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, 7th January 2026

Chapter 1

Introduction

Modern engineering increasingly relies on the structured management of information, treating data as the fundamental digital substance equivalent to physical matter. In the pursuit of understanding the world, we can categorize scientific inquiry into a matrix of paradigms. While mathematics explores necessary truths through natural thought and computer science analyzes the theoretical necessity of artificial computation, physics observes the world as it exists. Data science effectively acts as the "physics of computer science," utilizing machine-driven computation to observe and interpret the world through empirical evidence.
The objective of an information system is to transform raw observations into actionable intelligence. This process follows a strict hierarchy. Data consists of raw, uninterpreted facts that are stored and moved between systems. When these facts are associated with specific meanings, they become information. Finally, when this information is applied to meaningful tasks or decision-making, it evolves into knowledge.

Definition 1.0.1: Information System

A software program or a synchronized set of programs designed to manage, store, and provide efficient access to information.

Theory 1.0.1 The Knowledge Hierarchy

The structured progression from raw data to information through added meaning, culminating in knowledge through practical application.

Notes:-

In modern engineering, making superior decisions is no longer just about observing numbers but about leveraging knowledge derived through information systems.

1.1 The Historical Evolution of Data Management

The history of data management is a narrative of scaling human memory and communication. Before the advent of technology, information was transmitted via oral traditions, which were hindered by the limitations of human recall and distance. The invention of writing marked the first major turning point, allowing symbols to be preserved on durable media such as stone or clay.
Ancient civilizations intuitively adopted the tabular format for data. Clay tablets from thousands of years ago have been discovered containing relational data, such as Pythagorean triples, organized in rows and columns. This indicates that tables are a primary cognitive tool for human information organization. The invention of the printing press in the 16th century further enabled the mass distribution of data, leading eventually to the mechanical and electronic computing revolutions of the 20th century.
In the early decades of computing, specifically the 1960s, data management was handled through direct file systems. Programmers were required to know the physical location of data on a disk and write complex logic to retrieve it. This changed in 1970 when Edgar Codd introduced the relational model. He argued that users should interact with data through intuitive tables, while the underlying machine complexities remained hidden.

This principle of data independence paved the way for the Object Era in the 1980s and the NoSQL Era in the 2000s, the latter of which was driven by the massive scale of modern social networks and search engines.

Notes:-

The tabular format has remained the most intuitive and enduring method for humans to represent structured data, from ancient clay to modern SQL.

1.2 The Structure and Shapes of Data

Data is categorized based on its degree of organization. Unstructured data, such as natural language text, audio, images, and video, exists in a raw form that was historically difficult for computers to process. However, recent breakthroughs in linear algebra and vector-based mathematics have enabled modern systems to interpret and even generate this type of content.
Structured data is the highly organized information typically found in spreadsheets and relational databases. Between these lies semi-structured data, which uses tags (like XML or JSON) to provide some semantic context without the rigid requirements of a fixed schema. To manage these types, engineers utilize data models—mathematical notations for describing data structures, the operations allowed on them, and the constraints they must follow.

Definition 1.2.1: Data Model

A formal notation that describes the structure of data, the methods for querying and modifying it, and the rules that maintain its integrity.

Theory 1.2.1 The Three Vs of Big Data

The defining challenges of modern data management are Volume (the sheer amount of bytes), Variety (the diversity of data types), and Velocity (the speed at which data is generated and must be processed).

1.3 The Necessity of Database Management Systems

In primitive computing environments, applications directly accessed files on local disks. This approach resulted in severe problems as systems grew. Data was often redundant (stored in multiple places) and inconsistent (versions of the same data conflicting). It was also difficult to combine data from different sources or control who had access to specific information.
A Database Management System (DBMS) resolves these issues by serving as a central software layer. A robust DBMS is expected to fulfill five primary roles:

1. Allow users to define the structure (schema) of new databases.
2. Provide high-level languages for querying and changing data.
3. Facilitate the storage of massive datasets over long durations.
4. Ensure durability by recovering data after system failures.
5. Manage concurrent access by multiple users to prevent data corruption.

Definition 1.3.1: Database Management System (DBMS)

A specialized software suite used to create, manage, and query databases, shielding the user from physical storage details.

Notes:-

A "Database System" is the holistic term for the combination of the DBMS software and the actual data stored within it.

1.4 System Architecture and Data Independence

Most modern information systems utilize a three-tier architecture to ensure modularity and scalability. The top layer is the User Interface (UI), which handles human interaction. The middle layer is the Business Logic, where the rules of the application are processed. The bottom layer is the Persistence layer, where the DBMS manages data storage on a disk or in the cloud.
The most vital concept within this architecture is data independence, championed by Edgar Codd. This principle separates the logical level (the tables humans see) from the physical level (the bits stored on the machine). Because of this separation, an engineer can change the physical storage medium—from a hard drive to a data center or even DNA storage—without the user ever needing to change their queries.

Definition 1.4.1: Data Independence

The ability of a database system to provide a stable logical view of data that is entirely independent of its physical storage implementation.

Theory 1.4.1 Three-Tier Architecture

A design pattern that divides an application into the presentation, logic, and data management layers to simplify development and maintenance.

1.5 Query Languages and Internal Processes

Interaction with a DBMS occurs through specialized languages. The Data Definition Language (DDL) is used to define metadata—the "data about the data," such as the names of columns and their types. The Data Manipulation Language (DML), primarily SQL, is used to search for or update actual records.
SQL is distinct because it is a declarative language. In imperative languages like C++ or Python, a programmer must write the step-by-step instructions for how to perform a task. In a declarative language, the user only describes what result they want. The DBMS uses a query compiler to analyze the request and an execution engine to find the most efficient path—the "query plan"—to retrieve the data.

Notes:-

The efficiency of modern databases is largely due to the query compiler's ability to optimize a declarative request into a high-performance execution strategy.

1.6 Measurement and Scaling in the Era of Big Data

The scale of data generated today is exponential, often said to double every few years. Engineers must be familiar with the international system of units for volume. While standard kilo and mega represent powers of 10 (10^3 and 10^6), computer science often relies on binary prefixes like kibi ($2^{10} = 1024$) and mebi (2^{20}) to ensure precision in memory and storage calculations. We are now entering the age of Zettabytes and Yottabytes, requiring a deep understanding of how to scale information systems to meet these unprecedented demands.

Notes:-

The total amount of data created in just the last few years is estimated to be greater than the sum of all information produced in the entirety of previous human history.

Chapter 2

The Relational Model

The relational model serves as the theoretical cornerstone of modern database systems, providing a structured yet flexible framework for data management. Proposed by Edgar Codd in 1970, this model revolutionized the field by introducing the principle of data independence. This principle decouples the logical representation of data—how users perceive and interact with it—from its physical storage on hardware. By representing information through intuitive two-dimensional tables, the model bridges the gap between complex mathematical theory and practical business applications. Interestingly, the tabular format is not a modern invention; historical evidence shows that humans have used clay tablets for relational data organization since at least 1800 BC. This enduring utility underscores the model's alignment with human cognitive patterns for managing structured facts.

Theory 2.0.1 Data Independence

The separation of the logical data model from the physical storage implementation, allowing changes to the machine-level storage without affecting user queries or the logical view of the data.

2.1 Core Terminology and Structural Components

In relational theory, specific terminology is used to describe the components of a database, often with synonyms used across different technical and business contexts. The primary structure is the relation, commonly referred to as a table. A relation consists of a set of attributes, which are the named columns that define the properties of the data stored. The set of these attributes, combined with the name of the relation itself, constitutes the relation schema.

Definition 2.1.1: Relation Schema

The formal description of a relation, comprising its name and a set of attributes, typically denoted as $R(A_1, A_2, \ldots, A_n)$.

Each entry within a relation is called a tuple, which corresponds to a row in a table or a record in a file. A tuple contains a specific value for each attribute defined in the schema. These values, often called scalars, represent individual facts or characteristics.

Definition 2.1.2: Tuple

A single row or record within a relation, representing a specific instance of the entity or business object described by the schema.

Notes:-

While mathematicians prefer to index tuples by numbers, database scientists identify components by their attribute names to provide semantic clarity.

2.2 Domains and Atomic Values

Every attribute in a relation is associated with a domain. A domain is essentially a data type or a set of permissible values that can appear in a specific column. For example, a "year" attribute might be restricted to the domain of integers, while a "name" attribute is restricted to the domain of character strings. A fundamental requirement of the standard relational model is that these values must be atomic. This means they cannot be further decomposed into smaller components, such as nested tables, lists, or sets. This requirement is formally known as the First Normal Form.

Definition 2.2.1: Domain

A set of values of a specific elementary type from which an attribute draws its components.

Theory 2.2.1 Atomic Integrity

The rule that every component of every tuple must be an indivisible, elementary value rather than a structured or repeating group.

2.3 Mathematical Foundations of Relations

The relational model is built upon the mathematical concept of the Cartesian product. Given a family of domains $D_1, D_2, \ldots, D_n$, a relation is defined as a subset of the Cartesian product $D_1 \times D_2 \times \cdots \times D_n$. Each element of this subset is an n-tuple. This mathematical approach ensures that domain integrity and relational integrity are maintained by definition, as every value must belong to its prescribed set.

Definition 2.3.1: Cartesian Product

The set of all possible ordered tuples that can be formed by taking one element from each of the participating sets or domains.

An alternative mathematical representation views a record as a map. In this perspective, a record t is a partial function from a set of attribute names to a global set of values. This mapping approach is often preferred because it makes the order of attributes irrelevant, reflecting how databases actually operate in practice.

Notes:-

In a relation, the order of both the attributes and the tuples is immaterial; a relation remains the same regardless of how its rows or columns are permuted.

2.4 Integrity and Consistency Rules

For a collection of data to be considered a valid relational table, it must adhere to three primary integrity rules. These rules ensure the consistency and predictability of the data.

1. **Relational Integrity:** This requires that all records within a specific table have the exact same set of attributes. A table cannot have "holes" or missing attributes in some rows but not others.
2. **Atomic Integrity:** As previously noted, this prohibits the nesting of structures within a cell. A value must be a single fact.
3. **Domain Integrity:** This ensures that every value in a column is of the same kind, matching the type specified for that attribute in the schema.

Definition 2.4.1: Domain Integrity

The constraint that every value in a specific column must belong to the domain (data type) associated with that attribute.

Theory 2.4.1 Relational Integrity

The requirement that every record in a relation must possess the same support, meaning they all share the identical set of attributes defined in the schema.

2.5 Keys and Uniqueness

To distinguish between tuples, the relational model relies on the concept of keys. A key is a set of one or more attributes that uniquely identifies a tuple within a relation instance. No two tuples in a valid relation can share the same values for all attributes in the key. Typically, one key is designated as the primary key.

Definition 2.5.1: Primary Key

A specific attribute or minimal set of attributes chosen to uniquely identify each tuple in a relation, often indicated in a schema by underlining the attributes.

Notes:-

Identifying a primary key is essential for establishing relationships between different tables and maintaining data accuracy.

2.6 Relation Instances and Temporal Change

A relation is not a static object; it changes over time as tuples are inserted, deleted, or updated. The set of tuples present in a relation at any given moment is called an instance. Standard database systems typically only maintain the "current instance," representing the data as it exists right now. Changing a schema (adding or deleting columns) is a much more significant and expensive operation than changing an instance, as it requires restructuring every tuple currently stored.

Definition 2.6.1: Relation Instance

The specific set of tuples contained within a relation at a given point in time.

2.7 Alternative Storage Semantics

While the classical relational model is based on set semantics, where duplicate tuples are strictly forbidden, practical implementations often utilize different semantics based on the needs of the system.

1. **Set Semantics:** No duplicate records are allowed.
2. **Bag Semantics:** Duplicate records are permitted. This is common in SQL results, as eliminating duplicates is computationally expensive.
3. **List Semantics:** The specific order of the records is preserved and significant.

Theory 2.7.1 Bag Semantics

A variation of the relational model where duplicate tuples are allowed to exist within a relation, often used to improve the efficiency of query operations.

Notes:-

The choice between set, bag, and list semantics is often a trade-off between mathematical purity and the performance requirements of a real-world database engine.

2.8 Conclusion

The relational model's power lies in its simplicity and its firm mathematical grounding. By treating data as a collection of relations and providing a clear set of integrity rules, it allows for the creation of robust, scalable information systems. The use of schemas provides a stable contract for applications, while the principle of data independence ensures that the system can evolve technologically without breaking the logical structures that users depend on.

Notes:-

The relational model effectively acts as the "physics" of data, providing the laws that govern how digital information is structured and transformed.

Chapter 3

Data Definition with SQL

3.1 Overview of Data Definition

Data definition is the fundamental process of specifying the logical structure of a database, often referred to as the schema. In the context of SQL (Structured Query Language), this involves declaring the tables that will store information, identifying the types of data permitted in each column, and establishing rules to maintain the correctness and consistency of that data. The relational model, which serves as the foundation for modern database systems, represents information as two-dimensional tables called relations. By defining these relations, developers create a rigid framework that ensures data independence, allowing the underlying physical storage to be optimized without affecting the high-level queries used by applications.

3.2 Mathematical Foundations of Relations

The concept of a relational table originates from mathematical set theory. At its core, a relation is defined over a series of sets, which are known as attribute domains. While a general relation can represent any subset of a Cartesian product, SQL tables require more specific semantic structures to function effectively as data stores.
A collection of data can be viewed as a set of records, where each record acts as a "map" or a function from a set of attributes to values. To transform a simple collection into a formal relational table, three specific types of integrity must be enforced.

Definition 3.2.1: Relational Table (Set Semantics)

A relational table is a set of maps from attribute names to values that satisfies relational integrity, domain integrity, and atomic integrity.

Theory 3.2.1 The Three Rules of Relational Integrity

To qualify as a relational table, a collection must fulfill:

1. **Relational Integrity**: Every record in the collection must have the same support, meaning they all share the exact same set of attributes.
2. **Domain Integrity**: Each attribute is associated with a specific domain (type), and every value for that attribute must belong to that domain.
3. **Atomic Integrity**: Every value in the table must be atomic, meaning it cannot be broken down into smaller components (e.g., no tables within tables).

Notes:-

While mathematics primarily uses set semantics (no duplicates), practical SQL implementations often utilize bag semantics, allowing for duplicate records, or list semantics, where the order of records is preserved.

3.3 The Evolution and Nature of SQL

SQL was developed in the early 1970s at IBM's San Jose research facility, originally under the name SEQUEL (Structured English Query Language). Created by Don Chamberlin and Raymond Boyce, the language was designed to be more intuitive than earlier procedural languages by using English-like syntax.
The primary characteristic of SQL is that it is a declarative language. Unlike imperative languages such as Java or C++, where the programmer must define exactly how to retrieve or calculate data, a SQL user simply declares what the desired result looks like. The database engine then determines the most efficient way to execute the request.

Theory 3.3.1 Set-Based Processing

SQL is a set-based language, meaning it manipulates entire relations with a single command rather than processing one record at a time.

3.4 SQL Data Types and Domains

Every attribute in a SQL table must be assigned a data type. These types define the nature of the data and the operations that can be performed on it.

3.4.1 String Types

SQL provides several ways to store text. Fixed-length strings are defined as `char(n)`, where the system reserves exactly n characters. If the input is shorter, it is padded with spaces. Variable-length strings with a specified limit are defined as `varchar(n)`. For very long text without a specific limit, PostgreSQL uses the `text` type, while the SQL standard refers to this as `clob` (Character Large Object).

3.4.2 Numeric Types

Numbers are categorized into exact and approximate types. Exact numbers include integers (`smallint`, `integer`, `bigint`) and fixed-point decimals.

Definition 3.4.1: Fixed-Point Decimal

A numeric type defined by `decimal(p, s)`, where p is the total number of significant digits (precision) and s is the number of digits after the decimal point (scale).

Approximate numbers are represented as floating-point values using `real` (single precision) or `double precision`. These follow the IEEE 754 standard and are highly efficient because they are handled directly by computer hardware.

3.4.3 Temporal and Binary Types

SQL supports complex date and time tracking. The `date` type follows the Gregorian calendar, while `time` tracks hours, minutes, and seconds. `timestamp` combines both, and can optionally include time zone data to handle global information.

Notes:-

The `interval` type represents a duration. However, there is a "duration wall" between months and days because the number of days in a month is variable, making certain additions ambiguous.

Binary data, such as images or videos, is stored using `binary(p)`, `varbinary(p)`, or `blob` (referred to as `bytea` in PostgreSQL).

3.5 Structural Management of Tables

The Data Definition Language (DDL) subset of SQL provides commands to manage the lifecycle of tables.

3.5.1 Creating and Dropping Tables

The `CREATE TABLE` statement is used to define a new relation. It requires a unique table name, a list of attributes, and their associated domains. A newly created table is initially empty.

Notes:-

In SQL, names are generally case-insensitive. However, if a developer needs to force a specific case for an attribute name, they must surround it with double quotes. Single quotes are reserved exclusively for string literals (values).

To remove a table entirely from the database, the `DROP TABLE` command is used. If there is uncertainty about whether a table exists, the `IF EXISTS` clause can be added to prevent execution errors.

3.5.2 Modifying Tables

The `ALTER TABLE` command allows for changes to an existing table's schema. This includes adding new columns, removing existing ones, or renaming attributes and the table itself.

Theory 3.5.1 Adding Columns to Populated Tables

When a new column is added to a table that already contains data, the system must fill the new attribute for existing rows. By default, it uses `NULL`, but a specific `DEFAULT` value can be specified instead.

3.6 Data Population and Manipulation

While data modification is primarily part of the Data Manipulation Language (DML), it is closely tied to definition through constraints.

3.6.1 Insertion Strategies

The most basic way to populate a table is the `INSERT INTO` statement followed by `VALUES`. One can insert a single record or multiple records in one command. If certain columns are omitted, the system will attempt to fill them with `NULL` or a defined default value.

Theory 3.6.1 Insertion via Subqueries

Instead of providing explicit values, an `INSERT` statement can use a `SELECT` subquery to compute a set of tuples from other tables and insert them into the target relation.

3.6.2 Updates and Deletions

Data can be modified using `UPDATE`, which changes values in existing tuples based on a condition, or removed using `DELETE FROM`, which deletes specific rows while keeping the table's structure intact.

3.7 Consistency and Integrity Constraints

Constraints are rules used to prevent the entry of invalid data, effectively enforcing relational and domain integrity at the database level.

Definition 3.7.1: NULL Value

A special marker used in SQL to indicate that a data value is unknown, inapplicable, or kept secret. It is not equivalent to zero or an empty string.

3.7.1 Fundamental Constraints

- **NOT NULL**: This ensures that a column cannot have an empty or unknown value. This is a primary tool for pushing a database toward strict relational integrity.
- **UNIQUE**: This requires that every non-null value in a column be distinct. It can be applied to a single column or a combination of columns (table constraint).
- **CHECK**: This allows for arbitrary conditions that every row must satisfy, such as ensuring a price is positive or a date is within a valid range.

3.8 Primary and Foreign Keys

Keys are the most critical constraints in relational design as they define how records are identified and linked.

3.8.1 Primary Keys

A primary key is an attribute or set of attributes that uniquely identifies a row. By definition, a primary key must be `UNIQUE` and `NOT NULL`. Every table should ideally have one primary key to ensure each record can be referenced without ambiguity.

3.8.2 Foreign Keys and Referential Integrity

A foreign key is an attribute in one table that references a unique or primary key in another table. This creates a link between the two relations.

> **Theory 3.8.1** Referential Integrity
>
> This constraint ensures that every value in a foreign key column must either be `NULL` or exist in the referenced primary key column of the related table.

3.8.3 Handling Deletions in References

If a referenced value is deleted, the system must follow a specific policy to maintain integrity.

Notes:-

Common policies for handling the deletion of a referenced record include:

- **CASCADE**: Automatically delete or update the referencing rows.
- **RESTRICT/NO ACTION**: Prohibit the deletion if references exist.
- **SET NULL**: Reset the foreign key of the referencing rows to `NULL`.
- **SET DEFAULT**: Reset the foreign key to its default value.

3.9 Lexical vs. Value Space

A sophisticated concept in data definition is the distinction between how data is represented and what it actually is. The "Value Space" refers to the abstract mathematical object (e.g., the concept of the number four), while the "Lexical Space" refers to the various ways that value can be written in a query (e.g., '4', '4.0', '04', or '4e0'). SQL engines are responsible for mapping various lexical representations to the correct underlying value space to perform comparisons and arithmetic accurately.

Chapter 4

Relational Algebra

Relational algebra serves as the formal mathematical foundation for manipulating data within the relational model. While Data Definition Language (DDL) is concerned with the static structure of the database, relational algebra provides the dynamic framework for the Data Manipulation Language (DML). It is a notation consisting of a set of operators that take one or more relations as input and produce a new relation as output. This operational approach allows for the expression of complex queries by nesting and combining simpler operations.
The power of relational algebra lies in its ability to abstract away from the physical storage of data, focusing instead on the logical transformation of information. It is essentially to relational tables what basic arithmetic is to numbers or matrix algebra is to vectors. By defining precise rules for how tables are filtered, combined, and restructured, it ensures that query results are predictable and mathematically sound.

4.1 The Concept of Relational Variables and Closure

A fundamental aspect of relational algebra is the way it identifies and treats data structures. In most mathematical contexts, an object does not inherently know the variable name assigned to it. However, in database theory, we often use the term "relvar" (relational variable) to describe a relation that is explicitly associated with a name. This allows the system to refer to stored data and intermediate results throughout the execution of a query.
Another critical property of relational algebra is closure. This principle dictates that because the input and output of every algebraic operator is a relation, operators can be nested indefinitely. This is identical to how the addition of two integers always results in another integer, allowing for the construction of complex arithmetic expressions.

Theory 4.1.1 The Property of Closure

In relational algebra, the result of any operation is always another relation. This ensures that the output of one operator can serve as the valid input for any subsequent operator in a query tree.

Definition 4.1.1: Relational Variable (Relvar)

A relvar is a named variable that is assigned a specific relation as its value, effectively allowing the database to track and manipulate data through an explicit identifier.

4.2 Unary Operators: Selection, Projection, and Renaming

Unary operators are those that act upon a single relation. The three primary unary operators are selection, projection, and renaming. These tools allow a user to isolate specific rows, columns, or change the labels of the data structure.
Selection, denoted by the Greek letter sigma (σ), acts as a horizontal filter. It extracts only those records (tuples) that satisfy a specific condition, known as a predicate. This predicate can involve logical comparisons, arithmetic, and boolean operators. Importantly, selection does not change the schema of the table; the output has the exact same attributes and domains as the input.

Projection, denoted by the Greek letter pi (π), serves as a vertical filter. It allows a user to choose a specific subset of attributes from a relation, discarding the rest. Since the output is still a relation (under set semantics), any duplicate rows that might appear because of the removal of identifying columns must be eliminated.
Renaming, denoted by the Greek letter rho (ρ), does not change the data within a relation but alters the metadata. It can be used to change the name of the relation itself or the names of specific attributes. This is often necessary when joining a table with itself or preparing for set operations where attribute names must match.

Definition 4.2.1: Selection (σ)

The selection operator identifies and retrieves a subset of tuples from a relation that meet a defined logical condition.

Definition 4.2.2: Projection (π)

The projection operator creates a new relation consisting only of a specified subset of attributes from the original relation.

Notes:-

In modern query processors, an extended version of projection is often used. This allows not only the selection of attributes but also the creation of new columns through calculations or string manipulations based on existing data.

4.3 Binary Set Operations

Relational algebra incorporates traditional set theory operations, including union ($\cup$), intersection ($\cap$), and subtraction ($-$). However, these cannot be applied to any two arbitrary tables. They require the operands to be "union-compatible." This means the two relations must share the exact same set of attributes, and each corresponding attribute must share the same domain.

Theory 4.3.1 Rules for Set Operations

For two relations R and S to participate in a union, intersection, or difference, they must:

1. Possess the same set of attributes.
2. Have identical domains for each corresponding attribute.
3. (In mathematical relations) Maintain the same order of attributes to satisfy Cartesian product rules.

The union of R and S includes all tuples that appear in either R, S, or both. The intersection includes only those tuples found in both relations. Subtraction (or set difference) retrieves tuples that are present in the first relation but not the second. It is important to note that while union and intersection are commutative, subtraction is not; the order of operands changes the result.

4.4 Joining Relations: Products and Joins

Combining data from different relations is achieved through joining operations. The most basic of these is the Cartesian Product ($\times$), which pairs every tuple of one relation with every tuple of another. While mathematically simple, this operation is computationally expensive and rarely used alone in practice, as it creates a massive amount of often irrelevant data.
To make combinations more meaningful, we use the Join operator ($\bowtie$). The natural join looks for attributes common to both relations and pairs tuples only when they share identical values for those common attributes. A more general version is the Theta-join, which pairs tuples based on an arbitrary condition (such as "greater than" or "not equal") rather than just simple equality.

Theory 4.4.1 The Equivalence of Theta-Joins

Any Theta-join can be expressed as a Cartesian product followed immediately by a selection operation. Formally: $R \bowtie_\theta S = \sigma_\theta(R \times S)$.

Notes:-

A "dangling tuple" refers to a record that does not find a match in the other relation during a join. In standard joins, these tuples are discarded from the result.

4.5 Bag Semantics and Extended Relational Algebra

While mathematical relational algebra assumes set semantics (where every element is unique), real-world systems like SQL often utilize bag semantics. In a bag, the same tuple can appear multiple times. This affects how set operations are calculated. For example, in a bag union, if a tuple appears m times in R and n times in S, it will appear $m + n$ times in the result.
Extended relational algebra introduces operators to handle these practical requirements. These include the duplicate elimination operator (δ), which turns a bag into a set, and the sorting operator (τ), which treats the relation as a list to arrange tuples by specific values.
The grouping and aggregation operator, denoted by gamma (γ), is perhaps the most powerful extended operator. It partitions tuples into groups based on "grouping keys" and applies an aggregate function—such as SUM, AVG, MIN, MAX, or COUNT—to each group.

Definition 4.5.1: Aggregate Function

A function that summarizes a collection of values from a column to produce a single representative value, such as a total or an average.

4.6 Relational Algebra as a Constraint Language

Relational algebra is not just for querying; it can also be used to define the rules that data must follow to be considered valid. These constraints ensure the integrity of the database. We can express any constraint by stating that a specific algebraic expression must result in an empty set ($\emptyset$), or that the result of one expression must be a subset of another.
Key constraints can be represented by showing that if we join a table with itself and find two records with the same key but different attribute values, the set of such instances must be empty. Referential integrity (foreign keys) is expressed by asserting that the projection of a foreign key column in one table must be a subset of the projection of the primary key column in the referenced table.

Theory 4.6.1 Referential Integrity Constraint

In relational algebra, referential integrity is enforced by the subset inclusion: $\pi_A(R) \subseteq \pi_B(S)$, meaning every value of attribute A in relation R must exist in the set of values of attribute B in relation S.

4.7 Relational Algebra and Database Modifications

The concepts of relational algebra also extend to how we modify the database state.

- **Deletion**: Removing tuples from a relation R can be modeled as $R := R - \sigma_C(R)$, where C is the deletion condition.
- **Insertion**: Adding tuples is modeled as $R := R \cup S$, where S is the set of new tuples.
- **Update**: Updating a tuple is logically equivalent to deleting the old version and inserting a new one with modified values.

By viewing modifications through this lens, the system can use algebraic laws to optimize not only how we retrieve data, but also how we maintain it.

Chapter 5

Queries with SQL

Structured Query Language (SQL) serves as the primary interface for interacting with relational databases. While the Data Definition Language (DDL) handles the creation and modification of database structures, the Data Manipulation Language (DML) is used for the retrieval and modification of records. The querying aspect of SQL is essentially a high-level, declarative implementation of relational algebra. Because SQL is declarative, users specify the desired properties of the result set rather than the procedural steps required to compute it. This allows the database management system (DBMS) to utilize a query optimizer to determine the most efficient execution strategy, known as a query plan.
Modern SQL implementations typically follow a set-based or bag-based processing model. Under bag semantics, relations are treated as multisets where duplicate records are permitted, contrasting with the strict set theory used in pure relational algebra. SQL queries are processed by a query compiler that translates the high-level syntax into a tree of algebraic operators, such as selection, projection, join, and grouping.

Definition 5.0.1: Declarative Language

A programming paradigm in which the programmer defines what the result should look like (the logic of the computation) without describing its control flow (the procedural steps).

Theory 5.0.1 Query Plan

A structured sequence of internal operations, often represented as a tree of relational algebra operators, that the DBMS execution engine follows to produce the results of a query.

5.1 Basic Query Structure: SELECT-FROM-WHERE

The fundamental building block of a SQL query is the select-from-where expression. This structure corresponds to the three most common operations in relational algebra: projection, relation selection, and tuple selection.
The `FROM` clause identifies the relations (tables) from which data is to be retrieved. This is conceptually the first step of the query, as it defines the scope of the data. The `WHERE` clause specifies a predicate used to filter the tuples. Only records that satisfy this logical condition are passed to the next stage. Finally, the `SELECT` clause identifies which attributes (columns) should be returned in the output. This is equivalent to the projection operator (π) in relational algebra. If a user wishes to retrieve all columns, a wildcard asterisk (*) is used.

Theory 5.1.1 SELECT-FROM-WHERE Mapping

A basic SQL query of the form `SELECT L FROM R WHERE C` is equivalent to the relational algebra expression $\pi_L(\sigma_C(R))$.

Notes:-

In SQL, the select-list can include not only existing attributes but also constants and computed expressions, functioning like an extended projection.

5.2 Logic, Comparisons, and Three-Valued Logic

Filters in the `WHERE` clause are constructed using comparison operators such as equality (=), inequality (¡¿ or !=), and range comparisons (¡, ¿, ¡=, ¿=). SQL also supports pattern matching for strings through the `LIKE` operator, where the percent sign (
A critical aspect of SQL logic is the treatment of `NULL` values. Because a `NULL` represents an unknown or missing value, comparing anything to `NULL` results in a truth value of `UNKNOWN`. This necessitates a three-valued logic system.

Definition 5.2.1: Three-Valued Logic

A logical framework where expressions can evaluate to TRUE, FALSE, or UNKNOWN, specifically required to handle comparisons involving NULL values.

The behavior of logical operators under three-valued logic follows specific rules:

- **AND**: The result is TRUE only if both operands are TRUE. If one is FALSE, the result is FALSE regardless of the other. If one is TRUE and the other is UNKNOWN, the result is UNKNOWN.
- **OR**: The result is TRUE if at least one operand is TRUE. If one is TRUE, the result is TRUE regardless of the other. If one is FALSE and the other is UNKNOWN, the result is UNKNOWN.
- **NOT**: The negation of UNKNOWN remains UNKNOWN.

Notes:-

The `WHERE` clause only retains tuples for which the predicate evaluates to TRUE. Records that evaluate to FALSE or UNKNOWN are filtered out.

5.3 Ordering and Limiting Results

While relational algebra results are conceptually unordered sets or bags, SQL allows users to impose a specific order on the output using the `ORDER BY` clause. Sorting can be performed in ascending (`ASC`, the default) or descending (`DESC`) order. Multiple columns can be specified to handle ties.
Furthermore, SQL provides mechanisms to limit the size of the result set, which is particularly useful for performance and pagination. The `LIMIT` clause restricts the total number of rows returned, while the `OFFSET` clause skips a specified number of rows before beginning to return results.

Theory 5.3.1 List Semantics

When an `ORDER BY` clause is applied, the result set is treated as a list rather than a bag, meaning the sequence of records is guaranteed and meaningful for the application.

5.4 Multi-Relation Queries and Joins

SQL allows queries to involve multiple relations by listing them in the `FROM` clause. When multiple tables are listed without a joining condition, the result is a Cartesian product, where every tuple from the first relation is paired with every tuple from the second.
To perform meaningful combinations, join conditions must be specified. These conditions link related data across tables, typically by equating a primary key in one table with a foreign key in another. If attribute names are identical across tables, they must be disambiguated using the table name or a tuple variable (alias).

Definition 5.4.1: Tuple Variable (Alias)

A temporary name assigned to a table in the `FROM` clause, used to shorten queries, disambiguate column references, or allow a table to be joined with itself (self-join).

SQL provides explicit join syntax as an alternative to the comma-separated list in the `FROM` clause:

- **CROSS JOIN**: Produces the Cartesian product.
- **INNER JOIN**: Returns only the tuples that satisfy the join condition.
- **NATURAL JOIN**: Automatically joins tables based on all columns with matching names and removes the redundant duplicate column.
- **OUTER JOIN**: Preserves "dangling tuples" that do not have a match in the other relation, padding the missing values with `NULL`. These come in `LEFT`, `RIGHT`, and `FULL` varieties.

Notes:-

The `USING` clause is a safer alternative to `NATURAL JOIN` as it allows the user to explicitly specify which columns with shared names should be used for the join, preventing accidental matches on unrelated columns.

5.5 Subqueries and Nesting

SQL is highly recursive, allowing queries to be nested within other queries. A subquery can appear in the `WHERE`, `FROM`, or `SELECT` clauses.
Subqueries that return a single row and a single column are called scalar subqueries and can be used anywhere a constant is expected. Subqueries that return a single column (a list of values) can be used with operators like `IN`, `ANY`, or `ALL`. The `EXISTS` operator is used to check if a subquery returns any results at all.

> **Theory 5.5.1** Correlated Subquery
>
> A subquery that references an attribute from the outer query. It conceptually requires the subquery to be evaluated once for every row processed by the outer query.

5.6 Duplicate Elimination and Set Operations

Because SQL defaults to bag semantics, it often produces duplicate rows. The `DISTINCT` keyword can be added to the `SELECT` clause to force set semantics by removing these duplicates.
SQL also supports standard set operations: `UNION`, `INTERSECT`, and `EXCEPT` (or `MINUS`). By default, these operations eliminate duplicates. If bag semantics are desired, the `ALL` keyword must be appended (e.g., `UNION ALL`).

Notes:-

Duplicate elimination is a computationally expensive operation because it requires sorting or hashing the entire result set to identify matching tuples.

5.7 Aggregation and Grouping

Aggregation allows users to summarize large volumes of data into single representative values. SQL provides five standard aggregate functions: `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`.
The `GROUP BY` clause partitions the data into groups based on the values of one or more columns. Aggregate functions are then applied to each group independently. A critical restriction exists when using grouping: any column appearing in the `SELECT` list that is not part of an aggregate function must be included in the `GROUP BY` clause.

> **Theory 5.7.1** The Aggregation Rule
>
> In a query using grouping, the output can only consist of the attributes used for grouping and the results of aggregate functions applied to the groups.

For filtering data after it has been aggregated, SQL uses the `HAVING` clause. Unlike `WHERE`, which filters individual rows before they are grouped, `HAVING` filters the groups themselves based on aggregate properties.

Definition 5.7.1: Aggregate Function

A function that takes a collection of values as input and returns a single value as a summary, such as a total or an average.

5.8 Advanced Table Expressions and Common Table Expressions

To improve query readability and maintainability, SQL provides mechanisms to define temporary relations within a single query. The `VALUES` clause can be used to construct a constant table on the fly. More importantly, the `WITH` clause allows for the definition of Common Table Expressions (CTEs).

Theory 5.8.1 Common Table Expression (CTE)

A temporary named result set that exists only within the scope of a single query, providing a way to decompose complex queries into smaller, logical steps.

CTEs can also be recursive, allowing SQL to perform operations that are impossible in standard relational algebra, such as computing the transitive closure of a graph (e.g., finding all reachable cities in a flight network).

Notes:-

The `WITH RECURSIVE` statement typically consists of a base case (non-recursive query) and a recursive step joined by a `UNION` operator.

5.9 Conclusion on Query Logic

Querying with SQL represents a bridge between high-level human requirements and mathematical relational theory. By understanding the underlying relational algebra—selection, projection, products, and joins—users can write more efficient and accurate queries. The complexity of SQL arises from its need to handle real-world data nuances, such as missing information (`NULL`s) and the desire for summarized reports (aggregation). Mastering the order of operations—starting from the `FROM` clause, moving through `WHERE` and `GROUP BY`, and finally reaching `SELECT`, `HAVING`, and `ORDER BY`—is essential for any database engineer.
The relationship between SQL and its execution can be viewed as a translation process: the user speaks in "declarative" desires, while the database engine converts those desires into a "procedural" query plan, much like a chef translating a customer's order into a sequence of kitchen tasks.

Chapter 6

Database Design Theory

Database design theory provides a mathematical foundation for creating relational schemas that are both efficient and resilient to errors. The primary objective is to avoid anomalies—logical inconsistencies that arise when a schema is poorly structured. These issues usually stem from a single relation attempting to store too many distinct types of information. By applying formal techniques such as functional dependencies and normalization, designers can decompose complex tables into smaller, well-structured ones that preserve data integrity while reducing redundancy. This chapter explores the fundamental concepts of functional dependencies, the definition of various keys, and the criteria for Boyce-Codd Normal Form, concluding with algorithms to ensure that decompositions do not lose information.

6.1 Anomalies and the Need for Better Design

When a database schema is designed without adhering to theoretical principles, it often suffers from three major types of anomalies. These anomalies make the database difficult to maintain and prone to data corruption over time.

Definition 6.1.1: Update Anomaly

A situation where a piece of information is stored multiple times due to redundancy. If that information changes, every instance must be updated simultaneously. Failure to do so leads to an inconsistent state where different records provide conflicting information for the same logical entity.

Definition 6.1.2: Deletion Anomaly

Occurs when the deletion of certain data inadvertently results in the loss of other, unrelated information that was stored in the same record. For example, if customer information and product pricing are stored in a single table, deleting a customer's only order might result in the loss of all data regarding that product's existence.

Definition 6.1.3: Insertion Anomaly

Arises when it is impossible to store certain information because it requires the presence of other, currently unavailable data. A common example is being unable to record a new product's price because no customer has ordered it yet, or being unable to store a new customer's details until they make their first purchase.

The intuitive solution to these problems is the separation of functions. Instead of one massive table, information should be distributed across multiple tables, each dedicated to a single functional purpose (e.g., one for customers, one for products, and one for transactions). These tables are then linked through primary and foreign keys.

6.2 Functional Dependencies

The core mathematical tool in design theory is the functional dependency. It allows us to formalize the relationships between attributes and provides a method to determine if a schema is well-designed.

Definition 6.2.1: Functional Dependency (FD)

A statement about a relation R such that for any two tuples t_1 and t_2, if they agree on the values of a set of attributes S, they must also agree on the values of another set of attributes T. This is denoted as $S \to T$. We say that S functionally determines T.

Theory 6.2.1 The No Coincidences Assumption

In the study of design theory, we assume that if a functional dependency $S \to T$ holds, it is because of a structural requirement of the application, not an accidental coincidence in the current data. The dependency must hold for all possible valid instances of the database.

Functional dependencies can be visualized as a lookup process: if you know the value of S, there is a unique value of T associated with it. However, this is not a mathematical function that can be calculated from first principles; it is a relationship maintained by the database state.

6.3 Rules for Reasoning About Functional Dependencies

We can deduce new functional dependencies from a set of existing ones using logical rules. These rules allow us to simplify sets of dependencies or verify if a specific dependency holds.

Definition 6.3.1: Trivial Functional Dependency

An FD $S \to T$ is trivial if $T \subseteq S$. Such a dependency always holds regardless of the data, as it simply states that if you know a set of values, you know any subset of those values.

Theory 6.3.1 Armstrong's Axioms

A complete set of rules for inferring all functional dependencies that follow from a given set:

- **Reflexivity:** If $T \subseteq S$, then $S \to T$.
- **Augmentation:** If $S \to T$, then $SZ \to TZ$ for any set of attributes Z.
- **Transitivity:** If $S \to T$ and $T \to U$, then $S \to U$.

In addition to the axioms, two practical rules are frequently used to manipulate dependencies:

- **Splitting Rule:** $A \to BC$ is equivalent to $A \to B$ and $A \to C$.
- **Combining Rule:** $A \to B$ and $A \to C$ is equivalent to $A \to BC$.

Note that these rules only apply to the right-hand side of a dependency. One cannot split the left-hand side (e.g., $AB \to C$ does not imply $A \to C$).

6.4 Attribute Closure and Minimal Basis

To determine everything that a set of attributes X can determine, we calculate its closure.

Definition 6.4.1: Attribute Closure

The closure of a set of attributes X under a set of FDs F, denoted X^+, is the set of all attributes A such that $X \to A$ can be derived from F. The algorithm starts with X and repeatedly adds the right-hand side of any FD whose left-hand side is already contained within the current set.

Notes:-

The closure algorithm is essential for finding keys. If X^+ contains all attributes of the relation, then X is a superkey.

Sets of FDs can be redundant. To streamline the design process, we often look for a minimal basis.

Definition 6.4.2: Minimal Basis

A set of FDs B is a minimal basis for a set F if:

- B is equivalent to F.
- Every right-hand side in B is a single attribute.
- No FD in B can be removed without losing equivalence.
- No attribute can be removed from the left-hand side of an FD in B without losing equivalence.

6.5 Defining Keys in Relational Design

Keys are subsets of attributes that uniquely identify records. Understanding the hierarchy of keys is necessary for normalization.

Definition 6.5.1: Superkey

A set of attributes K is a superkey for relation R if K functionally determines all other attributes in R. Every relation has at least one superkey: the set of all its attributes.

Definition 6.5.2: Candidate Key

A candidate key is a minimal superkey. This means that if any attribute is removed from the set, it is no longer a superkey.

Definition 6.5.3: Primary Key

A specific candidate key chosen by the database designer to be the principal means of identifying tuples within a relation.

Notes:-

An attribute is considered **prime** if it is a member of at least one candidate key. Otherwise, it is **non-prime**.

6.6 Boyce-Codd Normal Form (BCNF)

The first level of normalization is often considered the First Normal Form (1NF), which requires that every attribute value be atomic (no nested tables or arrays). However, to eliminate the anomalies discussed earlier, we require stricter forms like BCNF.

Definition 6.6.1: Boyce-Codd Normal Form (BCNF)

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

Theorem 6.6.1 Two-Attribute BCNF

Any relation with exactly two attributes is guaranteed to be in BCNF, regardless of the dependencies present.

If a relation violates BCNF, it means there is a "determinant" (a left-hand side of an FD) that does not uniquely identify the entire row. This causes redundancy and the risk of anomalies.

6.7 The BCNF Decomposition Algorithm

To bring a relation R into BCNF, we follow a recursive decomposition process:

1. Identify a non-trivial FD $S \to T$ that violates BCNF (where S is not a superkey).
2. Compute S^+.
3. Decompose R into two relations:
   - R_1 with attributes in S^+.
   - R_2 with attributes S and all attributes of R that are not in S^+.
4. Recursively apply the algorithm to R_1 and R_2 until all resulting tables are in BCNF.

6.8 Lossless Join and the Chase Algorithm

A critical requirement of decomposition is that it must be "lossless." We must be able to reconstruct the original relation exactly by joining the decomposed relations.

Theory 6.8.1 The Lossless Join Property

A decomposition of R into $R_1, R_2, \ldots, R_k$ is lossless if the natural join of all R_i produces exactly the original instance of R. A decomposition can never result in fewer tuples than the original, but a "lossy" join creates "ghost tuples"—records that weren't in the original data.

To verify if a join is lossless, we use the Chase Algorithm.

Definition 6.8.1: Chase Algorithm

A test for a lossless join. We create a tableau representing a tuple in the join. We then apply the functional dependencies of the original relation to equate symbols in the tableau. If one row eventually becomes identical to the target tuple (all unsubscripted symbols), the join is proven to be lossless.

6.9 Dependency Preservation and the Impossibility Triangle

While BCNF eliminates redundancy and ensures a lossless join, it does not always preserve functional dependencies. A dependency $S \to T$ is preserved if it can be checked within a single relation of the decomposition.

Notes:-

Database designers often face a trade-off. It is not always possible to achieve BCNF, a lossless join, and dependency preservation simultaneously. If preserving a specific dependency is vital for the application, the designer might settle for Third Normal Form (3NF).

Definition 6.9.1: Third Normal Form (3NF)

A relation R is in 3NF if for every non-trivial FD $S \to T$, either S is a superkey or every attribute in $T \setminus S$ is prime (part of some candidate key). 3NF is more lenient than BCNF because it allows certain dependencies where the left side is not a superkey, provided the right side is prime.

6.10 Multivalued Dependencies and 4NF

Sometimes, BCNF is insufficient to remove all redundancy. This occurs when a table attempts to store two independent many-to-many relationships for the same key.

Definition 6.10.1: Multivalued Dependency (MVD)

A statement $S \twoheadrightarrow T$ which implies that for a given value of S, the associated values of T are independent of the values of the other attributes in the relation. If we fix S, we must see all possible combinations of T and the other attributes.

Theory 6.10.1 Fourth Normal Form (4NF)

A relation R is in 4NF if for every non-trivial MVD $S \twoheadrightarrow T$, S is a superkey. This is a generalization of BCNF that also accounts for redundancies caused by MVDs.

Decomposition into 4NF follows a similar logic to BCNF decomposition but uses MVDs to split the tables. By reaching 4NF, the designer eliminates nearly all forms of logical redundancy.

6.11 Design Heuristics and Practical Application

The theoretical process often involves a "representative" table—a snapshot of the data that includes enough examples to prevent the designer from assuming "accidental" dependencies. Designers must distinguish between structural dependencies that must always hold and coincidences in a specific dataset.

Theory 6.11.1 The Table Universe Concept

A table universe consists of all possible valid instances of a schema over time. Integrity constraints and normal forms must apply to every instance in this universe, ensuring that the database remains robust as it grows and changes.

By ensuring that every table does "one thing" and follows the rules of BCNF or 3NF, developers can build systems that are significantly cheaper to maintain, as the logic for handling anomalies does not need to be hard-coded into the application layer. Instead, the structure of the database itself prevents corruption.

Notes:-

Normal forms are additive: 4NF implies BCNF, which implies 3NF, which implies 2NF, which implies 1NF. The higher the normal form, the fewer anomalies exist in the schema.

In conclusion, database design theory moves database creation from an intuitive art to a rigorous science. By understanding the interplay between keys, dependencies, and normalization, engineers can create high-performance data systems that remain consistent through complex transactional workloads.

Chapter 7

Transactions and the Three Tiers

Modern database systems do not operate in isolation; they are embedded within complex multi-tier architectures designed to handle thousands of concurrent users. At the heart of this ecosystem is the concept of a transaction, a logical unit of work that ensures data integrity despite system failures or overlapping user actions. To maintain this integrity, databases adhere to the ACID properties—Atomicity, Consistency, Isolation, and Durability. This chapter explores the three-tier architecture that connects users to data, the hierarchical structure of the SQL environment, and the rigorous mechanics of transaction management, including isolation levels and locking protocols such as Two-Phase Locking (2PL).

7.1 The Three-Tier Architecture

Large-scale database installations typically utilize a three-tier architecture to separate concerns and improve scalability. This organization allows different components of the system to run on dedicated hardware, optimizing performance for each specific task.

Definition 7.1.1: Three-Tier Architecture

A system organization consisting of three distinct layers: the Web Server tier for user interaction, the Application Server tier for processing logic, and the Database Server tier for data management.

The first tier consists of **Web Servers**. These processes act as the entry point for clients, usually interacting via a web browser over the Internet. When a user enters a URL or submits a form, the browser sends an HTTP (Hypertext Transfer Protocol) request to the web server. The web server is responsible for returning an HTML page, which may include images and other data to be displayed to the user.

Notes:-

Common web server software includes Apache and Tomcat, which are frequently used in both professional and academic environments to bridge the gap between web browsers and database systems.

The second tier is the **Application Server**, often referred to as the **Business Logic** layer. This is where the core functionality of the system resides. When the web server receives a request that requires data, it communicates with the application tier. Programmers use languages such as Java, Python, C++, or PHP to write the logic that decides how to respond to user requests. This layer is responsible for generating SQL queries, sending them to the database, and formatting the returned results into a programmatically built HTML page or other responses.
The third tier is the **Database Server**. These are the processes running the Database Management System (DBMS), such as PostgreSQL or MySQL. This tier executes the queries requested by the application tier, manages data persistence on disk, and ensures that the system remains responsive through buffering and connection management.

7.2 The SQL Environment

Within the database tier, data is organized in a hierarchical framework known as the SQL environment. This structure allows for a clear namespace and organizational scope for all database elements.

Definition 7.2.1: SQL Environment

The overall framework under which database elements exist and SQL operations are executed, typically representing a specific installation of a DBMS.

The hierarchy begins with the **Cluster**, which represents the maximum scope for a database operation and the set of all data accessible to a particular user. Within a cluster, data is organized into **Catalogs**. A catalog is the primary unit for supporting unique terminology and contains one or more **Schemas**.
A schema is a collection of database objects, including tables, views, triggers, and assertions. In professional environments, a full name for a table might look like ‘CatalogName.SchemaName.TableName‘. If the catalog or schema is not explicitly specified, the system defaults to the current session's settings (e.g., ‘public‘ is often the default schema).

Theory 7.2.1 The Concept of Sessions

A session is the period during which a connection between a SQL client and a SQL server is active, encompassing a sequence of operations performed under a specific authorization ID.

7.3 Fundamentals of Transactions

A transaction is a single execution of a program or a batch of queries that must be treated as an indivisible unit. The goal of the transaction manager is to ensure that even if the system crashes or multiple users access the same record, the result remains correct.

Definition 7.3.1: Transaction

A collection of one or more database operations, such as reads and writes, that are grouped together to be executed atomically and in isolation from other concurrent actions.

To be considered reliable, every transaction must satisfy the **ACID** test. These four properties are the cornerstone of database design theory.

Theory 7.3.1 ACID Properties

- **Atomicity:** Often described as "all-or-nothing," this ensures that a transaction is either fully completed or not executed at all. If a failure occurs halfway through, any partial changes must be undone.
- **Consistency:** A transaction must take the database from one consistent state to another, satisfying all integrity constraints like primary keys and check constraints.
- **Isolation:** Each transaction should run as if it were the only one using the system, regardless of how many other users are active.
- **Durability:** Once a transaction has been committed, its effects must persist in the database even in the event of a power outage or system crash.

Notes:-

Atomicity in transactions should not be confused with atomic values in First Normal Form. In this context, it refers to the indivisibility of the execution process itself.

7.4 Concurrency and Isolation Levels

When multiple transactions run at the same time, their actions may interleave in a way that leads to inconsistencies. A **Schedule** is the actual sequence of actions (reads and writes) performed by these transactions. While a **Serial Schedule** (running one transaction after another) is always safe, it is inefficient. Schedulers instead aim for **Serializability**.

> **Theory 7.4.1** Serializability
>
> A schedule is serializable if its effect on the database is identical to the effect of some serial execution of the same transactions.

If isolation is not properly managed, several types of "anomalies" can occur. These phenomena describe undesirable interactions between concurrent processes.

Definition 7.4.1: Dirty Read

A situation where one transaction reads data that has been modified by another transaction but has not yet been committed. If the first transaction subsequently aborts, the second transaction has based its work on data that "never existed."

Definition 7.4.2: Non-repeatable Read

Occurs when a transaction reads the same data element twice but finds different values because another transaction modified and committed that element in the interim.

Definition 7.4.3: Phantom Read

A phenomenon where a transaction runs a query to find a set of rows, but upon repeating the query, finds additional "phantom" rows that were inserted and committed by a concurrent transaction.

SQL provides four **Isolation Levels** that allow developers to trade off strictness for performance.

- **Read Uncommitted:** The most relaxed level; allows dirty reads.
- **Read Committed:** Forbids dirty reads but allows non-repeatable reads.
- **Repeatable Read:** Forbids dirty and non-repeatable reads but may allow phantoms.
- **Serializable:** The strictest level; ensures the result is equivalent to some serial order.

7.5 Locking and Two-Phase Locking (2PL)

The most common way for a database to enforce serializability is through the use of **Locks**. Before a transaction can read or write a piece of data, it must obtain a lock on that element. These are managed via a **Lock Table** in the scheduler.

Definition 7.5.1: Shared and Exclusive Locks

A Shared (S) lock is required for reading and allows multiple transactions to read the same element. An Exclusive (X) lock is required for writing and prevents any other transaction from accessing that element.

Simply using locks is not enough to guarantee serializability; the timing of when locks are released is critical. If a transaction releases a lock too early, another transaction might intervene and change the data, leading to a non-serializable schedule. To prevent this, systems use the **Two-Phase Locking (2PL)** protocol.

> **Theory 7.5.1** Two-Phase Locking (2PL)
>
> A protocol requiring that in every transaction, all locking actions must precede all unlocking actions. This

> creates two distinct phases: a "growing phase" where locks are acquired and a "shrinking phase" where they are released.

Notes:-

Strict Two-Phase Locking is a variation where a transaction does not release any exclusive locks until it has committed or aborted. This prevents other transactions from reading dirty data and avoids the need for cascading rollbacks.

A potential downside of locking is the risk of a **Deadlock**. This occurs when two or more transactions are stuck in a cycle, each waiting for a lock held by the other. Schedulers must be able to detect these cycles—often using a **Waits-For Graph**—and resolve them by aborting one of the transactions.

In conclusion, the management of transactions requires a deep integration of architectural tiers, hierarchical environments, and rigorous concurrency control. By utilizing ACID properties, various isolation levels, and the 2PL protocol, database systems provide a robust platform where users can safely interact with data as if they were the sole occupants of the system.

Notes:-

In practice, many developers use higher-level APIs like JDBC for Java or PHP's PEAR DB library to handle the complexities of database connections and transaction boundaries programmatically.

To think of it another way, a transaction is like a single entry in a shared diary. Even if twenty people are writing in the same diary simultaneously, the system acts like a careful librarian, ensuring that each person's entry is written cleanly on its own line without anyone's ink smudging another's work.

Chapter 8

Views and Indecies

The management of information within a relational database system involves a balance between logical abstraction and physical performance. While base tables provide the primary storage for data, they are not always optimized for the specific ways in which users interact with information. To bridge this gap, database systems utilize two critical components: **Views** and **Indices**. Views allow designers to create virtual relations that simplify complex queries and provide security by filtering access to specific attributes or rows. Indices, on the other hand, focus on the physical layer, providing specialized data structures that accelerate the retrieval of tuples without requiring exhaustive table scans. Together, these tools ensure that a database is both easy to use for the developer and efficient for the machine. This chapter explores the declaration and querying of virtual views, the limitations of updating them, the mechanics of indexing, and the mathematical models used to determine when an index provides a genuine performance benefit.

8.1 Virtual Views and Logical Abstraction

A virtual view is a relation that does not exist as a separate entity on the physical disk but is instead defined by a query over one or more base tables. From the perspective of the user or the application layer, a view is indistinguishable from a standard table; it can be queried, joined with other relations, and used in subqueries.

Definition 8.1.1: Virtual View

A named relation defined by an expression or query that acts as a shortcut to data stored in other relations. It is persistent in definition but transient in representation, meaning its contents are recomputed or mapped back to base tables every time it is accessed.

The primary advantage of using views is **productivity**. Instead of repeating a complex subquery multiple times across different parts of an application, a developer can define that subquery as a view and refer to it by name.

Theory 8.1.1 The Interpretation of View Queries

When a query refers to a virtual view, the query processor logically replaces the view name with its underlying definition. This effectively turns the query into a larger expression that operates directly on the base tables, ensuring that the view always reflects the most current state of the database.

Notes:-

To declare a view in SQL, the `CREATE VIEW` statement is used followed by the keyword `AS` and a standard `SELECT-FROM-WHERE` block. If a developer wishes to change the column names presented by the view to be more descriptive or to avoid name collisions, they can list the new attribute names in parentheses immediately following the view name.

8.2 Updatable Views and Modification Criteria

While querying a view is straightforward, modifying one—through `INSERT`, `DELETE`, or `UPDATE`—presents a logical challenge. Since a view is virtual, any change must be translated into a corresponding change in the underlying base tables. SQL allows this only under specific, restrictive conditions to ensure that the translation is unambiguous.

Definition 8.2.1: Updatable View

A virtual view that the DBMS can modify by passing the changes through to the base relation. In standard SQL, this generally requires the view to be defined over a single relation and to include enough attributes so that a valid tuple can be formed in the underlying table.

To be considered updatable without the help of triggers, a view typically must meet several criteria:

- The `FROM` clause must contain exactly one relation.
- There can be no `DISTINCT` keyword, as this would make it impossible to determine which original tuple a change refers to.
- The `WHERE` clause cannot use the relation itself in a subquery.
- The `SELECT` list must include enough attributes to satisfy the `NOT NULL` and primary key constraints of the base table, or those omitted attributes must have default values.

Notes:-

If an insertion is made into a view and the view projects out the attribute used in the `WHERE` clause, the new tuple might disappear from the view immediately after being added. This is because the underlying table receives the tuple with a `NULL` or default value that may not satisfy the view's selection criteria.

8.3 Instead-Of Triggers

When a view is too complex to be automatically updatable (for instance, when it involve joins or aggregations), a database designer can use **Instead-Of Triggers**. These allow the programmer to explicitly define how a modification to a view should be handled by the system.

Theory 8.3.1 Instead-Of Trigger Principle

An instead-of trigger intercepts a modification command intended for a view and executes a specified block of code in its place. This code usually involves custom logic to distribute the modification across multiple base tables or to calculate missing values.

Notes:-

By using the `REFERENCING NEW ROW AS` clause, the trigger can access the values the user attempted to insert into the view and use them as parameters for updates to the actual stored tables.

8.4 Physical Storage and the Motivation for Indices

In a database without indices, the only way to find a specific record is through a **Full Scan**. This requires the system to read every block of the relation from the disk and check every tuple against the search condition. While this is feasible for small tables, it becomes a massive bottleneck as the data grows into millions or billions of rows.

Definition 8.4.1: Index

A supplementary data structure that associates specific values of one or more attributes (the search key) with pointers to the physical locations of the records containing those values. Its purpose is to allow the system to bypass irrelevant data and jump directly to the desired blocks.

Theory 8.4.1 The I/O Model of Computation

The cost of a database operation is primarily determined by the number of disk I/O actions it requires. Because moving data from the disk to main memory is orders of magnitude slower than CPU operations, the efficiency of a physical plan is measured by how many blocks must be read or written.

Notes:-

It is important to distinguish between the **search key** of an index and the **primary key** of a relation. An index can be built on any attribute or set of attributes, regardless of whether they are unique.

8.5 Clustered and Non-Clustered Indices

The relationship between the order of an index and the physical arrangement of tuples on the disk significantly impacts performance.

Definition 8.5.1: Clustering Index

An index where the physical order of the records in the data file matches or closely follows the order of the keys in the index. This ensures that all tuples with the same or similar key values are packed into the minimum possible number of blocks.

Theory 8.5.1 Clustering Efficiency

A clustering index is much more efficient for range queries than a non-clustering index. In a clustered scenario, the system can retrieve a range of values by reading consecutive blocks. In a non-clustered scenario, every matching tuple might reside on a different block, potentially requiring one disk I/O per tuple.

Notes:-

A relation can only have one clustering index because the data can only be physically sorted in one way. However, it can have multiple non-clustering (secondary) indices.

8.6 The Mechanics of B-Trees

The most prevalent index structure in modern database systems is the **B-Tree**, specifically the B+ Tree variant. This structure is a balanced tree that automatically scales with the size of the data.

Definition 8.6.1: B-Tree

A balanced tree structure where every path from the root to a leaf is of equal length. Each node corresponds to a single disk block and contains a sorted list of keys and pointers to either child nodes or data records.

B-Trees are characterized by a parameter n, which defines the maximum number of keys a block can hold. The rules for a B-Tree include:

- **Internal Nodes:** Must have between $\lceil (n+1)/2 \rceil$ and $n+1$ children, except for the root, which can have as few as two.
- **Leaves:** Hold the actual search keys and pointers to the records. They also include a pointer to the next leaf in sequence to facilitate range scans.

Theory 8.6.1 Logarithmic Search Complexity

In a B-Tree, the number of steps required to find any specific key is proportional to the height of the tree. For a tree with N records and a fan-out of f, the height is approximately $\log_f N$. Because f is typically large (often hundreds of keys per block), even a billion records can be searched in just three or four disk accesses.

Notes:-

B-Trees are dynamic; they grow by splitting nodes when they become too full and shrink by merging nodes when deletions leave them under-populated. This ensures that every block remains at least half-full, optimizing disk usage.

8.7 Hash Indices and Constant Time Lookups

As an alternative to tree-structured indices, databases may use **Hash Indices**. These are based on a "smoothie machine" analogy: a deterministic function that turns any input into a seemingly random integer.

Definition 8.7.1: Hash Index

A structure that uses a hash function to map search keys into specific buckets. Each bucket corresponds to one or more disk blocks holding pointers to the relevant records.

Theory 8.7.1 Constant Complexity

A hash index provides $O(1)$ lookup time for equality queries. No matter how large the table becomes, the time to locate a specific key remains constant, as it requires only the computation of the hash and a direct jump to the indicated bucket.

Notes:-

The major limitation of hash indices is their lack of support for range queries. Because the hash function randomizes the placement of keys, two keys that are close in value (e.g., 19 and 20) will likely end up in completely different parts of the index.

8.8 Index Selection and Cost Modeling

Creating an index is not a "free" performance boost. Every index added to a relation imposes costs that must be weighed against its benefits. The decision process involves analyzing a query workload and estimating the average disk I/O.

Definition 8.8.1: Index Creation Costs

The costs associated with indices include the initial CPU and I/O time to build the structure, the additional disk space required to store it, and the "write penalty"—the fact that every `INSERT`, `DELETE`, or `UPDATE` to the base table must also update every associated index.

Theory 8.8.1 The Selection Formula

If p is the probability that a query uses a certain attribute and $1 - p$ is the probability of an update, an index on that attribute is beneficial only if the time saved during the queries outweighs the extra time spent on updates. This can be expressed as a linear combination of costs based on the parameters $B(R)$ (blocks) and $T(R)$ (tuples).

Notes:-

In practice, many systems use an "automatic tuning advisor" that applies a greedy algorithm to suggest the best set of indices for a specific historical workload.

8.9 Indices and Complex Queries

Indices are particularly powerful when dealing with joins or multiple selection criteria.

Theory 8.9.1 The Index-Join Strategy

In a join $R \bowtie S$, if S has an index on the join attribute, the system can iterate through R and for each tuple, use the index on S to find matching records. This is significantly faster than a nested-loop join if R is small and the index on S is efficient.

Notes:-

When multiple indices are available for a single query, a technique called "pointer intersection" can be used. The system retrieves lists of pointers from several indices, intersects them in main memory, and only then reads the data blocks for the tuples that satisfy all conditions.

8.10 Information Retrieval and Inverted Indices

A specialized form of indexing used for documents is the **Inverted Index**. This is the technology that powers web search engines and large-scale document repositories.

Definition 8.10.1: Inverted Index

A mapping from words (keywords) to the list of documents in which those words appear. Often, these lists include metadata such as the position of the word in the document or whether it appeared in a title or anchor tag.

Notes:-

To optimize inverted indices, systems often use "stemming" (reducing words to their root form) and "stop words" (ignoring common words like "the" or "and" that do not help distinguish documents).

8.11 Summary of Design Principles

The theory of views and indices suggests that database design is as much about managing the physical medium as it is about logical modeling.

Theory 8.11.1 The Table Universe and Consistency

While views provide a filtered perspective of the data, they must remain consistent with the "table universe"—the set of all possible valid states of the database. Constraints and indices must be applied such that they hold true across all these potential states, ensuring that neither hardware failure nor concurrent access can corrupt the logical integrity of the system.

Notes:-

Ultimately, the goal of indices is to turn linear or quadratic problems into logarithmic or constant ones. By carefully selecting which attributes to index based on the $B, T,$ and V parameters, a designer can create a system that remains responsive even under the weight of massive datasets.

In summary, views provide the necessary abstraction to keep application code clean and secure, while indices provide the surgical precision required to extract data from high-volume storage. A database is essentially a large library; a view is a specific bookshelf curated for a student, while an index is the card catalog that allows a librarian to find one specific page in a million volumes without having to read every book in the building.

Chapter 9

Data Cubes

The history of data management has progressed through several distinct eras, each defined by the primary utility of information. The initial phase, spanning from the 1970s to the 2000s, is characterized as the **Age of Transactions**. During this period, the development of the relational model, SQL, and the concept of data independence allowed organizations to maintain consistent and reliable records. These systems were designed to handle a continuous stream of updates, inserts, and deletions, necessitating a focus on concurrency and integrity. However, in the mid-1990s, a transition occurred toward the **Age of Business Intelligence**. As computational power increased and data volumes grew, corporate leadership—such as CEOs and CFOs—began requiring high-level insights rather than individual record access. This shift led to the emergence of specialized systems designed for data analysis, reporting, and dashboarding. This evolution eventually culminated in the modern **Age of Big Data**, characterized by massive scale and distributed processing.

Definition 9.0.1: OLTP (Online Transactional Processing)

A paradigm of data management focused on the day-to-day operational tasks of a business. It emphasizes record-keeping, high-frequency write operations, and the maintenance of data integrity through ACID properties.

Definition 9.0.2: OLAP (Online Analytical Processing)

A data management paradigm designed for decision support and business intelligence. It involves the analysis of large, consolidated datasets that are typically frozen or "non-volatile," focusing on complex read-only queries rather than real-time updates.

9.1 Comparing OLTP and OLAP Paradigms

To understand the necessity of specialized analytical structures like data cubes, one must distinguish between the operational requirements of OLTP and the analytical requirements of OLAP. In an OLTP environment, the system is "zoomed in" on specific, detailed records, such as an individual customer's order or a specific product's inventory level. The goal is consistent record-keeping. Because these systems are interactive and face end-users directly, performance is measured in milliseconds, and the design relies heavily on normalization to prevent update, insertion, and deletion anomalies.
In contrast, OLAP systems are "zoomed out," providing a high-level view of the entire organization. Instead of individual transactions, OLAP focuses on aggregated data—such as total sales by region per quarter. These systems are used for decision support, where the speed of a query might range from seconds to several hours. Redundancy is often embraced in OLAP to improve query efficiency, leading to the use of denormalized structures.

Theory 9.1.1 The Trade-off of Freshness vs. Performance

Running complex analytical queries directly on a live OLTP system is generally avoided because it consumes significant resources and slows down the day-to-day business operations. Consequently, data is extracted

from OLTP systems and loaded into dedicated OLAP environments, typically during off-peak hours.

Notes:-

Backups are critical in OLTP because losing transaction records means losing the business history. In OLAP, data can often be re-imported from the original sources, making backup procedures slightly less existential but still important for efficiency.

9.2 The Data Cube Model

The logical foundation of analytical processing is the **Data Cube**. While the term suggests a three-dimensional structure, a data cube is an n-dimensional hypercube that can accommodate any number of dimensions. Each dimension represents a different axis of analysis, such as time, geography, or product category.

Definition 9.2.1: Dimension (Axis)

A specific category or perspective used to organize data within a cube. Common dimensions include "Where" (Geography), "When" (Time), "Who" (Salesperson), and "What" (Product).

Definition 9.2.2: Member

An individual value within a dimension. For example, "2024" is a member of the "Year" axis, and "Switzerland" is a member of the "Location" axis.

At the intersection of specific members from every dimension lies a **cell**, which contains a numerical **value** or **fact**. For instance, a cell might store the information that in the year 2024, in Switzerland, seven servers were sold. This highly structured model ensures that for every combination of dimensional coordinates, a specific metric is available.

9.3 The Fact Table and Normal Forms

In a relational implementation, a data cube is represented physically as a **Fact Table**. This table serves as the central hub of the analytical schema. Every row in a fact table represents a single cell from the hypercube.

Theory 9.3.1 Fact Tables and the Sixth Normal Form

A fact table represents the highest level of data structure, often described as being in the Sixth Normal Form (6NF). In this state, every column representing a dimension is part of a composite primary key, and there is typically only one non-key column representing the recorded value.

In practice, fact tables may have multiple "measure" columns, such as revenue, profit, and quantity. This is often preferred over a strict 6NF to reduce the number of rows. The process of moving between a single-measure fact table and a multi-measure table is known as **pivoting** and **unpivoting**.

9.4 Operations on Data Cubes: Slicing and Dicing

Analyzing a cube involves reducing its complexity to a format that can be visualized, typically on a two-dimensional screen or a sheet of paper. This is achieved through slicing and dicing.

Definition 9.4.1: Slicing

The process of selecting a single member from a specific dimension, thereby reducing the dimensionality of the cube. It is analogous to taking a slice of a physical cake; if you slice a 3D cube on a specific year, you are left with a 2D square representing all other dimensions for that year.

Definition 9.4.2: Dicing

The arrangement of remaining dimensions onto the rows and columns of a cross-tabulated view (or pivot table). Dicing allows the user to explicitly define the grid they wish to see, such as putting "Salesperson" on the rows and "Year" on the columns.

Notes:-

Dimensions that are not used as dicers (rows or columns) must be set as slicers. Slicers act as filters for the entire view, ensuring that the displayed data is logically consistent with the user's focus.

9.5 Hierarchies and Aggregation

Dimensions in a data cube are rarely flat lists; they are usually organized into **hierarchies**. For example, the "Location" dimension might move from City to Country to Continent to the whole World. The "Time" dimension might move from Day to Month to Quarter to Year.

Definition 9.5.1: Roll-up

The action of moving up a hierarchy to a higher level of granularity. Rolling up from "City" to "Country" involves aggregating (summing, averaging, etc.) all city values into a single total for the country.

Definition 9.5.2: Drill-down

The inverse of a roll-up, where a user moves down a hierarchy to view more specific details. Drilling down from "Year" might reveal the underlying data for each individual "Month."

In a cross-tabulated view, these hierarchies are visualized through **subtotals**. Column hierarchies are often shown using "L-shaped" headers, while row hierarchies typically use indentation, bolding, and underlining to distinguish between levels.

9.6 The ETL Process

Data does not exist in a cube format by default. It must be moved from heterogeneous operational sources (ERP, CRM, files) into the OLAP system through a process known as **ETL**.

Theory 9.6.1 The ETL Verb

ETL stands for Extract, Transform, and Load. It is often used as a verb in industry (e.g., "to ETL the data"), describing the complex engineering task of consolidating data into a unified analytical structure.

- **Extract:** Connecting to source systems, often via gateways and firewalls, to pull raw data. This can be done through triggers, log extraction, or incremental updates.
- **Transform:** The most labor-intensive phase, involving data cleaning (e.g., translating "Mr." and "Mister" into a single format), merging tables, filtering irrelevant records, and ensuring integrity constraints are met.
- **Load:** Inserting the transformed data into the data cube, building indices to accelerate future queries, and potentially partitioning the data across multiple machines.

9.7 Implementation Architectures: ROLAP and MOLAP

There are two primary flavors of OLAP implementation. **MOLAP (Multidimensional OLAP)** uses specialized, non-relational data structures to store the cube. **ROLAP (Relational OLAP)** implements the cube logic on top of standard relational tables.
In ROLAP, the schema often takes one of two shapes:

1. **Star Schema:** A central fact table surrounded by "satellite" dimension tables. Each row in the fact table contains foreign keys pointing to the members in the dimension tables.
2. **Snowflake Schema:** A more normalized version of the star schema where dimension tables are themselves decomposed into further satellite tables (e.g., a City table pointing to a Country table).

Theory 9.7.1 The Denormalized Fact Table

For extreme performance, some designers join all satellite information back into a single, massive fact table. This creates significant redundancy but allows for extremely fast aggregations as no joins are required during query time.

9.8 SQL Extensions for Analytical Processing

While standard SQL can be used to query fact tables, the code required to generate comprehensive reports with subtotals is often repetitive and prone to error. To address this, SQL was extended with specialized grouping functions.

Definition 9.8.1: GROUPING SETS

An extension of the GROUP BY clause that allows a user to specify multiple groupings in a single query. It is logically equivalent to a UNION of several GROUP BY queries, but more efficient.

Theory 9.8.1 The CUBE Operator

A syntactic sugar that generates the power set of all possible groupings for the specified attributes. For n attributes, GROUP BY CUBE produces 2^n grouping sets, providing subtotals for every possible combination.

Theory 9.8.2 The ROLLUP Operator

A specialized version of grouping sets that follows a hierarchical path. For n attributes, it produces $n + 1$ grouping sets by progressively removing attributes from right to left. This is the ideal tool for generating totals and subtotals in a dimension hierarchy.

Notes:-

The order of attributes matters significantly for ROLLUP but is irrelevant for CUBE. In a ROLLUP, you must list attributes from the most specific to the most general (e.g., City, Country, Continent).

9.9 Querying with MDX

For environments that require a dedicated multidimensional language, **MDX (Multi-Dimensional Expressions)** is used. Unlike SQL, which treats data as sets of rows, MDX natively understands the concept of dimensions, members, and cells.
MDX allows a user to explicitly define which dimensions should appear on the columns and which on the rows of a result set. It uses a "WHERE" clause not for relational selection, but as a "slicer" to pick a specific coordinate in the cube. While advanced users might write MDX, most interact with it indirectly through the drag-and-drop interfaces of spreadsheet or business intelligence software.

9.10 Standardized Reporting and XBRL

Data cube technology has significant real-world applications in financial and sustainability reporting. Regulatory bodies, such as the SEC in the United States and ESMA in the European Union, now require companies to submit reports in standardized electronic formats like **XBRL (eXtensible Business Reporting Language)**.

Definition 9.10.1: Inline XBRL

A technology that embeds machine-readable data cube information within a standard human-readable HTML webpage. This allows a single document to be viewed by a human in a browser while its individual values can be extracted and reconstructed into a cube by a computer.

In an XBRL report, every financial value is tagged with its dimensional coordinates: what the value is (e.g., Assets), who the company is (e.g., Coca-Cola), when the period was (e.g., Dec 31, 2024), and the currency used (e.g., USD). This creates a "table universe" of standardized, comparable data across entire industries.

Notes:-

The shift toward machine-readable reporting is often referred to as "interactive data," as it allows investors and regulators to automatically perform slicing and dicing operations across thousands of company filings simultaneously.

In essence, data cube theory provides the bridge between the chaotic, high-velocity world of transactional data and the structured, strategic world of corporate decision-making. By transforming "wheat" (raw transaction logs) into "bread" (actionable reports), these systems enable a level of organizational insight that was impossible in the era of paper ledgers or simple flat-file databases.

To visualize this, think of a fact table as a collection of thousands of individual lego bricks. Each brick has a specific color, size, and shape (its dimensions). While they are just a pile of plastic on their own, the dicing and rolling-up operations allow us to assemble them into a specific structure—a castle or a bridge—that reveals the overall pattern and strength of the data.

Chapter 10

Database Architecture

The management of transactions is the core mechanism that ensures a database remains reliable and consistent despite concurrent access and system failures. A transaction is defined as a logical unit of work, consisting of one or more database operations that must be executed as an indivisible whole. This chapter explores the multi-tier architecture that supports these operations, the physical storage layer that provides data independence, and the sophisticated logging and concurrency control protocols used to maintain the ACID properties. We investigate how the system handles crashes through undo and redo logging, how schedulers prevent interference between users through locking and timestamping, and how complex, long-running processes are managed through the use of sagas and compensating transactions.

10.1 The Architectural Context of Transactions

Modern database systems are typically deployed in a three-tier architecture to separate user interaction from business logic and data persistence.

Definition 10.1.1: Three-Tier Architecture

A system organization consisting of three distinct layers: the Web Server tier for managing client interactions, the Application Server tier for executing business logic and generating queries, and the Database Server tier for managing data storage and transaction execution.

The database tier is designed to provide data independence, allowing users to query data without needing to understand the underlying physical storage mechanics. Behind the scenes, the DBMS manages a complex hierarchy of hardware, moving data between volatile main memory (RAM) and nonvolatile storage (Disk).

Notes:-

Data independence is a fundamental principle established by Edgar Codd. it ensures that the logical representation of data in tables is decoupled from the physical directories and files on the disk, such as the ‘pg_data‘ directory in PostgreSQL.

10.2 Physical Storage and Data Movement

The unit of interaction between the disk and main memory is not the individual record, but the block or page. In many systems, such as PostgreSQL, these blocks are typically 8 KB in size.

Definition 10.2.1: Database Element

A unit of data that can be accessed or modified by a transaction. While elements can be tuples or relations, they are most effectively treated as disk blocks to ensure atomic writes to nonvolatile storage.

Theory 10.2.1 The I/O Model of Computation

The primary cost of database operations is measured by the number of disk I/O actions. Because accessing a disk is orders of magnitude slower than CPU cycles, efficiency is achieved by minimizing the transfer of blocks between the disk and memory buffers.

When a record is too large to fit within a standard page—such as large text objects or binary data—the system employs specialized techniques like TOAST (The Oversized-Attribute Storage Technique), which slices the data into chunks and stores them in separate tables. Physically, these pages are organized into larger files on the disk called chunks, often reaching sizes of 1 GB.

10.3 The ACID Properties of Transactions

To ensure the integrity of the database, every transaction must satisfy the ACID test. These properties guarantee that the database remains in a consistent state even if a program is interrupted or multiple users attempt to modify the same record.

Theory 10.3.1 ACID Properties

- **Atomicity:** The "all-or-nothing" execution of transactions. If any part fails, the entire unit is rolled back.
- **Consistency:** Every transaction must move the database from one valid state to another, satisfying all structural and business constraints.
- **Isolation:** Each transaction must appear to execute as if no other transaction were occurring simultaneously.
- **Durability:** Once a transaction is committed, its effects must persist permanently, surviving any subsequent system crash.

10.4 Undo Logging and Recovery

Logging is the primary method for achieving durability and atomicity. The log is an append-only file that records every important change to the database.

Definition 10.4.1: Undo Logging

A logging method where only the old values of modified data elements are recorded. It is designed to allow the recovery manager to cancel the effects of uncommitted transactions by restoring data to its previous state.

For undo logging to function correctly, two specific rules must be followed: 1. Every update record (the old value) must be written to the disk before the modified data element itself reaches the disk. 2. The commit record must be written to the disk only after all modified data elements have been successfully flushed to the disk.

Notes:-

In undo logging, the order of writes to disk is: Log record → Data element → Commit record. This ensures that if a crash occurs before the commit, we always have the old value available to undo the change.

10.5 Redo Logging and the Write-Ahead Rule

While undo logging requires immediate data flushes, redo logging offers more flexibility by recording only the new values of data elements.

Definition 10.5.1: Redo Logging

A logging method that records the new values of database elements. On recovery, the system repeats the changes of committed transactions and ignores those that did not commit.

Theory 10.5.1 Write-Ahead Logging (WAL) Rule

In redo logging, all log records pertaining to a modification, including the update record and the commit record, must appear on disk before the modified data element itself is written to disk.

The order of operations for redo logging is: Log record → Commit record → Data element. This allows the system to keep changed data in memory buffers longer, potentially reducing disk I/O, as the log provides a way to "redo" the work if memory is lost.

10.6 Undo/Redo Logging and Checkpointing

A hybrid approach, undo/redo logging, records both the old and new values of a database element ($< T, X, v, w >$). This provides the highest level of flexibility, as the commit record can be written either before or after the data elements are flushed to disk.

Notes:-

Undo/redo logging is the most common method in modern DBMSs because it allows the buffer manager to be more efficient. It only requires that the log record for a change reach the disk before the change itself does.

To avoid scanning the entire log during recovery, the system uses checkpointing.

Definition 10.6.1: Nonquiescent Checkpointing

A technique that allows the system to mark a "safe" point in the log without shutting down the database. It records the set of active transactions and ensures that all data changed by previously committed transactions has reached the disk.

10.7 Concurrency Control and Serializability

When multiple transactions run at once, their actions form a schedule. The goal of the scheduler is to ensure that this schedule is serializable.

Definition 10.7.1: Conflict-Serializable Schedule

A schedule that can be transformed into a serial schedule (where transactions run one after another) by a sequence of swaps of adjacent, non-conflicting actions.

Theory 10.7.1 The Precedence Graph Test

A schedule is conflict-serializable if and only if its precedence graph—where nodes are transactions and arcs represent conflicts—is acyclic. A conflict occurs if two transactions access the same element and at least one is a write.

10.8 Lock-Based Schedulers and Two-Phase Locking

The most common way to enforce serializability is through the use of locks. Before a transaction can access a database element, it must obtain a lock on that element from the scheduler's lock table.

Definition 10.8.1: Shared and Exclusive Locks

A Shared (S) lock allows multiple transactions to read an element simultaneously. An Exclusive (X) lock is required for writing and prevents any other transaction from reading or writing that element.

Simply taking locks is insufficient; the timing of when locks are released is vital for maintaining a consistent state.

Theory 10.8.1 Two-Phase Locking (2PL)

A protocol requiring that in every transaction, all lock acquisitions must precede all lock releases. This creates a "growing phase" where locks are gathered and a "shrinking phase" where they are surrendered.

Notes:-

Strict Two-Phase Locking is a variation where a transaction holds all its exclusive locks until it commits or aborts. This prevents other transactions from reading "dirty data"—values written by uncommitted transactions—and eliminates the need for cascading rollbacks.

10.9 Deadlock Management

Locking systems are inherently prone to deadlocks, where transactions are stuck in a cycle of waiting for one another. Schedulers must implement strategies to detect or prevent these states.

Definition 10.9.1: Waits-For Graph

A directed graph used for deadlock detection. Nodes represent transactions, and an arc from T to U indicates that T is waiting for a lock currently held by U. A cycle in this graph indicates a deadlock.

Prevention strategies often involve timestamps. Two popular methods are:

- **Wait-Die:** If an older transaction needs a lock held by a newer one, it waits. If a newer transaction needs a lock held by an older one, it dies (rolls back).
- **Wound-Wait:** An older transaction "wounds" (forces a rollback) a newer transaction that holds a lock it needs. A newer transaction must wait for an older one.

10.10 Alternative Concurrency Control: Timestamps and Validation

Beyond locking, systems may use optimistic concurrency control methods, which are particularly effective when conflicts are rare.

Definition 10.10.1: Timestamp-Based Scheduling

A method where each transaction is assigned a unique timestamp when it begins. The scheduler maintains read and write times for every database element and rolls back any transaction that attempts to perform a "physically unrealizable" action, such as reading a value written in its future.

Definition 10.10.2: Validation-Based Scheduling

An optimistic approach where transactions execute in a private workspace. Before committing, the transaction enters a validation phase where the scheduler checks its read and write sets against those of other active transactions to ensure no serializability violations occurred.

10.11 Long-Duration Transactions and Sagas

In environments like design systems or workflow management, transactions can last for hours or even days. Holding locks for such durations would paralyze the system.

Definition 10.11.1: Saga

A long-duration transaction consisting of a sequence of smaller, independent actions. Each action is its own transaction that commits immediately.

Theory 10.11.1 Compensating Transactions

For every action A in a saga, there must be a corresponding compensating transaction A^{-1} that logically undoes the effects of A. If the saga must abort, the system executes the compensating transactions in reverse order to return the database to a consistent state.

Notes:-

A saga does not strictly follow the traditional "Isolation" property of ACID, as the results of its intermediate actions are visible to other transactions. However, through the use of compensation, it maintains the logical consistency of the system.

In conclusion, transaction management requires a delicate balance between performance and correctness. By combining robust logging for durability, strict locking for isolation, and innovative structures like sagas for long-term processes, modern database systems provide a stable foundation for complex information ecosystems. These mechanisms ensure that even in the event of hardware failure or intense concurrent demand, the integrity of the data remains unassailable.

To visualize a transaction, think of it as a set of instructions for a complicated recipe. If you get halfway through and realize you are missing a vital ingredient, you cannot just stop and leave the half-mixed dough on the counter. You must either finish the recipe or clean up the mess so the kitchen is exactly as it was before you started. The database scheduler and log manager are like the head chef, ensuring that every cook has the tools they need and that no one's flour ends up in someone else's soup.