Information Systems for Engineers

Jirayu Ruh

Contents

DISCLAIMER

These notes were compiled based on the lecture “Information Systems for Engineers” by Dr. G. Fourny and the exercises by D. Isik.

I accept no liability for any potential errors within these notes. Please be aware that these materials were processed with the assistance of NotebookLM. While AI is a powerful tool, it may produce technical inaccuracies or “hallucinations”. Furthermore, I have paraphrased sections and added personal observations, which may introduce further errors.

Unless stated otherwise, all graphics were generated personally.

Errors or copyright concerns can be reported via email to jirruh@ethz.ch.

Jirayu Ruh, 29.12.2025

Chapter 1

Introduction

Modern technological infrastructure relies heavily on the ability to manage vast quantities of facts effectively. At its most fundamental level, we distinguish between data—which represents raw, stored facts—and information, which is data endowed with specific meaning. When this information is applied to solve meaningful problems or used in decision-making processes, it matures into knowledge. An information system is essentially a collection of software programs designed to manage this progression of information efficiently.
In the broader context of science, data management has shifted paradigms. While classical mathematics and physics focused on the world as it must be or as it is observed naturally, computer science and data science focus on computing and data-driven insights. Data can be viewed as the "matter" of the digital world, and studying its behavior and storage is central to modern engineering.

Definition 1.0.1: Information System

A software program or a complex suite of programs dedicated to the management, storage, and exchange of information.

Theory 1.0.1 Data-Information-Knowledge Hierarchy

The progression from raw facts (data) to interpreted meaning (information) and finally to the purposeful application of that information (knowledge).

1.1 Historical Context and the Evolution of Storage

The necessity of recording information spans human history, moving from oral traditions to the invention of writing, accounting, and eventually the printing press. However, the mid-20th century marked the beginning of the computational era. In the 1960s, data was primarily managed through file systems. These systems were rudimentary, as they essentially involved independent programs reading from local disks, often leading to data redundancy and inconsistency across different applications.
The 1970s ushered in the Relational Era, largely defined by the work of Edgar Codd. He proposed a model where data is organized into tables (relations), allowing users to interact with data logically rather than worrying about its physical placement on a disk. The 1980s saw the rise of object-oriented models, and the 2000s introduced the NoSQL era, which addressed the needs of massive, distributed data through key-value stores, document stores, and graph databases.

Definition 1.1.1: Relational Database

An organized collection of related data presented to the user as a set of two-dimensional tables called relations.

1.2 The Purpose and Functionality of a DBMS

A Database Management System (DBMS) is a specialized software suite designed to manage and query databases. Relying on simple file systems for complex applications is problematic because it is difficult to combine data from different files, and there is no built-in support for multiple users or protection against data loss. A DBMS provides five critical functionalities to solve these issues.

First, it allows for the creation of new databases and the definition of their schemas, or logical structures. Second, it enables efficient querying and modification of data through specialized languages. Third, it supports the storage of immense datasets—reaching into the terabytes and petabytes—over long periods. Fourth, it ensures durability, meaning the system can recover from failures or errors without losing data. Finally, it manages concurrent access, allowing many users to interact with the data simultaneously without causing inconsistencies.

Definition 1.2.1: Database Management System (DBMS)

A powerful tool for creating, managing, and efficiently querying large amounts of persistent, safe data.

Theory 1.2.1 Data Independence

The principle, championed by Edgar Codd, that separates the physical storage of data from its logical representation, allowing users to interact with a logical model that the software then translates into physical structures.

1.3 Data Classification and the Three Vs

Data within an information system typically takes one of three shapes. Structured data, such as that found in relational databases or spreadsheets, follows a rigid schema. Semi-structured data, including XML, JSON, and YAML, possesses some internal structure but is more flexible and can be validated against frames or schemas. Unstructured data, such as raw text, audio, images, or video, lacks a predefined format and often requires advanced linear algebra and vector-based processing to analyze.

The scale of modern data is often described by the ”Three Vs”: Volume (the sheer amount of data, moving from terabytes to zettabytes), Variety (the different formats and sources of data), and Velocity (the speed at which new data is generated and must be processed). Understanding the prefixes of the International System of Units, such as Peta (10 to the 15th) and Exa (10 to the 18th), is essential for engineers working at this scale.

Definition 1.3.1: Structured Data

Data that is organized into a highly formatted structure, typically using the relational model, which makes it easily searchable via languages like SQL.

1.4 System Architecture and the Three-Tier Model

Modern information systems are often organized into a three-tier architecture to separate concerns and improve scalability. The top layer is the user interface or front-end, which manages the presentation and user interaction. The middle layer is the business logic, where the specific rules and processes of the application are defined. The bottom layer is the database system, which handles data persistence and management.

Within this architecture, the DBMS itself is divided into various components. A storage manager controls how data is placed on disk and moved between the disk and main memory. The query processor parses and optimizes requests to find the most efficient execution plan. The transaction manager ensures that database operations are performed safely and reliably.

Definition 1.4.1: 3-Tier Architecture

A software design pattern consisting of three layers: the presentation layer (User Interface), the logic layer (Business Logic), and the data layer (Database System).

1.5 Database Languages: DDL and DML

Interaction with a DBMS occurs through two primary types of languages. The Data Definition Language (DDL) is used to establish and modify the metadata, which is the "data about data" describing the schema and constraints of the database. The Data Manipulation Language (DML) is used to search, retrieve, and modify the actual data stored within that schema.
These languages can be further categorized as imperative or declarative. Imperative languages require the programmer to specify *how* to perform a task (e.g., C++, Java), while declarative languages, most notably SQL, allow the user to specify *what* they want, leaving the "how" to the system's query optimizer.

Definition 1.5.1: Metadata

The structural information that defines the types and constraints of the data, essentially acting as a blueprint for the database.

Theory 1.5.1 Declarative Language Property

The characteristic of languages like SQL that allows users to describe the desired result of a query without defining the physical execution steps or algorithms required to reach that result.

1.6 Transaction Management and the ACID Test

A transaction is a single unit of work consisting of one or more database operations that must be treated as an indivisible whole. To maintain integrity, transactions must satisfy the ACID properties.

Definition 1.6.1: Transaction

A program or set of actions that manages information and must be executed as an atomic unit to preserve database consistency.

Theory 1.6.1 The ACID Properties

The fundamental requirements for reliable transaction processing:

- **Atomicity**: All-or-nothing execution; if any part of the transaction fails, the entire transaction is rolled back.
- **Consistency**: Every transaction must leave the database in a state that satisfies all predefined rules and constraints.
- **Isolation**: Each transaction must appear to execute in a vacuum, as if no other transactions are occurring simultaneously.
- **Durability**: Once a transaction is completed, its effects are permanent and must survive any subsequent system failures.

1.7 Database Roles and Ecosystems

A database environment involves several distinct roles. The Database Administrator (DBA) is responsible for coordination, monitoring, and access control. The Database Designer creates the structure and schema of the content. Power Users may interact with the system through complex programming, while Data Analysts use DML for updates and queries. Finally, Parametric Users interact with the database through simplified interfaces like menus and forms.
As systems grow, the challenge often becomes one of information integration. Large organizations may have many legacy databases that use different models or terms. Integration strategies include creating data warehouses—centralized repositories where data from various sources is translated and copied—or using mediators, which provide an integrated model of the data while translating requests for each individual source database.

Definition 1.7.1: Data Warehouse

A centralized database used for reporting and data analysis, which stores integrated data from one or more disparate sources.

Theory 1.7.1 Legacy Database Problem

The difficulty of decommissioning old database systems because existing applications depend on them, necessitating the creation of integration layers to combine their data with newer systems.

Chapter 2

The Relational Model

The relational model stands as the preeminent framework for managing data in contemporary information systems. Historically, the organization of data in tabular formats is a practice that stretches back nearly four millennia, beginning with early clay tablets. However, the modern digital iteration was pioneered by Edgar Codd in 1970. Codd's primary contribution was the principle of data independence, which strictly separates the logical representation of information from its physical implementation on storage devices. Before this shift, programmers were often required to understand the underlying physical structure of the data to perform even basic queries. The relational model replaced these complex, system-dependent methods with a high-level abstraction based on tables, which are referred to as relations.

In this model, data is represented as a collection of two-dimensional structures. This approach offers simplicity and versatility, allowing for anything from corporate records to scientific data to be modeled effectively. By restricting operations to a limited set of high-level queries, the relational model allows for significant optimization by the database management system, often performing tasks more efficiently than code written in general-purpose languages. This chapter details the structure, mathematical foundations, and design theories—specifically functional dependencies and normalization—that ensure data remains consistent and free from redundancy.

2.1 Core Terminology and Structural Components

The architecture of the relational model is defined by a specific set of terms that describe both the structure and the content of the data.

Definition 2.1.1: Attribute

An attribute is a named header for a column in a relation. It describes the meaning of the entries within that column. For example, in a table tracking information about movies, "title" and "year" would be typical attributes.

Definition 2.1.2: Tuple

A tuple is a single row in a relation, excluding the header row. It represents a specific instance of the entity described by the relation. A tuple contains one component for every attribute defined in the relation's schema.

Definition 2.1.3: Relation Schema

A relation schema consists of the name of the relation and the set of attributes associated with it. This is typically expressed as $R(A_1, A_2, \ldots, A_n)$. A database schema is the total collection of all relation schemas within a system.

Definition 2.1.4: Relation Instance

A relation instance is the specific set of tuples present in a relation at any given time. While schemas are relatively static, instances change frequently as data is inserted, updated, or deleted.

The components of a tuple must be atomic, meaning they are elementary types like integers or strings. The model explicitly forbids complex structures such as nested lists or sets as individual values. Every attribute is associated with a domain, which defines the set of permissible values or the specific data type for that column.

2.2 Mathematical Foundations of Relations

Mathematically, a relation is defined as a subset of the Cartesian product of the domains of its attributes. If an attribute A has a domain D, then the entries in the column for A must be elements of D. A record can be viewed as a partial function or a "map" from the set of attribute names to a set of atomic values.

Theory 2.2.1 Relation as a Set

In the abstract mathematical model, a relation is a set of tuples. This implies that the order of the rows is irrelevant and that every tuple must be unique. Furthermore, because attributes are a set, the order of columns does not change the identity of the relation, provided the components of the tuples are reordered to match.

While the theoretical model relies on set semantics, practical implementations often utilize alternate semantics:

- **Bag Semantics**: Used by SQL, this allows for duplicate records within a table.
- **List Semantics**: In this variation, the specific sequence of the records is preserved and carries meaning.

A database is formally defined as a set of these relational tables. To interact with this data, the model employs relational algebra, a system of operators that take one or more relations as input and produce a new relation as output.

2.3 Integrity Constraints and Consistency

To ensure the validity of data, the relational model enforces several categories of integrity.

Theory 2.3.1 Relational Integrity

The requirement that every record within a specific relation must possess the exact same set of attributes. Broken relational integrity occurs if attributes are missing or if redundant attributes appear in individual rows.

Theory 2.3.2 Atomic Integrity

Also known as the First Normal Form (1NF), this rule dictates that every value in a cell must be a single, indivisible unit. Complex data types cannot be stored within a single attribute field.

Theory 2.3.3 Domain Integrity

This constraint requires that every value for an attribute must belong to the predefined set of values or the data type associated with its domain.

2.4 Defining Relation Schemas in SQL

SQL (Structured Query Language) is the primary tool for implementing the relational model. It is divided into the Data-Definition Language (DDL) for creating and modifying schemas, and the Data-Manipulation Language (DML) for querying and updating data. The most fundamental command in DDL is the `CREATE TABLE` statement, which establishes the table name, its attributes, and their types.

2.4.1 SQL Data Types

Attributes must be assigned a primitive type. Common SQL types include:

- **CHAR(n)**: A fixed-length string of n characters.
- **VARCHAR(n)**: A variable-length string up to n characters.
- **INT / INTEGER**: Standard whole numbers.
- **FLOAT / REAL**: Floating-point numbers.
- **BOOLEAN**: Stores TRUE, FALSE, or UNKNOWN.
- **DATE / TIME**: Specific formats for calendar dates (e.g., YYYY-MM-DD) and clock times.

2.4.2 Keys and Uniqueness

Definition 2.4.1: Key

A key is a set of one or more attributes such that no two tuples in any possible relation instance can share the same values for all these attributes. A key must be minimal; no subset of its attributes can also be a key.

In SQL, keys are declared using the `PRIMARY KEY` or `UNIQUE` keywords. Attributes designated as a primary key are forbidden from containing NULL values, whereas `UNIQUE` columns may allow them depending on the system.

2.5 Functional Dependencies

A central concept in database design theory is the functional dependency (FD), which generalizes the idea of a key.

Theory 2.5.1 Functional Dependency

A functional dependency on a relation R is an assertion that if two tuples agree on a set of attributes $A_1, \ldots, A_n$, they must also agree on another set of attributes $B_1, \ldots, B_m$. This is written as $A \longrightarrow B$.

FDs are not merely observations about a specific instance of data but are constraints that must hold for every possible legal instance of the relation. They describe the relationships between attributes; for example, a movie's title and year might functionally determine its length and studio, as there is only one specific length and studio for a unique movie released in a given year.

Definition 2.5.1: Superkey

A superkey is a set of attributes that contains a key as a subset. Therefore, every superkey functionally determines all attributes of the relation, but it may not be minimal.

The closure of a set of attributes under a set of FDs is the collection of all attributes that are functionally determined by that set. Calculating the closure allows designers to identify all keys of a relation and test if a new FD follows from the existing ones.

2.6 Anomalies and the Need for Decomposition

Careless schema design leads to ”anomalies,” which are problems that occur when too much information is crammed into a single table. There are three primary types:

1. **Redundancy**: Information is repeated unnecessarily across multiple rows (e.g., repeating a studio's address for every movie they made).

2. **Update Anomalies**: If a piece of redundant information changes, it must be updated in every row. Failure to do so leads to inconsistent data.

3. **Deletion Anomalies**: Deleting a record might inadvertently destroy the only copy of unrelated information (e.g., deleting the last movie of a studio might remove the studio's address from the database entirely).

To eliminate these issues, designers use decomposition—the process of splitting a relation into two or more smaller relations whose attributes, when combined, include all the original attributes.

2.7 Normal Forms

The goal of decomposition is to reach a normal form that guarantees the absence of certain anomalies.

Theory 2.7.1 Boyce-Codd Normal Form (BCNF)

A relation R is in BCNF if and only if for every nontrivial functional dependency $A \longrightarrow B$, the set of attributes A is a superkey. In simpler terms, every determinant must be a key.

Any relation can be decomposed into a collection of BCNF relations. This process effectively removes redundancy caused by functional dependencies. However, while BCNF is very powerful, it does not always preserve all original dependencies. This leads to the use of a slightly relaxed condition.

Theory 2.7.2 Third Normal Form (3NF)

A relation R is in 3NF if for every nontrivial FD $A \longrightarrow B$, either A is a superkey, or every attribute in B that is not in A is "prime" (a member of some key).

3NF is useful because it is always possible to find a decomposition that is both lossless (the original data can be reconstructed) and dependency-preserving, which is not always true for BCNF.

2.8 Modifying and Removing Schemas

Database structures are dynamic. SQL provides the `DROP TABLE` command to remove a relation and all its data permanently. For structural changes, the `ALTER TABLE` command is used. This allows for the addition of new attributes via `ADD` or the removal of existing ones via `DROP`. When new columns are added, existing tuples typically receive a `NULL` value or a specified `DEFAULT` value.

Chapter 3

Data Definition with SQL

3.1 Overview of Data Definition and the SQL Language

The process of managing data within an information system begins with a rigorous definition of its structure. The Data Definition Language (DDL) serves as the primary tool for database administrators to specify the logical organization of data, known as the schema. Historically, the evolution of database languages was marked by the development of SEQUEL in the early 1970s at the IBM Almaden Research Center, which was eventually renamed SQL due to trademark concerns. SQL is a declarative, set-based language, meaning it allows users to specify what data they want to retrieve or manipulate without detailing the step-by-step physical procedures. This abstraction significantly enhances the productivity of developers by separating the conceptual model from the physical storage layer.

Data in a relational system is organized into two-dimensional tables called relations. These relations must adhere to fundamental integrity principles to maintain data quality. Relational integrity ensures that every entry in a table follows the structure defined by its attributes. Domain integrity mandates that every attribute value belongs to a specific, predefined set of acceptable values. Finally, atomic integrity requires that every component of a tuple is an indivisible unit, preventing the use of complex structures like lists or nested records as simple attribute values.

3.2 Core Domain Types and Atomic Integrity

A central aspect of data definition is the selection of appropriate domain types for each attribute. SQL provides a rich set of standardized types to handle various data categories. Character data is typically managed through fixed-length strings (CHAR) or variable-length strings with a maximum limit (VARCHAR). For exceptionally large textual content, types such as CLOB or TEXT are utilized. Numeric data is bifurcated into exact and approximate types. Exact numbers include various sizes of integers (SMALLINT, INTEGER, BIGINT) and fixed-point decimals (DECIMAL or NUMERIC), where precision and scale can be strictly defined. Approximate numbers, such as REAL and DOUBLE PRECISION, follow floating-point standards to represent scientific data where a degree of approximation is acceptable.

Definition 3.2.1: Atomic Integrity

The requirement that every value in a relational table must be a single, indivisible data item of an elementary type, such as an integer or a string.

Temporal data is equally vital, with SQL supporting types for dates, times, and timestamps. These allow for the storage of specific points in time, often including time zone information for global applications. Furthermore, intervals represent durations, such as ”two years and four months.” Binary data, such as images or passport scans, is stored using BLOB or BYTEA types. Boolean types provide the foundation for logical operations, supporting TRUE, FALSE, and the three-valued logic involving UNKNOWN when NULL values are present.

3.3 Structural Operations: Creating and Modifying Tables

The lifecycle of a database schema involves the creation, modification, and removal of tables. The `CREATE TABLE` command is the primary DDL statement used to introduce new relations. It requires the specification of a table name, a list of attributes, and their associated domains. A newly created table is initially empty, representing an extension of zero tuples.

Theory 3.3.1 Relational Schema
The formal definition of a relation, comprising its unique name and the set of attributes along with their corresponding data types or domains.

As requirements change, the `ALTER TABLE` statement allows administrators to evolve the schema without deleting existing data. This includes adding new columns, which may be initialized with NULL or a specific default value, and renaming or removing existing columns. When a table is no longer required, the `DROP TABLE` command removes both the schema and all stored data from the system. To avoid errors during automated scripts, the `IF EXISTS` clause is frequently employed to ensure a command only executes if the target relation is present.

3.4 Data Manipulation and Logic of Modifications

Once the schema is defined, Data Manipulation Language (DML) commands are used to populate and maintain the data. The `INSERT` statement adds new records to a relation. It can take specific values for a single tuple or use a subquery to perform bulk insertions from other tables. A critical rule in SQL modification is that the system must evaluate the query portion of an insertion entirely before any data is actually added to the target table. This prevents infinite loops or inconsistent states where a new tuple might satisfy its own insertion criteria.

Definition 3.4.1: Cascading Rollback

A situation where the abortion of one transaction necessitates the cancellation of other dependent transactions that have read data written by the initial transaction.

The `DELETE` and `UPDATE` commands provide the means to remove or modify existing tuples based on specific conditions. Similar to insertions, these operations apply the condition to every tuple in the relation and execute the change only on those that satisfy the predicate. Through these commands, the system transitions between different database states while aiming to preserve overall consistency.

3.5 Integrity Constraints and Key Definitions

Constraints are declarative rules that restrict the data permitted in the database to prevent inaccuracies. The most fundamental constraints are those defining keys. A primary key uniquely identifies each row in a table and, by definition, cannot contain NULL values. In contrast, the `UNIQUE` constraint ensures distinctness but may permit NULLs depending on the specific DBMS implementation.

Theory 3.5.1 The Thomas Write Rule
A principle in timestamp-based concurrency control that allows a write operation to be ignored if a later transaction has already updated the same data element, thereby maintaining the intended final state.

Beyond keys, `NOT NULL` constraints ensure that critical attributes always have a value. `CHECK` constraints provide more complex logic, allowing the system to validate that an attribute or an entire tuple meets specific boolean conditions. For instance, a check could ensure that a person's age is never negative or that a start date precedes an end date.

3.6 Referential Integrity and Foreign Keys

Referential integrity is maintained through foreign keys, which establish a link between tables. A foreign key in one table must reference a unique or primary key in another. This ensures that the relationship between entities remains valid; for example, every movie in a tracking system must be associated with an existing studio.

Definition 3.6.1: Foreign Key

An attribute or set of attributes in a relation that serves as a reference to a primary or unique key in a different relation, enforcing a logical connection between the two.

The management of these links during data removal or updates is governed by specific policies. The `CASCADE` policy ensures that changes in the parent table are automatically reflected in the child table. Alternatively, the `SET NULL` policy breaks the link by nullifying the foreign key when the referenced record is deleted. If neither is appropriate, the `RESTRICT` policy blocks any modification that would break referential integrity.

3.7 Advanced Constraints and Deferred Checking

For constraints that span multiple tables or require global validation, SQL offers assertions. Unlike table-based checks, assertions are standalone schema elements that the DBMS must verify whenever any involved relation is modified. This makes them powerful but potentially expensive to implement efficiently.

Theory 3.7.1 Two-Phase Locking (2PL)

A concurrency control protocol that guarantees conflict-serializability by requiring that all lock acquisitions by a transaction must occur before any of its locks are released.

In complex transactions where multiple interrelated tables are updated, immediate constraint checking can be problematic. SQL addresses this with deferred checking. By declaring a constraint as `DEFERRABLE`, the system can postpone validation until the very end of a transaction, just before it commits. This allows for temporary inconsistencies that are resolved by the time the transaction completes its entire sequence of actions.

3.8 Active Database Elements: Triggers

Triggers, or Event-Condition-Action (ECA) rules, represent the transition from a passive database to an active one. A trigger is awakened by a specific event—such as an insertion, deletion, or update—and then evaluates a condition. If the condition is true, the system executes a predefined set of actions.

Definition 3.8.1: Trigger

A stored procedure that is automatically invoked by the DBMS in response to specified changes to the database, consisting of a triggering event, a condition, and a resulting action.

Triggers offer significant flexibility compared to standard constraints. They can be set to execute either `BEFORE` or `AFTER` the triggering event and can operate at either the row level (executing for every modified tuple) or the statement level (executing once for the entire SQL command). They are frequently used to enforce complex business rules, maintain audit logs, or automatically fix data inconsistencies that simple constraints cannot handle.

Chapter 4

Relational Algebra

Relational algebra serves as the formal foundation for the manipulation of data within the relational model. It is a procedural language consisting of a set of operators that take one or more relations as input and produce a new relation as output. This mathematical framework is essential for database systems because it provides a precise way to represent queries and allows the system's query optimizer to reorganize expressions into more efficient execution plans. Unlike general-purpose programming languages such as C or Java, relational algebra is intentionally limited in power. For example, it cannot perform arbitrary calculations like factorials or determine if the number of tuples in a relation is even or odd. However, this limitation is a strategic advantage, as it enables the database engine to perform high-level optimizations that would be impossible in a more complex language.
The algebra is characterized by the property of closure, where every result is a relation that can immediately serve as an input for another operation. Operators are generally categorized into unary operators, which act on a single relation, and binary operators, which combine two relations. The core operations include set-based maneuvers—such as union, intersection, and difference—and relational-specific maneuvers like selection, projection, joins, and renaming. In modern database implementations, these concepts are often extended to support bag semantics (allowing duplicates) and complex operations like grouping and sorting.

Definition 4.0.1: Relational Algebra

A collection of mathematical operators that function on relations to perform data retrieval and manipulation. It is closed under its operations, ensuring that the output of any expression is itself a relation.

Theory 4.0.1 The Power of Optimization

By limiting the expressive power of the query language to relational algebra, database systems can effectively optimize code. This allows the system to replace inefficient execution strategies with mathematically equivalent but significantly faster algorithms.

4.1 The Unary Operators: Selection and Projection

Selection and projection are the primary filters used to reduce the size of a relation. Selection, denoted by the Greek letter sigma (σ), acts as a horizontal filter. It examines each tuple in a relation and retains only those that satisfy a specific logical condition. This condition can involve attribute comparisons to constants or other attributes using standard operators such as equality, inequality, and logical connectors like AND and OR.
Projection, denoted by the Greek letter pi (π), acts as a vertical filter. It is used to choose specific columns from a relation while discarding others. In its classical set-based form, projection also serves to eliminate any duplicate tuples that may arise when certain attributes are removed. This ensures that the result remains a valid set. Extended versions of projection allow for the creation of new attributes through calculations or renamings of existing fields.

Definition 4.1.1: Selection

An operation $\sigma_C(R)$ that produces a relation containing all tuples from R that satisfy the condition C. It does not change the schema of the relation but reduces the number of rows.

Definition 4.1.2: Projection

An operation $\pi_L(R)$ that creates a new relation consisting of only the attributes listed in L. It transforms the schema of the relation and may reduce the number of rows if duplicates are removed.

4.2 Set Operations and Compatibility Constraints

Relational algebra incorporates standard set operations: union ($\cup$), intersection ($\cap$), and difference ($-$). Because these operations are inherited from set theory, they require the participating relations to be "compatible." This means the relations must share the same schema—specifically, they must have the same set of attributes, and the domains (data types) associated with corresponding attributes must be identical.

Union combines all tuples from two relations into a single result. Intersection identifies tuples that appear in both input relations. Difference, which is not commutative, returns tuples found in the first relation but not the second. While these were originally defined for sets, modern systems often apply them to "bags" (multisets), where the rules for handling duplicates differ. For instance, in bag union, the number of occurrences of a tuple is the sum of its occurrences in the inputs, whereas in set union, it appears only once.

Definition 4.2.1: Schema Compatibility

The requirement that two relations involved in a set operation must have the same number of attributes, with matching names and identical data types for each corresponding column.

Theory 4.2.1 Commutativity and Associativity

Set union and intersection are both commutative ($R \cup S = S \cup R$) and associative ($(R \cup S) \cup T = R \cup (S \cup T)$), allowing the system to reorder these operations for better performance.

4.3 Renaming and Relational Variables

In complex queries, it is often necessary to change the name of an attribute or the relation itself to avoid ambiguity or to prepare a relation for a set operation. The renaming operator, denoted by the Greek letter rho (ρ), allows for this modification. This is particularly useful when joining a relation with itself, as it provides a way to distinguish between the two copies.

The concept of a "relvar" (relational variable) is also important here. A relvar is essentially a variable that has a name and is assigned a specific relation. In algebraic expressions, we use these names to refer to the data stored within the tables.

Definition 4.3.1: Renaming

An operator $\rho_S(R)$ that returns a new relation identical to R in content but renamed to S. It can also be used as $\rho_{S(A_1,\ldots,A_n)}(R)$ to rename individual attributes.

4.4 Combining Relations: Products and Joins

The most complex operations in relational algebra involve combining information from different relations. The Cartesian Product ($\times$) is the most basic of these, pairing every tuple of the first relation with every tuple of the second. While mathematically simple, the product often produces very large relations that contain many irrelevant pairings.

Joins are more refined versions of the product. The Theta-Join ($\bowtie_C$) performs a product followed by a selection based on a specific condition C. The most common join is the Natural Join ($\bowtie$), which automatically pairs tuples that have equal values in all attributes shared by the two relations. After the pairing, it removes the redundant columns, leaving a cleaner result.

Definition 4.4.1: Cartesian Product

A binary operator $R \times S$ that produces a relation whose schema is the union of the schemas of R and S, and whose tuples are all possible concatenations of a tuple from R and a tuple from S.

Definition 4.4.2: Natural Join

A specific type of join $R \bowtie S$ that connects tuples based on equality across all common attributes and then projects out the duplicate columns.

Theory 4.4.1 The Join-Product Relationship

A theta-join $R \bowtie_C S$ is mathematically equivalent to the expression $\sigma_C(R \times S)$. This relationship allows query optimizers to choose between different physical execution strategies for the same logical request.

4.5 Linear Notation and Expression Trees

Because relational algebra is a functional language, complex queries are built by nesting operations. These can be represented in two main ways. Linear notation involves a sequence of assignments to temporary variables, making the steps of a query easier to read. Alternatively, expression trees provide a graphical representation where leaves are stored relations and interior nodes are algebraic operators.
The query processor uses these trees to visualize the flow of data. By applying algebraic laws, the processor can ”push” selections and projections down the tree, closer to the data sources. This reduces the size of intermediate relations as early as possible, which is a hallmark of efficient query execution.

Theory 4.5.1 Selection Pushing

In a query tree, moving a selection σ below other operators like joins or unions is almost always beneficial, as it reduces the number of tuples that subsequent, more expensive operators must process.

4.6 Extended Relational Algebra

To meet the requirements of SQL, the basic algebra is often extended with additional operators. These include duplicate elimination (δ), which explicitly turns a bag into a set; sorting (τ), which orders the tuples of a relation; and grouping and aggregation (γ), which partitions tuples into groups and calculates summaries like sums or averages.
While these operators go beyond the original mathematical definition of the algebra, they are essential for practical database management. They allow the algebra to serve as a complete intermediate language for translating SQL queries into physical instructions for the machine.

Definition 4.6.1: Duplicate Elimination

The operator $\delta(R)$ that takes a bag R as input and returns a set containing exactly one copy of every distinct tuple found in the input.

Definition 4.6.2: Aggregation

The application of functions such as SUM, AVG, MIN, MAX, or COUNT to a column of a relation to produce a single summary value.

Chapter 5

Queries with SQL

5.1 Overview of SQL Querying

The Structured Query Language (SQL) serves as the primary interface for interacting with relational database management systems. At its core, SQL is a declarative language, meaning that a programmer specifies what data should be retrieved rather than providing the step-by-step procedure for how to find it. This approach allows the database's internal query optimizer to determine the most efficient path to the data, leveraging indexes and specific storage structures. SQL is fundamentally set-based, manipulating entire relations as units rather than processing individual records in a procedural loop. This aligns closely with the mathematical foundations of relational algebra, where operations like selection, projection, and join act on sets or bags of tuples.

The language is typically divided into two main components: the Data Definition Language (DDL) and the Data Manipulation Language (DML). DDL is concerned with the metadata and the structural level of the database, involving the creation, modification, and deletion of tables and columns. DML, which is the focus of this summary, operates at the record level. It allows for the insertion of new data, the updating of existing records, the deletion of information, and most importantly, the querying of the stored data.

Definition 5.1.1: SQL

Structured Query Language, a declarative and set-based language used to define and manipulate data within a relational database management system.

Theory 5.1.1 Declarative Programming

The principle where the programmer describes the desired result of a computation without specifying the control flow or algorithmic steps to achieve it.

5.2 Fundamental Selection and Projection

The most basic form of a SQL query follows the select-from-where structure. This construct allows a user to extract specific attributes from one or more tables based on certain criteria. To understand this in the context of relational algebra, we can view the three main clauses as distinct algebraic operators. The FROM clause identifies the source relations; if multiple relations are listed, it effectively represents a Cartesian product. The WHERE clause functions as a selection operator (σ), filtering the tuples produced by the product based on a logical predicate. Finally, the SELECT clause acts as a projection operator (π), narrowing the result set to only the desired columns.

In its simplest manifestation, a query can use the asterisk symbol (*) to denote that all columns from the source table should be included in the output. While this is useful for exploration, explicit projection is preferred in production environments to minimize data transfer and clarify the schema of the result set. Within the SELECT clause, we are not limited to just listing attributes; we can also include arithmetic expressions, such as calculating a value based on existing columns, or constants to provide context in the output.

Definition 5.2.1: Query

A formal request for information from a database, typically expressed in SQL to retrieve specific data matching a set of conditions.

Theory 5.2.1 SQL to Relational Algebra Mapping

A simple select-from-where query is equivalent to the relational algebra expression $\pi_L(\sigma_C(R_1 \times R_2 \times ... \times R_n))$, where L is the select list, C is the where condition, and R_i are the relations in the from list.

5.3 Renaming and Aliasing

When a query is executed, the resulting relation has column headers that defaults to the names of the attributes in the source tables. However, there are many cases where these names may be ambiguous or uninformative, particularly when calculations are involved. SQL provides the 'AS' keyword to assign an alias to a column or an expression. This allows the programmer to rename the output for better readability or to comply with the requirements of an application.
Aliases can also be applied to relations in the FROM clause. These are referred to as tuple variables or correlation names. Aliasing a relation is essential when a query must compare different rows within the same table, a process known as a self-join. By assigning different aliases to the same table, the query can treat them as two distinct sources of data, enabling comparisons between tuples.

Definition 5.3.1: Alias

A temporary name assigned to a table or a column within the scope of a single SQL query to improve clarity or disambiguate references.

5.4 String Patterns and Comparison Operators

SQL provides a robust set of comparison operators to filter data within the WHERE clause. These include equality (=), inequality (¡¿), and various ordering comparisons (¡, ¿, ¡=, ¿=). While numeric comparisons are straightforward, string comparisons follow lexicographical order.
For more flexible string matching, the 'LIKE' operator is used with patterns. This operator allows for partial matches using two special wildcard characters: the percent sign (

Definition 5.4.1: Predicate

A logical expression that evaluates to true, false, or unknown, used in the WHERE clause to determine which tuples satisfy the query criteria.

Theory 5.4.1 Lexicographical Comparison

The method of ordering strings based on the alphabetical order of their component characters, where a string is ”less than” another if it appears earlier in a dictionary.

5.5 Handling Incomplete Information with Null Values

In real-world databases, it is common for certain pieces of information to be missing or inapplicable. SQL represents this missing data with a special marker called 'NULL'. It is important to recognize that 'NULL' is not a value in the same way 0 or an empty string is; it is a placeholder indicating the absence of a value.
Because 'NULL' represents unknown data, comparisons involving 'NULL' cannot result in a standard true or false. Instead, SQL employs a three-valued logic system that includes 'UNKNOWN'. For example, if we compare a column containing a 'NULL' to a constant, the result is 'UNKNOWN'. To explicitly check for these placeholders,

SQL provides the ‘IS NULL‘ and ‘IS NOT NULL‘ operators. Standard equality comparisons like ‘= NULL‘ will always evaluate to ‘UNKNOWN‘ and therefore fail to filter the desired records.

Definition 5.5.1: NULL

A special marker in SQL used to indicate that a data value does not exist in the database, either because it is unknown or not applicable.

Theory 5.5.1 Three-Valued Logic

A system of logic where expressions can evaluate to TRUE, FALSE, or UNKNOWN, requiring specialized truth tables for AND, OR, and NOT operations.

5.6 Logic and Truth Tables in SQL

The presence of ‘UNKNOWN‘ values necessitates a clear understanding of how logical operators behave. When combining conditions with ‘AND‘, the result is the minimum of the truth values, where TRUE is 1, UNKNOWN is 0.5, and FALSE is 0. Conversely, ‘OR‘ takes the maximum of the truth values. The ‘NOT‘ operator subtracts the truth value from 1.
In the context of a WHERE clause, a tuple is only included in the final result set if the entire condition evaluates to ‘TRUE‘. Tuples for which the condition is ‘FALSE‘ or ‘UNKNOWN‘ are excluded. This behavior can lead to unintuitive results, such as a query for ”all records where X is 10 OR X is not 10” failing to return records where X is ‘NULL‘, because the result of that OR operation would be ‘UNKNOWN‘.

Definition 5.6.1: Truth Table

A mathematical table used to determine the result of logical operations given all possible combinations of input truth values.

5.7 Multi-Relation Queries and the Cartesian Product

When a query involves data spread across multiple tables, the FROM clause lists all the relevant relations. The logical starting point for such a query is the Cartesian product, which pairs every tuple from the first relation with every tuple from the second, and so on. This produces a very large intermediate relation where each row represents a potential combination of the source data.
To make this product useful, the WHERE clause must contain join conditions that link the relations based on common attributes. For instance, if we are joining a ’Movies’ table with a ’Producers’ table, we might equate the ’producerID’ column in both. This filtering process discards the vast majority of the Cartesian product, leaving only the rows where the related data actually matches. When attributes in different tables share the same name, we use the dot notation (e.g., TableName.AttributeName) to disambiguate the references.

Theory 5.7.1 The Join-Selection Equivalence

The principle that a natural join or an equijoin can be logically expressed as a selection performed on a Cartesian product of relations.

5.8 Interpretation of Multi-Relation Queries

There are multiple ways to interpret the execution of a query involving several relations. One helpful mental model is the ”nested loops” approach. In this model, we imagine a loop for each relation in the FROM clause. The outermost loop iterates through every tuple of the first relation, and for each of those, the next loop iterates through the second relation, and so on. Inside the innermost loop, the WHERE condition is tested against the current combination of tuples. If the condition is met, the SELECT clause produces an output row.
Another interpretation is based on parallel assignment. In this view, we consider all possible assignments of tuples to the variables representing the relations. We then filter for those assignments that satisfy the condition. While

the nested loop model is more algorithmic, the parallel assignment model highlights the declarative nature of the query, emphasizing that the order of the relations in the FROM clause should not, in theory, affect the result.

Definition 5.8.1: Tuple Variable

A variable that ranges over the tuples of a relation, often implicitly created for each table in the FROM clause or explicitly defined as an alias.

5.9 Set Operators and Bag Semantics

SQL provides operators for the traditional set-theoretic actions: ‘UNION‘, ‘INTERSECT‘, and ‘EXCEPT‘. These allow the results of two queries to be combined, provided they have the same schema (compatible attribute types and order). By default, these operators follow set semantics, meaning that they automatically eliminate duplicate tuples from the result.
However, since SQL is fundamentally based on bags (multisets), it also provides versions of these operators that preserve duplicates using the ‘ALL‘ keyword. ‘UNION ALL‘ simply concatenates the two result sets. ‘INTERSECT ALL‘ produces a tuple as many times as it appears in both inputs (taking the minimum count). ‘EXCEPT ALL‘ produces a tuple as many times as it appears in the first input minus the number of times it appears in the second (taking the difference). Using bag semantics is often more efficient because the system does not need to perform the expensive work of sorting or hashing the data to find and remove duplicates.

Definition 5.9.1: Bag

A collection of elements that allows for multiple occurrences of the same element, where the order of elements remains immaterial.

Theory 5.9.1 Closure of Bag Operations

The property that the result of any operation on bags is also a bag, ensuring that the relational model remains consistent through complex sequences of operations.

5.10 Nested Queries and Scalar Subqueries

A subquery is a query nested within another query. Subqueries can appear in various parts of a SQL statement, including the WHERE, FROM, and HAVING clauses. A scalar subquery is one that returns exactly one row and one column—a single value. Because it evaluates to a scalar, it can be used anywhere a constant or an attribute would be valid, such as in a comparison.
If a scalar subquery is designed to find, for instance, the specific ID of a person, and the data actually contains two people with that name, the query will fail at runtime. The system expects a single value and cannot resolve the ambiguity. If the subquery returns no rows, it is treated as a ‘NULL‘.

Definition 5.10.1: Scalar

A single atomic value, such as an integer or a string, as opposed to a collection of values like a row or a table.

5.11 Conditions on Relations: IN and EXISTS

When a subquery returns a set of values rather than a single scalar, it can be used with relational operators like ‘IN‘. The expression ‘x IN (subquery)‘ evaluates to true if the value of x is found in the result set produced by the subquery. This is a powerful way to filter data based on membership in a dynamically calculated set.
The ‘EXISTS‘ operator is another tool for dealing with subqueries. It takes a subquery as an argument and returns true if the subquery returns at least one row. Unlike ‘IN‘, ‘EXISTS‘ does not look at the actual values returned; it only checks for the existence of results. This is often used in correlated subqueries to check for the presence of related records in another table.

Theory 5.11.1 Existence Quantification

The logical principle of checking whether there is at least one element in a set that satisfies a given property, implemented in SQL via the EXISTS operator.

5.12 Correlated Subqueries and Scoping

A correlated subquery is a nested query that refers to attributes of the outer query. Because of this dependency, the subquery must, in concept, be re-evaluated for every row processed by the outer query. This creates a link between the two levels of the query, allowing for complex logic like "find all employees whose salary is higher than the average salary in their specific department."

Scoping rules in SQL dictate how attribute names are resolved. An attribute in a subquery will first be looked for in the tables mentioned in that subquery's own FROM clause. If it is not found there, the system looks at the FROM clause of the next level out, and so on. If the same attribute name appears in multiple levels, we must use aliases to ensure the correct column is referenced. Correlated subqueries are often more expressive than simple joins but can be more computationally expensive if the optimizer cannot unnest them into a join.

Definition 5.12.1: Correlated Subquery

A subquery that depends on the current row being processed by the outer query, identified by references to attributes defined in the outer scope.

5.13 Join Expressions and Syntax Variants

While the select-from-where structure can express most joins, SQL also provides explicit join syntax. A 'CROSS JOIN' is a direct representation of the Cartesian product. A 'JOIN ... ON' allows the join condition to be specified explicitly in the FROM clause, which many developers find clearer than placing the condition in the WHERE clause.

A 'NATURAL JOIN' is a specialized form of join that automatically equates all columns with the same name in both tables and removes the redundant copies of those columns. While natural joins are concise, they can be risky because they depend on attribute names. If a schema change adds a column to one table that happens to share a name with a column in another, the natural join logic will change automatically and potentially break the query. The 'USING' clause provides a middle ground, allowing the user to specify exactly which common columns should be used for the join.

Definition 5.13.1: Natural Join

A join operation that matches tuples based on all attributes that have the same name in both relations, producing a result that contains only one copy of each common attribute.

5.14 Outer Joins and Data Preservation

In a standard (inner) join, tuples that do not have a match in the other table are discarded. These are called "dangling tuples." If we wish to preserve these tuples in our result set, we use an 'OUTER JOIN'. There are three types: 'LEFT', 'RIGHT', and 'FULL'. A 'LEFT OUTER JOIN' includes all tuples from the left table; if a tuple has no match in the right table, the columns from the right table are filled with 'NULL'.

The 'RIGHT OUTER JOIN' is symmetric, preserving all rows from the right table. A 'FULL OUTER JOIN' preserves all rows from both tables, ensuring that no information from either source is lost. Outer joins are essential when we need a comprehensive list of items, even if some of those items lack certain related data.

Definition 5.14.1: Dangling Tuple

A tuple in one relation that does not match any tuple in another relation based on the join criteria.

Theory 5.14.1 The Outer Join Property

The guarantee that all tuples of the specified operand relations will be represented in the result, with NULL values used to fill in missing components for non-matching rows.

5.15 Aggregation and Data Summarization

SQL includes several built-in functions to perform calculations across entire columns of data. These are known as aggregation operators. The five standard operators are ‘SUM‘, ‘AVG‘, ‘MIN‘, ‘MAX‘, and ‘COUNT‘. ‘SUM‘ and ‘AVG‘ can only be applied to numeric data, while ‘MIN‘ and ‘MAX‘ can also be applied to strings (using lexicographical order) or dates.
The ‘COUNT‘ operator is versatile; ‘COUNT(*)‘ counts every row in a table, while ‘COUNT(attribute)‘ counts only the non-null values in that specific column. If we wish to count only the unique values, we can use the ‘DISTINCT‘ keyword inside the aggregation, such as ‘COUNT(DISTINCT studioName)‘. It is vital to remember that all aggregations except for ‘COUNT‘ return ‘NULL‘ if they are applied to an empty set of values. ‘COUNT‘ returns 0 for an empty set.

Definition 5.15.1: Aggregation

The process of summarizing multiple values into a single value through functions like summation or averaging.

5.16 Grouping and Partitioning

The ‘GROUP BY‘ clause allows us to partition the rows of a relation into groups based on their values in one or more attributes. When a query contains a ‘GROUP BY‘ clause, the SELECT clause is limited in what it can contain. Every attribute listed in the SELECT clause must either be an attribute used for grouping or be part of an aggregate function.
Conceptually, the system first creates the groups and then applies the aggregate functions to each group independently. The result is a single row for each unique combination of values in the grouping attributes. This is the primary way to generate reports and statistics, such as ”the total number of movies produced by each studio per year.”

Definition 5.16.1: Grouping Attribute

An attribute used in the GROUP BY clause to define the partitions upon which aggregation functions will operate.

5.17 Post-Aggregation Filtering with HAVING

Sometimes we want to filter the results of a query based on an aggregate value. However, the WHERE clause is evaluated before any grouping or aggregation takes place. Therefore, we cannot use a condition like ‘WHERE SUM(length) ¿ 500‘. To solve this, SQL provides the ‘HAVING‘ clause.
The ‘HAVING‘ clause is evaluated after the groups have been formed and the aggregations have been calculated. It allows the programmer to specify conditions that apply to the group as a whole. Only the groups that satisfy the ‘HAVING‘ condition will appear in the final output. While ‘HAVING‘ can technically contain any condition, it is best practice to only use it for conditions involving aggregates, leaving all tuple-level filtering to the WHERE clause.

Definition 5.17.1: HAVING

A clause in SQL used to specify conditions that filter groups of rows created by the GROUP BY clause, typically involving aggregate functions.

Theory 5.17.1 Query Execution Order

The logical sequence of operations in a SQL query: FROM (and JOINs), then WHERE, then GROUP BY, then HAVING, and finally SELECT (and DISTINCT) and ORDER BY.

5.18 Ordering and Sorting the Result

The final step in many queries is to present the data in a specific order for the user. The ‘ORDER BY‘ clause facilitates this, allowing for sorting by one or more columns in either ascending (‘ASC‘) or descending (‘DESC‘) order. Sorting is the last operation performed before the data is returned; even if a column is not projected in the SELECT clause, it can still be used for sorting if it was available in the source tables.
If multiple columns are listed in the ‘ORDER BY‘ clause, the system sorts by the first column first. If there are ties, it uses the second column to break them, and so on. This ensures a deterministic and readable presentation of the retrieved information.

Definition 5.18.1: Sorting

The process of arranging the rows of a result set in a specific sequence based on the values of one or more attributes.

5.19 Extended Projection and Constants

The extended projection operator allows for more than just choosing columns. It enables the use of expressions that combine attributes or apply functions to them. In SQL, this is manifested in the SELECT list, where we can perform additions, concatenations, or even call stored functions.
Constants are also frequently used in the SELECT list. For example, a query might select ”Movie”, title, year from a table. Every resulting row would have the string literal ”Movie” as its first column. This is often used to label different parts of a union or to provide fixed formatting for an external application.

Theory 5.19.1 Functional Dependency in Aggregation

The rule that in a grouped query, any attribute in the SELECT list that is not aggregated must be functionally determined by the grouping attributes to ensure the result is well-defined.

5.20 Nested Queries in the FROM Clause

SQL allows a subquery to be placed in the FROM clause. In this case, the subquery acts as a temporary table that exists only for the duration of the outer query. This is particularly useful when we need to perform multiple levels of aggregation or when we want to join a table with a summarized version of itself.
When a subquery is used in the FROM clause, it must be assigned an alias. This alias allows the outer query to refer to the columns produced by the subquery. This technique is often a cleaner alternative to using complex correlated subqueries in the WHERE clause, as it makes the flow of data more explicit.

Definition 5.20.1: Derived Table

A temporary result set returned by a subquery in the FROM clause, which is then used by the outer query as if it were a physical table.

5.21 Summary of Advanced SQL Syntax

Throughout our exploration of Chapters 6 and the accompanying presentation, we have seen that SQL is far more than a simple tool for data retrieval. Its ability to nest logic, perform complex aggregations across partitioned data, and handle various join types allows it to solve sophisticated data analysis problems. The transition from the mathematical abstractions of relational algebra to the practical syntax of SQL reveals how each keyword serves a specific logical function in the data-processing pipeline.

By understanding the declarative nature of the language and the underlying bag semantics, developers can write queries that are not only correct but also efficient. The careful management of NULLs, the strategic use of subqueries, and the mastery of grouping and having clauses form the foundation of expert database programming. This comprehensive summary has detailed the syntax and the theoretical justifications for the most critical features of SQL querying, providing a roadmap for complex data manipulation.

Theory 5.21.1 The Universal Query Form

The select-from-where block is the universal building block of SQL, capable of expressing any operation that can be represented by the core operators of relational algebra.

Chapter 6

Database Design Theory

Relational database design is fundamentally a process of identifying and addressing structural weaknesses within a schema. While a database can be constructed by simply creating tables for every entity, a lack of rigorous design often results in redundancy and anomalies that compromise data integrity. Design theory provides a mathematical framework, primarily through the study of functional dependencies and normal forms, to evaluate and refine these structures.

The core objective is to ensure that a relation is "about" one specific concept. When a single table attempts to store information concerning multiple distinct entities—such as products, customers, and individual sales transactions all in one place—it inevitably leads to maintenance difficulties. These difficulties are categorized as update, insertion, and deletion anomalies. By applying normalization techniques, designers can decompose a problematic relation into smaller, more focused relations that preserve the original data while eliminating these risks.

This summary explores the foundational elements of this theory, beginning with the formal definition of functional dependencies, the mechanics of attribute closures, the identification of keys, and the application of Boyce-Codd Normal Form (BCNF) to achieve a robust database architecture. We also examine the critical properties of these decompositions, such as the lossless-join property, ensuring that information is never lost when tables are split.

6.1 Functional Dependencies

A functional dependency is a constraint that describes the relationship between attributes within a relation. It generalizes the concept of a key by stating that if the values of one set of attributes are known, the values of another set are determined.

Definition:

Definition 6.1.1: Functional Dependency

A functional dependency (FD) on a relation R is an expression of the form $A_1, A_2, \ldots, A_n \rightarrow B_1, B_2, \ldots, B_m$. This indicates that for any two tuples t and u in R, if t and u agree on all components for attributes $A_1, A_2, \ldots, A_n$, then they must also agree on all components for attributes $B_1, B_2, \ldots, B_m$.

In simpler terms, we say that the set $\{A_1, A_2, \ldots, A_n\}$ functionally determines the set $\{B_1, B_2, \ldots, B_m\}$. This is essentially a statement about the "functions" that can exist within a table; for every unique value in the determining set, there is a unique corresponding value in the determined set. It is important to note that a functional dependency is a property of the schema itself and must hold for all possible instances of the relation, not just the current set of data.

Concept:

Theory 6.1.1 The Nature of Functional Dependencies

A functional dependency $X \rightarrow Y$ exists because there is a logical or physical relationship in the real world being modeled, such that the values in Y are functionally dependent on the values in X. This relationship allows the DBMS to ensure that no contradictory data exists within the relation.

6.2 Keys and Superkeys

Functional dependencies allow us to define precisely what constitutes a key for a relation. A key is a set of attributes that uniquely identifies every tuple in a relation.
Definition:

Definition 6.2.1: Superkey

A set of attributes $\{A_1, A_2, \ldots, A_n\}$ is a superkey for a relation R if those attributes functionally determine all other attributes of the relation. That is, no two distinct tuples in any legal instance of R can have the same values for all attributes in the superkey.

Definition:

Definition 6.2.2: Key

A set of attributes K is a key for a relation R if K is a superkey and no proper subset of K is also a superkey. This implies that a key must be minimal; removing even one attribute from a key destroys its ability to uniquely identify tuples.

Concept:

Theory 6.2.1 Existence of Keys
Every relational table must have at least one candidate key. In the extreme case where no smaller subset exists, the set of all attributes in the relation serves as a superkey.

When a relation has multiple candidate keys, a designer usually selects one to be the primary key. While the choice of a primary key is often a matter of implementation preference, the underlying theory treats all candidate keys as equally capable of enforcing data integrity.

6.3 Rules for Reasoning About Functional Dependencies

Often, the set of functional dependencies explicitly provided by a designer implies the existence of other dependencies. Understanding how to derive these implicit constraints is necessary for normalization.

6.3.1 The Splitting and Combining Rules

Functional dependencies can be simplified or grouped based on their right-hand sides.

- **Splitting Rule:** An FD $X \to YZ$ is equivalent to the pair of FDs $X \to Y$ and $X \to Z$.
- **Combining Rule:** A set of FDs $X \to A_1, X \to A_2, \ldots, X \to A_n$ can be expressed as a single FD $X \to A_1A_2 \ldots A_n$.

It is crucial to observe that these rules only apply to the right-hand side of a dependency. Splitting the left-hand side is generally invalid; knowing that $\{Title, Year\} \to Length$ does not imply that $Title \to Length$.

6.3.2 Trivial Dependencies

A dependency is considered trivial if it is guaranteed to hold in any relation, regardless of the data. Definition:

Definition 6.3.1: Trivial Functional Dependency

An FD $X \to Y$ is trivial if the set of attributes Y is a subset of the set of attributes X. For example, $Title, Year \to Title$ is a trivial dependency because any two tuples agreeing on both title and year must necessarily agree on the title.

6.3.3 Transitivity

Concept:

Theory 6.3.1 Transitive Rule

If a relation satisfies the functional dependencies $X \longrightarrow Y$ and $Y \longrightarrow Z$, then it must also satisfy the functional dependency $X \longrightarrow Z$.

6.4 Attribute Closure

To determine if a specific functional dependency $X \longrightarrow A$ follows from a given set of dependencies S, we compute the closure of the set of attributes X.
Definition:

Definition 6.4.1: Attribute Closure

The closure of a set of attributes X with respect to a set of functional dependencies S, denoted X^+, is the set of all attributes A such that $X \longrightarrow A$ can be derived from S using the rules of functional dependencies.

Concept:

Theory 6.4.1 Closure Algorithm

To compute X^+: 1. Start with a set of attributes $Y = X$. 2. Repeatedly find an FD $B \longrightarrow C$ in S such that B is a subset of Y, but C is not. Add C to Y. 3. When no more attributes can be added, the resulting set Y is X^+.

This algorithm is essential for finding keys. If X^+ contains all attributes of a relation R, then X is a superkey for R. If no subset of X has a closure containing all attributes, then X is a key.

6.5 Design Anomalies

Redundancy in a database is not merely a waste of storage space; it is the root cause of data inconsistency through three types of anomalies.

- **Redundancy:** Storing the same piece of information multiple times. For example, if we store the studio address in every movie record, a studio with many movies will have its address repeated many times.
- **Update Anomaly:** This occurs when a change to one instance of redundant data is not reflected in all other instances. If a studio moves, every single movie record associated with that studio must be updated, or the database will hold contradictory addresses.
- **Deletion Anomaly:** This happens when the deletion of a tuple results in the unintended loss of unrelated information. If we delete the only movie associated with a studio, we might lose the information about that studio's existence and address entirely.
- **Insertion Anomaly:** This occurs when we cannot insert information about one entity without having information about another. For instance, we might not be able to store a new studio's address until they produce their first movie.

The goal of database design theory is to eliminate these anomalies by ensuring that all functional dependencies are ”cleanly” associated with keys.

6.6 Boyce-Codd Normal Form (BCNF)

BCNF is a strict standard for relation design that effectively eliminates the anomalies caused by functional dependencies.

Definition:

Definition 6.6.1: Boyce-Codd Normal Form

A relation R is in BCNF if and only if for every non-trivial functional dependency $X \longrightarrow Y$ that holds in R, X is a superkey of R.

In a BCNF relation, every determinant (the left side of an FD) must be a superkey. This ensures that every functional relationship in the table is a relationship from a key to some other attribute, preventing the "storing of two things in one table" that leads to redundancy.
Concept:

Theory 6.6.1 The BCNF Decomposition Algorithm

If a relation R is not in BCNF because of a violating FD $X \longrightarrow Y$: 1. Compute the closure X^+. 2. Decompose R into two relations: R_1 with attributes X^+ and R_2 with attributes X and all attributes of R that are not in X^+. 3. Recursively apply this process to R_1 and R_2 until all resulting relations are in BCNF.

This algorithm ensures that the data is split such that each new relation focuses on a single functional theme.

6.7 Decomposition and the Lossless-Join Property

When we decompose a relation into two or more smaller relations, we must ensure that the process is reversible. If we join the smaller relations back together, we must obtain exactly the original relation—neither more nor less.
Definition:

Definition 6.7.1: Lossless Join

A decomposition of a relation R into R_1 and R_2 is a lossless join if the natural join of R_1 and R_2 is always equal to R. If the join produces "extra" tuples that were not in the original relation, the join is "lossy," and the decomposition is invalid.

Concept:

Theory 6.7.1 Condition for a Lossless Join

A decomposition of R into R_1 and R_2 is lossless if and only if the set of attributes common to R_1 and R_2 is a superkey for at least one of the two relations.

The BCNF decomposition algorithm is guaranteed to be lossless because it always splits a relation based on a functional dependency $X \longrightarrow Y$, ensuring that the common attribute X is a key for the relation containing X and its determined attributes.

6.8 The Chase Algorithm

The "chase" is a general-purpose method to test whether a decomposition into any number of relations is lossless. It uses a tableau of rows and columns to track how functional dependencies can be used to equate different components of tuples.
Concept:

Theory 6.8.1 The Chase Procedure

To test if a decomposition $\{S_1, S_2, \ldots, S_k\}$ of relation R is lossless: 1. Create a tableau where each row corresponds to a relation S_i in the decomposition. 2. For each attribute A in R, if A is in S_i, the entry in row i, column A is a special symbol a. Otherwise, it is a unique subscripted symbol b_i. 3. Repeatedly apply the given FDs to the tableau. If an FD $X \longrightarrow Y$ exists and two rows agree on all attributes in X, then they must agree on all attributes in Y. 4. If a row ever becomes all a symbols, the decomposition is lossless. Otherwise, if no more changes can be made and no such row exists, it is lossy.

The chase algorithm provides a rigorous check for recoverability, even in complex multi-table designs.

6.9 Dependency Preservation

Another desirable property of a decomposition is dependency preservation. A decomposition preserves dependencies if we can check all the original functional dependencies by looking at the decomposed relations individually. While the BCNF decomposition is always lossless, it does not always preserve all dependencies. In some cases, a dependency $X \to Y$ might involve attributes that end up in different relations after the split. If it is impossible to find a BCNF decomposition that preserves all dependencies, a designer might opt for a less strict normal form, such as Third Normal Form (3NF).
Concept:

Theory 6.9.1 The Trade-off in Normalization

There is a fundamental trade-off in database design: we can always achieve a BCNF decomposition that is lossless, but we might not be able to preserve all dependencies. Conversely, 3NF synthesis can always provide a lossless and dependency-preserving decomposition, though it may allow some minor redundancy.

6.10 Minimal Basis of Functional Dependencies

Before performing normalization or synthesis, it is often helpful to simplify the set of functional dependencies into a minimal basis (also called a minimal cover).
Definition:

Definition 6.10.1: Minimal Basis

A set of FDs B is a minimal basis for a set of FDs S if: 1. B is equivalent to S. 2. All FDs in B have a single attribute on the right-hand side. 3. No FD can be removed from B while maintaining equivalence. 4. No attribute can be removed from the left-hand side of any FD in B while maintaining equivalence.

Starting with a minimal basis prevents the creation of redundant tables during the 3NF synthesis process and clarifies the essential constraints acting upon the data.

6.11 Conclusion

Database design theory transforms the intuitive process of organizing data into a rigorous engineering discipline. By defining functional dependencies, we uncover the hidden structure of data. By identifying keys and superkeys, we establish the requirements for unique identification. Through attribute closure, we derive the full implications of our constraints.
The application of BCNF and the decomposition algorithm allows us to restructure relations to eliminate the threat of update, insertion, and deletion anomalies. Finally, by verifying the lossless-join property via the chase algorithm, we ensure that our specialized, efficient schema remains a faithful representation of the original information. While compromises between BCNF and dependency preservation may sometimes be necessary, the tools provided by design theory allow these decisions to be made with a clear understanding of the costs and benefits to data integrity and system performance.

Chapter 7

Transactions and the Three Tiers

The evolution of data management has shifted from localized, single-machine installations to complex, multi-tiered architectures that support massive user bases across the globe. This chapter explores the foundational structures of modern information systems, specifically focusing on how databases operate within a server environment. We examine the interaction between various layers of processing, known as the three-tier architecture, and the logical organization of data into environments, clusters, catalogs, and schemas. Furthermore, we investigate the mechanisms that allow general-purpose programming languages, such as Java, to interact with SQL through call-level interfaces like JDBC. Central to this discussion is the management of transactions, which ensure that even in highly concurrent and distributed settings, the integrity and consistency of data are maintained through the adherence to the ACID properties and the management of isolation levels.

Definition 7.0.1: Database Management System

A specialized software system designed to create, manage, and provide efficient, safe, and persistent access to large volumes of data over long periods of time.

7.1 The Three-Tier Architecture

Modern large-scale database installations typically utilize a three-tier or three-layer architecture. This structure is designed to separate different functional concerns, which allows for better scalability, security, and maintenance.

Theory 7.1.1 Three-Tier Architecture

A system organization that distinguishes between three interacting layers: the Web Server tier (user interface), the Application Server tier (business logic), and the Database Server tier (data management).

The first layer is the Web-Server Tier. This tier manages the primary interaction with the user, often through the Internet. When a customer accesses a service, a web server responds to the initial request and presents the interface, such as an HTML page with forms and menus. The client's browser handles the user's input and transmits it back to the web server, which then communicates with the application tier.

The middle layer is the Application-Server Tier. This is where the "business logic" of an organization resides. The responsibility of this tier is to process requests from the web server by determining what data is needed and how it should be presented. In complex systems, this tier might be divided into subtiers, such as one for object-oriented data handling or another for information integration, where data from multiple disparate sources is combined. The application tier performs the heavy lifting of turning raw database information into a meaningful response for the end user.

The final layer is the Database-Server Tier. This layer consists of the processes that run the Database Management System (DBMS). It receives query and modification requests from the application tier and executes them against the stored data. To ensure efficiency, this tier often maintains a pool of open connections that can be shared among various application processes, avoiding the overhead of constantly opening and closing connections.

7.2 The SQL Environment and Its Logical Organization

The SQL environment provides the framework within which data exists and operations are executed. This environment is organized into a specific hierarchy to manage terminology and scope.

Definition 7.2.1: SQL Environment

The overall framework, typically an installation of a DBMS at a specific site, under which database elements are defined and SQL operations are performed.

At the top of this hierarchy is the Cluster. A cluster is a collection of catalogs and represents the maximum scope over which a single database operation can occur. Essentially, it is the entire database as perceived by a specific user.
Below the cluster is the Catalog. Catalogs are used to organize schemas and provide a unique naming space. Each catalog must contain a special schema that holds information about all other schemas within that catalog. The most basic unit of organization is the Schema. A schema is a collection of database elements such as tables, views, triggers, and assertions. One can create a schema using a specific declaration or modify it over time.

7.3 Establishing Connections and Sessions

For a program or a user to interact with the database server, a link must be established. This is handled through connections and sessions. A connection is the physical or logical link between a SQL client (often the application server) and a SQL server. A user can open multiple connections, but only one can be active at any given moment.

Definition 7.3.1: Session

The sequence of SQL operations performed while a specific connection is active. It includes state information such as the current catalog, current schema, and the authorized user.

When a connection is established, it usually requires an authorization clause, which includes a username and password. This ensures that the current authorization ID has the necessary privileges to perform the requested actions. In this context, a ”Module” refers to the application program code, while a ”SQL Agent” is the actual execution of that code.

7.4 Transactions and the ACID Properties

Transactions are the fundamental units of work in a database system. To ensure that the database remains in a consistent state despite concurrent access or system failures, every transaction must follow a set of requirements known as the ACID properties.

Theory 7.4.1 ACID Properties
A set of four essential characteristics of a transaction: Atomicity (all-or-nothing execution), Consistency (preserving database invariants), Isolation (executing as if in isolation), and Durability (permanent storage of results).

Atomicity ensures that if a transaction is interrupted, any partial changes are rolled back, leaving the database as if the transaction never started. Consistency guarantees that a transaction moves the database from one valid state to another, respecting all defined rules. Isolation is managed by a scheduler to ensure that the concurrent execution of multiple transactions results in a state that could have been achieved if they were run one after another. Finally, Durability ensures that once a transaction is committed, its effects will survive even a subsequent system crash.

7.4.1 Transactional Phenomena and Isolation Levels

When multiple transactions run simultaneously, several problematic phenomena can occur if isolation is not strictly enforced.

1. **Dirty Read**: This happens when one transaction sees data that has been written by another transaction but has not yet been committed. If the first transaction eventually aborts, the data seen by the second transaction effectively never existed. 2. **Nonrepeatable Read**: A transaction reads the same data twice but finds different values because another transaction modified and committed that data in the meantime. 3. **Phantom Read**: A transaction runs a query multiple times and finds "phantom" rows that were inserted by another committed transaction during the process. 4. **Serialization Anomaly**: This occurs when the result of a group of concurrent transactions is inconsistent with any serial ordering of those same transactions.
To manage these risks, SQL defines various "Isolation Levels." The most stringent is "Serializable," which prevents all the aforementioned phenomena. Lower levels, such as "Read Committed" or "Read Uncommitted," allow for higher concurrency at the risk of encountering some of these issues.

7.4.2 Java Database Connectivity (JDBC)

One of the most common ways to implement the application tier is through Java, using the JDBC call-level interface. JDBC allows a Java program to interact with virtually any SQL database by using a standard set of classes and methods.

Definition 7.4.1: JDBC

A Java-based API that provides a standard library of classes for connecting to a database, executing SQL statements, and processing the results.

The process begins by loading a driver for the specific DBMS, such as MySQL or PostgreSQL. Once the driver is loaded, a connection is established using a URL that identifies the database, along with credentials for authorization.
In JDBC, there are different types of statements used to interact with the data. A simple 'Statement' is used for queries without parameters, while a 'PreparedStatement' is used when a query needs to be executed multiple times with different values. These parameters are denoted by question marks in the SQL string and are bound to specific values before execution.
The result of a query in JDBC is returned as a 'ResultSet' object. This object acts like a cursor, allowing the program to iterate through the resulting tuples one at a time using a 'next()' method. For each tuple, the programmer uses specific getter methods, such as 'getInt()' or 'getString()', to extract data based on the attribute's position in the result.

Theory 7.4.2 JDBC Interaction Pattern

The standard flow of database access in Java: Load Driver $\longrightarrow$ Establish Connection $\longrightarrow$ Create Statement $\longrightarrow$ Execute Query/Update $\longrightarrow$ Process Results via ResultSet $\longrightarrow$ Close Connection.

This interface effectively solves the "impedance mismatch" between the set-oriented world of SQL and the object-oriented world of Java. By providing a mechanism to fetch rows individually, it allows Java's iterative control structures to process data retrieved from SQL's relational queries. Furthermore, it supports the execution of updates, which encompass all non-query operations like insertions, deletions, and schema modifications. This robust framework is essential for building the business logic required in the application tier of the three-tier architecture.'''

Chapter 8

Views and Indecies

This chapter explores the conceptual and physical layers of database management, focusing on the mechanisms that allow users to interact with data flexibly while ensuring that the underlying hardware performs at its peak. The discussion is divided into two primary concepts: views and indexes.
Virtual views represent a method of logical abstraction. They allow a database designer to present users with data organized in a way that is most convenient for their specific tasks, without necessarily altering the structure of the base tables where the information is physically stored. These virtual relations are computed on demand and provide a layer of data independence, protecting applications from changes in the underlying schema and offering a simplified interface for complex queries.
On the physical side, indexes are specialized data structures used to circumvent the high cost of exhaustive table scans. By providing direct paths to specific tuples based on the values of search keys, indexes significantly reduce the number of disk accesses required for lookups and joins. However, the creation of an index is not a cost-free operation. It involves a fundamental trade-off between the acceleration of read operations and the increased overhead associated with insertions, deletions, and updates. This summary evaluates the criteria for view updatability, the mechanics of index implementation, and the rigorous cost models used to determine the optimal configuration of physical storage.

8.1 Virtual Views in a Relational Environment

In a standard database, relations created through table declarations are considered persistent or ”base” tables. These structures are physically stored on disk and remain unchanged unless modified by specific commands. In contrast, a virtual view is a relation defined by a SQL expression, typically a query. It does not exist in storage as a set of tuples; instead, its content is dynamically generated whenever it is referenced.

Definition 8.1.1: Virtual View

A named virtual relation defined by a query over one or more existing base tables or other views, which is not physically materialized in the database.

Theory 8.1.1 View Expansion

The query processing mechanism whereby the name of a view in a SQL query is replaced by the query expression that defines it, allowing the system to optimize the operation as if it were performed directly on the base tables.

When a view is defined, the system stores only its definition. From the perspective of a user, the view is indistinguishable from a base table. It possesses a schema and can be the target of queries. Furthermore, attributes in a view can be renamed during declaration to provide clearer identifiers for the end-user. This is particularly useful when the underlying table uses technical or ambiguous column names. For instance, a view might extract movie titles and production years from a comprehensive database to present a simplified list of films belonging to a specific studio.

8.2 Modification and Update Logic for Views

While querying a view is straightforward, modifying one—through insertions, updates, or deletions—is conceptually complex because the view contains no physical tuples. For a modification to be successful, the database management system must be able to translate the request into an equivalent sequence of operations on the underlying base tables.

Definition 8.2.1: Updatable View

A virtual view that is sufficiently simple for the system to automatically map modifications back to the original base relations without ambiguity.

Theory 8.2.1 Criteria for Updatability

To be updatable, a view must generally be defined by a simple selection and projection from a single relation. It cannot involve duplicate elimination, aggregations, or group-by clauses, and it must include all attributes necessary to form a valid tuple in the base relation.

If a view is defined over multiple relations, such as through a join, it is typically not updatable because the logic for handling the change is not unique. For example, if a tuple is deleted from a view joining movies and producers, it is unclear whether the system should delete the movie, the producer, or both. To overcome these limitations, SQL provides "instead-of" triggers. These allow the designer to intercept a modification attempt on a view and define a custom set of actions to be performed on the base tables instead. This ensures that the intended semantics of the operation are preserved regardless of the complexity of the view's definition.

8.3 Physical Indexes and Retrieval Performance

The efficiency of data retrieval is largely determined by the number of disk blocks the system must access. Without an index, the database must perform a full scan of a relation to find specific tuples. For large relations spanning thousands of blocks, this process is prohibitively slow. An index is a physical structure that maps values of a search key to the physical locations of the tuples containing those values.

Definition 8.3.1: Index

A physical data structure designed to accelerate the location of tuples within a relation based on specified attribute values, bypassing the need for an exhaustive scan of all blocks.

Definition 8.3.2: Multi-attribute Index

An index built on a combination of two or more attributes, allowing the system to efficiently find tuples when values for all or a prefix of those attributes are provided in a query.

Indexes are most commonly implemented as B-trees or hash tables. A B-tree is a balanced structure where every path from the root to a leaf is of equal length, ensuring predictable performance for both point lookups and range queries. In most modern systems, the B+ tree variant is used, where pointers to the actual data records are stored only at the leaf nodes. This structure allows the system to navigate through the index by comparing search keys, moving from a root block down to the appropriate leaf with minimal disk I/O.

8.4 Selection and Performance Analysis of Indexes

The decision of whether to build an index on a particular attribute requires a careful analysis of the expected workload. While an index speeds up queries, every modification to the underlying relation requires a corresponding update to the index. This secondary update involves reading and writing index blocks, which can double the cost of insertions and deletions.

Definition 8.4.1: Clustering Index

An index where the physical order of the tuples on disk corresponds to the order of the index entries, ensuring that all tuples with a specific key value are stored on as few blocks as possible.

Theory 8.4.1 The Index Selection Trade-off

The process of evaluating whether the time saved during the execution of frequent queries outweighs the time lost during the maintenance of the index for insertions, updates, and deletions.

To make this determination, database administrators use a cost model centered on disk I/O. If a relation is clustered on an attribute, the cost of retrieving all tuples with a specific value is approximately the number of blocks occupied by the relation divided by the number of distinct values of that attribute. If the index is non-clustering, each retrieved tuple may potentially reside on a different block, leading to a much higher retrieval cost. A tuning advisor or administrator will calculate the average cost of all anticipated operations (queries and updates) to decide which set of indexes minimizes the total weighted cost for the system.

8.5 Materialized Views and Automated Tuning

Beyond virtual views and physical indexes, database systems often employ materialized views. Unlike a virtual view, a materialized view is physically computed and stored on disk. This approach is beneficial for high-complexity queries that are executed frequently, such as those involving expensive joins or aggregations in a data warehousing environment.

Definition 8.5.1: Materialized View

A view whose query result is physically stored in the database, requiring an explicit maintenance strategy to synchronize its content with changes in the base tables.

The use of materialized views introduces a maintenance cost similar to that of indexes. Every time a base table changes, the materialized view must be updated, either immediately or on a periodic schedule. Because the number of possible views is virtually infinite, modern systems use automated tuning advisors. These tools analyze a query log to identify representative workloads and then use a greedy algorithm to recommend the combination of indexes and materialized views that will provide the greatest overall benefit to the system's performance.

8.6 Strategic Balance in Database Design

The successful implementation of a database requires a strategic balance between logical flexibility and physical efficiency. Virtual views provide the necessary abstraction to simplify application development and manage data security. Meanwhile, the careful selection of indexes and materialized views ensures that the system remains responsive as the volume of data grows.

By employing a formal cost model based on disk access times, designers can objectively evaluate the merits of different storage configurations. The goal is to reach a state where the most frequent and critical operations are prioritized, even if it necessitates a penalty for less common tasks. This continuous process of tuning and optimization is a hallmark of modern relational database management, allowing these systems to handle massive datasets while providing the illusion of instantaneous access to information. An index on a primary key, for example, is almost always beneficial because it is frequently queried and guarantees that only a single block needs to be retrieved to find a unique record. In contrast, an index on a frequently updated non-key attribute requires a more nuanced analysis to ensure it does not become a performance bottleneck.

Ultimately, the choice of views and indexes defines the operational efficiency of the entire information system. A well-designed logical and physical schema acts as the foundation for scalable, high-performance applications, enabling efficient data exploration and robust transaction processing in even the most demanding environments.“‘

Chapter 9

Data Cubes

The evolution of data management has transitioned through distinct eras, beginning with the Age of Transactions in the late 20th century, moving into the Age of Business Intelligence in the mid-1990s, and culminating in the modern Age of Big Data. This progression reflects a shift in focus from simple record-keeping to complex data-based decision support. Central to this transition is the distinction between two primary operational paradigms: Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP). While OLTP systems are designed for consistent and reliable record-keeping through numerous small, fast writes, OLAP systems are built for historical, summarized, and consolidated data analysis characterized by long, heavy, and complex aggregating queries.

To bridge the gap between raw data storage and analytical utility, database systems utilize specific architectural components. Views provide a logical layer over base tables, allowing users to interact with data without needing to manage its physical storage. Meanwhile, indexes serve as the engine for performance, enabling the rapid retrieval of specific tuples from massive datasets. In the context of OLAP, these concepts are expanded into the multidimensional data model, often visualized as a data cube.

Definition 9.0.1: Online Transaction Processing (OLTP)

A database paradigm focused on managing transaction-oriented applications. It is characterized by a high volume of short, fast, and concurrent transactions, primarily involving writes to small portions of normalized data to ensure consistent and reliable record-keeping.

Definition 9.0.2: Online Analytical Processing (OLAP)

A paradigm designed to support multi-dimensional data analysis for decision-making. It typically involves complex, long-running queries on large portions of consolidated, historical data, often stored in denormalized structures like data cubes.

9.1 Virtual Views and Data Interfaces

Virtual views are a cornerstone of modern database management, acting as relations that are defined by a query rather than being stored physically on a disk. These views exist only logically; when a user queries a view, the system's query processor substitutes the view name with its underlying definition. This mechanism offers several advantages, including simplified query writing for end-users and enhanced security by restricting access to sensitive columns of a base table.

Theory 9.1.1 View Preprocessing

The process by which a preprocessor replaces an operand in a query that is a virtual view with a piece of a parse tree or expression tree representing the view's construction from base tables. This allows the query to be interpreted entirely in terms of physical storage.

The management of views also extends to their modification. While most views are read-only, certain "updatable views" allow for insertions, deletions, or updates that are passed through to the underlying base tables. For a view to be considered updatable by standard SQL rules, it must generally be defined over a single relation without the use of aggregation or distinct clauses. Furthermore, database designers can use specialized triggers to define how modifications to a view should be handled if the standard pass-through logic is insufficient.

Definition 9.1.1: Instead-Of Trigger

A specialized trigger defined on a virtual view that intercepts an attempted modification (INSERT, UPDATE, or DELETE). Instead of executing the modification on the view, the system executes a predefined sequence of actions on the underlying base tables to achieve the intended result.

9.2 Performance Optimization through Indexing

As databases grow into the terabyte and petabyte range, the cost of scanning every block of a relation to find specific information becomes prohibitive. Indexes are auxiliary data structures designed to mitigate this cost by allowing the system to locate tuples with specific search-key values without a full table scan. The primary motivation for indexing is the reduction of disk I/O, which is the dominant cost in query execution.

Theory 9.2.1 Clustering Index Advantage

An index is considered clustering if all tuples with a specific search-key value are stored on as few disk blocks as possible. A clustering index typically provides a massive speedup for range queries and selections because once the first matching block is found, the system can read subsequent matching tuples with minimal additional seek time or rotational latency.

Selecting the appropriate set of indexes is one of the most critical decisions for a database administrator. While indexes significantly accelerate read-heavy OLAP queries, they impose a maintenance penalty on OLTP operations. Every time a tuple is inserted, deleted, or updated, the associated indexes must also be modified, requiring additional disk writes. Therefore, the optimal indexing strategy depends on the specific workload—balancing the frequency of specific query forms against the frequency of modifications.

Definition 9.2.1: Secondary Index

An index that does not determine the physical placement of records in the data file. Secondary indexes are necessarily dense, meaning they contain pointers to every record in the data file to facilitate retrieval by non-primary attributes.

9.3 The Multidimensional Data Model

In the analytical realm, data is often viewed through a multidimensional lens rather than as flat tables. This model organizes information around "facts"—events of interest like a specific sale—and "dimensions," which are the axes of the data space, such as time, location, or product type. This structure allows analysts to "slice and dice" data to find patterns.

Definition 9.3.1: Data Cube

A multidimensional representation of data where each point represents a fact and the axes represent various dimensions of the data. A formal data cube includes not only the raw data but also precomputed aggregations across all subsets of dimensions.

To support this model, ROLAP systems often use a "star schema." In this architecture, a central fact table contains the quantitative measures and foreign keys referencing "dimension tables." Dimension tables store descriptive information about the axes of the cube. If these dimension tables are further normalized, the structure is referred to as a "snowflake schema."

Theory 9.3.1 Slicing and Dicing

Slicing is the act of picking a specific value for one or more dimensions to focus on a particular subset of the cube. Dicing involves selecting ranges for several dimensions to define a smaller, focused sub-cube for analysis.

9.4 Data Cube Operations and Implementation

Navigating a data cube requires operations that change the level of granularity. ”Roll-up” is the process of moving from a fine-grained view to a more summarized view by aggregating along a dimension (e.g., viewing sales by year instead of by month). Conversely, ”drill-down” is the process of moving from a summarized view to a more detailed one.

The implementation of these operations varies between Relational OLAP (ROLAP) and Multidimensional OLAP (MOLAP). ROLAP utilizes standard relational tables and extended SQL operators, whereas MOLAP uses specialized, non-relational structures that store the cube and its aggregates directly. One of the most powerful tools in this environment is the CUBE operator.

Definition 9.4.1: CUBE Operator

An extension of the GROUP BY clause that computes all possible aggregations across a set of dimensions. It effectively augments a fact table with ”border” values representing ”any” or summarized totals for every combination of the specified attributes.

When implemented in SQL, the CUBE operator produces a result set where the ”all” or summarized values are represented by NULLs in the grouping columns. This allows a single query to return the detailed data, subtotals for every dimension, and a grand total for the entire dataset. To manage the potentially explosive growth of data in a cube, many systems use ”materialized views,” which are views whose results are physically stored on disk and incrementally updated as the base data changes.

Theory 9.4.1 The Thomas Write Rule

A principle in concurrency control that allows certain writes to be skipped if a write with a later timestamp is already in place, assuming that no other transaction needs to see the skipped value. This is relevant in the context of maintaining analytical data consistent with temporal versions.

In conclusion, the data cube represents a sophisticated integration of logical views, performance indexing, and multidimensional modeling. By leveraging these structures, database systems can provide the interactive, high-speed analysis required for modern decision support, even when operating on the vast scales of contemporary data warehouses.

Chapter 10

Database Architecture

Database architecture serves as the structural foundation that bridges high-level data models with the physical realities of computer hardware. It encompasses a spectrum of components ranging from the multi-layered memory hierarchy used to store bits and bytes to the logical abstractions like virtual views that provide tailored interfaces for users. A central goal of this architecture is to maintain data independence, allowing the underlying storage methods to change without affecting how application programs interact with the data. This summary explores the mechanics of physical storage, the implementation of virtual relations, and the optimization techniques involving index structures to ensure efficient data retrieval and system resilience.

10.1 The Storage and Memory Hierarchy

Behind the daily operations of a database system lies a sophisticated hardware hierarchy designed to manage the trade-off between access speed and storage capacity. At the fastest and most expensive level is the processor's internal cache (Levels 1 and 2), providing almost instantaneous access to data measured in nanoseconds. Below this is the main memory, or RAM, which acts as the primary workspace for the Database Management System (DBMS). While RAM provides significant capacity, it is volatile, meaning its contents are lost if power is interrupted. This volatility is a critical consideration for the ACID properties of transactions, specifically durability.

Secondary storage, primarily consisting of magnetic disks, serves as the non-volatile repository where data persists. Accessing disks involves mechanical movements, introducing latencies in the millisecond range, which is orders of magnitude slower than RAM. For massive data sets that exceed disk capacity, tertiary storage such as tapes or DVDs is utilized, offering enormous capacity (terabyte to petabyte range) at the cost of significantly longer access times.

Definition 10.1.1: Memory Hierarchy

A systematic organization of storage devices in a computer system, ranked by speed, capacity, and cost per bit, typically including cache, main memory, and secondary/tertiary storage.

Theory 10.1.1 The Dominance of I/O

The performance of a database system is largely determined by the number of disk I/O operations performed, as the time required to access a block on disk is significantly greater than the time needed to process data in main memory.

10.2 Physical Data Representation and PGDATA

The physical storage of a database is typically organized into a specific directory structure on the host machine, often referred to as the data directory or PGDATA. This directory contains the actual files representing tables, configuration parameters, and transaction logs. To manage this data effectively, the DBMS divides information

into blocks or pages, which are the fundamental units of transfer between disk and memory. In systems like PostgreSQL, these pages are usually 8KB in size.
Data within these pages is organized into records. Fixed-length records are straightforward to manage, but variable-length fields require more complex structures like offset tables within the block header. When fields become exceptionally large, such as multi-gigabyte video files or documents, techniques like TOAST (The Oversized-Attribute Storage Technique) are employed to store these values in separate chunks, preventing them from bloating the primary data pages.

10.3 Virtual Views and Data Abstraction

Virtual views are relations that do not exist physically in the database but are instead defined by a stored query over base tables. These views provide a layer of abstraction, allowing different users to see the data in formats that suit their specific needs without duplicating the underlying information. When a user queries a view, the system's query processor substitutes the view name with its corresponding definition, effectively transforming the query into one that operates directly on the stored base tables.

Definition 10.3.1: Virtual View

A relation that is not stored in the database but is defined by an expression that constructs it from other relations whenever it is needed.

Attributes within a view can be renamed for clarity using the AS keyword or by listing names in the declaration. This allows the architect to present a clean logical model to the end-user while hiding the complexity of the underlying join operations or attribute names.

10.4 Modification of Virtual Relations

Modifying a view is inherently more complex than modifying a base table because the system must determine how to map changes to the underlying physical data. SQL allows for "updatable views" under specific conditions: the view must be defined over a single relation, it cannot use aggregation or duplicate elimination, and the selection criteria must be simple enough that the system can unambiguously identify which base tuples are affected.

Definition 10.4.1: Updatable View

A virtual view that is sufficiently simple to allow insertions, deletions, and updates to be translated directly into equivalent modifications on the underlying base relation.

For more complex views that involve multiple tables or aggregations, "instead-of" triggers provide a solution. These triggers intercept a modification attempt on a view and execute a custom piece of logic—written by the database designer—that appropriately updates the base tables.

10.5 Index Structures and Motivation

As database relations grow, scanning every block to find specific information becomes prohibitively slow. Indexes are specialized data structures designed to accelerate this process. An index takes the value of a specific attribute, known as the search key, and provides pointers directly to the records containing that value.

Definition 10.5.1: Index

A stored data structure that facilitates the efficient retrieval of records in a relation based on the values of one or more attributes.

The primary motivation for indexing is the reduction of disk I/O. For example, finding a specific movie in a massive relation is much faster if the system can use an index on the title rather than performing a full table scan. In joins, indexes on the join attributes can allow the system to look up only the relevant matching tuples, avoiding the exhaustive pairing of every row from both relations.

10.6 Strategic Selection of Indexes

While indexes speed up queries, they impose a cost: every time a record is inserted, deleted, or updated, the associated indexes must also be modified. This creates a strategic trade-off for the database architect. A clustering index, where the physical order of records matches the index order, is exceptionally efficient for range queries as it minimizes the number of blocks that must be read. Non-clustering indexes are useful for locating individual records but may require many disk accesses if many rows match the search key, as the records might be scattered across different blocks.

Theory 10.6.1 Index Cost-Benefit Analysis

The decision to create an index depends on the ratio of queries to modifications; an index is beneficial if the time saved during data retrieval exceeds the additional time required to maintain the index during updates.

Architects often use a cost model based on the number of disk I/O's to evaluate the utility of a proposed index. This model considers factors like the number of tuples (T), the number of blocks (B), and the number of distinct values for an attribute (V).

10.7 Historical Foundations of Relational Theory

The modern concept of database architecture and data independence traces back to the seminal work of Edgar Codd in 1970. Codd's introduction of the relational model revolutionized the field by suggesting that data should be viewed as sets of tuples in tables, independent of their physical storage. This shift allowed for the development of high-level query languages like SQL and sophisticated optimization techniques that define the current state of database management systems. Subsequent research into integrity checking and semistructured data models like XML continues to build upon these relational foundations.

Definition 10.7.1: Data Independence

The principle that the logical structure of data (the schema) should be separated from its physical storage, ensuring that changes to the storage method do not require changes to application programs.