Université Ferhat Abbas Setif 1

Faculy Of Sciences

Computer Science Department

Databases Administration and Architecture

3rd Year Engineer Artificial Intelligence

By Dr. Lyazid TOUMI

Contents

Reference Books

- Fundamentals of Database Systems (7th Ed.), Ramez Elmasri, Shamkant Navathe Pearson, 2015.
- Physical Database Design: The Database Professional's Guide to Exploiting Indexes, Views, Storage, and More (4th Ed.), Sam S. Lightstone, Toby J. Teorey, Tom Nadeau , Morgan Kaufmann, 2007

1 Databases Physical design

1.1 History

In 1974, a pivotal debate unfolded at the annual ACM SIGFIDET (now SIGMOD) conference in Ann Arbor, Michigan, between two giants of the database world: Ted Codd, the visionary behind the relational database model, and Charlie Bachman, the driving force behind the network database model and the CODASYL report. Their discussion centered on which database model was superior, sparking a debate that spilled over into academic journals and industry publications for nearly 30 years, only concluding with Codd's passing in 2003. Over the years, countless database systems were developed to support both models. While the relational model ultimately became the industry standard, the underlying physical structures of both models continued to evolve side by side. Early on, physical design decisions revolved around indexing methods, with B+tree indexing emerging as the go-to standard. As time went on, concepts like clustering and partitioning gained traction, becoming increasingly distinct from the logical structures that dominated the debates of the 1970s.

Logical database design, which focuses on defining how data relates within a specific database system, is typically the domain of application designers and programmers. These professionals might use specialized tools like ERwin Data Modeler or Rational Rose with UML, or they might opt for more manual approaches. On the other hand, physical database design is all about creating efficient systems for storing and retrieving data.

The inner workings of your computing platform are usually overseen by the database administrator (DBA), who has access to a variety of tools provided by vendors to fine-tune database performance. This book zeroes in on the methods and tools that are widely used today for the physical design of relational databases. To illustrate key concepts, well draw on examples from well-known systems like Oracle, IBM's DB2, Microsoft SQL Server, and Postgres.

The database life cycle outlines the essential steps needed to design a logical database. It starts with conceptual modeling based on user needs,

followed by defining tables specific to the DBMS and creating a physical database optimized for performance through techniques like indexing, partitioning, clustering, and materialization. In distributed databases, physical design also involves spreading data across a network. Once the design phase is complete, the life cycle moves on to implementation and ongoing maintenance, as shown in Figure 1.2. Physical database design (step 3 below) is presented within the context of the full life cycle to highlight its connection to earlier design stages.

1.1.1 Requirements Analysis

The first step is to gather database requirements by interviewing both data producers and users. A formal requirements specification is then created, detailing the data needed for processing, the relationships within the data, and the software platform to be used for the database.

1.1.2 Logical Database Design:

This stage involves creating a conceptual model from user requirements and refining it into normalized SQL tables. The goal is to accurately capture users' data needs and relationships, making it easier to query and update the database. Using methods like Entity-Relationship (ER) modeling or the Unified Modeling Language (UML), a global scheme (a comprehensive conceptual data model diagram) is created to depict all data and their interrelationships. This schema is then transformed into normalized SQL tables, often in third normal form (3NF), to ensure data integrity. Some database tools refer to this conceptual model as a "logical model" and the physical data model as a "physical model." The implementation model specific to a DBMS, such as SQL tables, often comes from reverse engineering an existing schema rather than being built from the ground up [Silberschatz 2006]. Heres a breakdown of what the physical model definition entails:

1.1.3 Physical Database Design:

This phase involves making critical decisions about indexing, partitioning, clustering, and selectively materializing data. Physical design, as discussed in this book, begins once SQL tables are defined and normalized. The focus shifts to optimizing storage and access methods to ensure maximum disk efficiency. The ultimate goal of physical design is to enhance the performance of the database across all applications that rely on it. Key resources

that impact performance include CPU usage, I/O operations (such as disk access), and network latency. Performance is evaluated based on two main metrics: response time for individual queries and updates, and overall system throughput (measured in transactions per second).

1.1.4 Database Implementation, Monitoring, and Modification:

Once the logical and physical designs are finalized, the database is constructed using the DBMSs data definition language (DDL). The data manipulation language (DML) is then used for tasks like querying, updating, indexing, and enforcing constraints such as referential integrity. SQL combines both DDL and DML constructsfor example, "CREATE TABLE" is a DDL command, while "SELECT" falls under DML.

During the operational phase, the database is continuously monitored to ensure it meets performance standards. If issues arise, adjustments are made to fine-tune performance. As user needs evolve or grow, further modifications may be required, creating an ongoing cycle of monitoring, redesign, and updates.

1.2 Indexes

An index is a data structure designed to organize data in a way that speeds up data retrieval from database tables. Programmers can create indexes using SQL commands like:

```
1 CREATE UNIQUE INDEX supplierNum ON supplier(snum);
2 /* Creates a unique index on a key */
```

A unique index pairs attribute values with pointers, where each pointer directs to a specific record containing that attribute value. This is known as an ordered index because the attribute values (keys) are sorted in ASCII order. For example, if the values are alphabetic, they are arranged alphabetically. These ordered indexes are typically stored as B+ trees which allow for fast searches of key values. Once the key and its corresponding pointer are located, an additional step retrieves the actual record from memory.

When data is accessed using a non-key attribute (one that may repeat across records) a secondary index is used. This structure allows multiple pointers for the same attribute value, each pointing to a record containing that value. For example:

```
CREATE INDEX shippingDate ON shipment (shipdate);
/* Creates a secondary index on a non-key */
```

For queries involving multiple attributes, a concatenated index can be created. This type of index stores a combination of attribute values with pointers to records that match all values in the set:

```
CREATE INDEX shipPart ON shipment (pnum, shipdate);
/* Creates a secondary concatenated index */
```

Concatenated indexes are highly efficient for queries that require both attributes (e.g., part number and shipping date). However, they are less efficient for queries using only one of the attributes due to their larger size, which increases search time.

To further optimize query performance, a clustered index can be used. This organizes the database so that records with similar values are stored close together on disk. For example, if a table is frequently accessed using a non-unique index on shipping dates, the database can be arranged to group similar ship dates together. Each table can have only one clustered index because it requires a fixed physical organization. Indexes that dont enforce this physical grouping are called nonclustered indexes.

For unordered tables, a hash index can improve access speed. This type of index uses a hash function to map unique key values to specific starting blocks, known as bucket addresses. Once records are inserted using the hash function, the same function can be applied for efficient retrieval.

Another type of index, the bitmap index, is often used in data warehouses and for secondary indexes with multiple values. A bitmap index consists of bit vectors, where each bit represents whether a record has a specific attribute value (1 for yes, 0 for no). Bitmap indexes are particularly effective for attributes with few distinct values, such as gender or grade, and are efficient to store and access, especially if they fit in memory. However, they are less suitable for attributes with many possible values, like last names or ages.

1.3 Materialized Views

When querying one or more tables, the results can be saved in a structure called a materialized view. Unlike standard SQL views, which are stored

only as query definitions, materialized views are stored as physical tables in the database. In data warehouses, materialized views are often used to store aggregated data from base tables. They are particularly useful for speeding up frequently executed queries or those involving data aggregates, as they allow results to be retrieved directly from the materialized view without repeatedly querying the original tables.

While materialized views can significantly reduce query times, its impractical to store all possible views due to storage limitations. Therefore, only the most useful views are typically materialized. A key challenge with materialized views is keeping them updated. When the base tables are modified, the dependent materialized views must also be updated, which can reduce efficiency. This cascading update process must be carefully managed.

1.4 Partitioning

Partitioning is a technique used in physical database design to distribute data across multiple disks, balancing the workload and reducing bottlenecks on individual hardware components. In range partitioning, data values are sorted into ranges, with each range assigned to a specific disk. This allows independent processing of different ranges. More details on partitioning can be found in Chapter 4.

1.5 Additional Physical Database Design Techniques

Other techniques can improve data access efficiency:

1.5.1 Data Compression

Reduces storage space by compressing data, which can speed up access times for frequently scanned data. Compression algorithms store data in a compact format and convert it back for display.

1.5.2 Data Striping

Spreads related data across multiple disks to enhance parallel processing and system throughput, reducing query times. This technique works well with disk array systems like RAID, which allow parallel data access.

1.5.3 Data Redundancy

Methods like mirroring improve reliability by duplicating data across multiple disks. However, redundancy increases storage requirements and requires updates to all copies when data changes. While storage costs are decreasing, update time remains a consideration. Redundancy is most beneficial for data that is rarely updated.

In some cases, the global schema may be adjusted to improve processing efficiency. This process, called textbfdenormalization, involves making minor changes to tables to enhance query performance. It requires identifying high-priority processes, evaluating the costs of queries, updates, and storage, and weighing potential drawbacks, such as risks to data integrity.

1.6 Challenges of Physical Design

Physical database design is complex due to the numerous variables involved, often numbering in the hundreds. The interdependencies between these variables make it challenging to evaluate different design choices. Manually calculating performance for individual indexing or partitioning options can take hours, and performance analysis often involves testing multiple configurations under varying loads, leading to thousands of calculations.

To simplify this process, automated tools like IBMs DB2 Design Advisor, Oracles SQL Access Advisor, Oracles SQL Tuning Advisor, and Microsoft's Database Tuning Advisor (formerly the Index Tuning Wizard) have been developed. These tools handle complex computations, allowing analysts to focus on evaluating trade-offs. This book covers both manual and automated approaches to physical database design.

1.7 Lab: Using a benchmark TPC-C

1.7.1 Preliminaries

The TPC-C (Transaction Processing Performance Council - C) database is a benchmark model designed to evaluate the performance of OLTP (Online Transaction Processing) systems. It simulates a wholesale supplier managing multiple warehouses, focusing on the efficiency and scalability of transaction processing. The database schema includes tables representing warehouses, districts, customers, orders, stock, and items.

The objective of TPC-C is to assess a system's capability to handle a complex mix of five types of transactions New Order, Payment, Order Status, Delivery, and Stock-Level that involve different levels of data access and modification. This benchmark measures throughput (transactions per minute) and response time, aiming to provide an industry-standard metric for comparing transactional performance across database systems.

Create the role and the DB

1. CREATE ROLE tpcc WITH LOGIN PASSWORD 'Test2024++'
2. ALTER ROLE tpcc CREATEDB;
3. CREATE DATABASE tpcc_db OWNER tpcc;

1.7.2 In linux terminal do: Download and compile TPC-C gnerator

1. Download the TPC-C gnerator source code from Moodle
2. Unzip the tpcc-generator-master.zip
3. Compile the CPP source code inside tpcc-generator-master and generate the executable file ./tpcc-generator
4. Create the folder results inside the source folder
5. Run the command to generate CSV datasources files:

   ./tpcc-generator 205 results
6. nine (9) csv files are generated : warehouse.csv, district.csv, customer.csv, history.csv, stock.csv,orders.csv,new_order.csv,order_line.csv, item.csv

1.7.3 TPC-C Schema

Create the TPC-C script bellow called tpcc-scheme.sql

- In the linux terminal run the script ans insure that you have in the folder where tpcc-scheme.sql is stored : tpcc-scheme.sql to create the tables:

```
psql -U tpcc -d tpcc\_db -f tpcc-scheme.sql
```

```
-- WAREHOUSE table
CREATE TABLE WAREHOUSE (
    W_ID INTEGER PRIMARY KEY,
    W_NAME VARCHAR(10),
    W_STREET_1 VARCHAR(20),
    W_STREET_2 VARCHAR(20),
    W_CITY VARCHAR(20),
    W_STATE CHAR(2),
    W_ZIP CHAR(9),
    W_TAX NUMERIC(4, 4),
    W_YTD NUMERIC(12, 2)
);

-- DISTRICT table
CREATE TABLE DISTRICT (
    D_ID INTEGER,
    D_W_ID INTEGER REFERENCES WAREHOUSE(W_ID),
    D_NAME VARCHAR(10),
    D_STREET_1 VARCHAR(20),
    D_STREET_2 VARCHAR(20),
    D_CITY VARCHAR(20),
    D_STATE CHAR(2),
    D_ZIP CHAR(9),
    D_TAX NUMERIC(4, 4),
    D_YTD NUMERIC(12, 2),
    D_NEXT_O_ID INTEGER,
    PRIMARY KEY (D_ID, D_W_ID)
);

-- CUSTOMER table
CREATE TABLE CUSTOMER (
    C_ID INTEGER,
    C_D_ID INTEGER,
    C_W_ID INTEGER,
    C_FIRST VARCHAR(16),
    C_MIDDLE CHAR(2),
    C_LAST VARCHAR(16),
    C_STREET_1 VARCHAR(20),
    C_STREET_2 VARCHAR(20),
    C_CITY VARCHAR(20),
    C_STATE CHAR(2),
    C_ZIP CHAR(9),
```

```
    C_PHONE CHAR(16),
    C_SINCE_2 INTEGER,
    C_CREDIT CHAR(2),
    C_CREDIT_LIM NUMERIC(12, 2),
    C_DISCOUNT NUMERIC(4, 4),
    C_BALANCE NUMERIC(12, 2),
    C_YTD_PAYMENT NUMERIC(12, 2),
    C_PAYMENT_CNT INTEGER,
    C_DELIVERY_CNT INTEGER,
    C_DATA VARCHAR(500),
    PRIMARY KEY (C_W_ID, C_D_ID, C_ID),
    FOREIGN KEY (C_D_ID, C_W_ID) REFERENCES DISTRICT(D_ID, D_W_ID)
);

-- HISTORY table
CREATE TABLE HISTORY (
    H_C_ID INTEGER,
    H_C_D_ID INTEGER,
    H_C_W_ID INTEGER,
    H_D_ID INTEGER,
    H_W_ID INTEGER,
    H_DATE_2 INTEGER,
    H_AMOUNT NUMERIC(6, 2),
    H_DATA VARCHAR(24),
    FOREIGN KEY (H_C_W_ID, H_C_D_ID, H_C_ID) REFERENCES
     ↪  CUSTOMER(C_W_ID, C_D_ID, C_ID),
    FOREIGN KEY (H_W_ID, H_D_ID) REFERENCES DISTRICT(D_W_ID, D_ID)
);

-- ORDERS table
CREATE TABLE ORDERS (
    O_ID INTEGER,
    O_D_ID INTEGER,
    O_W_ID INTEGER,
    O_C_ID INTEGER,
    O_ENTRY_D_2 INTEGER,
    O_CARRIER_ID INTEGER,
    O_OL_CNT INTEGER,
    O_ALL_LOCAL INTEGER,
    PRIMARY KEY (O_W_ID, O_D_ID, O_ID),
    FOREIGN KEY (O_W_ID, O_D_ID, O_C_ID) REFERENCES CUSTOMER(C_W_ID,
     ↪  C_D_ID, C_ID)
```

```
);

-- NEW_ORDER table
CREATE TABLE NEW_ORDER (
    NO_O_ID INTEGER,
    NO_D_ID INTEGER,
    NO_W_ID INTEGER,
    PRIMARY KEY (NO_W_ID, NO_D_ID, NO_O_ID),
    FOREIGN KEY (NO_W_ID, NO_D_ID, NO_O_ID) REFERENCES ORDERS(O_W_ID,
     ↪  O_D_ID, O_ID)
);

-- ORDER_LINE table
CREATE TABLE ORDER_LINE (
    OL_O_ID INTEGER,
    OL_D_ID INTEGER,
    OL_W_ID INTEGER,
    OL_NUMBER INTEGER,
    OL_I_ID INTEGER,
    OL_SUPPLY_W_ID INTEGER,
    OL_DELIVERY_D_2 INTEGER,
    OL_QUANTITY NUMERIC(2),
    OL_AMOUNT NUMERIC(6, 2),
    OL_DIST_INFO CHAR(24),
    PRIMARY KEY (OL_W_ID, OL_D_ID, OL_O_ID, OL_NUMBER),
    FOREIGN KEY (OL_W_ID, OL_D_ID, OL_O_ID) REFERENCES ORDERS(O_W_ID,
     ↪  O_D_ID, O_ID),
    FOREIGN KEY (OL_SUPPLY_W_ID, OL_I_ID) REFERENCES STOCK(S_W_ID,
     ↪  S_I_ID)
);
-- ITEM table
CREATE TABLE ITEM (
    I_ID SERIAL PRIMARY KEY,
    I_IM_ID INTEGER,
    I_NAME VARCHAR(24) NOT NULL,
    I_PRICE NUMERIC(5, 2) NOT NULL,
    I_DATA VARCHAR(50)
);

- STOCK table
CREATE TABLE STOCK (
    S_I_ID INTEGER,
```

```
    S_W_ID INTEGER,
    S_QUANTITY NUMERIC(4),
    S_DIST_01 CHAR(24),
    S_DIST_02 CHAR(24),
    S_DIST_03 CHAR(24),
    S_DIST_04 CHAR(24),
    S_DIST_05 CHAR(24),
    S_DIST_06 CHAR(24),
    S_DIST_07 CHAR(24),
    S_DIST_08 CHAR(24),
    S_DIST_09 CHAR(24),
    S_DIST_10 CHAR(24),
    S_YTD NUMERIC(8),
    S_ORDER_CNT INTEGER,
    S_REMOTE_CNT INTEGER,
    S_DATA VARCHAR(50),
    PRIMARY KEY (S_W_ID, S_I_ID),
    FOREIGN KEY (S_W_ID) REFERENCES WAREHOUSE(W_ID)
    FOREIGN KEY (S_I_ID) REFERENCES ITEM(I_ID)
   );
```

1.7.4 In the postgres terminal: Load TPC-C in tpcc_db

- Connect t the tpcc_db with tpcc user :

  psql -U tpcc -d tpcc_db

```
tpcc_db=> \COPY WAREHOUSE FROM './results/warehouse.csv' DELIMITER
 ↪  ','
tpcc_db=> \COPY DISTRICT FROM './results/district.csv' DELIMITER ','
tpcc_db=> \COPY CUSTOMER FROM './results/customer.csv' DELIMITER ','
tpcc_db=> \COPY HISTORY FROM './results/history.csv' DELIMITER ','
tpcc_db=> \COPY STOCK FROM './results/stock.csv' DELIMITER ','
tpcc_db=> \COPY ORDERS FROM './results/order.csv' DELIMITER ',' CSV
 ↪  NULL 'null';
tpcc_db=> \COPY NEW_ORDER FROM './results/new_order.csv' DELIMITER
 ↪  ',' CSV NULL 'null';
tpcc_db=> \COPY ORDER_LINE FROM './results/order_line.csv' DELIMITER
 ↪  ',' CSV NULL 'null';
tpcc_db=> \COPY ITEM FROM './results/item.csv' DELIMITER ',' CSV
 ↪  NULL 'null';

```

1.7.5 Database tables size information after loading csv files

- WAREHOUSE : 250 rows
- DISTRICT : 2050 rows
- CUSTOMER : 6150000 rows
- HISTORY : 6150000 rows
- STOCK : 20500000 rows
- ORDERS : 6150000 rows
- NEW_ORDER : 1845000 rows
- ORDER_LINE : 61502422 rows
- ITEM_LINE : 100000 rows

2 Indexation

"If you don't find it in the index, look very carefully through the entire catalogue."
-Sears, Roebuck, and Co., Consumer's Guide, 1897

2.1 Introduction

The concept of indexing has long been employed in dictionaries, encyclopedias, manuscripts, catalogs, and books. In computing, searching for data in a file is analogous to looking up information in a dictionary. Typically, this process requires examining the entire dataset sequentially. However, the use of an index set eliminates the need for sequential examination of all data. By employing appropriate indexing techniques, database transactions can be significantly accelerated, enabling faster data access.

Since the late 1970s, various methods have been developed for indexing storage structures in relational and hierarchical databases. These include sequential, sequential indexed, hash-based, binary search trees, and B-trees. While these techniques provide database designers with a wide range of options, selecting the most suitable indexing method remains a complex task. For other database types, such as object-oriented, spatial, and temporal databases, specialized indexing methods have also been developed. Despite this diversity, the B-tree index remains the most widely used method in commercial relational database management systems (DBMSs).

This overview explores indexing techniques utilized in both traditional database systems and data warehouse environments. Special attention is given to the primary indexing methods in data warehouses, with a detailed discussion on Bitmap Join indexes.

2.2 Indexation techniques

2.2.1 B-tree index

The B-tree index is one of the most widely supported indexing methods in commercial DBMSs. Structurally, it resembles an inverted tree, with the lowest level containing actual data values and pointers to the corresponding rows. B-tree indexes maintain a nearly uniform complexity for both search and update operations, making them particularly suitable for OLTP environments, where search and update operations occur with similar frequency.

However, this indexing method is less effective in OLAP environments, which are characterized by a high frequency of search operations and a low frequency of updates. In a data warehouse context, B-tree indexes are best applied to unique columns or columns with very high cardinality. For instance, using a B-tree index on a Gender column, which has low cardinality, offers little benefit for OLAP queries, as it provides minimal reduction in Input/Output operations.

B-tree indexes can be created on single or multiple columns. Figure ?? illustrates an example of a B-tree index applied to the CATEGORY column in the PRODUCT table.

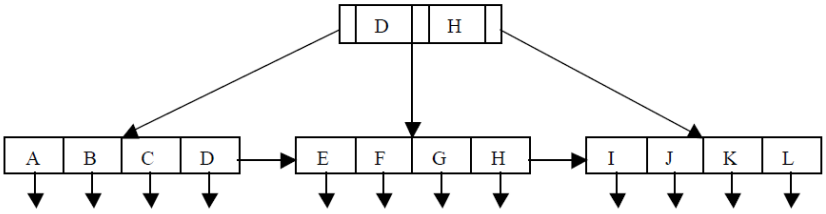


Figure 2.1: Example of B-Tree index.

2.2.2 Projection index

A projection index created on an attribute AA in a table RR stores all the values of AA in a sorted sequence, maintaining the same order as they appear in RR. Figure ?? illustrates a projection index built on the CATEGORY column in the PRODUCT table.

In a data warehouse (DW) environment, OLAP queries typically access a limited set of columns from a relation. By creating a projection index

on these frequently queried columns, query costs can be significantly reduced. For instance, SAP Sybase DBMS supports the creation of projection indexes, referred to as FastProjection Indexes, which optimize query performance in such scenarios.

Products Table		
PID	**Name**	**Category**
101	JAVA	Book
1O2	C++	Book
103	VLDB	Journal
104	DOLAP	Journal
106	DATA WAREHOUSE	Encyclopedia

Projection index created on Category
Book
Book
Journal
Journal
Encyclopedia

Figure 2.2: Example of projection index.

2.2.3 Hash index

A hash index is generated using a hash function provided by the DBMS, which maps primary key values to the physical locations of corresponding records. Figure ?? demonstrates an example of a hash index built on the PID column in the PRODUCT table.

The primary limitation of a hash index lies in the selection of the hash function. An improperly chosen hash function can significantly impact search efficiency, particularly if it produces identical hash values for a large number of keys, leading to increased collisions and reduced performance.

Products table		
PID	**Name**	**Category**
101	JAVA	Book
192	C++	Book
113	VLDB	Journal
104	DOLAP	Journal
154	DW & DM	Book
126	KDD	Book
117	ICDM	Journal

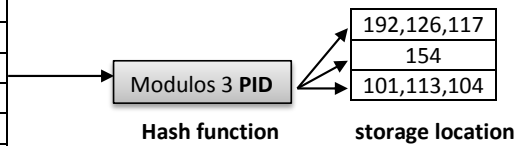


Figure 2.3: Example of hash index.

2.2.4 Bitmap Index

The Pure Bitmap Index was initially introduced in the Model 204 DBMS. A bitmap index is constructed using a table of bitmap vectors, where each bit in the vector represents a distinct value from the indexed column. For a given record ii in the indexed table, the bit corresponding to the value vv is set to 1 if the record contains that value. Figure ?? illustrates an example of a Pure Bitmap Index applied to the CATEGORY column in the PRODUCT table.

Bitmap indexes offer a simple and efficient way to represent row IDs. They are particularly advantageous in terms of storage and CPU usage, especially when the indexed column has a low number of distinct values. By leveraging Boolean operations like OR, AND, and NOT on restriction predicates, bitmap indexes optimize complex query performance. In data warehouse environments, bitmap indexes are most effective for non-unique columns, while B-tree indexes are more suitable for high-cardinality columns, such as Name or PhoneNumber.

When answering a query with a bitmap index, the relevant bitmap vectors are first loaded into memory, and Boolean operations are then performed on them. However, the primary limitation of bitmap indexes is their performance on high-cardinality columns. These columns require more storage space and result in longer query processing times. Many DBMSs, including Oracle, Sybase, Informix, and Red Brick, support the implementation of bitmap indexes.

Products Table		
PID	**Name**	**Category**
101	JAVA	Book
1O2	C++	Book
103	VLDB	Journal
104	J. Supercomp	Journal
106	DATA WAREHOUSE	Encyclopedia

Bitmap index created on Category		
Book	**Journal**	**Encyclopedia**
1	0	0
1	0	0
0	1	0
0	1	0
0	0	1

Figure 2.4: Example of bitmap index.

2.2.5 Join index

In a data warehouse (DW) environment, join operations are often computationally expensive. To address this, Valduriez introduced the Join Index,

which significantly enhances the processing speed of OLAP queries (see Fig. ??). The Join Index works by precomputing join operations in advance, reducing the overhead during query execution.

When executing a query using the Join Index, the DBMS typically follows these steps:

1. Read the Join Index JI
2. Perform $R \times JI$.
3. Internally sort the join index JI_k on s
4. Perform $S \times JI$.

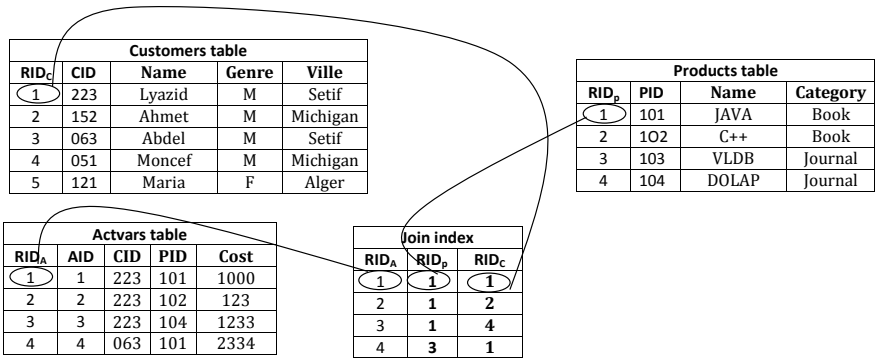


Figure 2.5: Example of join index.

The size of the Join Index depends on the selectivity factor of the join operation. A lower selectivity factor (close to 0) results in a smaller Join Index, while a higher selectivity factor (close to 1, which implies the join approaches a Cartesian product) leads to a larger Join Index.

2.2.6 Star join index

The Join Index is beneficial in OLTP environments for efficiently joining two relations. In data warehouse environments, where OLAP queries often involve multiple joins between dimension tables and a fact table (at least one join), Redbrick et al. Stohr00multi-dimensionaldatabase proposed an adaptation of the Join Index for star schema designs, known as the Star Join Index (SJI).

The SJI facilitates joining all dimension tables with the fact table, creating a Complete SJI. While a Complete SJI is advantageous for supporting all queries, it comes with significant drawbacks, including a large storage footprint and high maintenance costs. Additionally, the SJI is not suitable for other data warehouse designs, such as the Snowflake schema.

2.2.7 Bitmap join index

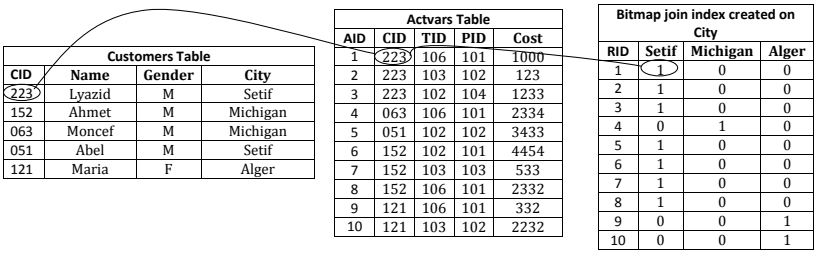
O'Neil introduced the Bitmap Join Index (BJI) by combining the concepts of the Join Index (JI) and the Bitmap Index (BI), providing a powerful tool for optimizing OLAP query performance. The BJI precomputes join operations between two or more tables, significantly enhancing query efficiency.

The BJI can be constructed using one or multiple attributes. For each value of an attribute, the BJI stores the row IDs of the corresponding rows in one or more related tables. In data warehouse environments, the join condition for a BJI is typically an equi-inner join between the primary key of dimension tables and the foreign key in the fact table. Figure ??

```
1 CREATE BITMAP INDEX cust_sales_bji
2 ON    sales(customers.city)
3 FROM  sales, customers
4 WHERE sales.cid = customers.cid;
```

Figure 2.6: Statement used to build BJI

illustrates a Bitmap Join Index (BJI) created on the CITY attribute in the CUSTOMER table and its association with the SALES table. Additionally, Figure ?? shows an example of an ORACLE statement used to define a BJI on the CITY attribute. Since the CUSTOMERS.CITY attribute is specified in the ON clause of the index, queries involving a join between the SALES table and the CUSTOMERS table using the CITY attribute can directly utilize the BJI, eliminating the need for performing the join operation at runtime.



Customers Table

CID	Name	Gender	City
223	Lyazid	M	Setif
152	Ahmet	M	Michigan
063	Moncef	M	Michigan
051	Abel	M	Setif
121	Maria	F	Alger

Actvars Table

AID	CID	TID	PID	Cost
1	223	106	101	1000
2	223	103	102	123
3	223	102	104	1233
4	063	106	101	2334
5	051	102	102	3433
6	152	102	101	4454
7	152	103	103	533
8	152	106	101	2332
9	121	106	101	332
10	121	103	102	2232

Bitmap join index created on City

RID	Setif	Michigan	Alger
1	1	0	0
2	1	0	0
3	1	0	0
4	0	1	0
5	1	0	0
6	1	0	0
7	1	0	0
8	1	0	0
9	0	0	1
10	0	0	1

Figure 2.7: Example of bitmap join index.

```
SELECT SUM(dollarAmount)
FROM   sales, customer
WHERE  sales.cid  = customer.cid
AND    customer.city = 'Setif';
```

Figure 2.8: Example of OLAP query

2.3 Lab

2.3.1 Objective

The objective of this lab is to explore all indexing techniques available in PostgreSQL, such as B-tree, Hash and GIN, and evaluate their performance and suitability for the TPC-C benchmark.

2.3.2 Prerequisites

- PostgreSQL installed on your system.
- TPC-C schema and data loaded into PostgreSQL.
- Basic understanding of PostgreSQL indexing mechanisms (Assistance in the courses).

2.3.3 Explore Existing Indexes

List all existing indexes using the following query:

```
SELECT tablename, indexname, indexdef
FROM pg_indexes
WHERE schemaname = 'public';
```

2.3.4 Create Category Indexes

Create various types of indexes on a sample table (e.g., `customer`):

- B-tree Index (default):

```
CREATE INDEX index_btree_customer_c_id ON customer USING btree
↪  (c_id);
```

- Hash Index:

```
CREATE INDEX hash_customer_id ON customer USING HASH (c_id);
```

- GIN Index (for full-text or JSON):

```
CREATE INDEX gin_customer_name ON customer USING
GIN (to_tsvector('english', c_last));
```

2.3.5 Drop and Recreate Indexes

Drop and recreate indexes to analyze their impact:

```
DROP INDEX btree_customer_id;

CREATE INDEX hash_customer_id ON customer USING HASH (c_id);
```

2.3.6 Benchmark Query Performance

Run typical TPC-C queries and measure performance using `EXPLAIN ANALYZE`:

- Without indexes (Drop all index, that you have created previously):

```
EXPLAIN ANALYZE SELECT * FROM customer WHERE c_id = 12345;
```

- With indexes:

```
EXPLAIN ANALYZE SELECT * FROM customer WHERE c_id = 12345;
```

Compare execution times, costs, and row retrieval efficiency.

2.3.7 Monitor Index Usage

Use the `pg_stat_user_indexes` view to monitor index usage:

```
SELECT relname AS table_name,
       indexrelname AS index_name,
       idx_scan AS index_scans,
       idx_tup_read AS tuples_read,
       idx_tup_fetch AS tuples_fetched
FROM pg_stat_user_indexes;
```

2.3.8 Evaluate Disk Usage

Check the size of different indexes:

```
SELECT relname AS index_name,
       pg_size_pretty(pg_relation_size(indexrelid)) AS size
FROM pg_stat_user_indexes
WHERE schemaname = 'public';
```

2.3.9 Analyze Results

- Compare execution times for each index type.
- Evaluate disk space usage.
- Discuss trade-offs for each type of index based on the query patterns.

3 Horizontal partitioning

"Nothing is particularly hard if you divide it into small jobs."
-Henry Ford (1863-1947)

3.1 Introduction

Horizontal partitioning (HP) is a crucial optimization technique in the physical design of databases, significantly influencing DBMS performance. It enables the division of tables, views, and indexes into smaller, manageable partitions. This chapter provides a comprehensive review of the partitioning methods employed in commercial DBMS. Additionally, it introduces a proposed approach for selecting the optimal partitioning schema, applicable to both database and data warehouse environments.

3.2 Horizontal partitioning

Horizontal partitioning of a relation R is performed based on the domains of its attributes, with each partition comprising a subset of R that shares a specific property. This approach provides database administrators and designers with greater flexibility to manage smaller, more manageable data units Inmon. Horizontal partitioning is generally classified into two main types: primary horizontal partitioning and referential horizontal partitioning.

- Primary Horizontal Partitioning (PHP): This type of horizontal partitioning is applied to a relation R using a set of restriction predicates defined on R. PHP minimizes query costs on R by reducing access to irrelevant data and supports parallel query execution, thereby achieving a high degree of parallelism. For instance, when a query Q contains a restriction predicate in its WHERE clause, the DBMS optimizer eliminates unnecessary partitions and loads only the relevant ones to process Q. Formally, given a database D with a set of

relations $R = \{R_1, \ldots, R_n\}$, each relation $R_i(A_1, \ldots, A_m)$ has a set of attributes A. Each attribute $A_j, 1 \leq j \leq M$, is associated with a domain $Dom(A_j) = \{d_{j1}, \ldots, d_{jN_j}\}$. The partitions $\{R_{i1}, \ldots, R_{ik}\}$ are the horizontal partitions of R_i, generated by the PHP process using a set of restriction predicates. To reconstruct the original relation, a union operation is performed: $R = \bigcup_{i=1}^{n} R_i$.

- Referential Horizontal Partitioning (RHP): This form of partitioning is based on restriction predicates defined on another relation. RHP is generally more complex than PHP. Formally, consider two relations R and S, where R has a foreign key referencing S. First, R is partitioned horizontally into a set of partitions $\{R_1, \ldots, R_k\}, (1 \leq k \leq M)$, using a set of restriction predicates. The relation S is then partitioned using referential horizontal partitioning, resulting in a set of partitions $\{S_1, \ldots, S_k\}, (1 \leq k \leq M)$. Each partition S_k is constructed as $S_k = S \ltimes R_k, (1 \leq k \leq M)$. The primary goal of RHP is to reduce the cost of join operations.

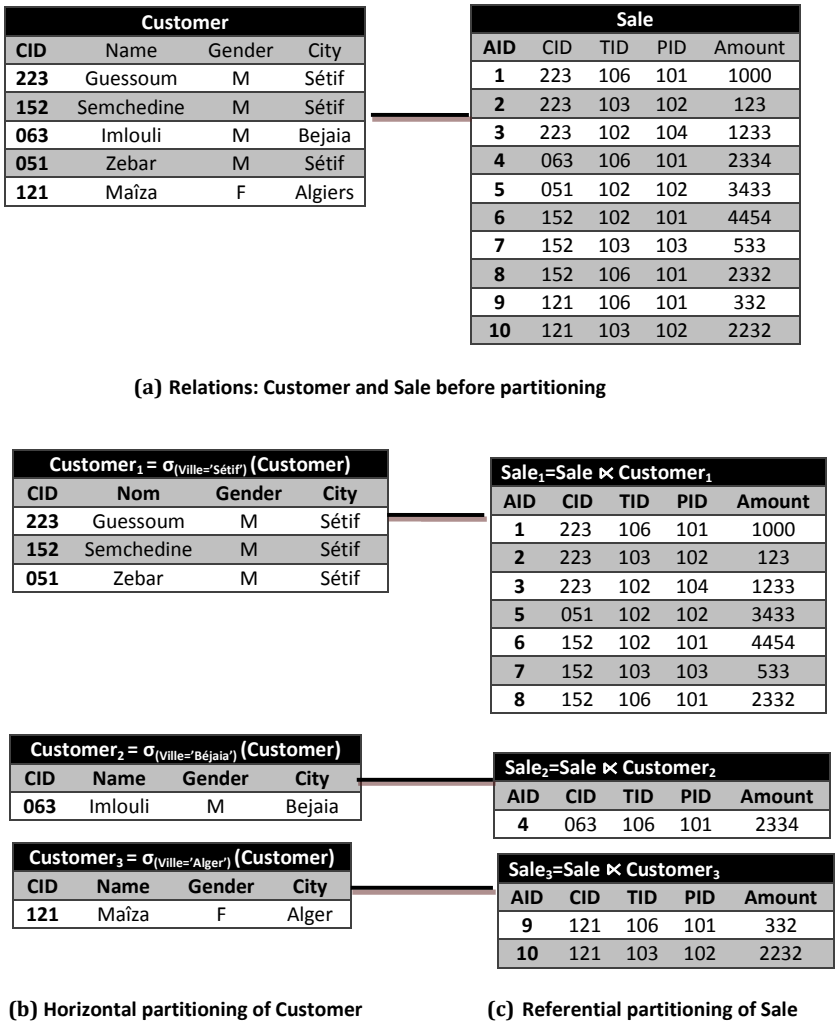
3.3 Horizontal partitioning example

Consider two relations, Customer and Sale, connected by the attribute Custlevel. The tuples of these relations are illustrated in Fig. ??. Initially, the Customer relation is divided into three horizontal partitions using Primary Horizontal Partitioning (PHP). These partitions are created based on the following restriction predicates:

- $\text{Customers}_1 = \sigma_{City='Setif'}(\text{Customer})$
- $\text{Customers}_2 = \sigma_{City='Bejaia'}(\text{Customer})$
- $\text{Customers}_3 = \sigma_{City='Algiers'}(\text{Customer})$

Fig. ??.c illustrates the referential horizontal partitioning of the SALE relation, which is derived from the partitions created during the PHP process on the Customer relation. Each Sale partition is generated through a semi-join operation between the SALE relation and the corresponding Customer partition, as described below:

- $\text{Sale}_1 = \text{Sales} \ltimes \text{Customer}_1$
- $\text{Sale}_2 = \text{Sales} \ltimes \text{Customer}_2$
- $\text{Sale}_3 = \text{Sales} \ltimes \text{Customer}_3$



Customer

CID	Name	Gender	City
223	Guessoum	M	Sétif
152	Semchedine	M	Sétif
063	Imlouli	M	Bejaia
051	Zebar	M	Sétif
121	Maîza	F	Algiers

Sale

AID	CID	TID	PID	Amount
1	223	106	101	1000
2	223	103	102	123
3	223	102	104	1233
4	063	106	101	2334
5	051	102	102	3433
6	152	102	101	4454
7	152	103	103	533
8	152	106	101	2332
9	121	106	101	332
10	121	103	102	2232

(a) Relations: Customer and Sale before partitioning

$Customer_1 = \sigma_{(Ville='Sétif')}(Customer)$

CID	Nom	Gender	City
223	Guessoum	M	Sétif
152	Semchedine	M	Sétif
051	Zebar	M	Sétif

$Sale_1 = Sale \ltimes Customer_1$

AID	CID	TID	PID	Amount
1	223	106	101	1000
2	223	103	102	123
3	223	102	104	1233
5	051	102	102	3433
6	152	102	101	4454
7	152	103	103	533
8	152	106	101	2332

$Customer_2 = \sigma_{(Ville='Béjaia')}(Customer)$

CID	Name	Gender	City
063	Imlouli	M	Bejaia

$Sale_2 = Sale \ltimes Customer_2$

AID	CID	TID	PID	Amount
4	063	106	101	2334

$Customer_3 = \sigma_{(Ville='Alger')}(Customer)$

CID	Name	Gender	City
121	Maîza	F	Alger

$Sale_3 = Sale \ltimes Customer_3$

AID	CID	TID	PID	Amount
9	121	106	101	332
10	121	103	102	2232

(b) Horizontal partitioning of Customer

(c) Referential partitioning of Sale

Figure 3.1: Horizontal partitioning example.

3.4 Complexity

Bellatreche et al. investigated the complexity of the Horizontal Partitioning Problem in Data Warehouses (HPPDW) and demonstrated that it is NP-complete. Their approach involved reducing the HPPDW to the 3-Partition problem, which determines whether a multiset of integers can be divided

into triples, each with the same sum. The 3-Partition problem is proven to be strongly NP-complete.

3.5 Horizontal partitioning advantages

Horizontal partitioning is a fundamental feature in database applications, designed to enhance administrative efficiency, system performance, and data availability.

3.5.1 Horizontal Partitioning for databases administration

Horizontal partitioning enables the division of a relation into smaller, more manageable units, simplifying data manipulation. This approach allows database administrators to adopt a "divide and rule" strategy for efficient data management. With horizontal partitioning, certain partitions can remain offline while others stay online. For instance, when a DBA is loading daily sales data, they can apply horizontal partitioning to the Sales table, ensuring that each partition corresponds to a single day's sales.

3.5.2 Partitioning for performance optimization

A significant challenge in managing very large databases is the growing volume of data, which negatively impacts DBMS performance due to the increasing amount of data processed during each new upload. Horizontal partitioning addresses this issue by reducing the volume of data examined.

This technique offers several advantages for optimizing DBMS performance. One key benefit is partition pruning, which allows the system to access only the relevant partition(s) required to answer a query. For example, if the Sales relation is divided into 48 partitions, each corresponding to sales data for a specific department (Wilaya), a query related to sales involving citizens of Setif would only require access to the partition containing data for Setif. The DBMS can thus ignore irrelevant partitions and focus solely on the pertinent ones, improving query efficiency.

Additionally, horizontal partitioning enhances the performance of join operations by enabling the pre-calculation of joins and breaking down large joins into smaller, more manageable operations. This reduces the computational complexity and improves overall system performance.

3.5.3 Partitioning for the availability

Partitioned relations ensure independence among partitions, particularly when each partition is stored in a separate Tablespace. If a Tablespace fails, only the partitions stored within that Tablespace are affected, while the remaining partitions continue to operate and remain accessible online.

3.6 Horizontal partitioning modes

3.6.1 Range partitioning mode

The range mode is the first partitioning mode integrated in ORACLE 8. This mode uses the domain D_k of the attribute A_k used as partitioning key of R. Each range has lower and upper bounds (see the example in Fig. 4.2 below)

Example The Fig. 3.2 illustrates a range partitioning of the Customers

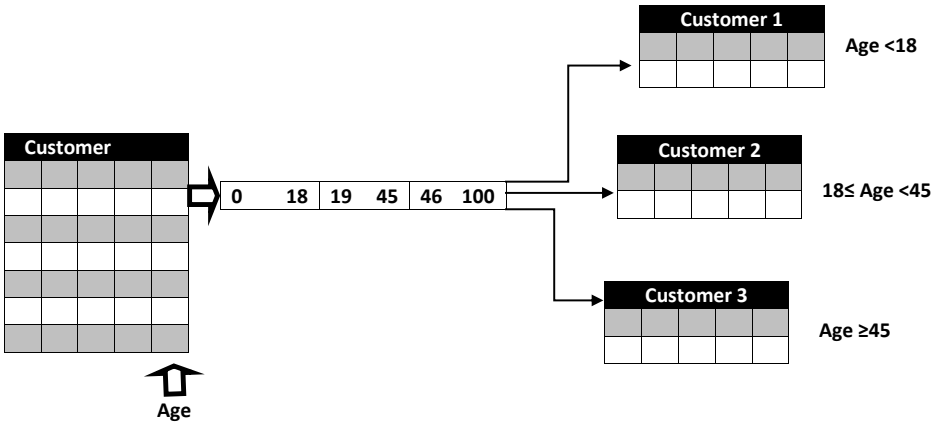


Figure 3.2: Range Mode.

on Age as partitioning key. The following ORACLE statement allows rang partitioning of Customers:

```
1 CREATE TABLE Customers
2 (CID number(9), Name varchar(25), City varchar(25),
3 Gender char(1), Age number(3)
4 PARTITION BY RANGE(Gender)
5 (PARTITION C-Childs VALUES LESS THAN (18) TABLESPACE TBS-Childs,
6 PARTITION C-Adults VALUES LESS THAN (45) TABLESPACE TBS-Adults,
7 PARTITION C-Olds VALUES LESS THAN (MAXVALUE) TABLESPACE TBS-Olds) ;
```

- The PARTITION BY RANGE clause specifies that range-based partitioning is being used. Each partition is assigned a name, such as $C\_Infants$, which represents the partition containing tuples where $Age < 18$.
- The TABLESPACE clause allows each partition to be stored in a predefined physical space.

When a tuple is inserted into the relation R, it is automatically placed into the appropriate partition based on the value of the ‘Age‘ column. For instance, if a tuple with $Age = 40$ is inserted, the DBMS first compares the ‘Age‘ value with the upper bound of the smallest partition. Finding that

40 > 18, the system proceeds to the next partition. It then checks 40 < 45 and inserts the tuple into the corresponding partition.

This partitioning mode is particularly effective for queries with range-based restriction predicates. For example:

```
1 SELECT Name FROM Customers
2 WHERE Age > 45;
```

In this case, the DBMS only loads the partition stored in the *TBS_Olds* Tablespace to answer the query, optimizing performance.

3.6.2 Hash partitioning mode

This mode utilizes a hashing algorithm provided by the DBMS. The user is

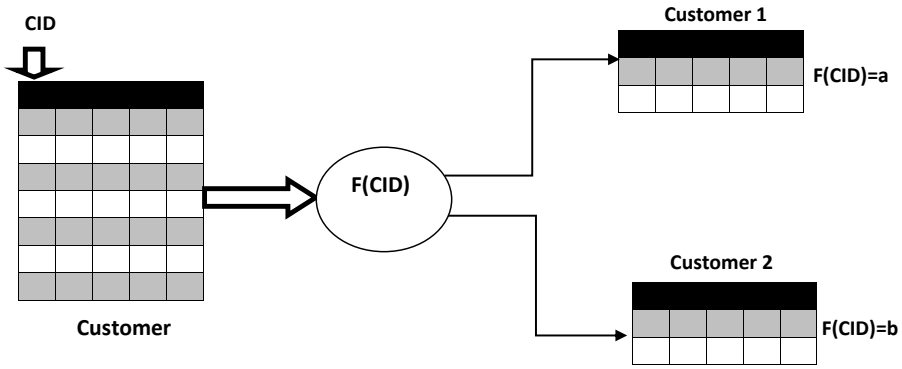


Figure 3.3: Hach Mode.

required to specify the partitioning key and the desired number of partitions. The hashing algorithm ensures an even distribution of tuples across the partitions, resulting in partitions of approximately equal size (see Fig. 3.4).

- Example: The following statement demonstrates the partitioning of the ‘Customers‘ table into four partitions using the ‘CID‘ attribute as the partitioning key. Each partition is stored in a separate TABLESPACE (TBS1, TBS2, TBS3, and TBS4).

```
CREATE TABLE CUSTOMER (CID number(9), Name varchar(25),
City varchar(25),Gender char(1), Age number(3))
PARTITION BY HASH (CID)
PARTITION 4 STORE IN (TBS1, TBS2, TBS4, TBS4) ;
```

The partitions names are automatically assigned by DBMS during the partitioning process.

3.6.3 List partitioning mode

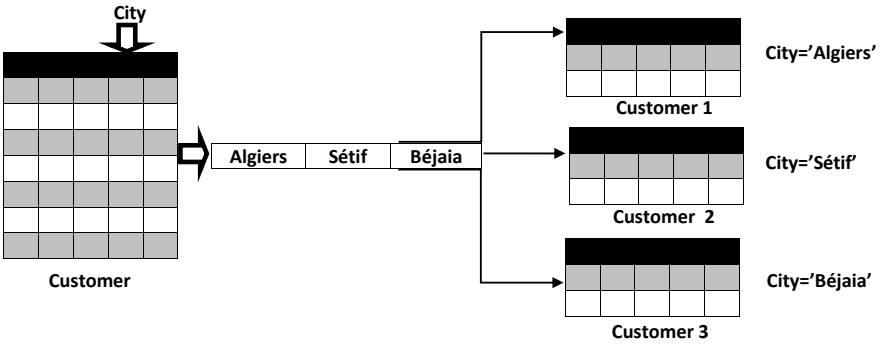


Figure 3.4: List mode.

List partitioning allows partitions to be defined based on a list of discrete values for the partitioning key. This method enables the grouping and organization of unordered and unrelated sets of data in an intuitive and logical manner.

- Example The following statement demonstrates the partitioning of the 'Customer' relation into four partitions using the list mode, with the 'City' attribute as the partitioning key. The four partitions contain customers from Setif, Bejaia, Algiers, and other cities, respectively (see Fig. 3.4).

```
CREATE TABLE CUSTOMER (CID number(9), Name varchar(25), City
  ↪ varchar(25),
Gender char(1), Age number(3))
PARTITION BY LIST (City)
```

```
4  (PARTITION C-Setif VALUES ('Setif'),
5  PARTITION C-Bejaia VALUES ('Bejaia'),
6  PARTITION C-Algiers VALUES ('Algiers'),
7  PARTITION C-Otherwise VALUES (DEFAULT)) ;
```

3.6.4 Composite partitioning mode

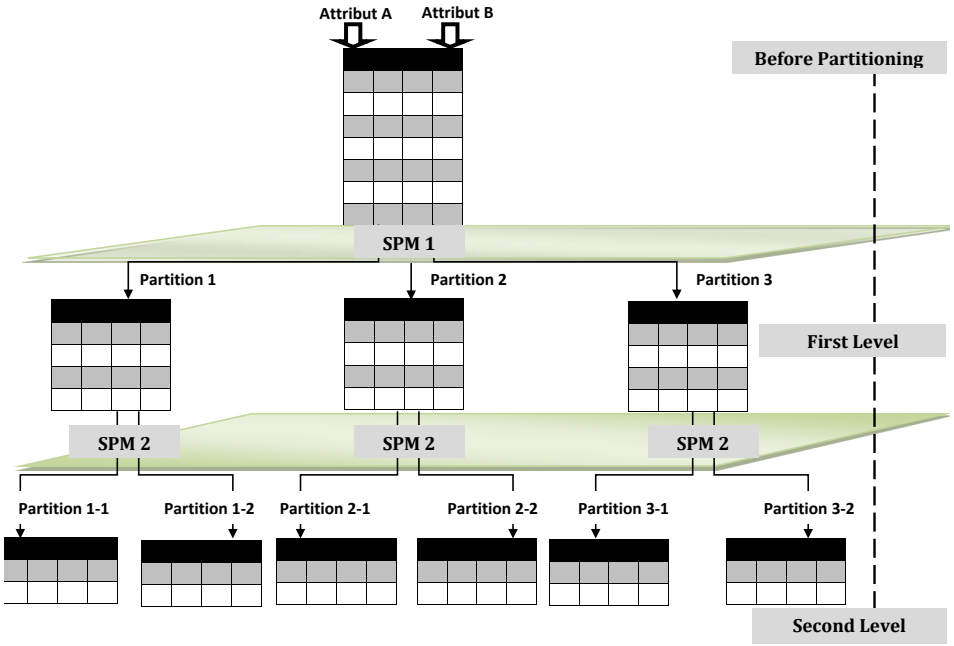


Figure 3.5: Composite partitioning mode.

Composite partitioning mode (CPO) combines two single partitioning modes, SPM1 and SPM2 (see Fig. ??). In this approach, the relation is first partitioned using SPM1, and then each resulting partition is further subdivided into sub-partitions using SPM2 whiteoracle. Several composite partitioning modes are obtained by combining single partitioning modes.

The ‘Customer‘ relation is first partitioned using ‘Gender‘ as the partitioning key. Each resulting partition is then further subdivided into sub-partitions using ‘Age‘ as the partitioning key (see Fig. 3.6). This is achieved

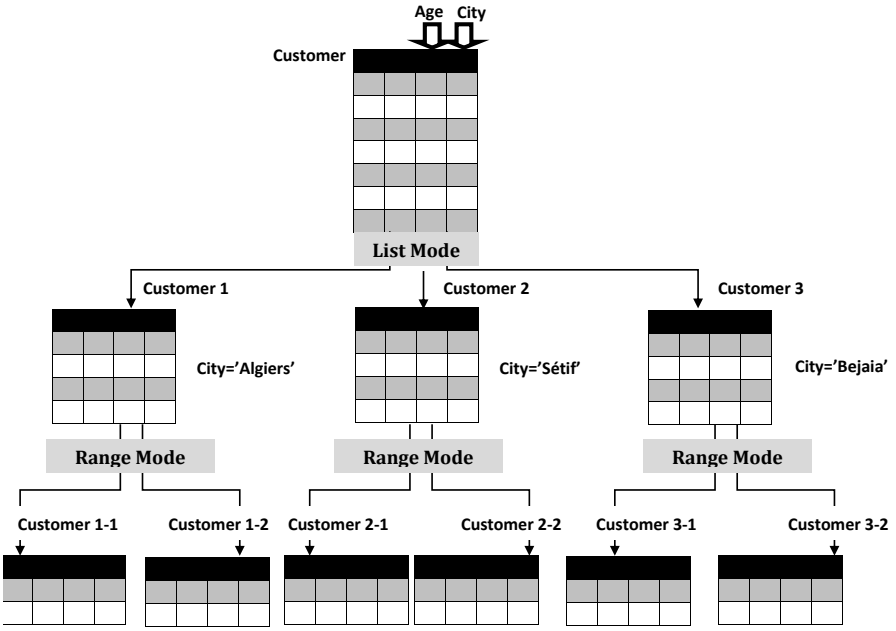


Figure 3.6: Example of composite partitioning mode.

using the following statement:

```
1  CREATE TABLE CUSTOMER
2  (CID number(9), Name varchar(25), City varchar(25),
3  Gender char(1), Age number(3)
4  PARTITION BY LIST (Gender)
5  SUBPARTITION BY RANGE (Age)
6  SUBPARTITION TEMPLATE
7  (SUBPARTITION C-Childs VALUES LESS THAN (16) TABLESPACE TBS-Childs,
8  SUBPARTITION C-Adults VALUES LESS THAN (MAXVALUE) TABLESPACE
     ↪ TBS-Adults))
9  (PARTITION C-Setif VALUES ('Setif'),
10 PARTITION C-Bejaia VALUES ('Bejaia'),
11 PARTITION C-Algiers VALUES ('Algiers')
12 PARTITION C-Otherwise VALUES (DEFAULT));
```

3.6.5 Multicolumn partitioning mode

The multicolumn partitioning mode combines range and hash partitioning methods, allowing up to 16 partitioning key columns. In this mode, the partitioning key, composed of multiple columns, provides finer granularity compared to single-column partitioning. A common example is a decomposed 'DATE' column, split into separate 'year', 'month', and 'day' columns. In DBMS, the n^{th} partitioning key is evaluated only when the values of the preceding $n-1$ keys exactly match the bounds of the corresponding $n-1$ partitions.

The following example illustrates the range partitioning of the relation Sales using two key partitioning Year and Month:

```
CREATE TABLE sales (
   Year          NUMBER,
   Month         NUMBER,
   Day           NUMBER,
   Amount   NUMBER)
PARTITION BY RANGE (Year,Month)
  (PARTITION before2014 VALUES LESS THAN (2014,1),
   PARTITION q1_2014    VALUES LESS THAN (2014,4),
   PARTITION q2_2014    VALUES LESS THAN (2014,7),
   PARTITION q3_2014    VALUES LESS THAN (2014,10),
   PARTITION q4_2014    VALUES LESS THAN (2014,1),
   PARTITION future     VALUES LESS THAN (MAXVALUE,0));
```

3.6.6 Reference partitioning mode

Previously, we discussed single and composite partitioning methods used for partitioning individual relations. In this section, we introduce the reference partitioning mode, as implemented in the Oracle 11g environment. Reference partitioning enables the partitioning of two related relations, R and S, which are connected through referential constraints. The partitioning key is determined based on the existing parent-child relationship, enforced by active and enabled primary key and foreign key constraints whiteoracle.

First, the relation R is partitioned using either a single or composite partitioning mode. If a single partitioning mode is applied to R, the number of partitions in R will be the same as the number of partitions in S. In contrast, if a composite partitioning mode is used for R, the number of partitions in R will correspond to the number of sub-partitions in S.

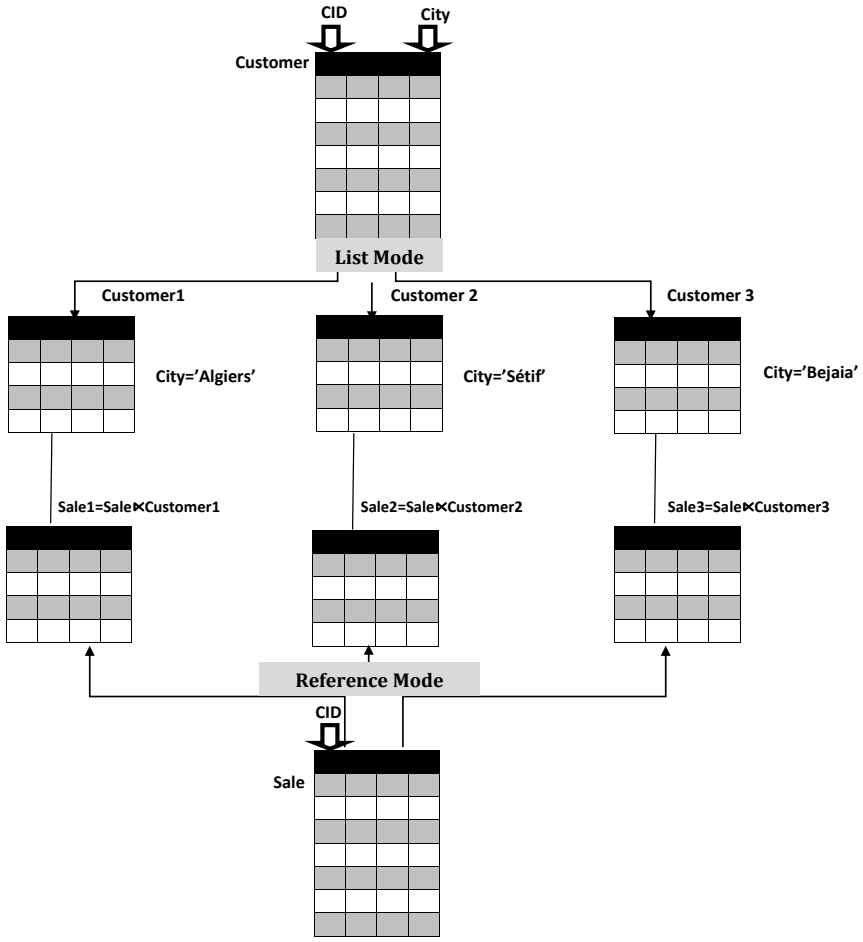


Figure 3.7: Example of reference partitioning mode.

- Example
  The 'Customer' relation is divided into three partitions: Customer1, Customer2, and Customer3 (see Fig. 3.7) using the List partitioning mode. Subsequently, three 'Sales' partitions are created, with each partition corresponding to a specific 'Customer' partition. The following statement demonstrates the partitioning of the 'Sales' relation into three partitions using the reference partitioning mode:

```
CREATE TABLE SALES
(CID number(9), Date DATE , Amount Number(10,2)
CONSTRAINT Customer_Cs FOREIGN KEY (CID) REFERENCES Customer(CID))
PARTITION BY REFERENCE(Customer_Cs);
```

3.6.7 Virtual column partitioning

This partitioning mode utilizes a virtual column in the same way as a regular column. All partitioning modes are supported with virtual columns, including range partitioning and various combinations of composite partitioning modes.

```
CREATE TABLE sales(
Pid NUMBER(6) NOT NULL
, Cid NUMBER NOT NULL
, Tid DATE NOT NULL
, CHid CHAR(1) NOT NULL
, PROMOid      NUMBER(6) NOT NULL
, quantitySold NUMBER(3) NOT NULL
, amountSold   NUMBER(10,2) NOT NULL
, totalAmount AS (quantitySold * amountSold)
)
PARTITION BY RANGE (Tid) INTERVAL (NUMTOYMINTERVAL(1,'MONTH'))
SUBPARTITION BY RANGE(totalAmount)
SUBPARTITION TEMPLATE
   ( SUBPARTITION Psmall VALUES LESS THAN (1000)
   , SUBPARTITION Pmedium VALUES LESS THAN (5000)
   , SUBPARTITION Plarge VALUES LESS THAN (10000)
   , SUBPARTITION Pextreme VALUES LESS THAN (MAXVALUE)
   )
 (PARTITION sales_before_2007 VALUES LESS THAN
       (TO_DATE('01-JAN-2007','dd-MON-yyyy'))
)
```

3.7 Lab

3.7.1 Objective

This lab will guide you through:

- Managing and updating `DATE` columns.
- Implementing all PostgreSQL partitioning modes:
  - Range Partitioning
  - List Partitioning
  - Hash Partitioning
- Migrating data to a new partitioned schema.

3.7.2 Managing **`DATE`** Columns

3.7.2.1 Create New Columns with **`DATE`** Data Type

Add `DATE` columns to existing tables:

```
ALTER TABLE CUSTOMER ADD COLUMN C_SINCE DATE;
ALTER TABLE HISTORY ADD COLUMN H_DATE DATE;
ALTER TABLE ORDERS ADD COLUMN O_ENTRY_D DATE;
ALTER TABLE ORDER_LINE ADD COLUMN OL_DELIVERY_D DATE;
```

3.7.2.2 Convert Unix Timestamps to `DATE`

Update the `DATE` columns using `to_timestamp`:

```
UPDATE CUSTOMER SET C_SINCE = to_timestamp(C_SINCE_2)::DATE;
UPDATE HISTORY SET H_DATE = to_timestamp(H_DATE_2)::DATE;
UPDATE ORDERS SET O_ENTRY_D = to_timestamp(O_ENTRY_D_2)::DATE;
UPDATE ORDER_LINE SET OL_DELIVERY_D =
  ↪ to_timestamp(OL_DELIVERY_D_2)::DATE;
```

3.7.2.3 Remove Temporary Columns

Delete the old columns after verification:

```
ALTER TABLE CUSTOMER DROP COLUMN C_SINCE_2;
ALTER TABLE HISTORY DROP COLUMN H_DATE_2;
ALTER TABLE ORDERS DROP COLUMN O_ENTRY_D_2;
ALTER TABLE ORDER_LINE DROP COLUMN OL_DELIVERY_D_2;
```

3.7.3 Backup and Save Existing TPC-C Tables

To facilitate migration, save data from the existing TPC-C tables to CSV files.

3.7.3.1 1. Export TPC-C Tables to CSV

Run the following commands:

```
\COPY CUSTOMER TO './backup/customer.csv' DELIMITER ',' CSV HEADER;
\COPY ORDERS TO './backup/orders.csv' DELIMITER ',' CSV HEADER;
\COPY ORDER_LINE TO './backup/order_line.csv' DELIMITER ',' CSV
  ↪ HEADER;
```

3.7.3.2 Verify Backup Files

Confirm the data using the `head` command or an equivalent file viewer.

```
!head ./backup/customer.csv
!head ./backup/orders.csv
!head ./backup/order_line.csv

```

3.7.4 PostgreSQL Partitioning Modes

3.7.4.1 Range Partitioning

Partition rows based on a range of values:

```
CREATE TABLE ORDER_LINE_2 (
    OL_O_ID INTEGER,
    OL_D_ID INTEGER,
    OL_W_ID INTEGER,
    OL_NUMBER INTEGER,
    OL_I_ID INTEGER,
    OL_SUPPLY_W_ID INTEGER,
    OL_QUANTITY NUMERIC(2),
    OL_AMOUNT NUMERIC(6, 2),
    OL_DIST_INFO CHAR(24),
    OL_DELIVERY_D DATE,
    PRIMARY KEY (OL_W_ID, OL_D_ID, OL_O_ID, OL_NUMBER,
    ↪ OL_DELIVERY_D),
    FOREIGN KEY (OL_W_ID, OL_D_ID, OL_O_ID) REFERENCES ORDERS(O_W_ID,
    ↪ O_D_ID, O_ID),
    FOREIGN KEY (OL_SUPPLY_W_ID, OL_I_ID) REFERENCES STOCK(S_W_ID,
    ↪ S_I_ID)
)PARTITION BY RANGE (OL_DELIVERY_D);

CREATE TABLE orders_2_1 PARTITION OF ORDER_LINE_2
FOR VALUES FROM ('1973-01-28') TO ('1973-01-29');

CREATE TABLE orders_2_2 PARTITION OF ORDER_LINE_2
FOR VALUES FROM ('1973-01-30') TO ('1973-01-31');

CREATE TABLE orders_2_others PARTITION OF ORDER_LINE_2 DEFAULT;

```

3.7.4.2 List Partitioning

Partition rows based on discrete values:

```
CREATE TABLE CUSTOMER_2 (
    C_ID INTEGER,
    C_D_ID INTEGER,
    C_W_ID INTEGER,
    C_FIRST VARCHAR(16),
    C_MIDDLE CHAR(2),
    C_LAST VARCHAR(16),
    C_STREET_1 VARCHAR(20),
    C_STREET_2 VARCHAR(20),
    C_CITY VARCHAR(20),
    C_STATE CHAR(2),
    C_ZIP CHAR(9),
    C_PHONE CHAR(16),
    C_CREDIT CHAR(2),
    C_CREDIT_LIM NUMERIC(12, 2),
    C_DISCOUNT NUMERIC(4, 4),
    C_BALANCE NUMERIC(12, 2),
    C_YTD_PAYMENT NUMERIC(12, 2),
    C_PAYMENT_CNT INTEGER,
    C_DELIVERY_CNT INTEGER,
    C_DATA VARCHAR(500),
    C_SINCE DATE,
    PRIMARY KEY (C_W_ID, C_D_ID, C_ID,C_STATE),
    FOREIGN KEY (C_D_ID, C_W_ID) REFERENCES DISTRICT(D_ID, D_W_ID)
) PARTITION BY LIST (C_STATE);

CREATE TABLE CUSTOMER_2_1 PARTITION OF CUSTOMER_2 FOR VALUES IN
 ↪  ('bD','UE');
CREATE TABLE CUSTOMER_2_2 PARTITION OF CUSTOMER_2 FOR VALUES IN
 ↪  ('kJ','W0');
CREATE TABLE CUSTOMER_2_3 PARTITION OF CUSTOMER_2 FOR VALUES IN
 ↪  ('pd','W3');
CREATE TABLE CUSTOMER_2_others PARTITION OF CUSTOMER_2 DEFAULT;

```

3.7.4.3 Hash Partitioning

Partition rows using a hash function:

```
CREATE TABLE ORDERS_2 (
    O_ID INTEGER,
    O_D_ID INTEGER,
    O_W_ID INTEGER,
    O_C_ID INTEGER,
    O_CARRIER_ID INTEGER,
    O_OL_CNT INTEGER,
    O_ALL_LOCAL INTEGER,
    O_ENTRY_D DATE,
    PRIMARY KEY (O_W_ID, O_D_ID, O_ID),
    FOREIGN KEY (O_W_ID, O_D_ID, O_C_ID) REFERENCES CUSTOMER(C_W_ID,
     ↪  C_D_ID, C_ID)
) PARTITION BY HASH (O_ID);

CREATE TABLE ORDERS_2_1 PARTITION OF ORDERS_2 FOR VALUES WITH
 ↪  (MODULUS 4, REMAINDER 0);
CREATE TABLE ORDERS_2_2 PARTITION OF ORDERS_2 FOR VALUES WITH
 ↪  (MODULUS 4, REMAINDER 1);
CREATE TABLE ORDERS_2_3 PARTITION OF ORDERS_2 FOR VALUES WITH
 ↪  (MODULUS 4, REMAINDER 2);
CREATE TABLE ORDERS_2_4 PARTITION OF ORDERS_2 FOR VALUES WITH
 ↪  (MODULUS 4, REMAINDER 3);

```

4 Materialized Views and the Selection Problem

"The more you know, the less you need to show."
- Australian Aboriginal Proverb

4.1 Introduction

In the relentless pursuit of performance within database management systems (DBMS) and data warehouses, the reduction of data access time stands as a paramount objective. While horizontal partitioning, as discussed in the previous chapter, addresses this by segmenting data into manageable physical units, another powerful technique exists at a higher level of abstraction: materialized views. A materialized view is a pre-computed result set stored as a physical table, based on a query against one or more base relations. This chapter provides a comprehensive examination of materialized views, their lifecycle, and the critical challenge of their selection. It delves into the formal problem of choosing an optimal set of views to materialize under resource constraints, a problem known to be NP-hard, and surveys the landscape of automated selection strategies that have emerged to tackle this complexity.

4.2 Materialized Views

A materialized view (MV), unlike a standard virtual view which is merely a saved query definition, physically stores the result of its defining query at the time of its creation or refresh. Formally, given a set of base relations $R = \{R_1, R_2, \ldots, R_n\}$ and a query Q defined over R, a materialized view V is a relation such that $V = Q(R)$, and the contents of V are persisted in storage.

The primary purpose of materialized views is to trade off storage space and maintenance overhead for potentially significant gains in query performance. When a user query can be answered entirely or partially from a

materialized view, the system can avoid the expensive operations of reading large volumes of base data, performing complex joins, and calculating aggregates on-the-fly.

Materialized views are broadly classified based on their completeness and the complexity of their defining query.

- Complete vs. Partial Materialization: In a complete materialization strategy, the entire result set of the defining query is stored. This is the most common approach. In contrast, a partial materialization strategy stores only a subset of the result, for instance, only the most frequently accessed rows or those meeting certain conditions.
- SPJ vs. Aggregation Views: The defining query of a materialized view often falls into one of two categories:
  - SPJ (Select-Project-Join) Views: These views involve selections (), projections (), and joins () but do not include aggregation. They are typically used to pre-compute common joins and filter conditions. Formally, an SPJ view V_{SPJ} on base relations $R_1, \ldots, R_n$ is defined as $V_{SPJ} = \pi_A(\sigma_P(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n))$, where A is a set of attributes and P is a predicate.
  - Aggregation Views: These views include group-by and aggregate functions (e.g., SUM, COUNT, AVG). They are indispensable in data warehousing for pre-computing summaries. Formally, an aggregation view V_{AGG} is defined as $V_{AGG} = \gamma_{G,AF}(R')$, where γ is the aggregation operator, G is a set of grouping attributes, AF is a set of aggregate functions, and R' is a relation (which could itself be the result of an SPJ query).

4.3 The Lifecycle of a Materialized View

The utility of a materialized view is governed by its lifecycle, which consists of three key phases: creation, maintenance, and exploitation.

4.3.1 Creation

The initial creation of a materialized view involves the execution of its defining query and the storage of the result. This can be a resource-intensive

operation, especially for views defined over large fact tables in a data warehouse. The SQL syntax for creating a materialized view, as seen in systems like Oracle, is an extension of the ‘CREATE VIEW‘ statement.

```
CREATE MATERIALIZED VIEW mv_sales_summary
BUILD IMMEDIATE
REFRESH COMPLETE ON DEMAND
ENABLE QUERY REWRITE
AS
SELECT s.cust_id,
       c.cust_name,
       t.fiscal_quarter,
       p.prod_category,
       SUM(s.amount_sold) AS total_sales,
       COUNT(*) AS num_transactions
FROM   sales s, customers c, times t, products p
WHERE  s.cust_id = c.cust_id
AND    s.time_id = t.time_id
AND    s.prod_id = p.prod_id
GROUP BY s.cust_id, c.cust_name, t.fiscal_quarter, p.prod_category;
```

The ‘BUILD IMMEDIATE‘ clause instructs the system to populate the view immediately. The ‘REFRESH‘ clause defines the maintenance strategy, and ‘ENABLE QUERY REWRITE‘ is crucial, as it allows the optimizer to transparently use the materialized view to answer queries that do not directly reference it.

4.3.2 Maintenance

A materialized view becomes stale when modifications (INSERT, UPDATE, DELETE) are applied to its underlying base relations. The process of updating the materialized view to reflect these changes is known as view maintenance. The choice of maintenance strategy is a critical trade-off between the currency of the data and the performance overhead.

- Refresh Modes:
  - On Commit: The view is updated automatically as part of the transaction that modifies the base data. This ensures strong consistency but can severely impact transaction throughput.
  - On Demand (Scheduled): The view is refreshed periodically (e.g., nightly, weekly). This is common in data warehousing where data is loaded in batches. It decouples maintenance from transaction processing but means the view data is temporarily stale.
- Refresh Methods:
  - Complete Refresh: The defining query is re-executed from scratch. This is simple but can be very expensive for large views.
  - Incremental (Fast) Refresh: Only the changes (deltas) since the last refresh are applied. This is far more efficient but is not always possible. It often requires the creation of materialized view logs on the base tables to track changes.

4.3.3 Exploitation: Query Rewriting

The true power of materialized views is realized through query rewriting. This is an optimization technique where the DBMS query optimizer transparently rewrites an incoming user query to use one or more materialized views instead of the base tables, provided the result is equivalent and correct.

For example, consider the materialized view ‘mv_sales_summary‘ defined above. A user might issue the following query to find the total sales for a specific customer in the first fiscal quarter:

```
1 SELECT cust_name, SUM(amount_sold)
2 FROM   sales s, customers c, times t
3 WHERE  s.cust_id = c.cust_id
4 AND    s.time_id = t.time_id
5 AND    t.fiscal_quarter = 'Q1-2024'
6 AND    c.cust_name = 'John Doe'
7 GROUP BY cust_name;
```

A sophisticated optimizer can recognize that this query can be answered entirely by scanning ‘mv_sales_summary‘ with the predicates ‘fiscal_quar-

ter = 'Q1-2024'‘ and ‘cust_name = 'John Doe'‘, and then summing the ‘total_sales‘ for that group. This avoids the multi-table join and aggregation at query time, leading to a dramatic performance improvement.

4.4 The Materialized View Selection Problem (MVSP)

While a single, well-chosen materialized view can be highly beneficial, the real challenge arises in a complex environment with thousands of potential views and a mixed workload of queries. The Materialized View Selection Problem (MVSP) is the task of selecting an optimal set of views to materialize, given finite resources.

4.4.1 Problem Formulation

The problem can be formally stated as follows:
Given:

- A database schema S.
- A set of queries (workload) $Q = \{Q_1, Q_2, \ldots, Q_m\}$, where each query Q_i has an associated frequency or weight f_i.
- A set of candidate materialized views $V_{cand} = \{V_1, V_2, \ldots, V_k\}$ derived from the queries in Q and the database schema S.
- A storage space constraint B (the total available space for storing materialized views).
- A maintenance cost constraint M (the maximum allowable time or resource cost for maintaining all selected views).

Find a set of views $V_{sel} \subseteq V_{cand}$ to materialize such that:

1. The total cost of evaluating the workload Q using V_{sel} is minimized. The cost is typically a weighted sum: $Total\_Cost = \sum_{i=1}^{m} f_i \times Cost(Q_i, V_{sel})$.
2. The total storage space required by V_{sel} does not exceed B: $\sum_{V_j \in V_{sel}} Size(V_j) \leq B$.
3. The total maintenance cost of V_{sel} (e.g., the time for a complete refresh) does not exceed M.

4.5 Complexity of the Materialized View Selection Problem

The MVSP has been proven to be NP-hard. This can be shown by a reduction from the well-known 0/1 Knapsack Problem.

In the Knapsack Problem, we have a set of items, each with a weight and a value, and we must select a subset that maximizes the total value without exceeding a given weight capacity. The MVSP can be seen as a Knapsack problem where:

- Each candidate view is an "item".
- The "weight" of a view is its storage cost (or maintenance cost).
- The "value" of a view is the reduction in query processing cost it provides for the entire workload.

The key complication that makes MVSP even harder than the basic Knapsack is that the "value" (benefit) of a view is not independent. The benefit of materializing a view V_a may be reduced if another view V_b, which can also be used to answer the same queries, is already selected. This interdependence introduces a complex combinatorial optimization problem. Consequently, finding an optimal solution for large problems is computationally infeasible, and research has focused on developing efficient heuristic and approximation algorithms.

4.6 Advantages of Materialized Views

The strategic use of materialized views offers profound advantages across different aspects of a data-intensive system.

4.6.1 Performance Acceleration

This is the most significant advantage. By pre-computing expensive operations, materialized views can reduce query response times by orders of magnitude. This is especially critical for:

- Complex Aggregation Queries: Calculations of sums, counts, and averages over millions of rows become simple lookups.

- Multi-Table Joins: Pre-joined tables eliminate the need for costly join operations during query execution.
- OLAP and Reporting Systems: Dashboards and reports that require consistent, fast response times heavily rely on materialized views (often called "aggregate tables" or "summary tables" in this context).

4.6.2 Reduced System Load

By serving queries from a materialized view, the system avoids placing a read load on the base tables. This is beneficial for:

- Base Table Contention: It frees up the base tables for transaction processing (OLTP), reducing lock contention.
- Resource Conservation: CPU and I/O resources that would have been used for joins and aggregations are conserved, allowing the system to support a higher number of concurrent users.

4.6.3 Simplified Application Logic

Complex queries can be encapsulated within a materialized view. Application developers can then write simpler queries against these views, making the code easier to write, understand, and maintain.

4.7 Materialized View Selection Strategies

Given the NP-hard nature of the MVSP, a variety of selection strategies have been proposed. These can be categorized as follows:

4.7.1 Manual Selection by Database Administrators (DBAs)

This is the traditional approach, where a DBA, based on experience and knowledge of the workload, manually identifies and creates a set of materialized views. While this can be effective for small systems, it does not scale to complex environments with hundreds of tables and queries. It is prone to human error and cannot systematically evaluate the vast space of possible view configurations.

4.7.2 Rule-Based Heuristics

These are simple, intuitive rules that guide the selection process. While not optimal, they provide a good starting point.

- Materialize Views from Expensive Queries: Focus on views that benefit queries with high execution cost or high frequency.
- Materialize Views Used by Multiple Queries: Prefer views that can be used to rewrite a larger number of queries in the workload.
- Prioritize Small, Aggregated Views: A small view that summarizes a large fact table often provides the highest benefit per unit of storage.

4.7.3 Cost-Based Greedy Algorithms

This is the most prevalent approach in automated selection tools. The algorithm starts with an empty set of selected views and iteratively adds the view that provides the highest benefit per unit cost (e.g., per megabyte of storage) until the storage or maintenance budget is exhausted.

Algorithm 1 Greedy Algorithm for Materialized View Selection

$V_{sel} \leftarrow \emptyset$ ▷ Set of selected views to materialize
$RemainingSpace \leftarrow B$ ▷ Initialize remaining storage budget
$V_{candidates} \leftarrow$ Set of candidate views
while $RemainingSpace > 0$ and $V_{candidates}$ is not empty do
    for each view v in $V_{candidates}$ do
        $BenefitPerMB[v] \leftarrow \frac{\text{Total Query Cost Reduction using } \{V_{sel} \cup v\}}{\text{Size}(v)}$
    end for
    $v_{best} \leftarrow$ the view in $V_{candidates}$ with the highest $BenefitPerMB$
    if $\text{Size}(v_{best}) \leq RemainingSpace$ then
        $V_{sel} \leftarrow V_{sel} \cup \{v_{best}\}$
        $RemainingSpace \leftarrow RemainingSpace - \text{Size}(v_{best})$
        $V_{candidates} \leftarrow V_{candidates} \setminus \{v_{best}\}$
    else
        $V_{candidates} \leftarrow V_{candidates} \setminus \{v_{best}\}$ ▷ Remove view that doesn't fit
    end if
end while
return V_{sel}

The major advantage of the greedy algorithm is its efficiency. Its drawback is that it can get stuck in a local optimum, as a myopic choice early on may preclude a globally better combination of views later.

4.7.4 Evolutionary and Randomized Algorithms

For very large and complex problems, algorithms inspired by natural evolution, such as Genetic Algorithms (GAs), can be effective. In a GA, a population of potential solutions (each solution is a set of views to materialize) is evolved over multiple generations. Solutions are evaluated using a fitness function (e.g., total query cost + penalty for constraint violation). Through operations like crossover (combining two solutions) and mutation (randomly altering a solution), the population converges towards a near-optimal solution. Simulated Annealing is another randomized technique that can escape local optima.

4.7.5 Integer Linear Programming (ILP)

For problems of moderate size, the MVSP can be formulated as an Integer Linear Program. Binary decision variables x_j are defined for each candidate view V_j, where $x_j = 1$ means the view is selected. The objective is to minimize total query cost, subject to linear constraints on storage and maintenance. While ILP solvers can guarantee an optimal solution, the approach becomes computationally prohibitive as the number of candidate views grows into the thousands.

4.8 Materialized View Selection in Data Warehouses

The MVSP is particularly salient in the context of data warehouses. The workload is typically read-intensive, consisting of complex analytical queries, and the data is updated in controlled batch windows. This environment is ideally suited for materialized views.

4.8.1 Integration with Data Cube Lattices

In a data warehouse, a common set of candidate views corresponds to the aggregates in a data cube. The cube lattice is a hierarchical structure where each node represents a possible group-by view. For example, from a base sales fact table (with dimensions time, product, customer, location), we can

generate views grouped by (time, product), (product), (customer, location), etc.

The selection problem then becomes: given a storage budget, which nodes of the cube lattice should be materialized to minimize the average query time for any aggregate query? The greedy algorithm has been shown to perform very well in this specific, structured context, often providing a solution that is guaranteed to be within a constant factor of the optimal.

4.9 A Unified Physical Design Tool

In modern DBMSs like Microsoft SQL Server, Oracle, and IBM DB2, the materialized view selection problem is not tackled in isolation. It is integrated into a larger Physical Design Advisor or Automatic Database Diagnostic Monitor. These tools take a workload as input (a SQL trace) and recommend a unified configuration that may include a combination of:

- A set of materialized views (or indexes) to create.
- A set of existing indexes to drop.
- A horizontal partitioning schema for key tables.

The advisor evaluates the trade-offs between these structures. For instance, creating a new index might speed up a query, but it also adds maintenance overhead. The tool's goal is to find the configuration that provides the maximum net benefit for the entire workload.

4.10 Example of Materialized View Selection

Consider a simplified data warehouse for a retail chain with a central 'sales' fact table and 'products', 'customers', and 'times' dimension tables. A typical workload might consist of the following queries:

- Q_1: Daily total sales per product category.
- Q_2: Monthly total sales per customer region.
- Q_3: Quarterly total sales and number of transactions.

The DBA runs the Physical Design Advisor, providing the trace of these queries. The advisor generates a set of candidate views, including:

- V_1: ‘sales‘ joined with ‘products‘, grouped by ‘prod_category‘ and ‘day‘.
- V_2: ‘sales‘ joined with ‘customers‘, grouped by ‘cust_region‘ and ‘month‘.
- V_3: ‘sales‘ grouped by ‘quarter‘.
- V_4: ‘sales‘ joined with all dimensions, grouped by ‘prod_category‘, ‘cust_region‘, and ‘month‘.

The advisor then executes a selection algorithm (e.g., a greedy algorithm). It calculates that V_1 is perfect for Q_1 and is relatively small. V_2 is excellent for Q_2. V_3 is very small and good for Q_3. V_4, while useful for all queries, is very large. Under a tight storage constraint, the advisor might recommend V_1, V_2, and V_3, as this set fits the budget and covers all queries effectively, leaving out the large V_4. The final recommendation would be a SQL script to create these three materialized views.

4.11 Conclusion

Materialized views represent a powerful denormalization technique that can dramatically accelerate query performance by pre-computing and storing expensive results. However, their effectiveness is contingent upon a careful and strategic selection process. The Materialized View Selection Problem is a complex, constrained optimization challenge that lies at the heart of physical database design automation. While NP-hard, the development of pragmatic greedy heuristics, evolutionary algorithms, and their integration into unified physical design advisors has made it feasible to automate this process, delivering significant performance benefits for large-scale database and data warehouse systems. The choice of which views to materialize remains a critical decision, balancing the tantalizing promise of instant query results against the very real costs of storage and maintenance.

4.12 Lab

4.12.1 Objective

This lab will guide you through:

- Creating and managing materialized views using the TPC-C schema

- Implementing different types of materialized views:
  - SPJ (Select-Project-Join) Views
  - Aggregation Views
  - Nested Materialized Views
- Managing materialized view refresh strategies
- Using materialized views for query optimization on TPC-C data
- Comparing performance with and without materialized views

4.12.2 Part 1: Understanding the TPC-C Schema

4.12.2.1 TPC-C Table Structure

The TPC-C benchmark consists of the following main tables:

```
-- Warehouse table
WAREHOUSE (
    W_ID INTEGER PRIMARY KEY,
    W_NAME VARCHAR(10),
    W_STREET_1 VARCHAR(20),
    W_STREET_2 VARCHAR(20),
    W_CITY VARCHAR(20),
    W_STATE CHAR(2),
    W_ZIP CHAR(9),
    W_TAX NUMERIC(4,4),
    W_YTD NUMERIC(12,2)
)

-- District table
DISTRICT (
    D_ID INTEGER,
    D_W_ID INTEGER,
    D_NAME VARCHAR(10),
    D_STREET_1 VARCHAR(20),
    D_STREET_2 VARCHAR(20),
    D_CITY VARCHAR(20),
    D_STATE CHAR(2),
    D_ZIP CHAR(9),
    D_TAX NUMERIC(4,4),
    D_YTD NUMERIC(12,2),
    D_NEXT_O_ID INTEGER,
    PRIMARY KEY (D_W_ID, D_ID),
    FOREIGN KEY (D_W_ID) REFERENCES WAREHOUSE(W_ID)
)

-- Customer table
CUSTOMER (
    C_ID INTEGER,
    C_D_ID INTEGER,
    C_W_ID INTEGER,
    C_FIRST VARCHAR(16),
    C_MIDDLE CHAR(2),
    C_LAST VARCHAR(16),
    C_STREET_1 VARCHAR(20),
    C_STREET_2 VARCHAR(20),
    C_CITY VARCHAR(20),
    C_STATE CHAR(2),
```

```
    C_ZIP CHAR(9),
    C_PHONE CHAR(16),
    C_SINCE TIMESTAMP,
    C_CREDIT CHAR(2),
    C_CREDIT_LIM NUMERIC(12,2),
    C_DISCOUNT NUMERIC(4,4),
    C_BALANCE NUMERIC(12,2),
    C_YTD_PAYMENT NUMERIC(12,2),
    C_PAYMENT_CNT INTEGER,
    C_DELIVERY_CNT INTEGER,
    C_DATA VARCHAR(500),
    PRIMARY KEY (C_W_ID, C_D_ID, C_ID),
    FOREIGN KEY (C_W_ID, C_D_ID) REFERENCES DISTRICT(D_W_ID, D_ID)
)

-- Orders table
ORDERS (
    O_ID INTEGER,
    O_D_ID INTEGER,
    O_W_ID INTEGER,
    O_C_ID INTEGER,
    O_ENTRY_D TIMESTAMP,
    O_CARRIER_ID INTEGER,
    O_OL_CNT INTEGER,
    O_ALL_LOCAL INTEGER,
    PRIMARY KEY (O_W_ID, O_D_ID, O_ID),
    FOREIGN KEY (O_W_ID, O_D_ID, O_C_ID) REFERENCES CUSTOMER(C_W_ID,
     ↪  C_D_ID, C_ID)
)

-- Order Line table
ORDER_LINE (
    OL_O_ID INTEGER,
    OL_D_ID INTEGER,
    OL_W_ID INTEGER,
    OL_NUMBER INTEGER,
    OL_I_ID INTEGER,
    OL_SUPPLY_W_ID INTEGER,
    OL_QUANTITY INTEGER,
    OL_AMOUNT NUMERIC(6,2),
    OL_DELIVERY_D TIMESTAMP,
    OL_DIST_INFO CHAR(24),
```

```
    PRIMARY KEY (OL_W_ID, OL_D_ID, OL_O_ID, OL_NUMBER),
    FOREIGN KEY (OL_W_ID, OL_D_ID, OL_O_ID) REFERENCES ORDERS(O_W_ID,
     ↪  O_D_ID, O_ID),
    FOREIGN KEY (OL_SUPPLY_W_ID, OL_I_ID) REFERENCES STOCK(S_W_ID,
     ↪  S_I_ID)
)

-- Item table
ITEM (
    I_ID INTEGER PRIMARY KEY,
    I_NAME VARCHAR(24),
    I_PRICE NUMERIC(5,2),
    I_IM_ID INTEGER,
    I_DATA VARCHAR(50)
)

-- Stock table
STOCK (
    S_I_ID INTEGER,
    S_W_ID INTEGER,
    S_QUANTITY INTEGER,
    S_DIST_01 CHAR(24),
    S_DIST_02 CHAR(24),
    S_DIST_03 CHAR(24),
    S_DIST_04 CHAR(24),
    S_DIST_05 CHAR(24),
    S_DIST_06 CHAR(24),
    S_DIST_07 CHAR(24),
    S_DIST_08 CHAR(24),
    S_DIST_09 CHAR(24),
    S_DIST_10 CHAR(24),
    S_YTD INTEGER,
    S_ORDER_CNT INTEGER,
    S_REMOTE_CNT INTEGER,
    S_DATA VARCHAR(50),
    PRIMARY KEY (S_W_ID, S_I_ID),
    FOREIGN KEY (S_I_ID) REFERENCES ITEM(I_ID),
    FOREIGN KEY (S_W_ID) REFERENCES WAREHOUSE(W_ID)
)

-- New Orders table
NEW_ORDER (
```

```
    NO_O_ID INTEGER,
    NO_D_ID INTEGER,
    NO_W_ID INTEGER,
    PRIMARY KEY (NO_W_ID, NO_D_ID, NO_O_ID),
    FOREIGN KEY (NO_W_ID, NO_D_ID, NO_O_ID) REFERENCES ORDERS(O_W_ID,
     ↪  O_D_ID, O_ID)
)

-- History table
HISTORY (
    H_C_ID INTEGER,
    H_C_D_ID INTEGER,
    H_C_W_ID INTEGER,
    H_D_ID INTEGER,
    H_W_ID INTEGER,
    H_DATE TIMESTAMP,
    H_AMOUNT NUMERIC(6,2),
    H_DATA VARCHAR(24),
    FOREIGN KEY (H_C_W_ID, H_C_D_ID, H_C_ID) REFERENCES
     ↪  CUSTOMER(C_W_ID, C_D_ID, C_ID),
    FOREIGN KEY (H_W_ID, H_D_ID) REFERENCES DISTRICT(D_W_ID, D_ID)
)
```

4.12.3 Part 2: Creating Materialized Views on TPC-C Schema

SPJ Materialized View - Customer Orders

Create a materialized view that pre-joins customers with their orders:

```
CREATE MATERIALIZED VIEW mv_customer_orders AS
SELECT
    c.C_W_ID as warehouse_id,
    c.C_D_ID as district_id,
    c.C_ID as customer_id,
    c.C_FIRST as first_name,
    c.C_LAST as last_name,
    c.C_BALANCE as balance,
    c.C_CREDIT as credit,
    o.O_ID as order_id,
    o.O_ENTRY_D as order_date,
    o.O_CARRIER_ID as carrier_id,
    o.O_OL_CNT as order_line_count
FROM CUSTOMER c
JOIN ORDERS o ON c.C_W_ID = o.O_W_ID
                AND c.C_D_ID = o.O_D_ID
                AND c.C_ID = o.O_C_ID
WHERE o.O_ID > 0;  -- Exclude special orders

-- Create indexes for common query patterns
CREATE INDEX idx_mv_cust_orders_warehouse ON
 ↪  mv_customer_orders(warehouse_id);
CREATE INDEX idx_mv_cust_orders_customer ON
 ↪  mv_customer_orders(customer_id, district_id, warehouse_id);
CREATE INDEX idx_mv_cust_orders_date ON
 ↪  mv_customer_orders(order_date);
```

Aggregation Materialized View - Sales Summary

Create a materialized view for sales aggregations by warehouse and district:

```
CREATE MATERIALIZED VIEW mv_sales_summary AS
SELECT
    ol.OL_W_ID as warehouse_id,
    ol.OL_D_ID as district_id,
    DATE_TRUNC('month', o.O_ENTRY_D) as sales_month,
```

```
    COUNT(DISTINCT o.O_ID) as order_count,
    COUNT(ol.OL_NUMBER) as order_line_count,
    SUM(ol.OL_QUANTITY) as total_quantity,
    SUM(ol.OL_AMOUNT) as total_sales,
    AVG(ol.OL_AMOUNT) as avg_order_line_amount,
    MAX(ol.OL_QUANTITY) as max_quantity,
    MIN(ol.OL_QUANTITY) as min_quantity
FROM ORDERS o
JOIN ORDER_LINE ol ON o.O_W_ID = ol.OL_W_ID
                    AND o.O_D_ID = ol.OL_D_ID
                    AND o.O_ID = ol.OL_O_ID
WHERE ol.OL_AMOUNT > 0
GROUP BY
    ol.OL_W_ID,
    ol.OL_D_ID,
    DATE_TRUNC('month', o.O_ENTRY_D);

-- Create indexes for analytical queries
CREATE INDEX idx_mv_sales_warehouse_month ON
 ↪ mv_sales_summary(warehouse_id, sales_month);
CREATE INDEX idx_mv_sales_district ON mv_sales_summary(district_id,
 ↪ warehouse_id);
CREATE INDEX idx_mv_sales_month ON mv_sales_summary(sales_month);
```

Complex Join Materialized View - Order Analysis

Create a comprehensive materialized view with multiple joins:

```
CREATE MATERIALIZED VIEW mv_order_analysis AS
SELECT
    o.O_W_ID as warehouse_id,
    o.O_D_ID as district_id,
    o.O_ID as order_id,
    o.O_C_ID as customer_id,
    c.C_FIRST as customer_first_name,
    c.C_LAST as customer_last_name,
    c.C_CREDIT as customer_credit,
    o.O_ENTRY_D as order_date,
    o.O_CARRIER_ID as carrier_id,
    o.O_OL_CNT as order_line_count,
    COUNT(ol.OL_NUMBER) as actual_line_count,
```

```
    SUM(ol.OL_QUANTITY) as total_items,
    SUM(ol.OL_AMOUNT) as order_total,
    AVG(ol.OL_QUANTITY) as avg_quantity_per_line,
    MAX(ol.OL_QUANTITY) as max_quantity,
    BOOL_AND(ol.OL_DELIVERY_D IS NOT NULL) as fully_delivered,
    COUNT(ol.OL_NUMBER) FILTER (WHERE ol.OL_DELIVERY_D IS NULL) as
     ↪  pending_deliveries
FROM ORDERS o
JOIN CUSTOMER c ON o.O_W_ID = c.C_W_ID
                 AND o.O_D_ID = c.C_D_ID
                 AND o.O_C_ID = c.C_ID
JOIN ORDER_LINE ol ON o.O_W_ID = ol.OL_W_ID
                    AND o.O_D_ID = ol.OL_D_ID
                    AND o.O_ID = ol.OL_O_ID
GROUP BY
    o.O_W_ID, o.O_D_ID, o.O_ID, o.O_C_ID,
    c.C_FIRST, c.C_LAST, c.C_CREDIT,
    o.O_ENTRY_D, o.O_CARRIER_ID, o.O_OL_CNT;

-- Create composite indexes for performance
CREATE INDEX idx_mv_order_analysis_warehouse ON
 ↪  mv_order_analysis(warehouse_id, order_date);
CREATE INDEX idx_mv_order_analysis_customer ON
 ↪  mv_order_analysis(customer_id, warehouse_id, district_id);
CREATE INDEX idx_mv_order_analysis_date ON
 ↪  mv_order_analysis(order_date);
CREATE INDEX idx_mv_order_analysis_carrier ON
 ↪  mv_order_analysis(carrier_id) WHERE carrier_id IS NOT NULL;
```

Stock Level Materialized View

Create a materialized view for inventory analysis:

```
CREATE MATERIALIZED VIEW mv_stock_level AS
SELECT
    s.S_W_ID as warehouse_id,
    s.S_I_ID as item_id,
    i.I_NAME as item_name,
    i.I_PRICE as item_price,
    s.S_QUANTITY as current_quantity,
    s.S_YTD as year_to_date_sold,
```

```
    s.S_ORDER_CNT as order_count,
    s.S_REMOTE_CNT as remote_order_count,
    CASE
        WHEN s.S_QUANTITY < 10 THEN 'LOW'
        WHEN s.S_QUANTITY < 50 THEN 'MEDIUM'
        ELSE 'HIGH'
    END as stock_level,
    w.W_NAME as warehouse_name,
    w.W_CITY as warehouse_city
FROM STOCK s
JOIN ITEM i ON s.S_I_ID = i.I_ID
JOIN WAREHOUSE w ON s.S_W_ID = w.W_ID
WHERE s.S_QUANTITY >= 0;

-- Create indexes for inventory queries
CREATE INDEX idx_mv_stock_warehouse ON mv_stock_level(warehouse_id);
CREATE INDEX idx_mv_stock_level ON mv_stock_level(stock_level);
CREATE INDEX idx_mv_stock_quantity ON
 ↪  mv_stock_level(current_quantity);
CREATE INDEX idx_mv_stock_item ON mv_stock_level(item_id);
```

4.12.4 Part 3: Materialized View Management and Refresh

Manual Refresh Operations

Practice different refresh strategies on TPC-C materialized views:

```
-- Refresh a single materialized view
REFRESH MATERIALIZED VIEW mv_customer_orders;

-- Refresh with concurrent access (allows reads during refresh)
REFRESH MATERIALIZED VIEW CONCURRENTLY mv_sales_summary;

-- Refresh all materialized views
REFRESH MATERIALIZED VIEW mv_customer_orders, mv_sales_summary,
    mv_order_analysis, mv_stock_level;

-- Check materialized view metadata
SELECT
    schemaname,
    matviewname,
    matviewowner,
    ispopulated,
    definition
FROM pg_matviews
WHERE matviewname LIKE 'mv_%'
ORDER BY matviewname;

-- Check storage usage
SELECT
    schemaname,
    matviewname,
    pg_size_pretty(pg_total_relation_size(schemaname||'.'||matviewname)) as total_size,
    pg_size_pretty(pg_relation_size(schemaname||'.'||matviewname)) as table_size,
    pg_size_pretty(pg_total_relation_size(schemaname||'.'||matviewname) -
                   pg_relation_size(schemaname||'.'||matviewname)) as index_size
FROM pg_matviews
WHERE schemaname = 'public'
ORDER BY pg_total_relation_size(schemaname||'.'||matviewname) DESC;
```

Automated Refresh with TPC-C Specific Functions

Create functions to automate refresh processes for TPC-C workload:

```
-- Create function to refresh TPC-C materialized views
CREATE OR REPLACE FUNCTION refresh_tpcc_materialized_views()
RETURNS VOID AS \$\$
DECLARE
    refresh_start TIMESTAMP;
BEGIN
    refresh_start := clock_timestamp();
    RAISE NOTICE 'Starting TPC-C materialized view refresh at %',
     ↪  refresh_start;

    -- Refresh in order of dependency and importance
    RAISE NOTICE 'Refreshing mv_customer_orders...';
    REFRESH MATERIALIZED VIEW CONCURRENTLY mv_customer_orders;

    RAISE NOTICE 'Refreshing mv_sales_summary...';
    REFRESH MATERIALIZED VIEW CONCURRENTLY mv_sales_summary;

    RAISE NOTICE 'Refreshing mv_order_analysis...';
    REFRESH MATERIALIZED VIEW CONCURRENTLY mv_order_analysis;

    RAISE NOTICE 'Refreshing mv_stock_level...';
    REFRESH MATERIALIZED VIEW CONCURRENTLY mv_stock_level;

    RAISE NOTICE 'TPC-C materialized view refresh completed in %',
        clock_timestamp() - refresh_start;
END;
\$\$ LANGUAGE plpgsql;

-- Create log table for TPC-C refresh operations
CREATE TABLE tpcc_mv_refresh_log (
    log_id SERIAL PRIMARY KEY,
    refresh_start TIMESTAMP,
    refresh_end TIMESTAMP,
    duration INTERVAL,
    views_refreshed TEXT[],
    success BOOLEAN DEFAULT TRUE,
    error_message TEXT,
    log_timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

```

```
-- Enhanced refresh function with logging
CREATE OR REPLACE FUNCTION refresh_tpcc_mv_with_logging()
RETURNS INTEGER AS \$\$
DECLARE
    log_id INTEGER;
    start_time TIMESTAMP;
    end_time TIMESTAMP;
BEGIN
    start_time := clock_timestamp();

    INSERT INTO tpcc_mv_refresh_log (refresh_start, views_refreshed)
    VALUES (start_time, ARRAY['mv_customer_orders',
     ↪  'mv_sales_summary', 'mv_order_analysis', 'mv_stock_level'])
    RETURNING log_id INTO log_id;

    BEGIN
        PERFORM refresh_tpcc_materialized_views();

        end_time := clock_timestamp();

        UPDATE tpcc_mv_refresh_log
        SET refresh_end = end_time,
            duration = end_time - start_time,
            success = TRUE
        WHERE log_id = log_id;

        RETURN log_id;
    EXCEPTION
        WHEN OTHERS THEN
            UPDATE tpcc_mv_refresh_log
            SET refresh_end = clock_timestamp(),
                duration = clock_timestamp() - start_time,
                success = FALSE,
                error_message = SQLERRM
            WHERE log_id = log_id;
            RAISE;
    END;
END;
\$\$ LANGUAGE plpgsql;
```

4.12.5 Part 4: Query Performance Comparison with TPC-C Data

TPC-C Query Performance Testing

Compare performance of typical TPC-C analytical queries:

```
-- Enable timing for performance measurement
\timing on

-- Query 1: Stock Level Query (TPC-C Standard) - Using base tables
EXPLAIN (ANALYZE, BUFFERS)
SELECT COUNT(DISTINCT s.S_I_ID)
FROM ORDER_LINE ol
JOIN STOCK s ON ol.OL_SUPPLY_W_ID = s.S_W_ID AND ol.OL_I_ID =
 ↪  s.S_I_ID
JOIN ORDERS o ON ol.OL_W_ID = o.O_W_ID AND ol.OL_D_ID = o.O_D_ID AND
 ↪  ol.OL_O_ID = o.O_ID
WHERE ol.OL_W_ID = 1
  AND ol.OL_D_ID = 1
  AND o.O_ID < 2100
  AND o.O_ID >= 2000
  AND s.S_QUANTITY < 15;

-- Same query using materialized view
EXPLAIN (ANALYZE, BUFFERS)
SELECT COUNT(DISTINCT item_id)
FROM mv_stock_level sl
JOIN mv_order_analysis oa ON sl.warehouse_id = oa.warehouse_id
WHERE sl.warehouse_id = 1
  AND oa.district_id = 1
  AND oa.order_id < 2100
  AND oa.order_id >= 2000
  AND sl.current_quantity < 15;

-- Query 2: Customer Balance Query - Using base tables
EXPLAIN (ANALYZE, BUFFERS)
SELECT c.C_FIRST, c.C_LAST, c.C_BALANCE, w.W_NAME, d.D_NAME
FROM CUSTOMER c
JOIN WAREHOUSE w ON c.C_W_ID = w.W_ID
JOIN DISTRICT d ON c.C_W_ID = d.D_W_ID AND c.C_D_ID = d.D_ID
WHERE c.C_W_ID = 1
  AND c.C_D_ID = 1
  AND c.C_BALANCE < -1000
ORDER BY c.C_BALANCE ASC
```

```
LIMIT 10;

-- Same query using materialized view
EXPLAIN (ANALYZE, BUFFERS)
SELECT co.first_name, co.last_name, co.balance, w.W_NAME, d.D_NAME
FROM mv_customer_orders co
JOIN WAREHOUSE w ON co.warehouse_id = w.W_ID
JOIN DISTRICT d ON co.warehouse_id = d.D_W_ID AND co.district_id =
 ↪  d.D_ID
WHERE co.warehouse_id = 1
  AND co.district_id = 1
  AND co.balance < -1000
ORDER BY co.balance ASC
LIMIT 10;

-- Query 3: Sales Summary by Warehouse - Using base tables
EXPLAIN (ANALYZE, BUFFERS)
SELECT
    o.O_W_ID,
    DATE_TRUNC('month', o.O_ENTRY_D) as month,
    COUNT(DISTINCT o.O_ID) as order_count,
    SUM(ol.OL_AMOUNT) as total_sales
FROM ORDERS o
JOIN ORDER_LINE ol ON o.O_W_ID = ol.OL_W_ID AND o.O_D_ID =
 ↪  ol.OL_D_ID AND o.O_ID = ol.OL_O_ID
WHERE o.O_ENTRY_D >= CURRENT_DATE - INTERVAL '6 months'
GROUP BY o.O_W_ID, DATE_TRUNC('month', o.O_ENTRY_D)
ORDER BY o.O_W_ID, month;

-- Same query using materialized view
EXPLAIN (ANALYZE, BUFFERS)
SELECT
    warehouse_id,
    sales_month,
    order_count,
    total_sales
FROM mv_sales_summary
WHERE sales_month >= DATE_TRUNC('month', CURRENT_DATE - INTERVAL '6
 ↪  months')
ORDER BY warehouse_id, sales_month;
```

Advanced TPC-C Performance Analysis

Create specialized functions for TPC-C performance analysis:

```
-- Function to compare TPC-C query performance
CREATE OR REPLACE FUNCTION compare_tpcc_query_performance(
    base_query TEXT,
    mv_query TEXT,
    description TEXT DEFAULT 'TPC-C Query',
    iterations INTEGER DEFAULT 3
)
RETURNS TABLE(
    test_description TEXT,
    iteration INTEGER,
    base_execution_time_ms DOUBLE PRECISION,
    mv_execution_time_ms DOUBLE PRECISION,
    improvement_percent DOUBLE PRECISION,
    improvement_factor DOUBLE PRECISION
) AS \$\$
DECLARE
    i INTEGER;
    start_time TIMESTAMP;
    end_time TIMESTAMP;
    base_duration INTERVAL;
    mv_duration INTERVAL;
BEGIN
    FOR i IN 1..iterations LOOP
        -- Clear cache between runs (requires superuser or
        ↪  appropriate permissions)
        -- DROP TABLE IF EXISTS temp_clear_cache;
        -- CREATE TEMP TABLE temp_clear_cache AS SELECT 1;

        -- Test base query
        start_time := clock_timestamp();
        EXECUTE base_query;
        end_time := clock_timestamp();
        base_duration := end_time - start_time;

        -- Clear cache between runs
        -- DROP TABLE IF EXISTS temp_clear_cache2;
        -- CREATE TEMP TABLE temp_clear_cache2 AS SELECT 1;

        -- Test materialized view query
        start_time := clock_timestamp();
```

```
        EXECUTE mv_query;
        end_time := clock_timestamp();
        mv_duration := end_time - start_time;

        -- Return results
        test_description := description;
        iteration := i;
        base_execution_time_ms := EXTRACT(EPOCH FROM base_duration) *
         ↪  1000;
        mv_execution_time_ms := EXTRACT(EPOCH FROM mv_duration) *
         ↪  1000;
        improvement_percent := ((base_execution_time_ms -
         ↪  mv_execution_time_ms) / base_execution_time_ms) * 100;
        improvement_factor := base_execution_time_ms /
         ↪  NULLIF(mv_execution_time_ms, 0);

        RETURN NEXT;
    END LOOP;
END;
\$\$ LANGUAGE plpgsql;

-- Example usage for TPC-C queries
SELECT * FROM compare_tpcc_query_performance(
    'SELECT COUNT(*) FROM ORDER_LINE ol JOIN STOCK s ON
     ↪  ol.OL_SUPPLY_W_ID = s.S_W_ID AND ol.OL_I_ID = s.S_I_ID WHERE
     ↪  ol.OL_W_ID = 1',
    'SELECT COUNT(*) FROM mv_stock_level WHERE warehouse_id = 1',
    'Stock Level Count Query',
    3
);
```

4.12.6 Part 5: Nested Materialized Views for TPC-C

Creating Nested TPC-C Materialized Views

Build materialized views on top of existing TPC-C materialized views:

```
-- Create a high-level business intelligence view
CREATE MATERIALIZED VIEW mv_tpcc_business_intelligence AS
SELECT
    ss.warehouse_id,
```

```
    ss.district_id,
    ss.sales_month,
    w.W_NAME as warehouse_name,
    d.D_NAME as district_name,
    ss.order_count,
    ss.total_sales,
    ss.avg_order_line_amount,
    COUNT(DISTINCT co.customer_id) as active_customers,
    AVG(co.balance) as avg_customer_balance,
    COUNT(sl.item_id) FILTER (WHERE sl.stock_level = 'LOW') as
    ↪ low_stock_items,
    COUNT(sl.item_id) FILTER (WHERE sl.stock_level = 'HIGH') as
    ↪ high_stock_items
FROM mv_sales_summary ss
JOIN WAREHOUSE w ON ss.warehouse_id = w.W_ID
JOIN DISTRICT d ON ss.warehouse_id = d.D_W_ID AND ss.district_id =
↪ d.D_ID
LEFT JOIN mv_customer_orders co ON ss.warehouse_id = co.warehouse_id
                                AND ss.district_id = co.district_id
                               AND DATE_TRUNC('month', co.order_date)
                                ↪ = ss.sales_month
LEFT JOIN mv_stock_level sl ON ss.warehouse_id = sl.warehouse_id
GROUP BY
    ss.warehouse_id, ss.district_id, ss.sales_month,
    w.W_NAME, d.D_NAME, ss.order_count, ss.total_sales,
    ↪ ss.avg_order_line_amount;

-- Create indexes for the business intelligence view
CREATE INDEX idx_mv_tpcc_bi_warehouse ON
↪ mv_tpcc_business_intelligence(warehouse_id, sales_month);
CREATE INDEX idx_mv_tpcc_bi_month ON
↪ mv_tpcc_business_intelligence(sales_month);
CREATE INDEX idx_mv_tpcc_bi_composite ON
↪ mv_tpcc_business_intelligence(warehouse_id, district_id,
↪ sales_month);

-- Create customer segmentation view
CREATE MATERIALIZED VIEW mv_customer_segments AS
SELECT
    warehouse_id,
    district_id,
    customer_credit as credit_type,
```

```
    COUNT(DISTINCT customer_id) as customer_count,
    AVG(balance) as avg_balance,
    SUM(balance) as total_balance,
    COUNT(DISTINCT order_id) as total_orders,
    SUM(order_total) as total_revenue,
    MAX(order_date) as last_order_date
FROM mv_order_analysis
GROUP BY warehouse_id, district_id, customer_credit;

-- Refresh nested materialized views
REFRESH MATERIALIZED VIEW mv_tpcc_business_intelligence;
REFRESH MATERIALIZED VIEW mv_customer_segments;
```

Managing TPC-C Dependencies

```
-- Function to refresh TPC-C materialized views in dependency order
CREATE OR REPLACE FUNCTION refresh_tpcc_views_cascading()
RETURNS VOID AS \$\$
BEGIN
    RAISE NOTICE 'Starting cascading refresh of TPC-C materialized
     ↪  views...';

    -- Step 1: Refresh base materialized views
    RAISE NOTICE 'Refreshing base materialized views...';
    REFRESH MATERIALIZED VIEW CONCURRENTLY mv_customer_orders;
    REFRESH MATERIALIZED VIEW CONCURRENTLY mv_sales_summary;
    REFRESH MATERIALIZED VIEW CONCURRENTLY mv_order_analysis;
    REFRESH MATERIALIZED VIEW CONCURRENTLY mv_stock_level;

    -- Step 2: Refresh nested materialized views
    RAISE NOTICE 'Refreshing nested materialized views...';
    REFRESH MATERIALIZED VIEW CONCURRENTLY
     ↪  mv_tpcc_business_intelligence;
    REFRESH MATERIALIZED VIEW CONCURRENTLY mv_customer_segments;

    RAISE NOTICE 'Cascading refresh completed successfully.';
END;
\$\$ LANGUAGE plpgsql;

-- View dependency analysis
SELECT
```

```
    mv.matviewname as view_name,
    pg_size_pretty(pg_total_relation_size('public.' ||
     ↪ mv.matviewname)) as size,
    (SELECT COUNT(*) FROM pg_depend
     WHERE objid = (SELECT oid FROM pg_class WHERE relname =
      ↪ mv.matviewname)
     AND refobjid IN (SELECT oid FROM pg_class WHERE relname LIKE
      ↪ 'mv_%')) as dependency_count
FROM pg_matviews mv
WHERE mv.schemaname = 'public'
ORDER BY pg_total_relation_size('public.' || mv.matviewname) DESC;
```

4.12.7 Part 6: TPC-C Specific Maintenance and Optimization

TPC-C Monitoring and Maintenance

```
-- Comprehensive TPC-C materialized view monitoring
SELECT
    m.matviewname,
    pg_size_pretty(pg_relation_size('public.' || m.matviewname)) as
     ↪ table_size,
    pg_size_pretty(pg_total_relation_size('public.' ||
     ↪ m.matviewname) -
                   pg_relation_size('public.' || m.matviewname)) as
                    ↪ index_size,
    pg_size_pretty(pg_total_relation_size('public.' ||
     ↪ m.matviewname)) as total_size,
    m.ispopulated,
    (SELECT COUNT(*) FROM pg_indexes WHERE tablename = m.matviewname)
     ↪ as index_count,
    s.n_live_tup as row_count,
    s.last_analyze,
    s.last_autoanalyze
FROM pg_matviews m
LEFT JOIN pg_stat_all_tables s ON m.matviewname = s.relname
WHERE m.schemaname = 'public'
ORDER BY pg_total_relation_size('public.' || m.matviewname) DESC;

-- Analyze TPC-C materialized views for query optimization
ANALYZE mv_customer_orders;
ANALYZE mv_sales_summary;
```

```
ANALYZE mv_order_analysis;
ANALYZE mv_stock_level;
ANALYZE mv_tpcc_business_intelligence;
ANALYZE mv_customer_segments;

-- Check refresh history and patterns
SELECT
    view_name,
    COUNT(*) as refresh_count,
    AVG(EXTRACT(EPOCH FROM duration)) as avg_duration_seconds,
    MAX(refresh_start) as last_refresh,
    NOW() - MAX(refresh_start) as time_since_last_refresh
FROM tpcc_mv_refresh_log
WHERE success = true
GROUP BY view_name
ORDER BY last_refresh DESC;
```

TPC-C Cleanup and Best Practices

```
-- Safely drop materialized views if needed (in reverse dependency
 ↪  order)
/*
DROP MATERIALIZED VIEW IF EXISTS mv_customer_segments;
DROP MATERIALIZED VIEW IF EXISTS mv_tpcc_business_intelligence;
DROP MATERIALIZED VIEW IF EXISTS mv_stock_level;
DROP MATERIALIZED VIEW IF EXISTS mv_order_analysis;
DROP MATERIALIZED VIEW IF EXISTS mv_sales_summary;
DROP MATERIALIZED VIEW IF EXISTS mv_customer_orders;
*/

-- Backup TPC-C materialized view definitions
SELECT
    'CREATE MATERIALIZED VIEW ' || matviewname || ' AS ' ||
    definition || E';\n\n-- Indexes:\n' ||
    (SELECT string_agg('CREATE INDEX ' || indexname || ' ON ' ||
     ↪  tablename ||
                       ' USING ' || indexdef, E';\n')
     FROM pg_indexes
     WHERE tablename = matviewname) || ';' as backup_script
FROM pg_matviews
WHERE schemaname = 'public' AND matviewname LIKE 'mv_%';
```

```
21
22 -- Document TPC-C specific refresh strategies
23 COMMENT ON MATERIALIZED VIEW mv_customer_orders IS
24 'TPC-C Customer Orders - Pre-joins CUSTOMER and ORDERS tables.
   ↪ Refresh: Hourly (high frequency due to new orders)';
25 COMMENT ON MATERIALIZED VIEW mv_sales_summary IS
26 'TPC-C Sales Summary - Aggregates ORDER_LINE data. Refresh: Daily
   ↪ (batch processing)';
27 COMMENT ON MATERIALIZED VIEW mv_order_analysis IS
28 'TPC-C Order Analysis - Comprehensive order analytics. Refresh:
   ↪ Every 4 hours';
29 COMMENT ON MATERIALIZED VIEW mv_stock_level IS
30 'TPC-C Stock Level - Inventory monitoring. Refresh: Real-time
   ↪ (critical for stock checks)';
31 COMMENT ON MATERIALIZED VIEW mv_tpcc_business_intelligence IS
32 'TPC-C Business Intelligence - High-level aggregated view. Refresh:
   ↪ Weekly (summary data)';
```

4.12.8 Exercises

Exercise 1: Create TPC-C Payment Analysis View

Create a materialized view that analyzes payment patterns by combining HISTORY and CUSTOMER tables.

Exercise 2: Implement TPC-C Specific Refresh Strategy

Design a refresh strategy that considers TPC-C workload patterns (peak order times, stock updates).

Exercise 3: TPC-C Query Performance Benchmarking

Compare the performance of 3 standard TPC-C queries with and without materialized views.

Exercise 4: Storage Optimization for TPC-C Views

Analyze storage usage and propose partitioning strategies for the largest TPC-C materialized views.

Exercise 5: TPC-C Business Intelligence Dashboard

Create a nested materialized view that provides executive-level summary data for all warehouses.

5 Query Optimization in Database Systems

5.1 Introduction to Query Optimization

5.1.1 The Multifaceted Nature of Problem Solving

The process of learning to play a piano serves as an excellent analogy for understanding query optimization in database systems. Just as there are numerous approaches to mastering a musical piecefrom traditional one-on-one lessons to self-instruction methods using books, videos, or sheet music-database queries can also be executed through multiple paths to achieve the same correct result.

Each approach to piano playing may produce variations in quality and efficiency, yet all can eventually lead to the desired outcome. Similarly, in database systems, various execution plans can process the same query correctly, but with significantly different performance characteristics. The essence of query optimization lies in analyzing these alternative paths and selecting the one that delivers results with optimal efficiency.

Sir Arthur Conan Doyle's wisdom"It is a capital mistake to theorize before one has data"resonates profoundly in this context. Query optimizers must gather and analyze statistical information about the data before determining the most efficient execution strategy, rather than forcing predetermined theories onto the data.

5.1.2 The Importance of Query Optimization

In modern database systems, query optimization represents one of the most critical components affecting overall system performance. While early database systems relied on programmers to specify exactly how queries should be executed, contemporary systems employ sophisticated optimizers that automatically determine efficient execution strategies. This automation relieves application developers from the burden of manual performance

tuning while ensuring that queries execute efficiently regardless of changes in data distribution or volume.

The significance of query optimization becomes particularly evident in:

- Online Transaction Processing (OLTP) systems where rapid response times are crucial for user satisfaction
- Data Warehousing and Business Intelligence applications involving complex queries over large datasets
- Web applications serving numerous concurrent users with strict performance requirements
- Scientific databases processing massive volumes of experimental or observational data

This course explores the fundamental principles, techniques, and practical considerations that underpin effective query optimization in relational database management systems.

5.2 Query Processing Fundamentals

5.2.1 Architecture of Query Processing

Database systems process SQL queries through a series of well-defined stages that transform high-level declarative statements into executable operations. Understanding this pipeline is essential for comprehending how optimization fits into the broader context of query execution.

5.2.1.1 Three-Stage Query Processing Model

1. Scanning, Parsing, and Decomposition: This initial phase validates the syntactic correctness of SQL queries and generates appropriate error messages when necessary. The output is an intermediate representation of the query, typically in the form of a query tree or initial execution plan. This stage ensures that the query follows proper SQL syntax and identifies all referenced database objects.
2. Query Optimization: This crucial phase encompasses both local and global optimization techniques. Global optimization determines the

order of join operations and the sequencing of selections and projections relative to joins. It also involves transforming nested queries into equivalent flat queries. Local optimization focuses on selecting appropriate index methods for selections and joins. Both optimization types rely on cost estimates based on database statistics and schema information.

3. Query Code Generation and Execution: The final stage employs classical compiler techniques to generate executable code from the optimized query plan. This code interfaces with the storage engine to retrieve and process data, ultimately delivering the query results to the user or application.

5.2.2 Modern Optimization Features

Contemporary database management systems incorporate numerous advanced features that enhance query optimization capabilities beyond the basic three-stage model.

5.2.2.1 Query Transformation and Rewriting

Modern database systems (such as Oracle, DB2, and SQL Server) automatically transform queries into more efficient forms before the optimization phase begins. Common query rewriting techniques include:

- Subquery to Join Transformation: Converting correlated subqueries into equivalent join operations, which often execute more efficiently
- Group By Pushdown: Moving group by operations below joins when possible to reduce the volume of data being joined
- Join Elimination: Removing unnecessary joins on foreign keys when the joined tables don't contribute to the final result
- Outer to Inner Join Conversion: Transforming outer joins to inner joins when they produce equivalent results, taking advantage of more efficient inner join algorithms
- View Merging: Replacing view references with their actual definitions to enable more comprehensive optimization

- Materialized View Rewrite: Substituting parts of queries with precomputed materialized views when available

These transformations are particularly beneficial in data warehouse environments using star schemas, where joins between fact and dimension tables can be optimized using specialized indexing techniques.

5.2.2.2 Query Execution Plan Visualization

Understanding how a database plans to execute a query is essential for performance tuning. Modern database systems provide tools like `EXPLAIN` or `EXPLAIN PLAN` (with graphical versions such as Visual Explain in DB2) that reveal:

- The sequence of operations in the execution plan
- The order in which tables are accessed
- Index usage information
- Join methods employed
- Estimated costs for each operation

These tools enable database administrators and developers to identify potential performance bottlenecks and verify that the optimizer is making appropriate choices.

5.2.2.3 Histograms for Selectivity Estimation

Accurate selectivity estimation is crucial for effective query optimization. Histograms provide detailed statistical information about the distribution of attribute values within tables. Unlike simple average-based estimates, histograms capture:

- Value distributions across different ranges
- Skewed data patterns
- Outlier concentrations
- Actual frequency counts for specific value ranges

This detailed distribution information enables optimizers to make more accurate cardinality estimates, leading to better plan selection.

5.2.2.4 Query Execution Plan Hints

Most major database systems support plan hintsdirectives embedded in SQL queries that influence the optimizer's plan selection. While hints should be used judiciously, they serve important purposes:

- Overriding suboptimal optimizer choices in complex scenarios
- Enabling performance comparisons between different execution strategies
- Locking in known efficient plans for critical queries
- Working around optimizer limitations in special cases

Common types of hints include those specifying join orders, join methods, index usage, and parallel execution degrees.

5.2.2.5 Optimization Depth Configuration

Database optimizers employ various search strategies to explore the space of possible execution plans, ranging from simple greedy algorithms to comprehensive dynamic programming approaches. Many systems allow administrators to configure optimization depth through parameters that control:

- The thoroughness of the search for optimal plans
- Timeouts for optimization processes
- Memory limits for plan enumeration
- Trade-offs between optimization time and execution time

5.3 Query Cost Evaluation: A Comprehensive Example

5.3.1 Foundational Concepts

To illustrate the principles of query optimization, we examine a practical scenario involving a simple three-table database containing part, supplier, and shipment information. This example demonstrates how different execution strategies for the same query can yield dramatically different performance characteristics.

5.3.1.1 Database Schema and Characteristics

Consider the following database tables:

Part (P)	Supplier (S)	Shipment (SH)
pnum: Primary Key pname: Part Name wt: Weight	snum: Primary Key sname: Supplier Name city: Supplier City status: Status Code	snum: Foreign Key to Supplier pnum: Foreign Key to Part qty: Quantity shipdate: Shipping Date

Table characteristics:

- supplier: 200 records, 37 bytes per record
- part: 100 records, 23 bytes per record
- shipment: 100,000 records, 26 bytes per record
- Block size: 15,000 bytes

5.3.1.2 Example Query

The query we'll optimize is: "What are the names of parts supplied by suppliers in New York City?"

Translated to SQL:

```
SELECT p.pname
FROM P, SH, S
WHERE P.pnum = SH.pnum
  AND SH.snum = S.snum
  AND S.city = 'NY';
```

5.3.2 Join Order Analysis

Given the two join conditions in the query, there are 3! = 6 possible join orders:

1. S join SH join P
2. SH join S join P

3. P join SH join S
4. SH join P join S
5. S × P join SH (Cartesian product)
6. P × S join SH (Cartesian product)

Due to the commutativity of joins (Rule 1), orders 1 and 2 are equivalent, as are orders 3 and 4. Orders 5 and 6 involve Cartesian products and should generally be avoided as they generate prohibitively large intermediate results.

This leaves us with two reasonable join order families to consider: Orders 1 and 3. For each, we can evaluate strategies that execute joins before selections versus selections before joins.

5.3.3 Cost Estimation Methodology

We use Sequential Block Accesses (SBA) as our primary cost metric, based on the linear relationship between SBA and I/O time. This approach provides a practical foundation for comparing alternative execution plans.

Key formulas used in cost estimation:

- Blocking Factor: $BF = \lfloor \text{Block Size/Row Size} \rfloor$
- Blocks to Scan Table: $\lceil \text{Number of Records}/BF \rceil$
- Sort Cost: $2 \times nb \times \log_M nb$ where nb is number of blocks and M is merge factor

5.3.4 Option 1A: Joins First, Selections Last

This approach represents a naive strategy that executes all joins before applying selective operations.

5.3.4.1 Execution Plan

1. Join Supplier and Shipment over snum → TEMPA (100,000 records)
2. Join TEMPA and Part over pnum → TEMPB (100,000 records)
3. Select TEMPB where city = 'NY' → TEMPC (10,000 records)
4. Project TEMPC over pname → RESULT (10,000 records)

5.3.4.2 Cost Calculation

- Step 1: Join S and SH = 1 (read S) + 174 (read SH) + 388 (write TEMPA) = 563 SBA
- Step 2: Join TEMPA and P = 1 (read P) + 4,214 (sort TEMPA) + 388 (read TEMPA) + 488 (write TEMPB) = 5,091 SBA
- Step 3: Select city = 'NY' = 488 (read TEMPB) + 49 (write TEMPC) = 537 SBA
- Step 4: Project pname = 49 (read TEMPC) = 49 SBA
- Total Cost: 6,240 SBA

This approach demonstrates the inefficiency of processing large intermediate results, particularly evident in the expensive sort operation required for the merge join.

5.3.5 Option 1B: Selections First, Joins Last

This optimized strategy applies selective operations early to reduce the volume of data processed in join operations.

5.3.5.1 Execution Plan

1. Select S where city = 'NY' → TEMP1 (20 records)
2. Project SH over snum, pnum → TEMP2 (100,000 records)
3. Project P over pnum, pname → TEMP3 (100 records)
4. Semi-join TEMP1 and TEMP2 over snum → TEMP4 (10,000 records)
5. Semi-join TEMP4 and TEMP3 over pnum → TEMP5 (10,000 records)
6. Project TEMP5 over pname → RESULT (10,000 records)

5.3.5.2 Cost Calculation

- Step 1: Select city = 'NY' = 1 (read S) + 1 (write TEMP1) = 2 SBA
- Step 2: Project SH = 174 (read SH) + 87 (write TEMP2) = 261 SBA
- Step 3: Project P = 1 (read P) + 1 (write TEMP3) = 2 SBA
- Step 4: Semi-join TEMP1 and TEMP2 = 1 (read TEMP1) + 87 (read TEMP2) + 9 (write TEMP4) = 97 SBA
- Step 5: Semi-join TEMP4 and TEMP3 = 9 (read TEMP4) + 1 (read TEMP3) + 13 (write TEMP5) = 23 SBA
- Step 6: Project pname = 13 (read TEMP5) = 13 SBA
- Total Cost: 398 SBA

5.3.6 Performance Comparison

The dramatic difference in cost between the two approaches (6,240 SBA vs. 398 SBA) highlights the critical importance of applying selective operations before joins. This represents a 94% reduction in I/O cost through intelligent plan selection.

5.4 Query Execution Plan Development

5.4.1 Query Execution Plan Representation

A Query Execution Plan (QEP) is a tree-structured representation of the sequence of operations required to process a query. Each node in the tree corresponds to a database operation (selection, projection, join), with edges representing data flow between operations.

QEPs can be represented using either top-down or bottom-up approaches. The bottom-up approach, as illustrated in our examples, starts with base table access and builds toward the final result, making it particularly intuitive for understanding how intermediate results are constructed.

5.4.2 Transformation Rules for Query Execution Plans

The flexibility to reorganize operations without changing the semantic meaning of queries enables optimizers to explore alternative execution strategies. The following transformation rules form the mathematical foundation for query optimization:

1. Commutativity of Joins: $R1 \bowtie R2 = R2 \bowtie R1$
2. Associativity of Joins: $R1 \bowtie (R2 \bowtie R3) = (R1 \bowtie R2) \bowtie R3$
3. Order Independence of Selections: The sequence of selection operations on the same table does not affect the result
4. Commutation of Selections and Projections: Selections and projections on the same table can be reordered, provided the projection doesn't eliminate attributes needed for selection
5. Commutation of Selections and Joins: Selections on a table before a join produce equivalent results to selections after a join
6. Commutation of Projections and Joins: Projections and joins can be reordered if the projection doesn't remove join attributes
7. Commutation of Selections/Projections with Unions: These operations can be pushed down through union operations

These rules enable the systematic restructuring of query execution plans to achieve more efficient forms while guaranteeing equivalent results.

5.4.3 Query Execution Plan Restructuring Algorithm

A practical heuristic algorithm for optimizing query execution plans incorporates the following steps:

1. Decompose Complex Selections: Separate selection operations with multiple AND conditions into sequences of individual selections
2. Push Selections Down: Move selection operations as early as possible in the execution sequence
3. Consolidate Selections: Group multiple selection operations on the same table

4. Push Projections Down: Move projection operations to eliminate unnecessary attributes early in processing
5. Consolidate Projections: Combine multiple projection operations on the same table and remove redundant projections

This algorithm systematically applies the transformation rules to create plans that minimize intermediate result sizes, particularly before expensive join operations.

5.5 Selectivity Factors and Cost Estimation

5.5.1 Foundations of Selectivity Estimation

Selectivity (S) quantifies the proportion of records in a table that satisfy a given condition, with values ranging from 0 to 1. Accurate selectivity estimation is crucial for predicting intermediate result sizes and comparing alternative execution plans.

Key statistical measures for selectivity estimation:

- $\text{card}(R)$: Cardinality (number of rows) in table R
- $\text{card}_A(R)$: Number of distinct values of attribute A in table R
- $\max_A(R)$: Maximum value of attribute A in table R
- $\min_A(R)$: Minimum value of attribute A in table R

5.5.2 Selectivity Estimation Formulas

5.5.2.1 Equality Selection

For an attribute A having a specific value a:

$$S(A = a) = \frac{1}{\text{card}_A(R)}$$

This formula assumes uniform data distribution, which provides reasonable estimates for many practical scenarios despite potential inaccuracies with skewed data.

5.5.2.2 Range Selections

For inequality conditions:

$$S(A > a) = \frac{\max_A(R) - a}{\max_A(R) - \min_A(R)}$$

$$S(A < a) = \frac{a - \min_A(R)}{\max_A(R) - \min_A(R)}$$

These formulas also rely on the uniform distribution assumption and work best when data values are evenly distributed across the range.

5.5.2.3 Compound Selections

For conjunctions (AND conditions):

$$S(P \wedge Q) = S(P) \times S(Q)$$

For disjunctions (OR conditions):

$$S(P \vee Q) = S(P) + S(Q) - S(P) \times S(Q)$$

These formulas assume predicate independence, which generally holds for unrelated conditions but may introduce errors for correlated predicates.

5.5.3 Histograms for Enhanced Estimation

Histograms address the limitations of uniform distribution assumptions by capturing actual data distribution patterns. Modern database systems maintain histograms that:

- Divide attribute value ranges into buckets
- Count the number of rows in each bucket
- Track frequent values and their exact frequencies
- Identify outliers and skewed distributions

The superiority of histogram-based estimation becomes evident in scenarios with skewed data distributions, where uniform assumptions can lead to significant estimation errors and consequently poor plan selection.

5.5.4 Join Selectivity Estimation

Estimating join result sizes presents unique challenges, particularly for non-key joins. For the common case of primary key-foreign key joins:

$$\text{card}(R1 \bowtie R2) = S \times \text{card}(R1) \times \text{card}(R2)$$

where S represents the selectivity of the join attribute when used as a primary key.

A useful rule of thumb: A join between a table with a primary key and a table with the corresponding foreign key produces a result with the same number of rows as the table containing the foreign key.

5.5.5 Comprehensive Example: Selectivity in Practice

5.5.5.1 Query and Statistical Information

Consider the query:

```
SELECT supplierName
FROM supplier S, shipment SH
WHERE S.snum = SH.snum
  AND S.city = 'London'
  AND SH.shipdate = '01-JUN-2006';
```

Statistical information:

- $\text{card}(\text{supplier}) = 200$
- $\text{card}_{\text{city}}(\text{supplier}) = 50$
- $\text{card}(\text{shipment}) = 100,000$
- $\text{card}_{\text{shipdate}}(\text{shipment}) = 1,000$

5.5.5.2 Case 1: Join Executed First

- Join result size: $S(\text{snum}) \times \text{card}(\text{supplier}) \times \text{card}(\text{shipment}) = \frac{1}{200} \times 200 \times 100,000 = 100,000$
- Final result: $S(\text{city} = \text{London}) \times S(\text{shipdate} = \text{01-JUN-2006}) \times 100,000 = \frac{1}{50} \times \frac{1}{1000} \times 100,000 = 2$ rows

5.5.5.3 Case 2: Selections Executed First

- Selected suppliers: $S(\text{city} = \text{London}) \times \text{card}(\text{supplier}) = \frac{1}{50} \times 200 = 4$ rows
- Selected shipments: $S(\text{shipdate} = \text{01-JUN-2006}) \times \text{card}(\text{shipment}) = \frac{1}{1000} \times 100,000 = 100$ rows
- Join result: $S(\text{snum}) \times 4 \times 100 = \frac{1}{200} \times 4 \times 100 = 2$ rows

Both approaches yield the same final result (2 rows), but the early-selection strategy processes significantly smaller intermediate tables, resulting in substantially better performance.

5.6 Advanced Optimization Considerations

5.6.1 Physical Design Impact on Optimization

The effectiveness of query optimization is intimately connected to physical database design decisions:

- Index Selection: Appropriate indexes can dramatically reduce the cost of selection and join operations, but introduce overhead for update operations
- Partitioning: Horizontal and vertical partitioning strategies can improve query performance by reducing the amount of data that needs to be scanned
- Materialized Views: Precomputed query results can eliminate expensive join and aggregation operations at query time
- Clustering: Physically organizing data based on query patterns can minimize I/O operations

Optimizers must consider these physical design elements when generating and comparing alternative execution plans.

5.6.2 Limitations and Challenges in Query Optimization

Despite sophisticated optimization techniques, several challenges persist:

- Cost Model Accuracy: Cost estimates are only as good as the underlying statistical information and the accuracy of the cost model itself
- Search Space Limitations: The exponential growth of possible plans with query complexity forces optimizers to use heuristic approaches rather than exhaustive search
- Correlation Estimation: Estimating selectivities for correlated predicates remains challenging with standard statistical approaches
- Runtime Dynamics: Static optimization cannot account for runtime factors like system load, memory availability, or concurrent resource contention
- Complex Data Types: Optimization techniques for traditional data types don't always extend well to complex types like spatial, temporal, or multimedia data

5.6.3 Emerging Trends in Query Optimization

Recent developments in query optimization include:

- Adaptive Query Processing: Techniques that adjust execution strategies based on runtime feedback
- Machine Learning Approaches: Using historical query performance data to improve future optimization decisions
- Multi-Objective Optimization: Balancing competing goals like execution time, resource consumption, and result freshness
- Cloud-Native Optimization: Techniques tailored for distributed database architectures and elastic resource environments
- Approximate Query Processing: Delivering timely approximate results for interactive analytics on massive datasets

5.7 Conclusion

Query optimization represents a cornerstone of modern database system performance. Through the systematic application of transformation rules, cost-based evaluation, and statistical analysis, optimizers can identify execution strategies that deliver query results orders of magnitude faster than naive approaches.

The fundamental principle that emerges consistently throughout optimization theory and practice is the importance of reducing data volume as early as possible in query processingparticularly before expensive join operations. This principle, combined with accurate cost estimation and intelligent plan enumeration, enables database systems to process complex queries efficiently despite the combinatorial explosion of possible execution strategies.

As data volumes continue to grow and application requirements evolve, query optimization remains an active area of research and development. The techniques covered in this course provide a solid foundation for understanding both current optimization practices and future advancements in this critical field of database technology.

Exercises

1. Using the three-table database example from Section 3, calculate the I/O cost for executing the query using join order 3 (P join SH join S) with selections before joins.

2. Design a set of histograms for the shipment table that would help the optimizer make better decisions for queries filtering on shipdate, and explain how these histograms would improve selectivity estimation.

3. Prove the correctness of the transformation rule for commuting selections and joins using relational algebra.

4. Compare the advantages and disadvantages of dynamic programming versus greedy search strategies for query optimization in terms of optimization time and plan quality.

5. Design an experiment to determine when query hints should be used instead of relying on the optimizer's built-in decision processes.

6 Transactions

6.1 Introduction to Transactions

A database transaction is a sequence of operations performed as a single logical unit of work in a database. It ensures data integrity and consistency by adhering to the ACID properties: Atomicity, Consistency, Isolation, and Durability. Transactions begin with a BEGIN command, end with COMMIT (to save changes) or ROLLBACK (to undo changes), and can include intermediate savepoints. They are essential in multi-user environments to prevent conflicts and maintain data reliability. Examples include transferring funds in banking systems or managing orders in e-commerce.

6.1.1 Definition and Importance

A database transaction is a sequence of operations performed as a single logical unit of work in a database management system (DBMS). It groups related operations, such as INSERT, UPDATE, or DELETE, into a single entity to ensure data integrity and consistency. Transactions follow the ACID properties: Atomicity ensures all operations succeed or none do; Consistency maintains valid database states; Isolation prevents concurrent transactions from interfering; and Durability ensures committed changes are permanent. Transactions begin with a BEGIN statement and end with COMMIT (to save changes) or ROLLBACK (to undo changes). They are essential for managing complex workflows, handling errors, and maintaining reliability in multi-user environments. By treating multiple operations as a single unit, transactions ensure that databases remain accurate and consistent, even during failures or concurrent access. Importance:

Database transactions are vital for maintaining data integrity, consistency, and reliability in modern applications. They ensure that all operations within a transaction are completed successfully or rolled back, preventing partial updates and data corruption. Transactions enable concurrency control, allowing multiple users to access and modify data simultaneously without conflicts. They also provide error recovery mechanisms, ensuring

databases can revert to a consistent state in case of failures. By enforcing ACID properties, transactions support critical operations like financial transactions, inventory management, and order processing. They improve system stability, enhance user experience, and ensure compliance with regulatory standards. In distributed systems, transactions maintain consistency across multiple databases, enabling scalability and high availability. Overall, transactions are the backbone of reliable and efficient database management, supporting complex workflows and ensuring data accuracy in real-world applications.

6.1.2 ACID Properties

Transactions follow the ACID properties:

- Atomicity: Ensures that all steps in a transaction are completed or none at all.
- Consistency: Guarantees that a transaction brings the database from one valid state to another.
- Isolation: Ensures transactions operate independently, avoiding interference.
- Durability: Once committed, the effects of a transaction persist, even after system failures.

6.1.3 Example of a Simple Transaction

```
1 BEGIN;
2 UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
3 UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
4 COMMIT;
```

In this transaction, money is transferred between accounts. The transaction either completes fully, transferring money from one account to another, or fails, leaving both accounts in their original states.

6.2 Transaction Lifecycle

The lifecycle of a transaction includes several phases: initiation, execution, validation, and completion.

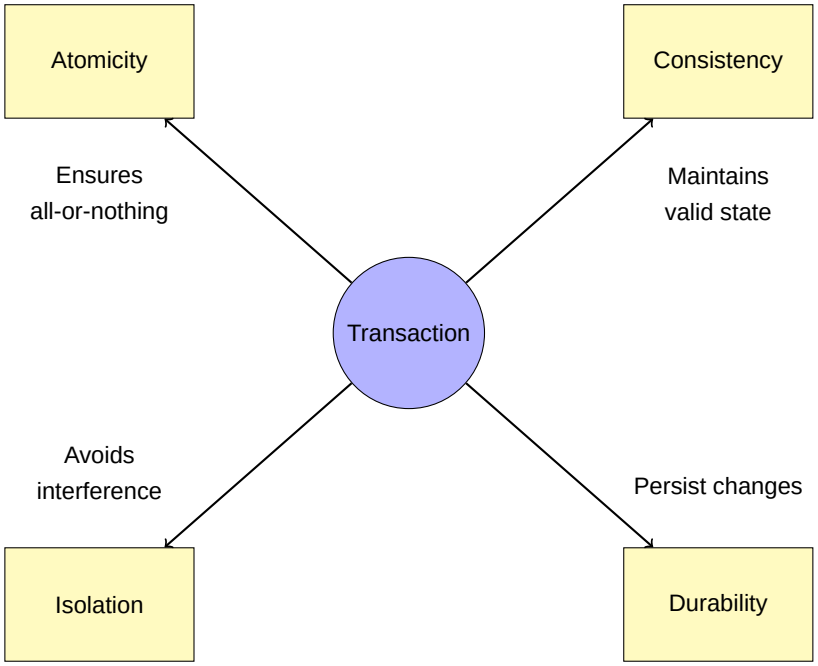


Figure 6.1: ACID Properties of a Transaction

6.2.1 Phases of a Transaction

1. Begin Transaction: Marks the start of a transaction.
2. Execute Statements: Database operations are performed.
3. Validation (Check for Success): Determines if the transaction can proceed.
4. Commit or Rollback: Commits changes if successful; otherwise, rolls back.

6.3 Isolation Levels

Isolation levels define how strictly transactions are isolated from each other, affecting consistency and concurrency. The following levels are common:

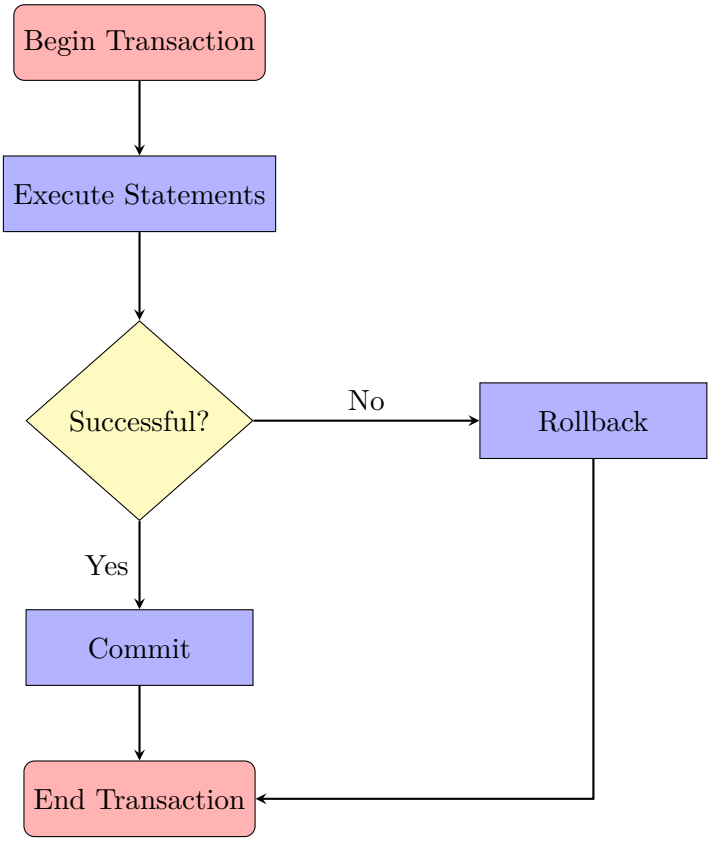


Figure 6.2: Transaction Lifecycle

6.3.1 Read Uncommitted

In this level, a transaction can see changes made by other transactions that have not yet been committed. This can lead to dirty reads, where uncommitted, potentially inconsistent, or invalid data is read. While this level offers the highest concurrency, it provides the lowest level of data consistency and integrity. It is rarely used in scenarios where data accuracy is critical.

6.3.2 Read Committed

In this level, transactions can only see data that has been committed by other transactions, preventing dirty reads. However, it allows non-repeatable

reads, meaning if the same data is read multiple times within a transaction, it may change if other transactions commit modifications in between. This level strikes a balance between consistency and concurrency, making it a common choice for many applications.

6.3.3 Repeatable Read

In this level, a transaction ensures that once it reads a row, no other transaction can modify that row until the initial transaction is complete. This prevents non-repeatable reads, where a transaction might see different values for the same row if read multiple times. However, it still allows phantom reads, where new rows added by other transactions can appear in subsequent reads. This level provides a higher degree of consistency compared to Read Committed but with reduced concurrency.

6.3.4 Serializable

The Serializable isolation level is the strictest, ensuring full isolation by locking rows or ranges of rows. This prevents other transactions from modifying or inserting data that would affect the current transaction. It eliminates dirty reads, non-repeatable reads, and phantom reads, providing the highest level of data consistency. However, this comes at the cost of reduced concurrency and potential performance overhead, as transactions may need to wait for locks to be released. It is typically used in scenarios where absolute data integrity is critical.

6.4 Concurrency Control

Concurrency control manages access in a multi-user environment, ensuring transaction isolation while balancing performance.

6.4.1 Locking Mechanisms

Locking mechanisms include:

- Pessimistic Locking: Locks data preemptively, blocking other transactions.
- Optimistic Locking: Allows concurrent access but checks for conflicts at commit.

6.4.2 Deadlock Handling

A deadlock occurs when two or more transactions wait on each other to release locks. Deadlocks are resolved through techniques like timeout and deadlock detection.

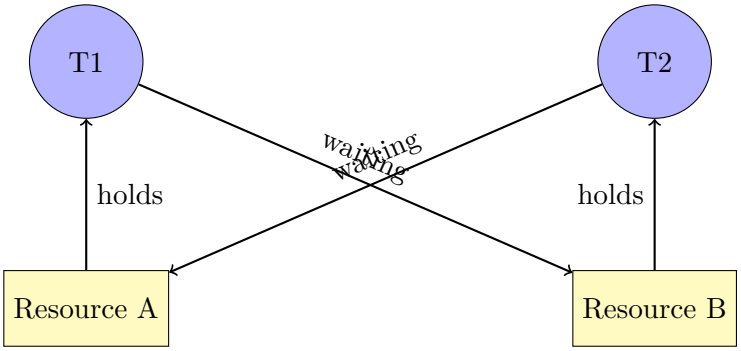


Figure 6.3: Deadlock Example: Transaction T1 and T2 each waiting for resources held by the other.

6.5 Error Handling and Recovery

Error handling in databases ensures that if a transaction fails, the system can revert to its previous state, maintaining data integrity. This is achieved through mechanisms like rollbacks, which undo changes made during the failed transaction. By implementing proper error handling, databases prevent partial updates or corruption, ensuring consistency. It also helps in identifying and logging errors for debugging and future improvements. Overall, error handling is crucial for reliable and stable database operations, safeguarding against unexpected failures.

6.5.1 Rollback Mechanisms

A rollback ensures that if a transaction encounters an error or fails, all changes made by that transaction are undone. This restores the database to its previous consistent state, maintaining data integrity. Rollbacks are a key feature of transactional systems, ensuring that partial or incomplete changes do not persist. They help enforce the atomicity property of transactions, where either all operations succeed or none do. This mechanism is crucial for reliability, especially in complex or critical systems.

6.5.2 Savepoints

Savepoints allow a transaction to roll back to a specific point without canceling the entire transaction. They provide finer control by marking intermediate stages within a transaction. If an error occurs, you can revert to a savepoint, undoing only the changes made after that point, while preserving earlier work. This is useful for complex transactions with multiple steps, enabling partial rollbacks. Savepoints enhance flexibility and efficiency in error handling, allowing transactions to continue from a known good state. They help maintain progress without starting over.

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
SAVEPOINT savepoint1;
UPDATE accounts SET balance = balance / 0 WHERE account_id = 2; --
  ↪ Intentional error
ROLLBACK TO savepoint1;
COMMIT;
```

6.6 Best Practices for Transactions

Following best practices for transactions optimizes database performance while maintaining consistency.

- Keep Transactions Short: Avoid lengthy transactions to minimize lock times.
- Use Appropriate Isolation Levels: Balance performance and isolation needs.
- Handle Errors Gracefully: Implement savepoints for controlled rollbacks.

6.7 Advanced Concepts

Advanced concepts include distributed transactions and protocols for handling them.

6.7.1 Distributed Transactions

These transactions span multiple databases, requiring a coordination protocol like Two-Phase Commit (2PC).

6.7.2 Two-Phase Commit (2PC)

The 2PC protocol ensures that all participating databases either commit or rollback to prevent inconsistencies.

1. Phase 1 - Prepare: Each participant secures resources.
2. Phase 2 - Commit/Rollback: All participants commit if prepared; otherwise, they rollback.

6.8 Lab

The purpose of this lab is to deepen your understanding of how transactions work, focusing on advanced topics such as deadlocks, transaction performance, and nested transactions. Additionally, you will experiment with different isolation levels and observe how they affect concurrent transactions. Pair up with your project partner to complete this lab.

This is a graded lab. Record your answers to the questions using LibreOffice Writer or LATEX, and submit one document per pair. Ensure that the names of both partners are included at the beginning of the document.

6.8.1 Preliminaries

Complete this section individually.

1. Download `bank.sql` and `films.sql` from the sciences-courses website and save them in your working directory.
2. You must decide how to handle autocommit mode. You can disable autocommit or manually manage transactions by using `BEGIN` and `COMMIT`.
3. To test concurrency, you will need to run multiple psql sessions (Several terminal sessions).

6.8.2 Transactions with Deadlocks

Deadlocks occur when two or more transactions wait for each other to release locks. In this section, you'll intentionally create a deadlock and observe how PostgreSQL resolves it.

- In the first psql session, start a transaction and run the `bank.sql` script to create and populate a bank accounts table.
- In the second psql session, start a transaction and run the `films.sql` script to create and populate a films table.
- In session 1, update an account balance in the bank table without committing.
- In session 2, update a film title in the films table without committing.
- In session 1, attempt to update the films table and, in session 2, attempt to update the bank accounts table. Both transactions should now be waiting for each other.
- After a short time, PostgreSQL should detect the deadlock and abort one of the transactions. Observe which session is aborted.

Describe the result of the deadlock and explain how PostgreSQL resolves the issue. Why was one transaction aborted and not the other?

6.8.3 Measuring Transaction Performance

In this section, you will compare the performance of different transaction isolation levels.

- Create a new table for recording performance metrics:

```
CREATE TABLE performance_metrics (
    isolation_level TEXT,
    transaction_duration INTERVAL
);
```

- Write a script that runs 1000 transactions at different isolation levels (`READ COMMITTED`, `REPEATABLE READ`, and `SERIALIZABLE`) and records the duration of each transaction in the `performance_metrics` table.

- Run the script at each isolation level and record the average transaction time.

Which isolation level resulted in the fastest transactions? Which was the slowest? Explain why the results differ between isolation levels.

6.8.4 Nested Transactions

Nested transactions are supported in some databases, allowing sub-transactions to be committed or rolled back independently. PostgreSQL does not natively support nested transactions but does allow you to mimic them using `SAVEPOINT`.

- Start a transaction and insert several rows into the `films` table.
- Create a `SAVEPOINT` named `sp1`.
- Insert additional rows into the `films` table, then create another savepoint named `sp2`.
- Simulate an error by updating all records with an invalid value for one of the columns.
- Roll back to `sp2` and verify that the second set of inserts remains intact.
- Now roll back to `sp1` and verify that only the first set of inserts remains.

Describe the behavior of PostgreSQL when using `SAVEPOINT`. How does this emulate nested transactions?

6.8.5 Concurrent Updates and Row-Level Locking

In this section, you will explore how PostgreSQL handles row-level locking when two transactions attempt to update the same rows concurrently.

- In the first psql session, start a transaction and update a few rows in the `films` table.
- Without committing, in the second psql session, start another transaction and attempt to update the same rows.

- Observe the behavior of PostgreSQL. One of the transactions will be blocked until the other commits or rolls back.
- Commit or roll back the first transaction and observe the effect on the second transaction.

What did you observe when both transactions attempted to update the same rows? Describe PostgreSQL's handling of row-level locking and why one transaction was blocked.

6.8.6 Bonus: Simulating Lost Updates

A lost update occurs when two transactions update the same data without awareness of each others changes. In this bonus section, you will simulate a lost update scenario.

- Start two psql sessions.
- In the first session, start a transaction and read a row from the `bank` table.
- In the second session, start a transaction and read the same row from the `bank` table.
- In the first session, update the balance of the account, then commit.
- In the second session, update the balance of the same account without re-reading the row and commit.

Did you successfully simulate a lost update? Why does this occur, and how could you prevent this issue in a real-world system?