Data Stripping:

- Refers to the technique used to allocate

a set of data to several disks to read and write the same time.

→ To stripe data, split either single bits or entire data blocks.

## Data block size:

- Number of bytes that are read or written to disk at one time.

### Small disk block size:

- Can improve performance for random access queries.

### Large disk block size:

- Can improve performance for sequential access queries.

---

## Tools for DWH

- MS SQL
- Teradata

### 1. MS SQL

→ Microsoft SQL Server.

Flexibility & versatility:

- Can handle wide range of work loads, including DWH.

Integration with MS ecosystem:

- Integrates with other microsoft products like Power BI, Azure & SSIS

Scalability:

- Can scale horizontally and vertically to handle large datasets.

Uses:

- Smaller to medium sized DWHs

Cost: Lower cost

## (2) Teradata:

High Performance & scalability:

- Designed specifically for large-scale data warehousing & analytics.

Reliability & availability:

- High availability & Data recovery features.

Use:

- Extremely large DWH

Cost:

- Higher cost