Introduction

Optimizing OLAP queries involves understanding the specific characteristics of data retrieval and aggregation in OLAP systems. This lecture focuses on indexing, partitioning, and caching strategies to improve query performance for OLAP queries, particularly in large data warehouses. Key topics include a recap of indexing strategies from the OLTP perspective, bitmap indexing, row vs columnar storage, partitioning, and the use of materialized views.

Section 1: Indexing Recap (OLTP Database Perspective)

Overview of OLTP Indexing

In OLTP systems, indexing helps to quickly retrieve data by reducing the need for full table scans. This is especially beneficial for transactional systems, where data is frequently updated or inserted. Key advantages of indexing in OLTP systems include:

- **Faster Data Retrieval**: Indexes allow the database to locate data faster, similar to using an index in a book.
- **Reduced I/O Operations**: Direct access to data's physical location minimizes disk I/O, improving performance.
- **Support for Primary/Foreign Keys**: Indexes are automatically created for primary keys and often for foreign keys, which optimizes relationships between tables.
- **Improved Efficiency on Normalized Data**: Multi-table joins in OLTP systems benefit significantly from indexing, as indexes minimize the need to scan entire tables.

Why OLTP Works Well with Indexes

In OLTP, queries are optimized to touch the smallest number of rows possible, allowing for rapid response times. Common OLTP operations include:

- **Point Lookups**: Queries like "get customer where id = ?" fetch a single row, making indexes like B-trees highly efficient.
- **Small Updates**: Queries like "set order.status = 'SHIPPED' where order_id = ?" only affect a small set of rows, making it ideal for indexed access.

Why Indexing Does Not Work Well for OLAP

In OLAP, queries often aggregate or filter large data sets, requiring the engine to access a significant portion of the data. Indexes like B-trees, which work efficiently for small, targeted lookups, are not as effective in this context due to the large-scale scanning required for aggregation and filtering over multiple columns.

OLTP vs OLAP Query Patterns

Aspect	OLTP	OLAP
Query Type	Fetch small amounts of data (point lookups)	Aggregate large datasets (wide scans)
Index Type	B-tree for fast lookup	Bitmap index for efficient aggregation

---

Section 2: Indexing That Works for OLAP (Bitmap Indexing)

Bitmap Index Overview

A **bitmap index** is an indexing technique particularly suited for OLAP systems, where queries involve complex aggregations and filtering over large data sets. It differs from B-tree indexes, which work better in OLTP systems with small, indexed queries.

B-Tree vs Bitmap Index

- **B-Tree Index**: Suitable for unique or highly selective columns (e.g., "customer_id"), where quick lookups are required.
- **Bitmap Index**: Works well for columns with a limited number of distinct values, like "gender" or "status," where the data is mostly categorical.

How Bitmap Indexing Works

Instead of storing pointers to rows, a bitmap index creates a bit array for each distinct value in a column:

- For example, if the "Gender" column has values 'Male' and 'Female', a bitmap could look like:

- **Male**: 01010 (indicating rows 2 and 4 are 'Male')
- **Female**: 10101 (indicating rows 1, 3, and 5 are 'Female')

Bitwise Operations for Querying

The real power of bitmap indexing comes from **bitwise operations**. For example:

- **AND Operation**: Find all "Male" AND "Part-time" employees by performing an AND on the "Male" and "Part-time" bitmaps.
- **OR Operation**: Find "Male" OR "Full-time" employees by performing an OR on the respective bitmaps.
  This makes the query processing incredibly fast, as bitwise operations are highly efficient.

Roll_no	Name	Gender	Result
01	Geeta Raj	F	Fail
02	Deep Singh	M	Fail
03	Ria Sharma	F	Pass
04	Ajit Singh	M	Fail
05	Jitu Bagga	M	Pass
06	Neha Kapoor	F	Pass

M	F
0	1
1	0
0	1
1	0
1	0
0	1

Bitmap Indices for Gender

Pass	Fail
0	1
0	1
1	0
0	1
1	0
1	0

Bitmap Indices for Result

$$\text{Bitmap('Male') AND Bitmap('Part-time')}$$

$$01010 \text{ AND } 01100 = 01000$$

Trade-offs

- **Pros**: Bitmap indexes are excellent for OLAP queries involving complex filtering and aggregation. They are highly efficient for multi-condition queries.
- **Cons**: Bitmap indexes are not suitable for columns with high cardinality (many distinct values), as the bitmaps become too large and difficult to manage.

Section 3: Row vs Column Stores

Understanding Data Storage

In OLAP and OLTP systems, the way data is stored on disk greatly impacts query performance.

Row-Oriented vs Column-Oriented Storage

- **Row-Oriented Storage (OLTP)**: Data is stored as complete rows on disk, which is optimal for transactional systems where entire records are accessed frequently.
- **Column-Oriented Storage (OLAP)**: Data is stored by column rather than by row, which is optimal for analytical queries that need to scan and aggregate data over specific columns.

Comparing the Two Approaches

OLTP (Row-Oriented)

- **Data Organization**: Entire rows are stored together.
- **Access Pattern**: Quickly fetches complete records.

- **I/O Efficiency**: Low I/O for single-row queries but high I/O for column-based aggregation.

OLAP (Column-Oriented)

- **Data Organization**: Each column is stored separately.
- **Access Pattern**: Efficient for aggregating and scanning specific columns across many rows.
- **I/O Efficiency**: Low I/O for column scans but high for row assembly during writes.

Analogy: Kitchen Racks

- **Style 1**: Store items by type (e.g., all cups together).
- **Style 2**: Store items by item (e.g., each rack has a different type of item).

Style 1

Style 2

For **OLTP**, Style 1 is preferred as it allows for quick retrieval of all details about a customer. For **OLAP**, Style 2 is more efficient as it allows easy scanning of a single type of item (e.g., counting all cups).

Section 4: Partitioning

What is Partitioning?

Partitioning divides large data sets into smaller, manageable segments based on certain criteria, such as date, to optimize query performance. This is especially useful in OLAP systems for efficient analysis of historical data.

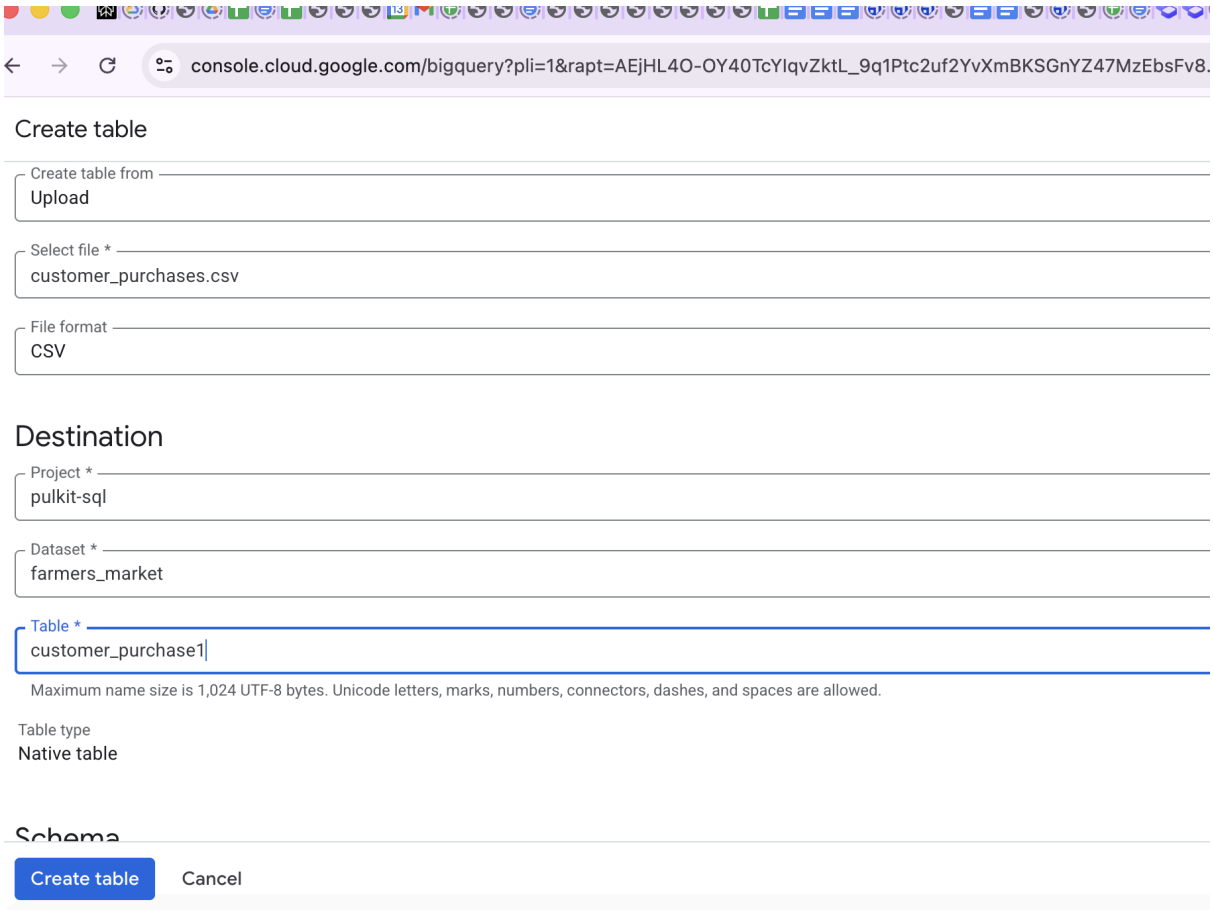
OLAP vs OLTP Partitioning

- **OLAP**: Partitioning helps in managing large analytical datasets, reducing I/O during complex aggregations.
- **OLTP**: Rarely uses partitioning, as it focuses on high-frequency transactions with minimal data overhead.

Aspect	**OLAP Partitioning**	**OLTP**
Purpose	Query pruning on large reads	Transaction speed/consistency
Data Volume	Historical, terabytes+	Current, gigabytes
Operations	Batch analytical queries	Real-time INSERT/UPDATE/DELETE
Schema	Denormalized (star)	Normalized (3NF)

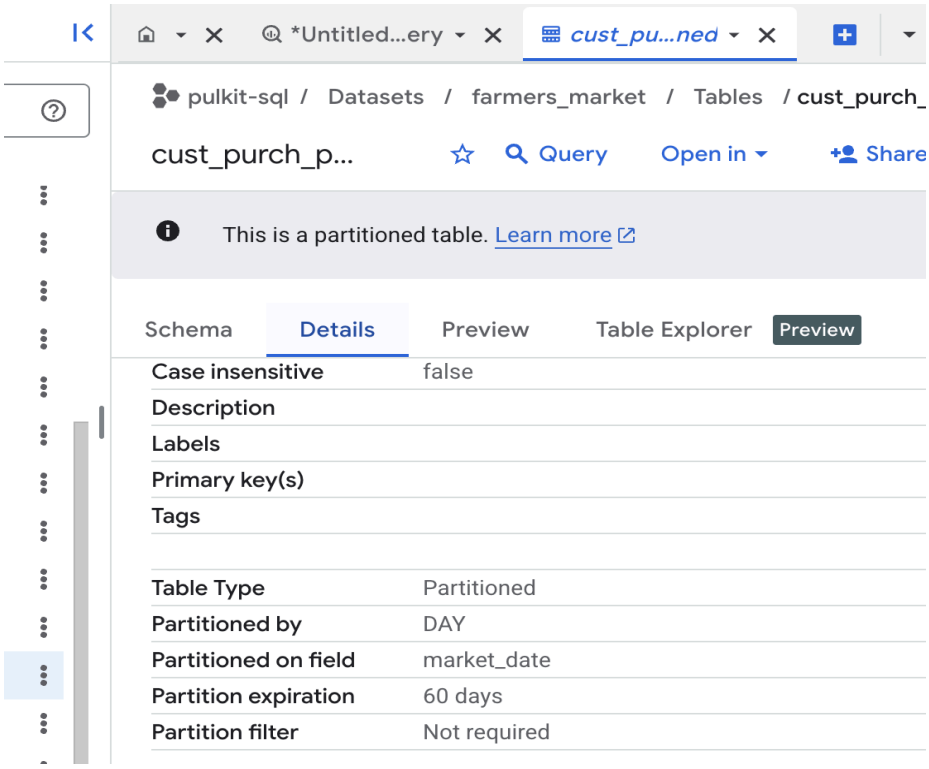
BigQuery Setup and Upload the Customer_Purchase Table using Bigquery:

Emphasise that Bigquery is a data-warehouse by GCP for analytical workloads



Create Table Partitioned By Market-Date

```
create table `farmers_market.cust_purch_partitioned`
partition by market_date
as
select * from `farmers_market.customer_purchases1`;
```



Execute Filter clause on partitioned and non-partitioned Data

```
select *
from `farmers_market.customer_purchases1`
where market_date = '2019-07-03';
```

This query will process 54.85 KB when run.

```
select *
from `farmers_market.cust_purch_partitioned`
where market_date = '2019-07-03';
```

This query will process 0 B when run.

Section 5: Materialized Views

Overview of Views and Materialized Views

- **Views**: Logical representations of data that are computed on-demand.
- **Materialized Views**: Physically store the results of a query, improving performance for complex, repeated queries.

Key Differences

Aspect	Views	Materialized Views
Storage	Logical only (no data)	Physical storage of results
Refresh	Always real-time	Incremental background refresh
Performance	Slower (recomputes each query)	Faster (cached results)
Cost	No storage fees	Storage + refresh costs
Maintenance	None	Automatic but limited SQL support

Creating a Materialized View in BigQuery

For example, creating a monthly sales summary from the `customer_purchases` table using a materialized view:

```
CREATE MATERIALIZED VIEW `farmer_market.monthly_sales`
OPTIONS(
  enable_refresh = TRUE,
  refresh_interval_minutes = 60
)
AS
SELECT
  DATE_TRUNC(market_date, MONTH) as sales_month,
  product_id,
  vendor_id,
  SUM(quantity * cost_to_customer_per_qty) as total_revenue,
  COUNT(*) as transaction_count
FROM `your-project.your-dataset.customer_purchases`
GROUP BY sales_month, product_id, vendor_id;
```

Conclusion

Key Insights

- OLTP systems benefit from traditional indexing (e.g., B-trees) due to small, targeted queries.
- OLAP systems require specialized indexing strategies like bitmap indexing to handle large-scale aggregations and filtering efficiently.
- Row-oriented storage is ideal for OLTP, while column-oriented storage excels in OLAP for fast aggregations.
- Partitioning and materialized views are powerful techniques for improving query performance in OLAP systems.

Key Terms

- **Bitmap Index**: A type of index that uses bit arrays to represent distinct values in a column for efficient querying.

- **Column-Oriented Storage**: Data storage in OLAP systems where columns are stored separately for efficient aggregation.

- **Partitioning**: Dividing large datasets into smaller, manageable segments to improve query performance.

- **Materialized View**: A precomputed, physically stored result of a query, improving the speed of repeated queries.