Advanced Database Design

Partitioning Strategies

Partitioning strategies in database design help improve performance by dividing tables into smaller, more manageable segments. This can reduce I/O, optimize query performance, and allow the database to work more efficiently. Two main types of partitioning are:

1. Vertical Partitioning
2. Horizontal Partitioning

1. Vertical Partitioning:

Divides a table by columns, separating frequently accessed columns from infrequently accessed ones to reduce data overhead. This approach reduces the data retrieved for common queries, enhancing performance.

Use Case: When a large portion of queries involve only a subset of a table's columns, vertical partitioning minimizes data access overhead by isolating the frequently queried columns.

Suppose we have a table `Claims` table with **420 bytes** per row and with the following columns:

Claim_ID	Claim_Date	Customer_ID	Claim_Amount	Claim_Details	Resolution_Status	Additional_Info
1001	2024-01-15	C001	$500	Minor Damage	Resolved	...
1002	2024-02-10	C002	$1200	Major Damage	Pending	...
1003	2024-02-20	C003	$250	Minor Damage	Resolved	...

Let's say that **90%** of queries only need the `Claim_ID`, `Claim_Date`, `Customer_ID`, and `Claim_Amount` columns. The first **60 bytes** cover data used in 90% of queries. We can vertically partition the `Claims` table into two parts:

1. Frequently Accessed Partition (e.g., 60 bytes)

This partition contains columns frequently used in queries.

Claim_ID	Claim_Date	Customer_ID	Claim_Amount
1001	2024-01-15	C001	$500
1002	2024-02-10	C002	$1200
1003	2024-02-20	C003	$250

2. Infrequently Accessed Partition (e.g., 360+ bytes)

This partition holds columns that are rarely queried.

Claim_ID	Claim_Details	Resolution_Status	Additional_Info
1001	Minor Damage	Resolved	...
1002	Major Damage	Pending	...
1003	Minor Damage	Resolved	...

2. Horizontal Partitioning

Horizontal partitioning splits a table into multiple smaller tables, or partitions, by row, distributing the data across several storage units. Each partition holds a subset of the rows, allowing for parallel processing and the efficient targeting of specific partitions during queries, also known as **partition elimination**.

Partition Elimination: Restricting data access to only the relevant partitions, eliminating unnecessary scanning of unrelated data.

By distributing rows across multiple hardware components (e.g., disks, CPUs), horizontal partitioning allows for simultaneous data access and processing.

There are several common methods of horizontal partitioning: **Hash Partitioning**, **Range Partitioning**, **Expression Partitioning**, and **Round-Robin Partitioning**. Each approach has its benefits, depending on the requirements for data distribution and performance.

1. Hash Partitioning

Hash Partitioning distributes rows across partitions based on a hashing algorithm applied to one or more columns, often called the **partitioning key**. This results in an even data spread across partitions, making it efficient for load balancing.

Example

Let's consider a `Customer` table with the following structure:

CustomerID	Name	Country
1	Alice	USA
2	Bob	Canada
3	Carlos	Mexico
4	Diana	USA
5	Emily	UK
6	Frank	Canada
...	...	...

Assuming a hash function that distributes data into four partitions based on the `CustomerID`, here's how the data might be distributed:

Partition	CustomerID (Hashed)	Rows
P1	1, 5, 9, ...	Alice, Emily, ...
P2	2, 6, 10, ...	Bob, Frank, ...
P3	3, 7, 11, ...	Carlos, ...
P4	4, 8, 12, ...	Diana, ...

- Here, the `CustomerID` is hashed to allocate each row to a particular partition.
- **Benefits**: Hash partitioning distributes rows across partitions evenly, making it easier to handle large data loads.
- **Limitation**: Since rows are randomly assigned, **partition elimination** is limited, as queries involving specific criteria (e.g., `Country = 'USA'`) will need to scan all partitions.

2. Range Partitioning

Range Partitioning organizes data into partitions based on specified ranges of values, making it particularly useful for sequential data like dates or numerical IDs. This allows partition elimination, as only the relevant partitions are queried.

Example

Consider an `Orders` table with a large number of rows, containing data on orders by date.

OrderID	OrderDate	Amount
1	2024-01-15	$100
2	2024-02-20	$200
3	2024-03-10	$150
4	2024-06-18	$250
5	2024-09-05	$300
...	...	...

If we partition this table by `OrderDate`, creating a new partition for each quarter, the data might be divided as follows:

Partition	Date Range	Rows
Q1	`2024-01-01` to `2024-03-31`	Orders 1, 2, 3
Q2	`2024-04-01` to `2024-06-30`	Order 4
Q3	`2024-07-01` to `2024-09-30`	Order 5
Q4	`2024-10-01` to `2024-12-31`	(None)

- Queries for orders within specific date ranges (e.g., Q1) will automatically target only the Q1 partition, resulting in **partition elimination**.
- **Benefits**: This allows efficient data retrieval for range-based queries.
- **Limitation**: If most queries focus on recent data, certain partitions can become "hot spots" with high query loads, while older data partitions receive less access.

3. Expression Partitioning

Expression Partitioning divides data into partitions based on custom expressions, which group data according to specific attributes. This technique is useful when data access patterns are based on logical categories, such as geographical regions or business departments.

Example

Consider a `Sales` table for a business with multiple departments.

SaleID	Department	SaleAmount
1	Electronics	$500
2	Furniture	$750
3	Electronics	$300
4	Clothing	$250
5	Furniture	$650
...	...	...

In this case, we might partition the table by `Department`, so each department's data is stored separately:

Partition	Department	Rows
Electronics	Electronics	Sales 1, 3, ...
Furniture	Furniture	Sales 2, 5, ...

Clothing	Clothing	Sale 4, ...

- **Benefits**: Queries targeting specific departments will only need to access the relevant partition.
- **Limitation**: As with range partitioning, if one department sees significantly more activity, that partition may become a bottleneck.

4. Round-Robin Partitioning

Round-Robin Partitioning distributes rows sequentially across partitions without regard to data values. This ensures an even spread of data across partitions but does not facilitate partition elimination, as there's no relationship between rows and partitioning criteria.

Example

Imagine a `Transactions` table where rows are divided across four partitions in a round-robin fashion.

TransactionID	CustomerName	Amount
1	Alice	$100
2	Bob	$200
3	Carlos	$150
4	Diana	$250
5	Emily	$300
6	Frank	$350

With round-robin partitioning, the rows might be distributed as follows:

Partition	Rows
P1	Transaction 1, 5, ...
P2	Transaction 2, 6, ...
P3	Transaction 3, ...
P4	Transaction 4, ...

- **Benefits**: Round-robin ensures an even distribution of data, preventing hot spots.
- **Limitation**: This method doesn't allow for partition elimination because rows are not distributed according to any meaningful data attribute.

5. Hybrid (Composite) Partitioning

Hybrid Partitioning combines two partitioning strategies, typically hash and range, to benefit from both load balancing and partition elimination. It's ideal when you need the parallel processing advantages of hash partitioning along with the query efficiency of range or expression partitioning.

Example

Consider a `SalesData` table partitioned by `StoreID` (using hash) and `Date` (using range) within each hash partition.

SaleID	StoreID	Date	Amount
1	101	2024-01-15	$100
2	102	2024-02-20	$200
3	103	2024-03-10	$150
...	...	...	...

The hybrid partitioning setup may look like this:

1. **First Level**: Hash partitioning by `StoreID`
   - Partition 1: StoreID 101, 102
   - Partition 2: StoreID 103, 104
2. **Second Level**: Within each hash partition, further split by date range.

Main Partition	Sub-Partition	Date Range	Rows
P1 (StoreID 101, 102)	Jan-March	`2024-01-01` to `2024-03-31`	Sales 1, 2
P1	April-June	`2024-04-01` to `2024-06-30`	(None)
P2 (StoreID 103, 104)	Jan-March	`2024-01-01` to `2024-03-31`	Sale 3
P2	April-June	`2024-04-01` to `2024-06-30`	(None)

With hybrid partitioning:

- **Benefits**: Data is balanced by hash partitioning, while range partitioning enables efficient partition elimination.
- **Limitation**: More complex to set up and maintain due to the two-layer partitioning strategy.

Materialized Views

Materialized views are database objects that store the results of a query physically. Unlike traditional views, which are computed each time they are referenced, materialized views are pre-computed and stored, improving query performance significantly. They are especially useful in data warehousing contexts, where complex queries involving large amounts of data are common. The concept of materialized views has been adopted by various high-end RDBMS vendors, each with slightly different implementations, such as Oracle's "Materialized Views," DB2's "Automatic Summary Tables (ASTs)," and Teradata's "Aggregate Join Indexes."

Traditional Views vs. Materialized Views

- **Traditional Views**: These are virtual views that do not store data physically. Each time the view is queried, the database recomputes it by joining, filtering, or aggregating the underlying base tables at runtime. This process can be resource-intensive, especially for complex queries, and may result in poor performance. Traditional views do not consume storage space except for the definitions stored in the database schema.
- **Materialized Views**: These views store the results of the query physically and are pre-computed. Once created, the materialized view does not require re-computation unless it is refreshed (manually or automatically). This provides significant performance improvements, especially for complex queries with multiple joins and aggregations. However, materialized views consume storage space, and there is a maintenance cost when base table data changes, as the materialized view must be updated to reflect those changes.

Key Features of Materialized Views

- **Pre-built Joins and Aggregations**: Materialized views pre-compute complex joins and aggregations, saving time and computational resources during query execution.
- **Storage Allocation**: Unlike traditional views, materialized views consume storage to save the results of the query execution. The storage is allocated when the materialized view is created, regardless of whether the view is queried.

- **Performance Improvement**: Since materialized views store pre-computed results, querying them is much faster compared to traditional views, which need to compute the results at runtime.
- **Maintenance Cost**: Materialized views need to be refreshed when the underlying data in the base tables changes. This can introduce maintenance overhead, especially if the base tables are frequently updated.

Example of Materialized View

Let's consider a simple scenario where we have two tables: `Sales` and `Products`.

Sales Table		Products Table
Sale_ID	Product_ID	Amount
1	101	500
2	102	300
3	101	150
4	103	200

Now, assume we create a materialized view to pre-compute the total sales by product category.

Materialized View Query:

```
CREATE MATERIALIZED VIEW Total_Sales_By_Category AS
SELECT p.Category, SUM(s.Amount) AS Total_Sales
FROM Sales s
JOIN Products p ON s.Product_ID = p.Product_ID
GROUP BY p.Category;
```

Materialized View (Stored Result):

Category	Total_Sales
Electronics	950
Apparel	350

In this example:

- The materialized view `Total_Sales_By_Category` stores the total sales for each product category. The results are pre-computed and stored.
- When you query this materialized view, the database does not have to join the `Sales` and `Products` tables and perform the aggregation again. Instead, it retrieves the pre-computed results directly from the materialized view, resulting in faster query execution.

Types of Materialized Views

1. **Join Indexing**:
   - This is an advanced version of pre-join denormalization. The database creates indexes on joins between tables, which improves query performance by avoiding the need to rejoin the tables at runtime.
   - Example: A join index between the `Sales` and `Products` tables would store pre-computed results for frequently used join conditions (e.g., joining `Product_ID`).
2. **Aggregate Join Indexing**:

- This type of materialized view stores pre-computed aggregates (like sums or averages) along with joins between tables.
- Example: A materialized view that computes the total sales for each category, as shown in the previous example.

3. **Table Re-Distribution (Reversal Tables)**:
   - This method redistributes data to improve query performance, especially in distributed systems. It involves moving data across nodes to optimize data retrieval. This method is typically used in shared-nothing architectures.
   - Example: Data for a specific product category could be grouped together in one partition or node, reducing the time needed for queries on that category.
4. **Table Duplication/ Replicated Tables**:
   - In this approach, tables are duplicated across different storage locations to reduce the access time. This method is also typically used in shared-nothing architectures.
   - Example: A copy of the `Products` table could be stored in multiple locations to optimize access speed.

Advantages and Disadvantages of Materialized Views

Advantages:

- **Improved Performance**: By storing pre-computed results, materialized views significantly reduce query execution time.
- **Efficient Aggregation**: Aggregations are performed once during materialization, not at runtime, saving computational resources.

Disadvantages:

- **Storage Overhead**: Materialized views consume additional storage since they physically store the results of queries.
- **Maintenance Overhead**: Materialized views need to be refreshed when base table data changes, which can add complexity and overhead.
- **Data Inconsistency**: If a materialized view is not refreshed immediately after a change in base table data, there may be temporary data inconsistencies.

SQL Grouping Sets, Rollup, and Cube Operations

Grouping Sets Overview

Grouping sets allow multiple grouping clauses to be specified in a single SQL statement, effectively creating a union of different groupings in one pass of the data.

Basic Syntax

```
SELECT column1, column2, aggregate_function(column3)
FROM table_name
GROUP BY GROUPING SETS ((column1, column2), (column1), ());
```

Key Concepts

- Enables multiple levels of aggregation in a single query
- More efficient than using UNION ALL between different GROUP BY queries
- Can combine with GROUPING() function to identify aggregation levels

ROLLUP Operation

ROLLUP generates a result set containing sub-total rows in a hierarchical order, moving from most detailed to least detailed.

```
SELECT year, city, dept, SUM(sales)
FROM sales
GROUP BY ROLLUP (year, city, dept);
```

This creates subtotals in the following hierarchy:

- (year, city, dept)
- (year, city)
- (year)
- grand total

CUBE Operation

CUBE generates all possible combinations of the specified dimensions, creating a full ROLAP (Relational Online Analytical Processing) cube.

```
SELECT year, city, dept, SUM(sales)
FROM sales
GROUP BY CUBE (year, city, dept);
```

CUBE vs ROLLUP

Feature	CUBE	ROLLUP
Combinations	All possible	Hierarchical only
Result rows	2^n combinations	n+1 combinations
Use case	Multi-dimensional analysis	Hierarchical summaries

GROUPING Function

The GROUPING function helps identify which columns are aggregated in each row:

- Returns 1 when the column is aggregated (NULL due to grouping)
- Returns 0 when the column contains regular data

```
SELECT year, city, dept,
       SUM(sales),
       GROUPING(year) as yr_group,
       GROUPING(city) as city_group,
       GROUPING(dept) as dept_group
```

```
FROM sales
GROUP BY GROUPING SETS ((year, city, dept), (city), ());
```