📚 CSE 132A Data Management Overview

Brief Overview

This note covers **Data Management** and was created from a PDF presentation of 19 pages on CSE 132A lecture slides. It provides an introduction to database fundamentals, including system architecture, data models, and key concepts that underpin modern data-centric applications.

Key Points

- Grasp the **relational model** and its importance in structuring data.
- Master **SQL** syntax for querying and manipulating data.
- Understand the three-level database architecture and data independence.
- Learn how ER modeling and normal forms guide effective schema design.

📅 Logistics

- **Lectures**: in-person and podcast
- **Attendance**: optional but strongly recommended
- **Slides**: posted prior to lecture
- **Weekly discussion**: on-demand Zoom session
- **Office hours**: TA and instructor, both in-person and via Zoom
- **Discussion board**: Piazza
- **Exams**: in-person only
- **Materials**: all posted on Canvas or Piazza

📚 Data Management Overview

- **Field**: rapidly evolving, expanding beyond classical stand-alone DBMS (SQL Server, Oracle, DB2)
- **Trends**: data-centric computer science → data science = data management + machine learning
- **Applications**: scientific databases, networks, data mining, streaming sensor monitoring, social networks, bioinformatics, GIS, digital libraries, data-driven web services, analytics
- **Core**: classical database concepts and algorithms remain foundational → focus of this course

🎯 Course Scope (CSE 132A)

- **Core concepts**: relational model, SQL, query processing, schema design, transactions, concurrency control
- **Theoretical foundations**: relational algebra, relational calculus, formal languages underlying SQL
- **Pathways**: prepares for 132B (applications), 132C (implementation), 135 (online analytics), 190 (beyond relational)

📋 Requirements

Assessment	Weight
5 homework assignments (individual) – 3 SQL programming, 2 non-programming	—
Midterm	30 %
Final	30 %
Academic Integrity	– (policy posted on class website)

🗃️ What Is a Database?

> *A database is a persistent collection of data together with a query and update language for accessing and modifying that data, supporting query optimization, and providing transaction and concurrency control.*

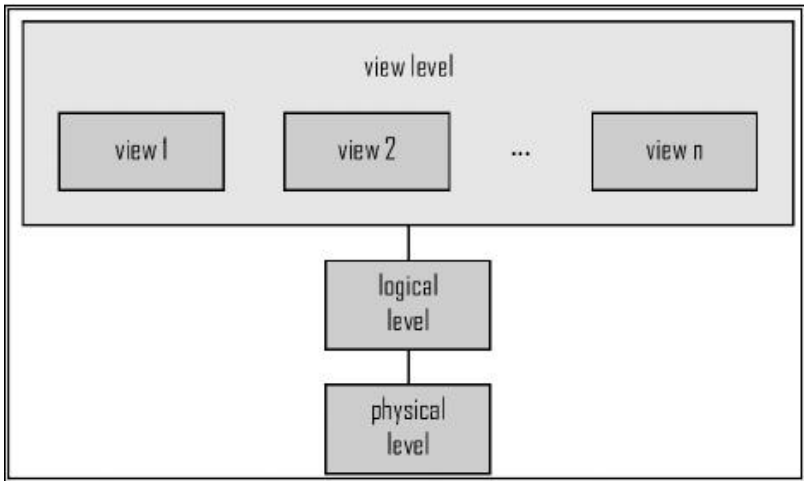
- **Typical data**: many instances of similarly structured records (e.g., airline reservations, library catalogs, medication advisors)

🎈 Top-Level Goals of a Database System

- **Meaning-based view**: abstract away irrelevant details for users
- **Support operations**: queries and updates
- **Data control**: security, integrity, recovery, concurrency

🏗️ Basic Architecture of a Database System

The architecture consists of three abstraction levels:



> *The diagram illustrates the hierarchical relationship: the **view level** (multiple user-specific views) sits atop the **logical level** (schema description), which in turn sits above the **physical level** (actual storage).*

- **Data independence**: logical and physical levels are independent, allowing changes in storage without affecting logical schema

🔍 Levels of Abstraction

- **Logical level**: describes data in terms close to applications (e.g., a flight reservation table)
- **Physical level**: specifies how data is stored and processed on disk
- **View level**: customized, possibly restricted, representations of data (e.g., hiding employee salaries for security)

🖥️ Database System vs. DBMS

Aspect	Database System	Database Management System (DBMS)
Scope	Tailored to a specific application	Generalized, usable across many applications
Responsibilities	Application-specific logic, data handling	Data organization, storage, access, control

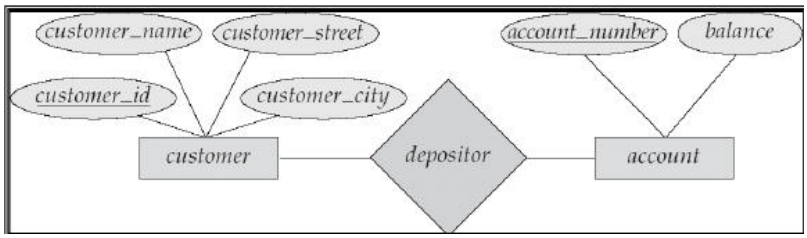
Examples	A hospital's patient-record system	PostgreSQL, MySQL, Oracle, DB2, SQL Server

📊 Data Models

- **Purpose**: concepts & tools for describing relationships, semantics, constraints; includes a query/modification language
- **Major models**:
  - *Relational model*
  - *Entity-Relationship (ER) model* – primarily for design
  - *Object-oriented / Object-relational models*
  - *Semi-structured* (XML, graphs)
  - *Older* models: network, hierarchical

🧩 Example: Entity-Relationship Model

The ER model represents an application as entities (objects) with attributes and relationships among them.



> *In this diagram, **Customer** and **Account** are linked by the associative entity **Depositor**, indicating a many-to-many relationship.*

📐 Schemas and Instances

- **Schema**: logical structure of the database (analogous to a variable's type)
- **Instance**: actual data at a point in time (analogous to a variable's value)

📘 Relational Model – Schema Example

Relation	Attributes

Customer	customer_id (PK), customer_name, customer_street, customer_city
Account	account_number (PK), balance
Depositor	customer_id (FK), account_number (FK)

*An instance would contain concrete rows for each relation, e.g., a specific customer "C001", account "A123", and a linking row in **Depositor**.*

🛠️ Data Definition Language (DDL)

> *DDL defines the database schema.*

Example in SQL:

```
create table account (
    account_number char(10),
    balance integer
);
```

- **DDL compiler** → populates the *data dictionary* (metadata) with:
  - schema definitions
  - integrity constraints
  - authorization information

📦 Data Manipulation Language (DML)

> *DML provides a language for accessing and modifying data.*

- Two query language styles:
  1. **Declarative (non-procedural)** – specify *what* data is needed (e.g., SQL)
  2. **Procedural** – specify *how* to obtain the data
- **SQL** is the dominant declarative language.

📚 Core Topics Covered in This Course

1. Relational model
2. Standard relational query language: **SQL**
3. Formal query languages: relational algebra & calculus
4. Query processing
5. Schema design: normal forms, ER model
6. Concurrency control
7. Additional topics as time permits

🏫 Databases at UCSD

- Faculty: Alin Deutsch, Arun Kumar, Victor Vianu
- Group website: https://dbucsd.github.io/ (papers, seminars, etc.)
- Intersections with other CSE groups: storage, multimedia, machine learning, networks
- Collaboration with SCIDS (e.g., Prof. Babak Salimi, Dr. Amarnath Gupta)

📚 Algebra & Relational Overview

Brief Overview

This note covers **Algebra** and was created from a 44-page PDF presentation. The notes cover algebraic structures, time-based algebras, tuples, relational algebra, JSON data.

Key Points

- Definitions of algebras and examples (natural numbers, set, Boolean).
- Time-based algebra operations and relations.
- Core relational algebra operations (selection, projection, join, division).
- JSON as semi-structured data representation.

Algebras and Data Models 🔢

What is an Algebra?

An **algebra** is a mathematical structure consisting of:

1. **A Set (A)**: The domain or universe of elements
2. **A Signature (F)**: A set of operations defined on the set A
3. **Axioms (Optional)**: Properties that the operations must satisfy

> **Formal definition**: An algebra is a tuple $A = (A, {f_i}*{i \in I})where A is the carrier set and {f_i : A^n \to A}*{i \in I}$ is a family of n-ary operations.

Examples of Algebras

Natural Numbers Algebra

The algebra of natural numbers under addition and multiplication:

- $A = \mathbb{N}$ (set of natural numbers)
- Operations: $+$, $\times$ (both binary operations: $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$)

Set Algebra

Formal representation: $A = (P(U), \cup, \cap, |, \emptyset, U)$

Components:

- **Carrier Set**: $P(U)$ (power set of universe U)
- **Operations**:
  - **Union**: $A \cup B$
  - **Intersection**: $A \cap B$
  - **Set Difference**: $A|B$ (elements in A but not in B)
  - **Complement**: $\overline{A}$ (elements in $U|A$)
- **Distinguished Elements**: $\emptyset$ (empty set), U (universal set)

Properties:

- **Associativity**: $(A \cup B) \cup C = A \cup (B \cup C)$
- **Commutativity**: $A \cup B = B \cup A$
- **Distributivity**: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
- **Identity**: $A \cup \emptyset = A$, $A \cap U = A$
- **Complement**: $A \cup \overline{A} = U$, $A \cap \overline{A} = \emptyset$

Boolean Algebra

Formal representation: $B = (B, \vee, \wedge, \neg, 0, 1)$

Components:

- **Carrier Set**: $B = 0, 1$
- **Operations**: OR ($\vee$), AND ($\wedge$), NOT ($\neg$)
- **Distinguished Elements**: 0 (false), 1 (true)

Properties:

- **Commutativity**: $a \vee b = b \vee a$
- **Associativity**: $(a \vee b) \vee c = a \vee (b \vee c)$
- **Distributivity**: $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
- **Identity**: $a \vee 0 = a$, $a \wedge 1 = a$
- **Complement**: $a \vee \neg a = 1$, $a \wedge \neg a = 0$

Time-Based Algebras ⏰

Algebra of Time Points

Carrier Set (P): All possible time points (e.g., timestamps)

Operations:

- **Addition**: $f_{add}(p, d) = p + d$ (add duration d to time point p)
- **Subtraction**: $f_{sub}(p, d) = p - d$ (subtract duration d from time point p)

Relations:

- **Before**: $p_1 < p_2$ (true if p_1 occurs before p_2)
- **After**: $p_1 > p_2$ (true if p_1 occurs after p_2)
- **Equals**: $p_1 = p_2$ (true if p_1 equals p_2)

Algebra of Time Intervals

Carrier Set (I): $I = [p_1, p_2] | p_1, p_2 \in P, p_1 < p_2$

Operations:

- **Interval Addition**: $f_{add}([p_1, p_2], d) = [p_1 + d, p_2 + d]$
- **Interval Intersection**: $f_{intersect}([p_1, p_2], [q_1, q_2]) = [\max(p_1, q_1), \min(p_2, q_2)]$
- **Interval Union**: Combines intervals if they overlap or are adjacent

Relations:

- **Overlaps**: True if intervals share common time points
- **Before**: True if first interval ends before second begins
- **Contains**: True if one interval fully contains another

Tuples and Many-Sorted Algebras 📊

Tuples

A **tuple** is a finite ordered collection of elements where each element can be of different type.

Definition: $t = (t_1, t_2, ..., t_n)$ where:

- n: arity (length) of the tuple
- t_i: i-th element from domain D_i

Key Properties:

- **Ordered**: Order matters $(1, 2) \neq (2, 1)$
- **Finite Length**: Fixed number of elements

- **Heterogeneous Types**: Elements can be from different domains

Operations for Atomic Tuples:

Operation	Definition	Example
Accessing Elements	Retrieves i-th element	$(1, 2, 3)[2] = 2$
Concatenation	Combines two tuples	$(1, 2) + (3, 4) = (1, 2, 3, 4)$
Projection	Extracts specific attributes	$\pi_{1,3}((1, 2, 3)) = (1, 3)$
Slicing	Extracts contiguous subset	$\text{slice}((1, 2, 3, 4, 5), 2, 4) = (2, 3, 4)$
Tuple Equality	Checks equality	$(1, 2, 3) = (1, 2, 3)$ → true

Many-Sorted Algebra

A **many-sorted algebra** generalizes single-sorted algebra by allowing multiple carrier sets.

Formal definition: $A = (S, \{A_s\}_{s \in S}, \{f_i\}_{i \in I})$ where:

- S: Finite set of sorts (types)
- $A_{s s \in S}$: Family of carrier sets, one per sort
- $f_{i i \in I}$: Family of operations between sorts

Example: 2-sorted Algebra:

- **Sorts**: String, Number
- **Carrier Sets**: $A_{string} = \text{"apple"}, \text{"banana"}, \text{"cherry"}$, $A_{number} = 1, 2, 3, 4, 5, 6$
- **Operations**:
  - String concatenation within strings
  - Addition/multiplication within numbers
  - Cross-sort operations (e.g., repeat string by number)

Relational Algebra 🗄️

Core Concepts

Relational algebra is a many-sorted algebra with sorts:

- **Domains (D)**: Sets of atomic values
- **Attributes (A)**: Names associated with domains
- **Tuples (T)**: Ordered collections of values
- **Relations (R)**: Sets of tuples with schema

Basic Operations

Selection

$$R_1 := \sigma_C(R_2)$$

- **C**: Boolean condition on attributes
- **Result**: Tuples from R_2 satisfying condition C

Projection

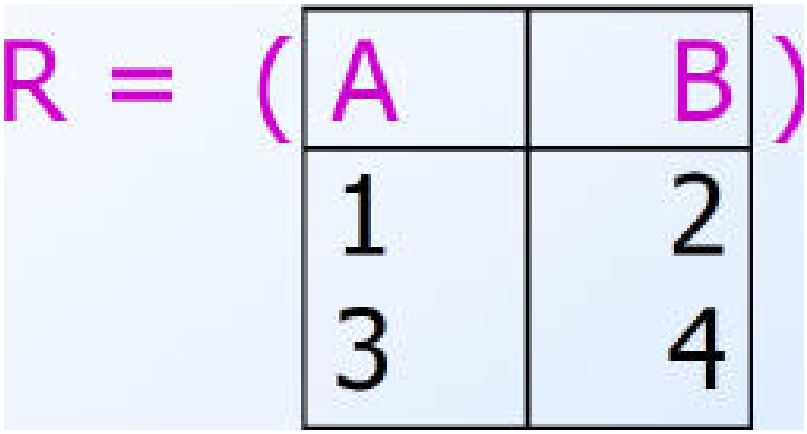
$$R_1 := \pi_L(R_2)$$

- **L**: List of attributes to extract
- **Result**: Tuples with only specified attributes, duplicates removed

Extended Projection

Allows expressions in projection list:

- Arithmetic operations
- Concatenation
- Duplicate attributes

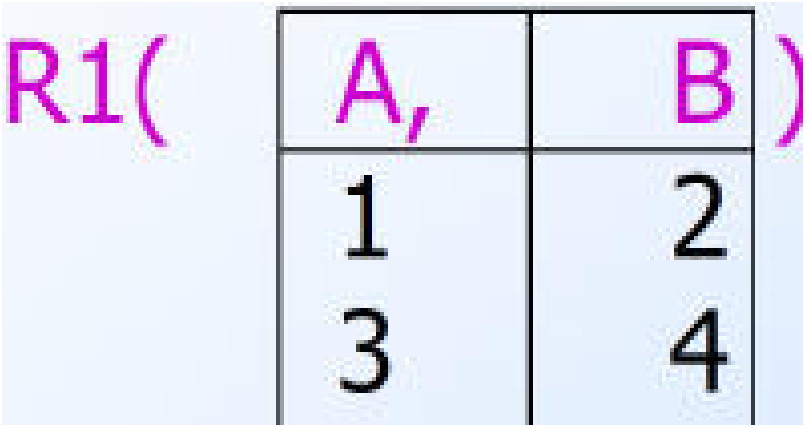


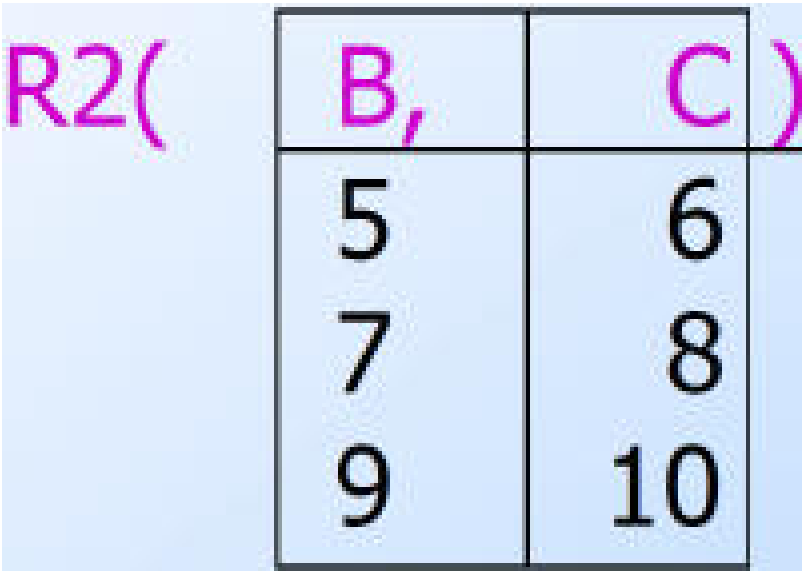
The image demonstrates how extended projection can perform calculations on attributes, transforming a simple 2×2 matrix relation into a new relation with computed values.

Product (Cartesian)

$R_3 := R_1 \times R_2$

- Pairs each tuple from R_1 with each tuple from R_2
- Schema: Attributes of both relations (use prefixes for duplicates)





These images show the component relations used in product operations, illustrating how tuples from different relations are paired together.

Join Operations

Theta Join: $R_3 := R_1 \bowtie_C R_2$

- Product followed by selection with condition C

Natural Join: $R_3 := R_1 \bowtie R_2$

- Equates same-named attributes
- Projects out duplicate attributes

Division

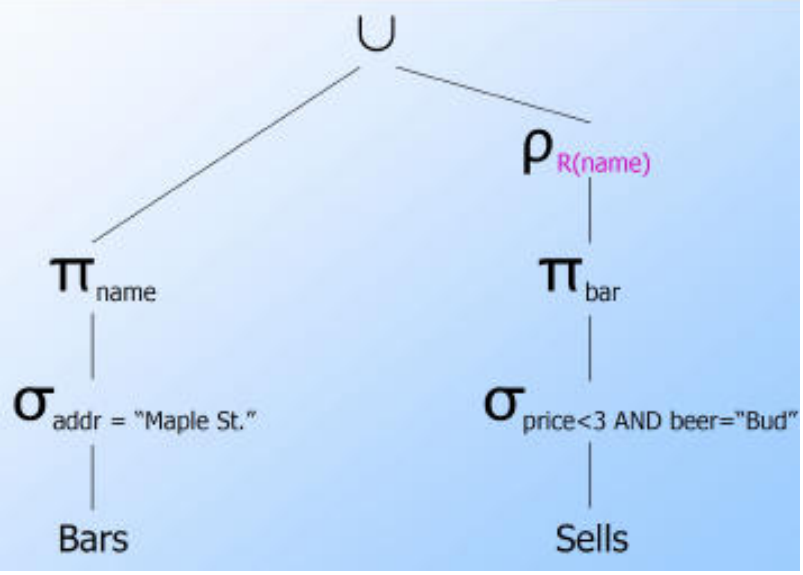
$$R \div S = x \in \pi_X(R) | \forall y \in S, (x, y) \in R$$

- Finds X values associated with every y in S
- Captures "for all" queries

Query Trees and Complex Expressions

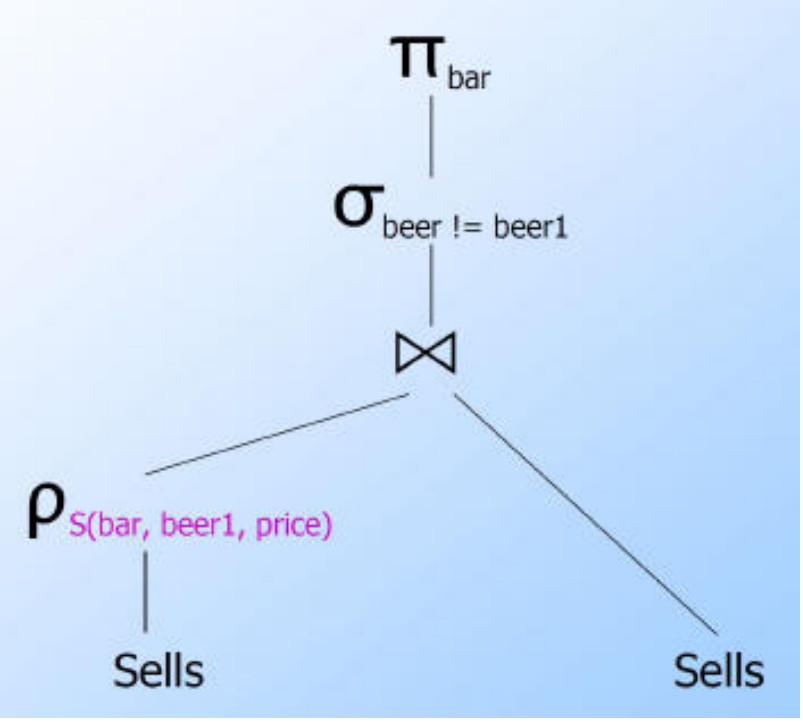
Operator Precedence:

1. $[\sigma, \pi, \rho]$ (highest)
2. $[\times, \bowtie]$
3. $[\cap]$
4. $[\cup, -]$ (lowest)



This query tree illustrates how to combine selection, projection, and union operations to find bars meeting specific criteria, demonstrating the visual representation of complex relational algebra expressions.

Self-Join Example: Finding bars selling different beers at same price using relation renaming and natural join.



The image shows how self-join operations work by creating a copy of a relation, renaming it, and then joining to compare tuples within the same original relation.

Bag (Multiset) Operations

Bag vs Set differences:

- Elements can appear multiple times
- Union: Sum of occurrences
- Intersection: Minimum occurrences
- Difference: Subtract occurrences (minimum 0)

Bag Laws:

- Commutative law holds
- Idempotent law fails: $S \cup S \neq S$ when S is a bag

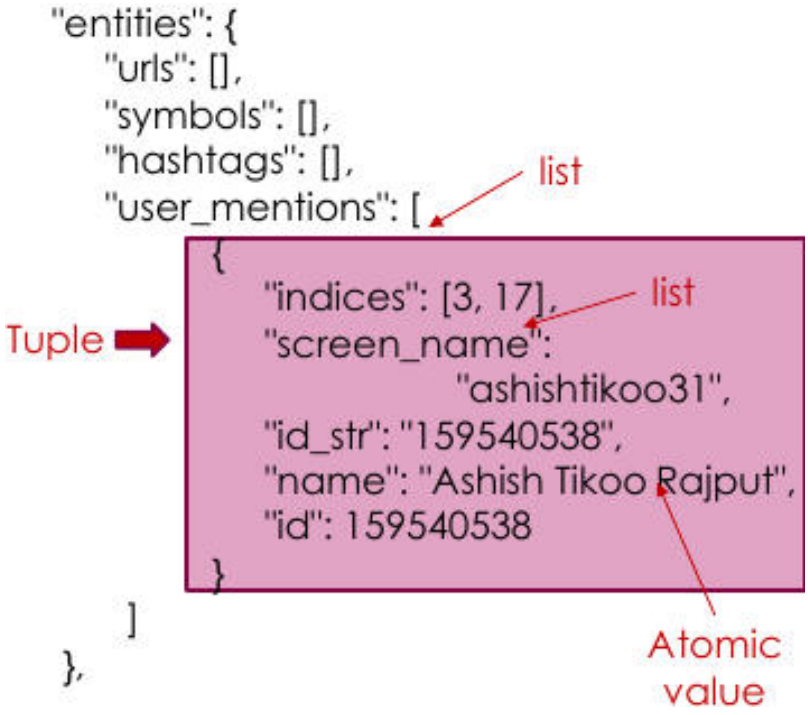
Semi-Structured Data 📱

JSON as Semi-Structured Data

Semi-structured data lacks rigid schema but contains tags/markers to separate elements.

Example JSON Structure:

- **Tuples**: Ordered collections with different types
- **Lists**: Ordered sequences
- **Atomic values**: Strings, numbers, booleans



This image demonstrates the hierarchical nature of JSON data, showing how semi-structured data can nest different data types (tuples, lists, atomic values) to represent complex real-world information like social media data.

📖 ER Modeling Fundamentals

Brief Overview

This note covers **entity-relationship modeling** and was created from a 36-page PDF presentation. It provides a step-by-step guide through entities, attributes, and relationships, with a focus on **ER diagrams**, cardinality, and advanced ER concepts.

Key Points

- Quick reference to entity sets, attribute types, and relationship cardinality.
- Visual examples of ER diagrams for customers, orders, products, and an educational support system.
- Practical guidance on handling composite, multi-valued, derived, and weak entities.

📦 Entities

> **Entity**: An “object” or “thing of interest” that stands alone with its own properties, is distinguishable from other objects, and has a unique name in the schema.

- Examples of entity sets:
  - University schema → **Student**, **Faculty**, **Course**, **Classroom**
  - Shop schema → **Receipt**, **Customer**, **Employee**, **Product**
  - Hospital schema → **Doctor**, **Patient**, **Nurse**, **Disease**, **Procedure**, **Medicine**

🔎 Attributes

> **Attribute**: A property of an entity; an entity may have one or many attributes.

- **Attribute Types**

Type	Description	Example
Single-valued	Holds one atomic value	Name

Composite	Consists of multiple sub-attributes	FullName (First, Middle, Last)
Multi-valued	Can store several values for the same entity	PhoneNumbers
Nested	Attributes that themselves contain sub-attributes (e.g., an address)	Address (StreetNumber, StreetName, AptNumber, City, State, Zip)
Derived	Computed from other attributes	Age from DateOfBirth

- Sample student attributes: Name, Age, Student_Id, Year_of_study, Current_GPA, Phone_number, Email_address, Home_Address, Current_Address.

🔗 Relationships

> **Relationship**: An association between two or more entities; may also possess its own attributes.

- Captures how rows in one table relate to rows in another.
- Can be **binary**, **ternary**, or **n-ary** depending on the number of participating entities.

Types of Relationships – Cardinality

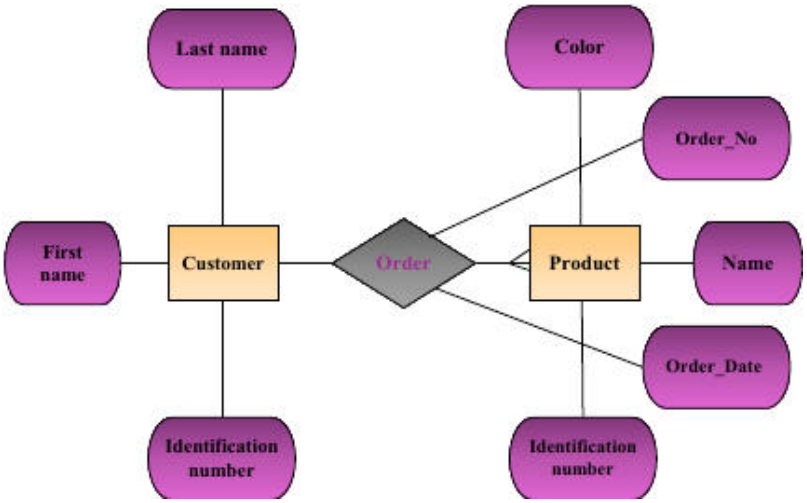
Cardinality	Symbol	Description	Example
One-to-One	1:1	Each entity in set A relates to at most one entity in set B	Student ↔ PID
One-to-Many	1:N	One entity in A relates to many in B	Instructor → multiple email addresses
Many-to-One	N:1	Many entities in A relate to one in B	Students → College
Many-to-Many	N:M	Entities in both sets can have multiple associations	Student ↔ Course enrollment

Types of Relationships – Entity Count

- **Binary** – involves two entities (e.g., Student-Course).
- **Ternary** – involves three entities (e.g., Student-Course-Instructor).
- **N-ary** – involves more than three (e.g., Student-Course-Classroom-Term).

📊 What an ER Model Looks Like

The diagram below illustrates an ER model for customers, orders, and products.
It shows entities as rectangles, attributes as ovals, and the central **Order** diamond linking **Customer** and **Product**.



The visual helps see how an order serves as a junction (many-to-many) between customers and products, with attributes such as Order_No and Order_Date attached to the relationship.

❓ Example Query Questions

- Identify a customer's **Identification_number**.
- Retrieve the **color** of a specific product.
- Find the **date** of a particular order.
- List all products whose color is **red**.
- Determine whether **customer a** bought **product p** and, if so, the order date.
- Count how many times **customer a** purchased **product p**.
- Find customers who bought **products a** and **b** in the same order.
- Locate the first purchase date for **customer a**.
- Insert a new **unique** customer.
- Insert a new **unique** order linked only to known customers and products.

🏫 Creating the ER Model for EdSupp

Scenario Overview

- Backend database for an educational support system (Q&A handling & grading).
- **Classes** are instances of **Courses**; each class has users (students or teaching staff) who can post comments, questions, or announcements.
- Assignments belong to classes; grades are recorded per assignment and per class.
- Instructors need analytics (top students, grade distribution, etc.).

Detailed Entity & Attribute Specification

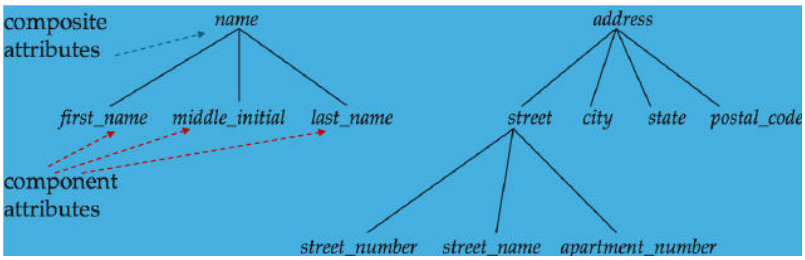
Entity	Primary Key	Key Attributes	Additional Attributes
User	user_id (system-generated)	first_name, last_name, email	phone_numbers (type + number, multivalued)
Student (subclass of User)	user_id	major	—
Instructor (subclass of User)	user_id	title (Full Prof, Assoc Prof, Asst Prof, Lecturer, Guest)	—
Course	Composite key (dept, course_number)	dept, course_number	title, level (Lower/Upper Division, Graduate)
Class	Composite key (course_id, class_number)	class_number	start_date, end_date, max_students, term, year
Assignment	assignment_id	assignment_name, deadline	class_id (FK)
Grade	Composite key (student_id, assignment_id)	grade_value	—

Post (Comment/Question/Announcement)	post_id	content, post_type	user_id (FK), class_id (FK), timestamp

Relationship Overview

- **User ↔ Class** – *Enrolls* (students) or *Teaches* (instructors).
- **Class ↔ Assignment** – *Contains*.
- **Student ↔ Assignment** – *Receives* grade.
- **User ↔ Post** – *Creates*.

ER Diagram for EdSupp (excerpt)



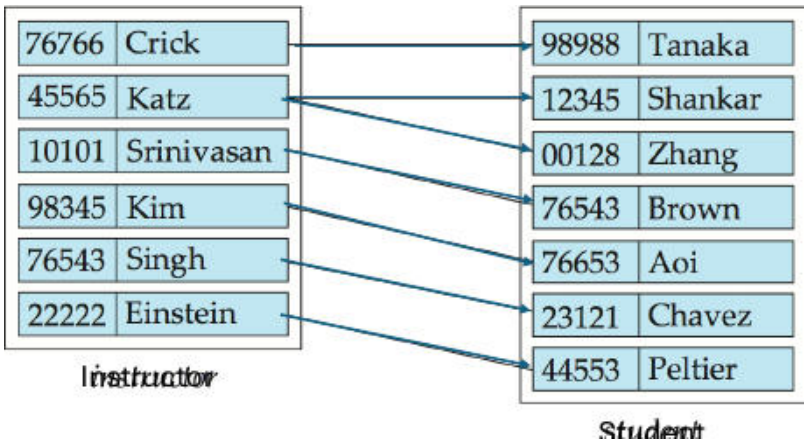
*The diagram demonstrates composite attributes (e.g., name broken into first/middle/last) and multivalued attributes (multiple phone numbers). It clarifies how subclasses inherit the primary key from **User** while adding role-specific attributes.*

🔑 Keys in ER Design

- A **key** is a minimal set of attributes that uniquely identifies an entity.
- Example keys:
  - IID → I_name for **Instructor**.
  - SID → S_name for **Student**.

👨‍🏫 Advisor Relationship Example

The schema below models an instructor advising students (one-to-many).



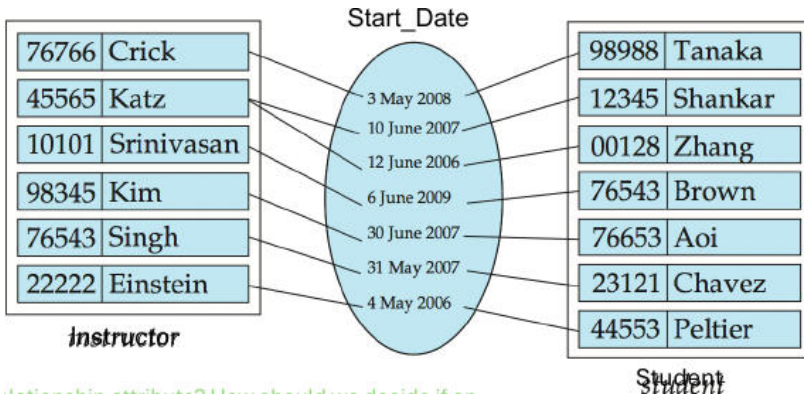
One instructor may have many advisees, while each student has exactly one advisor.

Cardinality Questions

- **What if** an instructor has no advisee? → Allow optional participation on the instructor side.
- **What if** a student can have multiple advisors? → Change to a many-to-many relationship, adding an associative entity (e.g., AdvisorAssignment).

📐 Attributes in Relationships

A relationship can own attributes (e.g., Start_Date linking instructors and students).



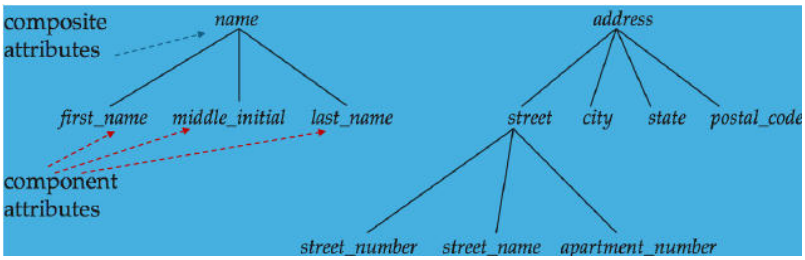
*The Start_Date attribute belongs to the **advisor** relationship rather than either entity.*

> **Relationship Attribute**: An attribute that describes a property of the association itself, not of the participating entities.

🧩 Complex Attributes

- **Simple vs. Composite** – Composite attributes are broken into atomic components for relational storage.
- **Multi-valued** – Requires a separate relation (e.g., PhoneNumber(user_id, type, number)).
- **Derived** – Computed on demand (e.g., age() from date_of_birth).
- **Domain** – Permitted set of values for an attribute (e.g., title ∈ {Full Professor, ...}).

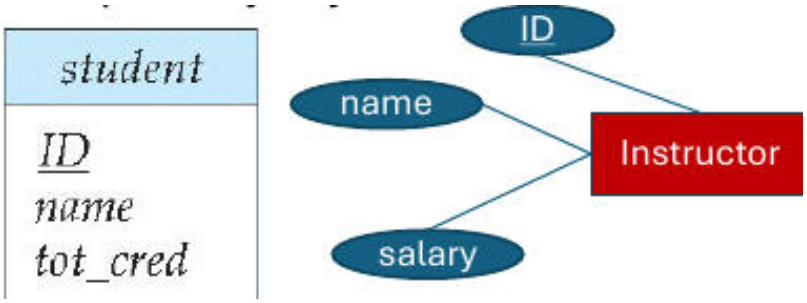
🏗️ Composite Attribute Example



The name attribute decomposes into first_name, middle_initial, last_name; the address attribute further splits into street details, city, state, zip.

📦 Entity Sets

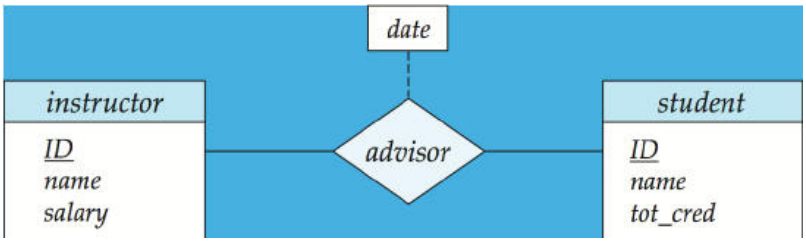
- Represented by rectangles; primary key attributes are underlined.



🔗 Relationship Sets

- Represented by diamonds; can also carry attributes.

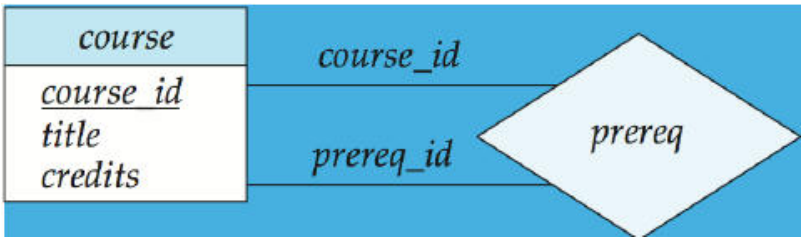
With Attributes



The date attribute records when an advisor relationship was established.

🎭 Roles in Relationships

- An entity set can appear multiple times in a relationship, each occurrence playing a distinct **role** (e.g., course_id vs. prereq_id).

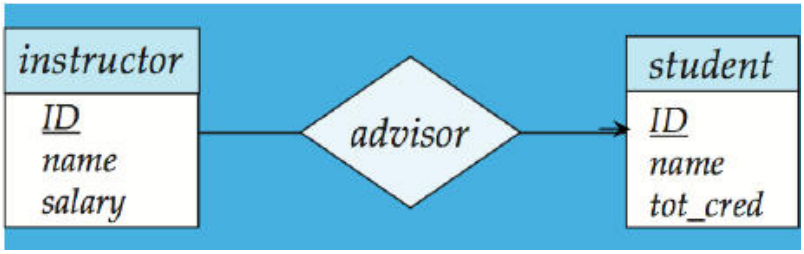


📏 Cardinality Constraints

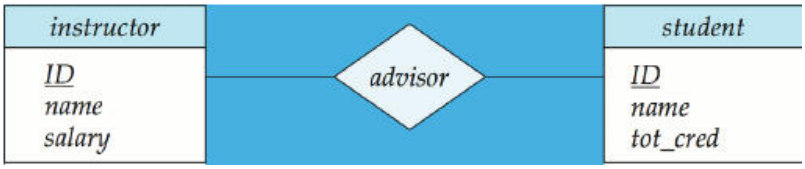
- **Directed line (→)** → “one”.
- **Undirected line (—)** → “many”.

One-to-Many Example

Many-to-One Example

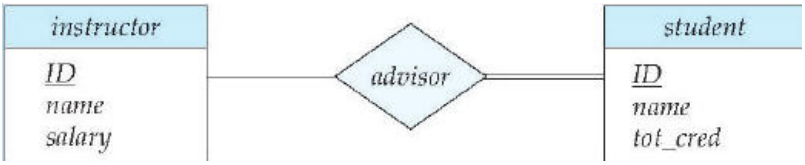


Many-to-Many Example



⭕ Total vs. Partial Participation

- **Total participation** (double line) → every entity must be involved in the relationship.
- **Partial participation** → some entities may have no related tuple.



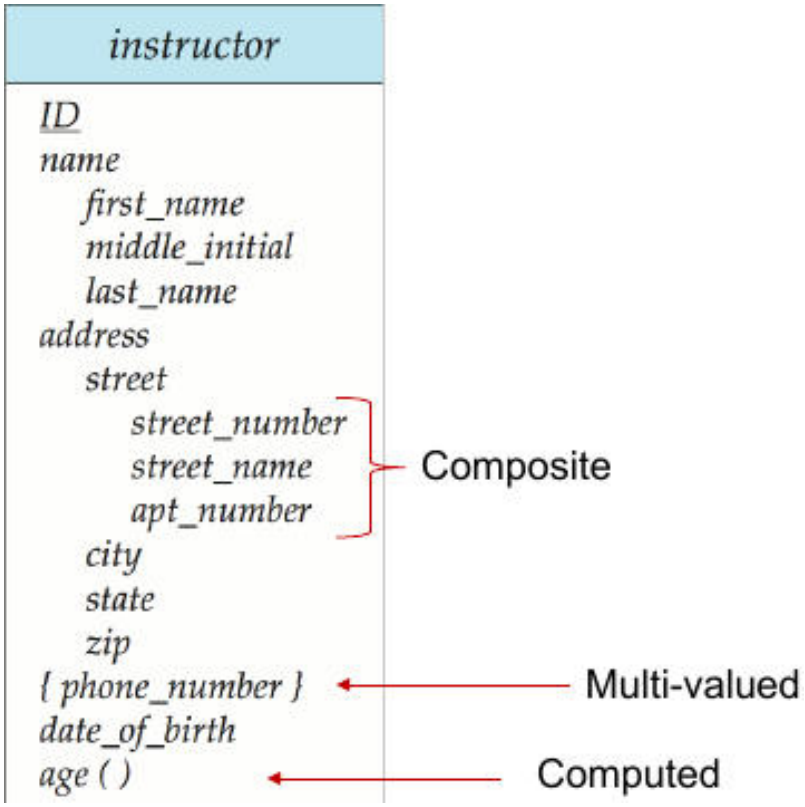
Students have total participation in the advisor relationship (each must have an advisor); instructors have partial participation (may advise none).

📏 Expressing Complex Cardinalities

- Use notation l..h on relationship lines:
  - l = minimum cardinality (1 → total participation).
  - h = maximum cardinality (1 → at most one, * → unlimited).

Example: Instructor → 0..* students, Student → 1..1 instructor.

🗃️ Notation for Complex Attributes



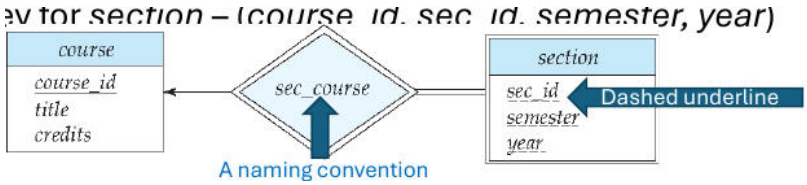
- Composite attributes (address, name) are broken into components.
- Multi-valued attribute {phone_number} indicates a set of values.
- Derived attribute age() is computed, not stored.

🏚 Weak Entities

- A **weak entity** lacks its own primary key and borrows identifying attributes from a strong entity via an **identifying relationship**.
- The weak entity's primary key = strong entity's key + discriminator(s).

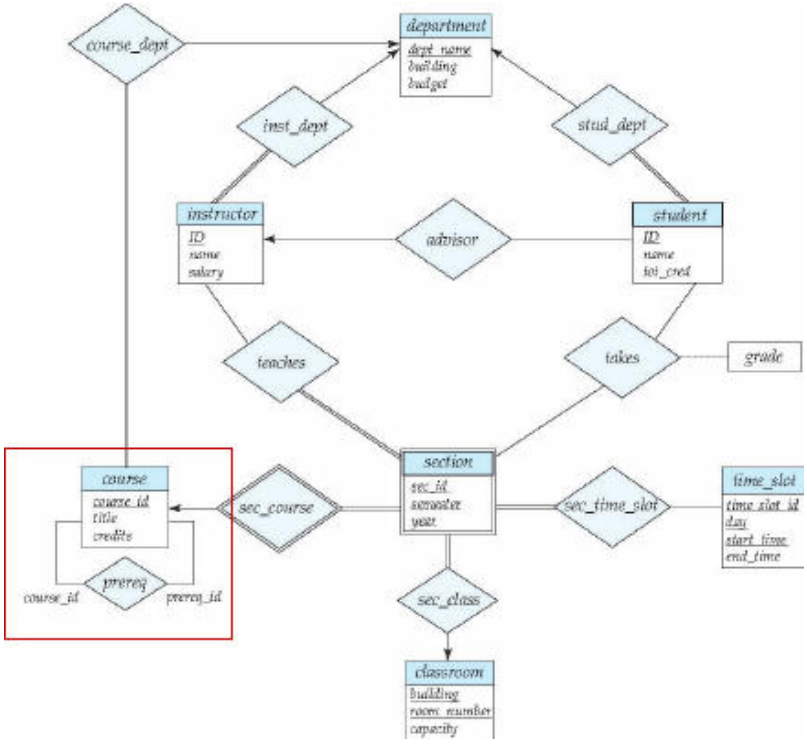
Visual Representation

- Double rectangle for the weak entity.
- Dashed underline for the discriminator.
- Double diamond for the identifying relationship.



Section is weak; its key combines course_id (from Course) with sec_id, semester, year.

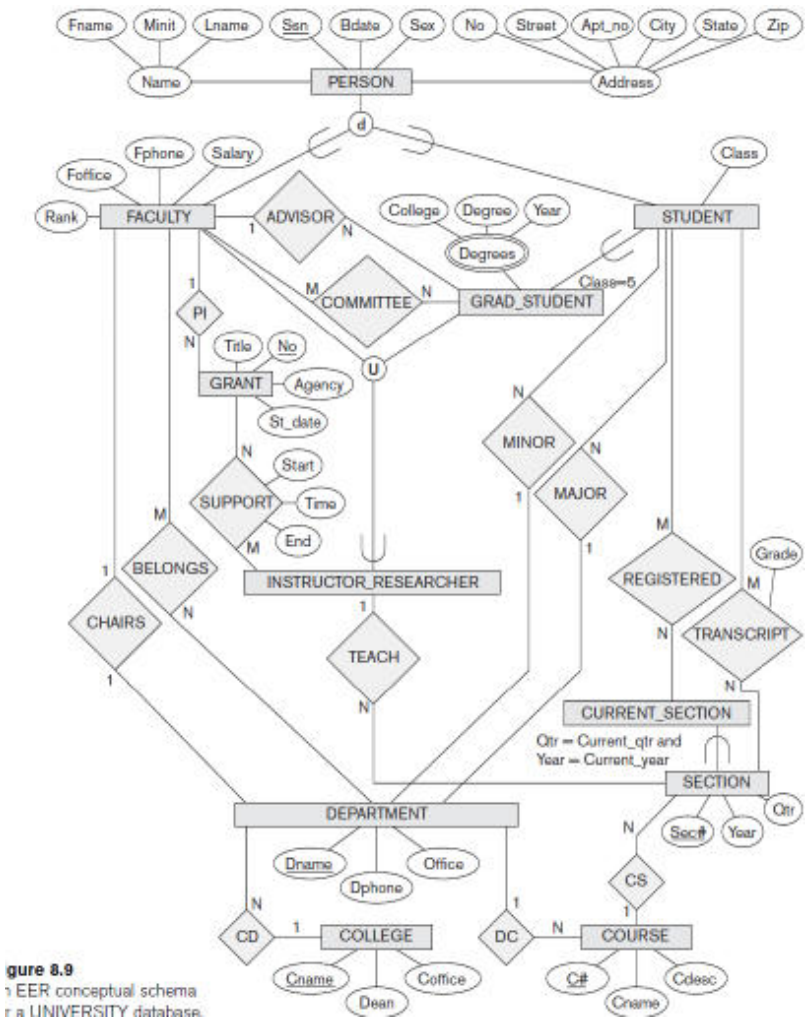
🏛️ University Enterprise ER Diagram



- Shows entities: **Course**, **Instructor**, **Department**, **Student**, **Section**, **Classroom**, **TimeSlot**, etc.
- Includes self-referential prerequisite relationship (course ↔ prerequisite).

📚 Extended ER Model

- Adds **generalization/specialization**, **disjoint unions**, and **additional constraints** (aggregations not covered).



gure 8.9
1 EER conceptual schema
r a UNIVERSITY database

The diagram demonstrates subclass entities (e.g., GraduateStudent, InstructorResearcher) and their inheritance of attributes from Person.

🔒 Database Concurrency & Recovery

Brief Overview

This note covers **Database Concurrency & Recovery** and was created from a 24-page PDF presentation. The note dives into **locking mechanisms**, transaction isolation, recovery strategies, and practical SQL transaction examples.

Key Points

- Lock types and compatibility tables
- Two-Phase Locking stages
- Isolation level guarantees
- WAL and checkpoint recovery

Concurrency Control 😊

> **Concurrency control** manages simultaneous transactions so they do not interfere with one another.

- **Objective:** allow many users to perform different operations at the same time.
- **Performance gain:** concurrent transactions increase system throughput and reduce waiting time.
- **Risk without control:** loss of data integrity and consistency.

Lock-Based Protocol 🔒

> A **lock** is a mechanism that controls concurrent access to a data item, ensuring that one process cannot retrieve or update a record that another process is updating.

Lock Modes

- **Shared lock (S)** – item can be **read** but not modified.
- **Exclusive lock (X)** – item can be **read and written**.

Compatibility of Lock Modes

Requested \ Existing	S	X	IS	IX
S	✓	✗	✓	✗
X	✗	✗	✗	✗
IS	✓	✗	✓	✓

IX	❌	❌	✓	❌

- Any number of transactions may hold **shared locks** on an item.
- An **exclusive lock** blocks all other locks on that item.

Lock Structure & Operations

Field	Description
object_id	Identifier of table, page, row, tuple, etc.
current granted mode	Current lock mode (S, X, IS, IX, ...)
holders	List of (transaction_id, mode) holding the lock
wait_queue	Queue of (transaction_id, requested_mode) waiting for the lock

Intention Locks (Hierarchy)

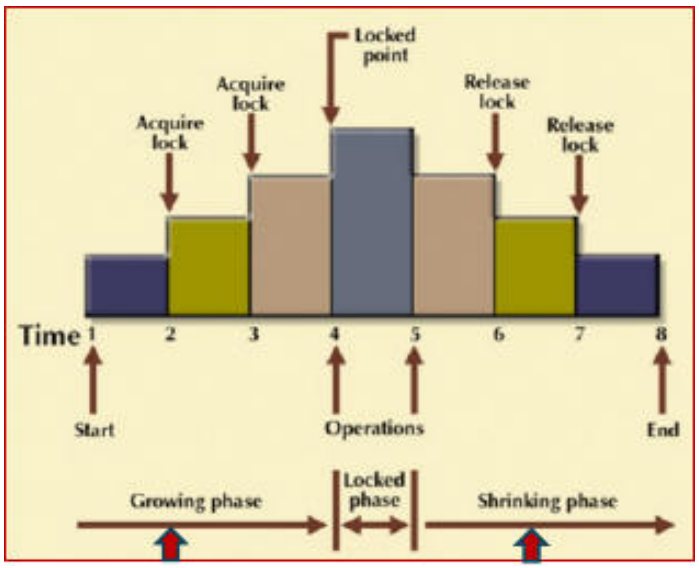
- **IS (Intention-Shared):** placed on a higher-level object (e.g., table) when a transaction intends to acquire **shared** locks on lower-level items (rows).
- **IX (Intention-Exclusive):** placed on a higher-level object when a transaction intends to acquire **exclusive** locks on lower-level items.

Example: To update a row in table **T**

1. Acquire **IX(T)**.
2. Acquire **X(row)**.
   The IX lock prevents another transaction from taking **IS(T)**, which would block the row update.

Two-Phase Locking (2PL) 🕒

> **Two-Phase Locking (2PL)** divides a transaction into a *growing phase* (acquiring locks) and a *shrinking phase* (releasing locks). No new locks are requested after the first release.



*The diagram shows the timeline of a transaction: the **growing phase** where locks are acquired, the **locked point**, and the **shrinking phase** where locks are released. This visual reinforces the rule that a transaction must not request new locks after it starts releasing them.*

- Guarantees **conflict-serializable** schedules.

Timestamp-Based Protocol ⏱️

> A **timestamp-based protocol** assigns a unique timestamp to each transaction, establishing a serializable order for conflicting reads and writes.

- Timestamp assigned when the transaction **enters** the system (using system time or a logical counter).
- Notation:
  - Transaction timestamp: **TS(T)**
  - Read-timestamp of data item **X**: **R-timestamp(X)**
  - Write-timestamp of data item **X**: **W-timestamp(X)**

Transaction Isolation Levels 🔐

> **Isolation** determines how the effects of one transaction are visible to others.

Isolation Level	Guarantees	Phenomena Prevented
Read Uncommitted	Allows reading uncommitted changes (dirty reads).	None
Read Committed	Reads only committed data; holds a lock on the current row.	Dirty read

Repeatable Read	Holds read locks on all rows read and write locks on modified rows.	Dirty read, non-repeatable read
Serializable	Execution appears as if transactions ran **serially**.	All three (dirty, non-repeatable, phantom)

> **PostgreSQL** does not implement *Read Uncommitted*; it defaults to *Read Committed*.

Phenomena Definitions

> **Dirty Read:** Transaction reads data written by another transaction that has not yet committed.

> **Non-Repeatable Read:** A transaction reads the same row twice and obtains different values because another transaction modified and committed the row in between.

> **Phantom Read:** Re-executing a query yields a different set of rows because another transaction inserted or deleted rows that satisfy the query's condition.

Database Recovery 💾

> **Recovery** restores a database to a correct state after a failure, ensuring consistency.

- **Rolling Forward:** Apply **redo** records to data blocks.
- **Rolling Back:** Apply **undo** segments to revert changes.

Log-Based Recovery

> Logs are ordered records of transaction actions stored on stable storage; they contain transaction identifiers, old and new values, and transaction states.

Example Log Sequence (transaction updating an employee address)

Modification Strategies

- **Deferred Modification:** All log records are written first; the database is updated after the transaction commits.

- **Immediate Modification:** The database is updated after each log entry; each step is logged before the change.

Write-Ahead Log (WAL)

- Ensures that log records are written to stable storage **before** the corresponding data pages are flushed.

Checkpointing

> A **checkpoint** marks a point where the database and log are synchronized: all previous logs are flushed, and a checkpoint record containing active transaction IDs is written.

- After a checkpoint, only transactions **active** at that point need undo/redo processing.

Recovery Process

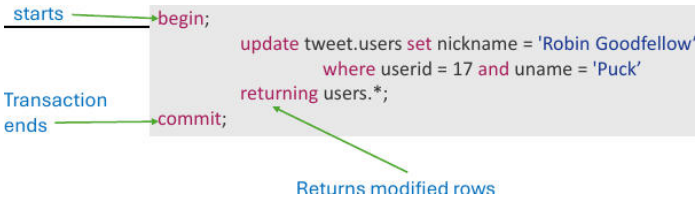
1. **Read log** backward from the last checkpoint.
2. **Undo** transactions that started but did not commit.
3. **Redo** committed transactions to reapply their effects.
- **Undo:** Use a log record to restore the old value.
- **Redo:** Use a log record to apply the new value.

Recovery Using Log Records & Checkpoints

- Transaction **Ti** is **undone** if appears without a matching or .
- Transaction **Ti** is **redone** if appears with a matching or .
- A checkpoint record lists active transactions **L** at checkpoint time; only those transactions (and any started afterward) require undo/redo.

SQL Transaction Examples 📋

Updating a User's Nickname (with diagram)



The image illustrates a simple transaction: BEGIN; → UPDATE ...; → COMMIT;, highlighting the start and end of the transaction.

```
BEGIN;
UPDATE tweet.users
```

```
SET nickname = 'Robin Goodfellow'
WHERE userid = 17 AND uname = 'Puck'
RETURNING *;
COMMIT;
```

Deleting a User Record

```
BEGIN;
DELETE FROM tweet.users
WHERE userid = 22 AND uname = 'CLAUDIUS'
RETURNING *;
COMMIT;
```

- Records are **not** physically removed until the **VACUUM** process runs automatically.

Truncate with Rollback

```
SELECT count(*) FROM foo;
BEGIN;
TRUNCATE foo;
ROLLBACK;
SELECT count(*) FROM foo;
```

- TRUNCATE efficiently purges all rows from a table in a single operation.

Concurrency Situation (Read-Committed effect)

1. **Console 1:** BEGIN; UPDATE tweet.message SET rts = rts + 1 WHERE messageid = 1 RETURNING messageid, rts; (no commit yet)
2. **Console 2:** Executes the same statement and receives **no rows** because the first transaction holds a lock.
3. After **Console 1** issues COMMIT;, **Console 2** proceeds and sees the updated row.

Isolation Level Demonstration (Repeatable Read)

```
\set ON_ERROR_ROLLBACK on
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
UPDATE tweet.message SET rts = rts + 1 WHERE messageid = 1 RETURNING messageid, rts;
-- In a second session run the same statement
-- The second transaction will wait until the first commits
COMMIT;
```

- If the first transaction commits, the second receives an error:
  ERROR: could not serialize access due to concurrent update → transaction rolls back.

📚 Database Indexes Overview

Brief Overview

This note covering **Database Indexes** was created from a 29-page PDF presentation. It provides a clear walkthrough of indexing concepts, practical examples, and algorithmic details to help you grasp how databases efficiently locate and manage records.

Key Points

- Understand dense vs. sparse index structures.
- Learn how primary, clustering, and secondary indexes differ.
- Apply B-Tree insertion and deletion algorithms.
- Evaluate multi-level index advantages for large datasets.

📚 Single-Level Indexes

Definition

> An **index** is an auxiliary file that makes searching for a record in a data file more efficient.
> It usually consists of ordered entries and is built on one or more fields.

Dense vs. Sparse (Nondense) Indexes

- **Dense index** – an entry for **every** search-key value (i.e., for every record).
- **Sparse (nondense) index** – entries only for **some** search values, typically one per disk block.

Math Example

Assume a data file **EMPLOYEE** with the following parameters:

Parameter	Value
Record size R	150 bytes
Block size B	512 bytes

Number of records r	30 000
Field size (SSN) V_{ssn}	9 bytes
Record-pointer size P_R	7 bytes

Calculations

- Blocking factor $B_{fr}= \frac{B}{R}= \frac{512}{150}=3$ records/block
- Number of data-file blocks $b= \frac{r}{B_{fr}}= \frac{30 000}{3}=10 000$ blocks

Index entry size $R_i = V_{ssn}+P_R = 9+7 = 16$ bytes

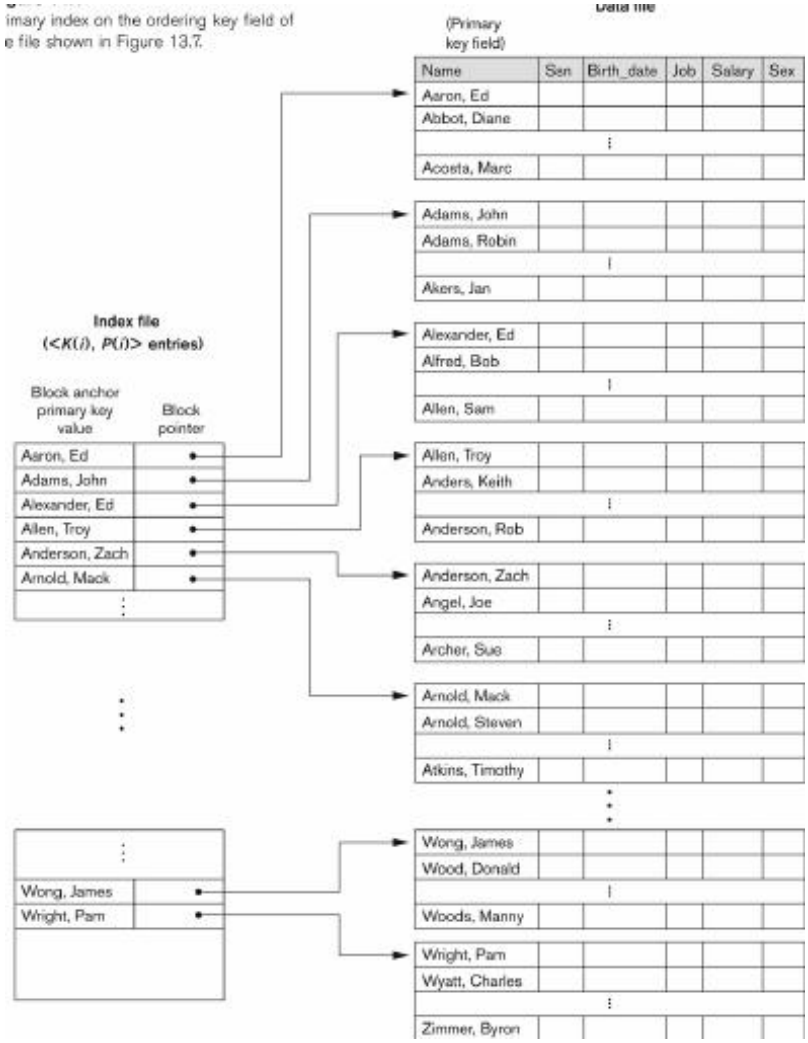
- Index blocking factor $B_{fr}^{(i)} = \frac{B}{R_i}= \frac{512}{16}=32$ entries/block
- Number of index blocks $b_i = \frac{r}{B_{fr}^{(i)}}= \frac{30 000}{32}\approx 938$ blocks

Binary search on the index needs
$\log_2 b_i \approx \log_2 938 \approx 10$ block accesses,
versus an average linear search cost of $\frac{b}{2}=5 000$ block accesses.

If the data file were ordered, binary search on the data would need
$\log_2 b = \log_2 10 000 \approx 14$ block accesses, but ordering 30 000 records is far more costly than ordering < 1 000 index blocks.

Primary Index 📂

> A **primary index** is built on an **ordered** data file. It stores one entry per **data block**, using the key of the first (or last) record in the block as the *block anchor*. This makes it a **sparse** index.

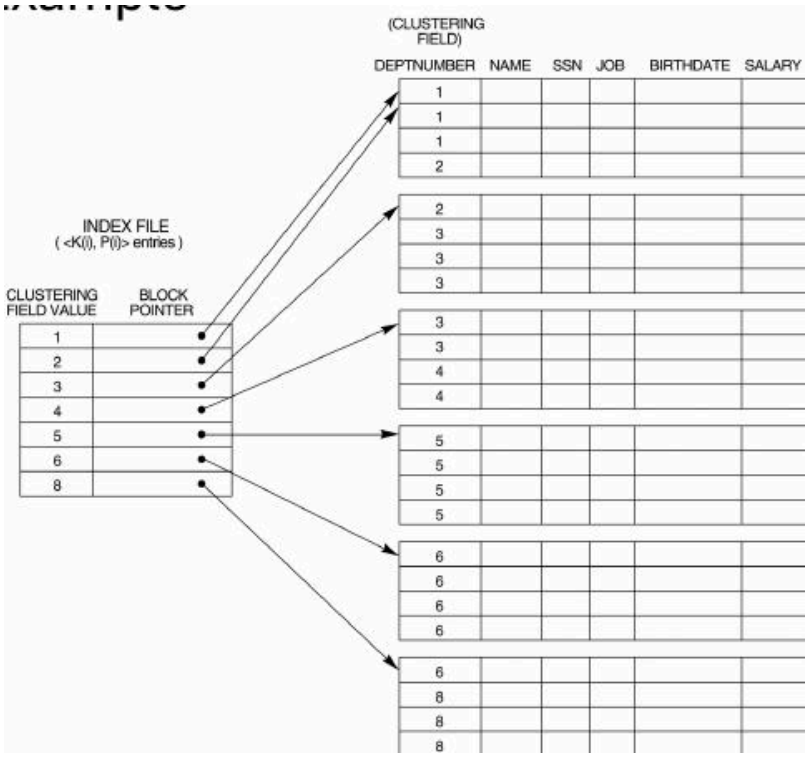


The diagram shows the index file on the left with entries mapping primary keys to block pointers in the data file.

Clustering Index 🌐

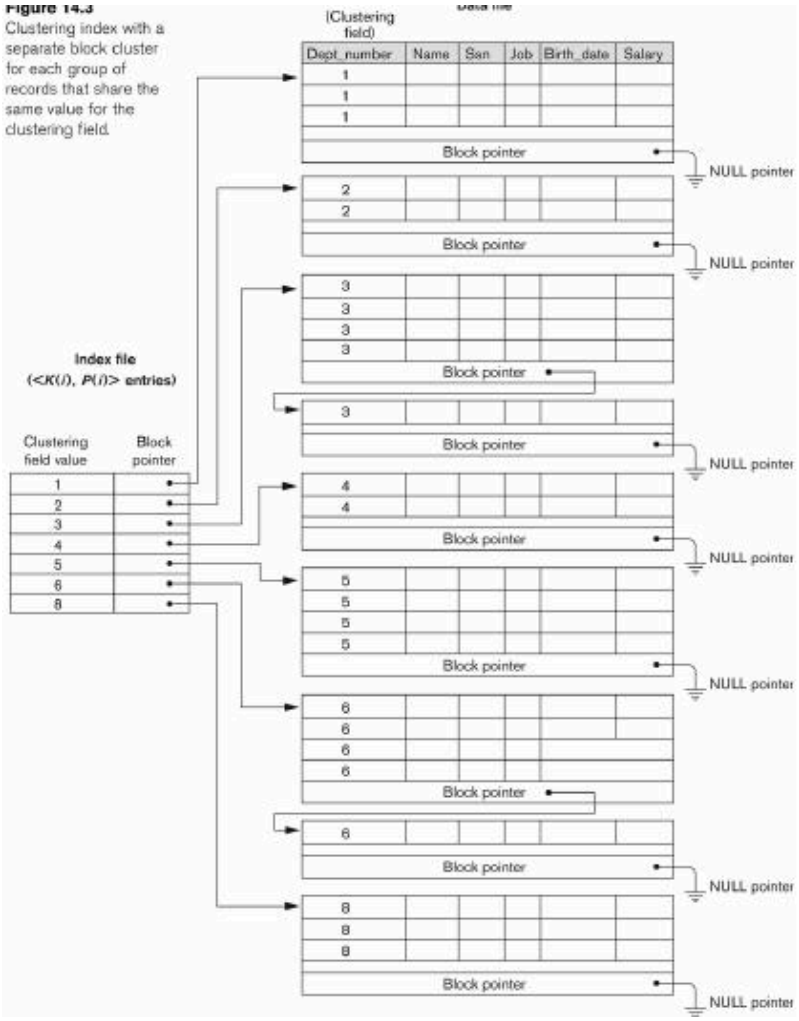
> A **clustering index** is defined on a **non-key** ordering field. It contains one entry for each distinct value of that field, pointing to the first block that holds records with that value. Like the primary index, it is also **sparse**, but insertion and deletion are simpler because the ordering field need not be unique.

Example 1 – Clustering index on DEPTNUMBER



The left side lists clustering field values (1-8) with block pointers; the right side shows the data blocks grouped by those values.

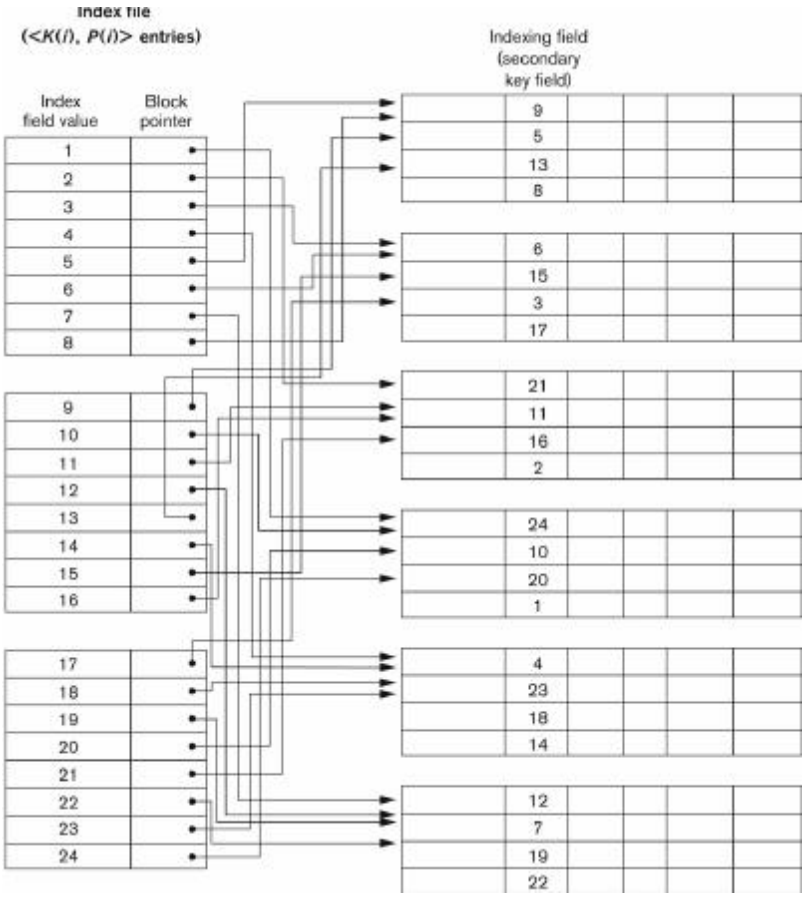
Example 2 – Separate block clusters for each clustering value



Each index entry points to a block cluster containing all records sharing the same department number.

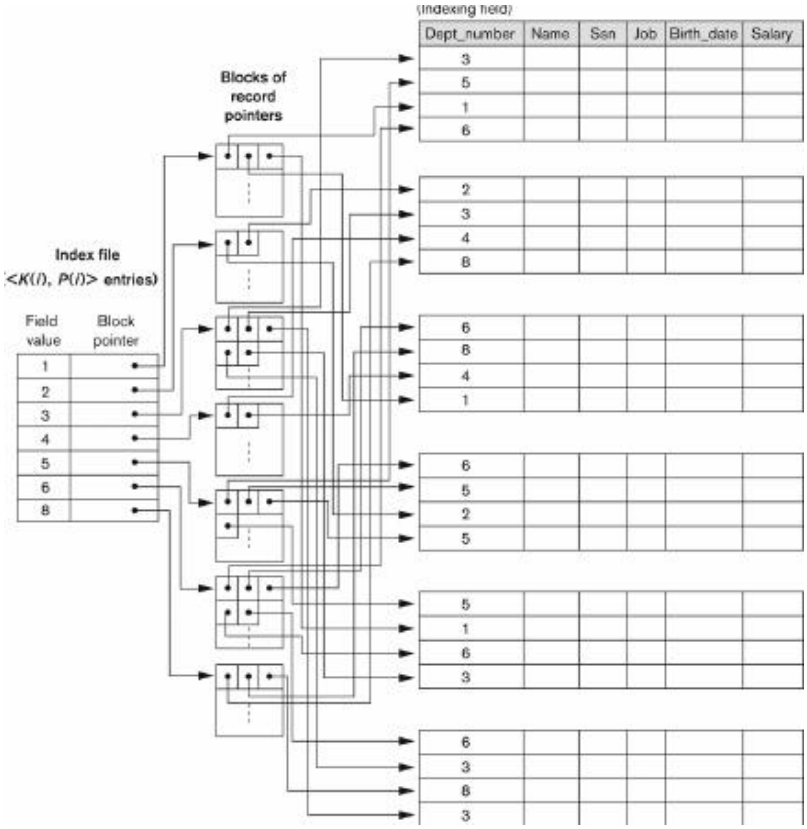
Secondary Index 🔎

> A **secondary index** offers an additional access path to a file that already has a primary access method. It may be built on a candidate key (unique) or a non-key (duplicates). The index file is ordered and typically **dense**, containing one entry for each record.



Shows an index file with field values, block pointers, and the secondary key.

Secondary index with indirection – the index points to blocks of record pointers rather than directly to records.



Illustrates how entries map to blocks that contain the actual record pointers.

📊 Properties of Index Types

Index Type	First-Level Entries	Dense?	Anchor on Data File?	Nondense?
Primary	One per block	No	Yes (key of first/last record)	Yes
Clustering	One per distinct ordering-field value	No	Yes (non-key ordering)	Yes

Secondary	One per record	Yes	No (independent of data-file order)	No

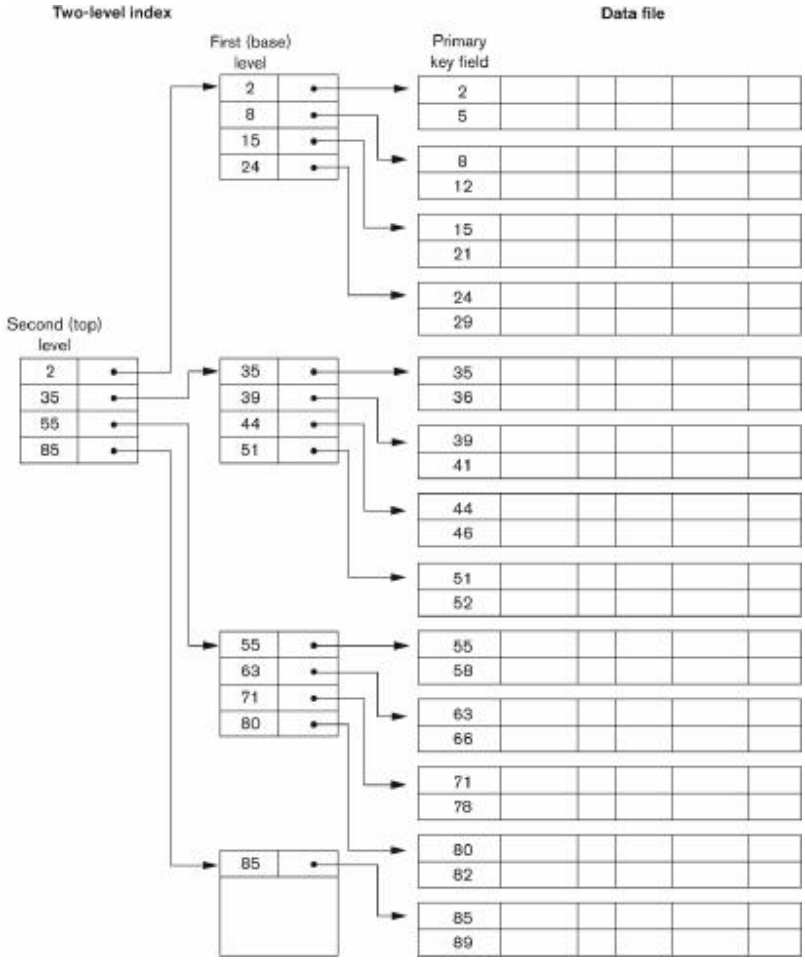
Every distinct value of the ordering field starts a new block in primary and clustering indexes; secondary indexes are dense, covering every record.

🏗️ Multi-Level Indexes

When a single-level index spans many blocks, it can be indexed itself:

- **First-level index** – the original index file.
- **Second-level index** – an index on the first-level index.
- Additional levels can be added until the topmost index fits in a single disk block.

Two-Level Primary Index Example



The lower level contains block anchors (2, 8, 15, 24) pointing to data; the upper level points to those anchors, reducing the number of disk I/O operations.

🌳 B-Tree and B⁺-Tree Indexes

Why B-Tree / B⁺-Tree?

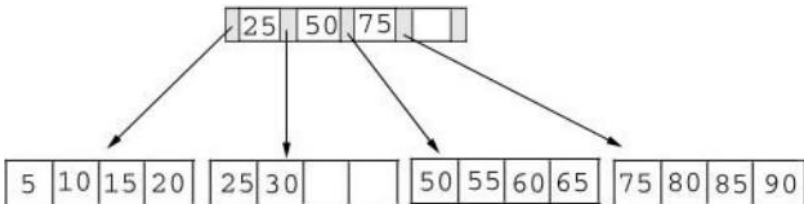
- Provide **dynamic** multi-level indexing.
- Support efficient **insertion** and **deletion** while keeping nodes between half-full and completely full.
- Each node corresponds to a disk block.

B⁺-Tree Characteristics

- Balanced: all root-to-leaf paths have equal length.

- Non-leaf nodes have between $\lceil n/2\rceil$ and n children (where n is the order).
- Leaf nodes store actual data pointers and are linked via a doubly-linked list for fast range queries.
- Leaf occupancy ≥ 50 % of the maximum.

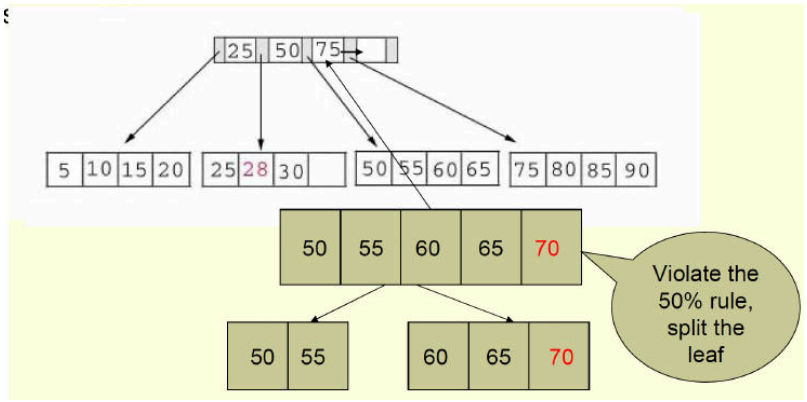
B-Tree Example (order $n=4$)



Root contains keys 25, 50, 75; each child covers a numeric range. Searching for 45 or 55 follows the appropriate branch.

Insertion Process

1. Locate the appropriate leaf.
2. Insert the key in sorted order.
3. If the leaf exceeds capacity, **split** it and promote the middle key to the parent.
4. Propagate splits upward as needed.



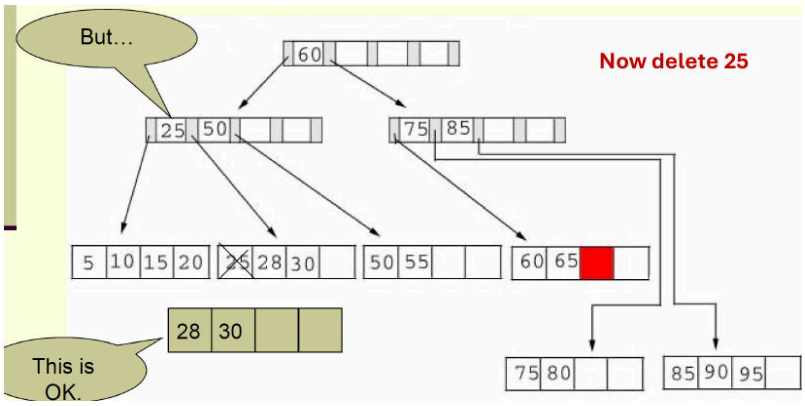
The leaf node (50 55 60 65 70) violates the 50 % rule, so it splits into two leaves (50 55) and (60 65 70).

Insertion Algorithm Summary

Situation	Action
Leaf page not full	Insert key in sorted position.
Leaf page full	Split leaf; promote middle key to index page.
Index page full after promotion	Split index page and continue upward.

Deletion Process

1. Remove the key from the leaf.
2. If the leaf falls below the minimum fill, **borrow** from a sibling or **merge** with a sibling, updating the parent accordingly.
3. Propagate merges upward if necessary.



The node containing 25 is deleted; its sibling (28 30) absorbs the gap, maintaining the tree balance.

Deletion Algorithm Summary

Situation	Action
Leaf page still ≥ minimum fill	Delete key; re-order remaining keys.
Leaf underfull	Combine with sibling; adjust parent index entries.
Parent underfull after combine	Recursively combine or redistribute upward.

📐 Designing B / B⁺ Trees

- **Maximum keys per node**: $n h$ (where h = height, n = order).
- **Minimum keys per node**: $2\bigl(\frac{n}{2}\bigr)^{h-1}$.

Example order $n = 200$

- Each leaf can hold up to 199 keys.
- Assuming the root has at least 100 children:
  - **2-level B⁺ tree** → ≥ 100 leaves × ≥ 99 keys ≈ **9 900 records**.
  - **3-level B⁺ tree** → about **1 million keys**.
  - **4-level B⁺ tree** → up to **100 million keys**.

These capacities illustrate how B⁺-trees scale efficiently with modest growth in height.

📚 Recursive Queries in SQL

Brief Overview

This note covers **recursive relational queries** and was created from a 24-page PDF presentation. It covers **SQL expressiveness** for flight paths, **recursive CTEs** in PostgreSQL, Datalog basics, and fixpoint evaluation methods.

Key Points

- Understanding how SQL limits stopovers versus using recursion for unbounded paths
- Implementing reachability with recursive CTEs in PostgreSQL
- Comparing naive and semi-naive fixpoint evaluation strategies
- Grasping Datalog's least fixpoint semantics for family-tree examples

---

Expressiveness of SQL 🚀

- **Direct flight**

  > *Definition:* Checks whether there is a flight from **City1** to **City2** without any stopovers.

  ```
  SELECT *
  FROM flight
  WHERE from = 'City1' AND to = 'City2';
  ```

- **At most one stopover**

  > *Definition:* Determines if a route exists with either a direct flight or exactly one intermediate city.

  ```
  SELECT *
  FROM flight
  WHERE (from = 'City1' AND to = 'City2')
     OR EXISTS (
         SELECT 1
         FROM flight f1
         JOIN flight f2 ON f1.to = f2.from
         WHERE f1.from = 'City1' AND f2.to = 'City2'
     );
  ```

- **At most two stopovers**

> *Definition:* Extends the previous idea to allow two intermediate cities.

```
SELECT *
FROM flight
WHERE (from = 'City1' AND to = 'City2')
   OR EXISTS (
       SELECT 1
       FROM flight f1
       JOIN flight f2 ON f1.to = f2.from
       WHERE f1.from = 'City1' AND f2.to = 'City2'
   )
   OR EXISTS (
       SELECT 1
       FROM flight f1
       JOIN flight f2 ON f1.to = f2.from
       JOIN flight f3 ON f2.to = f3.from
       WHERE f1.from = 'City1' AND f3.to = 'City2'
   );
```

- **General *k* stopovers**

> *Observation:* Requires **k + 1** tuple variables; the query size grows linearly with *k*.

- **Unbounded stopovers**

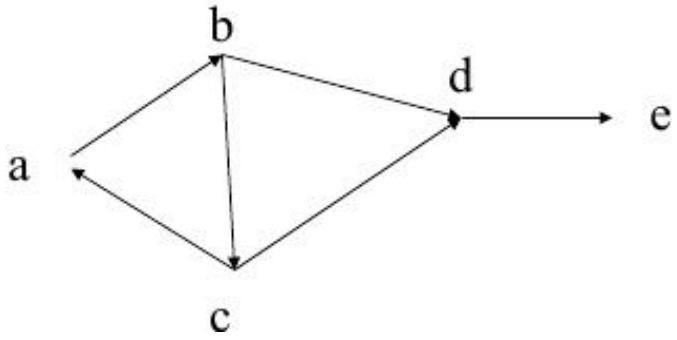
> *Result:* **Cannot** be expressed in basic SQL; needs recursion.

Recursive Relational Queries 🔁

> *Definition:* A **recursive query** computes the *transitive closure* of a relation, i.e., all pairs of nodes reachable via any length of directed path.

Graph Example

The diagram below illustrates a simple directed graph used to explain reachability.



The image shows five nodes (a-e) connected by directed edges, forming a path that can be traversed to demonstrate recursive reachability.

Edge (from → to)
a → b
a → d
a → e
b → d
b → e
…

The task is to find all ordered pairs $\langle x, y \rangle$ such that a directed path exists from *x* to *y*.

Datalog Foundations 📚

> *Definition:* **Datalog** expresses inference declaratively via rules; evaluation follows the **least fixpoint** semantics.

Family Relation Example

```
% Base facts
person('Alice', 'Bob',   'Carol').
person('Bob',   'David', 'Emma' ).
person('Carol', 'Frank', 'Grace').
person('Henry', 'Bob',   'Carol').

% Parent relation derived from family facts
parent(C, P) :- person(C, P, _).
parent(C, P) :- person(C, _, P).

% Recursive ancestor definition
```

```
ancestor(C, A) :- parent(C, A).
ancestor(C, A) :- parent(C, X), ancestor(X, A).
```

> *Definition:* A **least fixpoint (LFP)** is the smallest set of facts that satisfies all Datalog rules. It is reached by repeatedly applying the *immediate consequence operator* until no new facts appear.

Fixpoint Evaluation 🛠️

Naïve Evaluation

```
R = BaseFacts;
REPEAT
    New = ApplyRules(R);
    R = R UNION New;
UNTIL New IS EMPTY;
```

- Recomputes joins over the entire relation each iteration.
- Leads to many redundant matches, especially on large graphs.

Semi-Naïve Evaluation (Δ-based)

```
Δ0 = BaseFacts;
R  = Δ0;
REPEAT
    Δnext = ApplyRules(Δ, R) MINUS R;
    R = R UNION Δnext;
    Δ = Δnext;
UNTIL Δ IS EMPTY;
```

- Uses only **newly derived tuples** (Δ) from the previous iteration.
- Guarantees each tuple participates in a join exactly once.

Δ-Table Example (Family Tree)

Iteration	Δ contents (new pairs)
0	(Alice, Bob), (Alice, Carol)
1	(Bob, David), (Bob, Emma)
2	(Alice, David), (Alice, Emma)
3	(Alice, Frank), (Alice, Grace)
…	…

final	∅ (no new tuples)

The **fixpoint** is the union of all Δ tables.

Recursive Queries in PostgreSQL 🐘

Schema & Data

```
CREATE TABLE family (
    person TEXT,
    father TEXT,
    mother TEXT
);

INSERT INTO family VALUES
('Alice','Bob','Carol'),
('Bob','David','Emma'),
('Carol','Frank','Grace'),
('Henry','Bob','Carol');
```

Binary parent Relation (CTE)

```
WITH parent AS (
    SELECT person AS child, father AS parent FROM family WHERE father IS NOT NULL
    UNION ALL
    SELECT person AS child, mother AS parent FROM family WHERE mother IS NOT NULL
)
SELECT * FROM parent;
```

child	parent
Alice	Bob
Alice	Carol
Bob	David
Bob	Emma
Carol	Frank
Carol	Grace
Henry	Bob
Henry	Carol

Ancestor Query (Depth-agnostic)

```
WITH RECURSIVE
parent(child, parent) AS (
    SELECT person, father FROM family WHERE father IS NOT NULL
    UNION ALL
    SELECT person, mother FROM family WHERE mother IS NOT NULL
),
ancestor(child, ancestor) AS (
    SELECT child, parent FROM parent                    -- base case
    UNION
    SELECT p.child, a.ancestor
    FROM parent p
    JOIN ancestor a ON p.parent = a.child               -- recursive step
    WHERE p.child <> a.ancestor                         -- avoid cycles
)
SELECT DISTINCT child, ancestor
FROM ancestor
ORDER BY child, ancestor;
```

Ancestor Query with Depth

```
WITH RECURSIVE
parent(child, parent) AS ( …same as above… ),
ancestors(child, ancestor, depth) AS (
    SELECT child, parent, 1 FROM parent                   -- base depth = 1
    UNION
    SELECT p.child, a.ancestor, a.depth + 1
    FROM parent p
    JOIN ancestors a ON p.parent = a.child
    WHERE p.child <> a.ancestor
)
SELECT DISTINCT child, ancestor, depth
FROM ancestors
ORDER BY child, depth;
```

Graph Reachability (Any Number of Stopovers)

```
WITH RECURSIVE reachable(origin, destination, stops) AS (
    SELECT origin, destination, 0 FROM flights                  -- direct flights
    UNION
    SELECT r.origin, f.destination, r.stops + 1
    FROM reachable r
    JOIN flights f ON r.destination = f.origin
    WHERE r.origin <> f.destination                              -- avoid self-loops
)
SELECT DISTINCT origin, destination, stops
FROM reachable
ORDER BY origin, destination, stops;
```

Origin	Destination	Stops
San Diego	Los Angeles	0
San Diego	Denver	1
San Diego	Chicago	2
San Diego	New York	3
Los Angeles	Denver	0
Los Angeles	Chicago	1
Los Angeles	New York	2
Denver	Chicago	0
Denver	New York	1

Existence Check for a Specific Route

```
WITH RECURSIVE reachable AS (
    SELECT origin, destination FROM flights
    UNION
    SELECT r.origin, f.destination
    FROM reachable r
    JOIN flights f ON r.destination = f.origin
)
SELECT EXISTS (
    SELECT 1 FROM reachable
    WHERE origin = 'San Diego' AND destination = 'New York'
) AS flight_exists;
```

Avoiding Cycles (Path Enumeration)

```
WITH RECURSIVE reachable(origin, destination, path) AS (
    -- Anchor: direct flights
    SELECT origin, destination, ARRAY[origin, destination]
    FROM flights
    UNION ALL
    -- Recursive step: extend path by one flight, prevent revisiting cities
    SELECT r.origin, f.destination, path || f.destination
    FROM reachable r
    JOIN flights f ON r.destination = f.origin
    WHERE NOT f.destination = ANY(r.path)
)
SELECT origin, destination, path
FROM reachable;
```

Origin	Destination	Path
San Diego	Los Angeles	{San Diego, Los Angeles}
San Diego	Denver	{San Diego, Los Angeles, Denver}
San Diego	Chicago	{San Diego, Los Angeles, Denver, Chicago}
San Diego	New York	{San Diego, Los Angeles, Denver, Chicago, New York}
Los Angeles	Denver	{Los Angeles, Denver}
...	...	...

📚🔍 Relational Calculus Primer

Brief Overview

This note covers relational calculus and was created from a 28-page PDF presentation. It includes **historical background**, **translation to SQL**, example queries, and variable scoping in tuple calculus.

Key Points

- The evolution from Frege, Tarski, to Codd and the algebraic foundation of relational databases.
- Predicate logic definitions and concrete examples such as even and prime numbers.
- Syntax and semantics of tuple relational calculus, including atomic conditions and quantifiers.
- Techniques for translating universal quantification into SQL using NOT EXISTS and NOT IN.

📚 Origins of Relational Calculus

- **Frege** – developed **first-order (FO) logic**.
- **Tarski** – provided an **algebraic foundation** for FO logic.
- **Codd** – introduced **relational databases**, using FO logic as the theoretical core.

🔍 Relational Calculus Overview

> **Definition:** *A logic-based query language that specifies **what** data to retrieve rather than **how** to retrieve it. It models data manipulation at the heart of SQL.*

- General form:
  ${,t \mid \text{property}(t),}$
  where **property**(t) is expressed in **predicate calculus** (first-order logic).

📐 Predicate Calculus Examples

- **Even numbers**:
  ${,x \mid \exists y;(x = 2 \times y),}$
- **Prime numbers**:
  ${,x \mid x \neq 1 ;\wedge; \forall y,\forall z,[,x = y \times z ;\Rightarrow; (y = 1 \vee z = 1),],}$

Symbols:

- $\exists$ – “there exists” (existential quantifier)
- $\forall$ – “for all” (universal quantifier)

🗃️ Tuples in Relational Calculus

- Tuple membership is written **$m \in R$** (“tuple *m* is in relation *R*”).
- Example table **movie**:

  SQL: SELECT * FROM movie

  Tuple view: “The answer consists of tuples **m** such that **m ∈ movie**.”

📋 Example Queries

Directors & Actors of Currently Playing Movies

Tuple calculus expression
$ {,t : \text{Director}, \text{Actor} \mid \exists m \in \text{movie};\exists s \in \text{schedule}; [, t(\text{Director}) = m(\text{Director}) ,\wedge, t(\text{Actor}) = m(\text{Actor}) ,\wedge, m(\text{Title}) = s(\text{Title}) ,] ,} $

Universal Quantification Example

“Every director is also an actor”

$ \forall m \in \text{movie};\exists t \in \text{movie}; [, m(\text{Director}) = t(\text{Actor}) ,] $

The query evaluates to **true** or **false**.

✏️ Tuple Relational Calculus Syntax

- **Tuple variables**: $t, s, \dots$
- **Atomic conditions** (atoms):
  - $t(A) = \text{constant}$, $t(A) \neq \text{constant}$, $t(A) \ge \text{constant}$
  - $t(A) = s(B)$, $t(A) \neq s(B)$
- **Boolean operators**: $\land$, $\lor$, $\lnot$ (with abbreviation $p \rightarrow q \equiv \lnot p \lor q$)
- **Quantifiers**:
  - Existential: $\exists t \in R; \phi(t)$
  - Universal: $\forall t \in R; \phi(t)$

📊 Free and Bound Variables

- **Scope** of a quantifier is the formula that follows it.
- A **free variable** is not bound by any quantifier; it acts as a parameter of the formula.

Example

$ {,t : \text{Director}, \text{Actor} \mid \exists m \in \text{movie};\exists s \in \text{schedule}; [,\dots,] ,} $

- Free variables: t (the answer tuple).
- Bound variables: m, s (introduced by $\exists$).

🖨️ Tuple Calculus Query Form

$ {,t : \langle\text{attributes}\rangle \mid \phi(t) ,} $

- $\phi(t)$ contains **only one free variable** (t).
- The result is a table with the listed attributes, containing all tuples v for which $\phi(v)$ evaluates to **true**.
- If the range of t isn't explicitly given, it defaults to the **active domain** (all values present in the database or mentioned in the query).

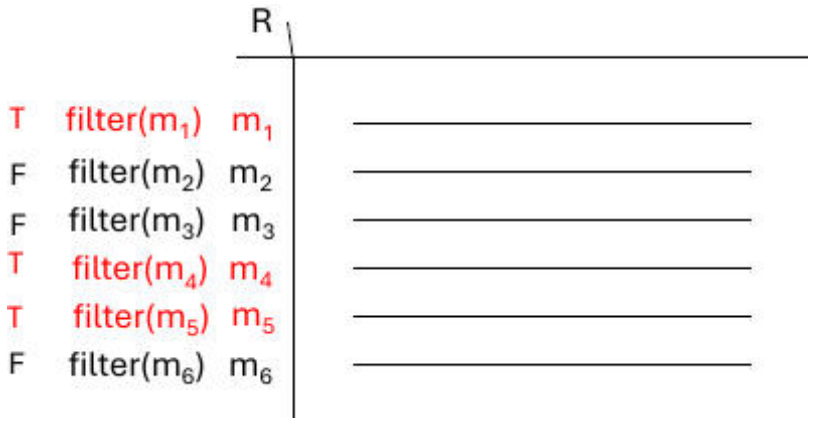
🎬 Movie Database Example Queries

Query	Tuple Calculus
Titles of currently playing movies	${,t:\text{title} \mid \exists s \in \text{schedule};[,s(\text{title}) = t(\text{title}),] ,}$

Titles of movies directed by **Berto**	${,t:\text{title} \mid \exists m \in \text{movie};[,m(\text{director}) = \text{“Berto”} ,\wedge, t(\text{title}) = m(\text{title}),] ,}$
Title & director of currently playing movies	${,t:\text{title},\text{director} \mid \exists s \in \text{schedule};\exists m \in \text{movie};[,s(\text{title}) = m(\text{title}) ,\wedge, t(\text{title}) = m(\text{title}) ,\wedge, t(\text{director}) = m(\text{director}),] ,}$
Employees with the highest salary	${,x:\text{name} \mid \exists y \in \text{employee};[,x(\text{name}) = y(\text{name}) ,\wedge, \forall z \in \text{employee};(y(\text{salary}) \ge z(\text{salary})),] ,}$
Actors appearing in **every** movie by **Berto**	${,a:\text{actor} \mid \exists y \in \text{movie};[,a(\text{actor}) = y(\text{actor}) ,\wedge, \forall m \in \text{movie};(m(\text{director}) = \text{“Berto”} \rightarrow \exists t \in \text{movie};(m(\text{title}) = t(\text{title}) ,\wedge, t(\text{actor}) = y(\text{actor}))),] ,}$

Visualizing Universal Quantification

The following diagram illustrates the typical pattern **$\forall m \in R\,[\,\text{filter}(m) \rightarrow \text{property}(m)\,]$**:



*Rows marked **T** satisfy the filter; the corresponding **property(m)** column shows the evaluated condition. Rows marked **F** are irrelevant ("don't care").*

🔄 Translating to SQL

Basic SQL Template

```
SELECT A1, …, An
FROM R1, …, Rk
WHERE cond(R1, …, Rk);
```

- Each tuple combination $(r_1,\dots,r_k)$ that satisfies **cond** yields a result row.
- The SQL form implicitly uses only **existential** quantification; there is no direct **∀** construct.

Example: Theaters Showing Movies by Bertolucci

- **SQL**

```
SELECT s.theater
FROM schedule s, movie m
WHERE s.title = m.title
  AND m.director = 'Bertolucci';
```

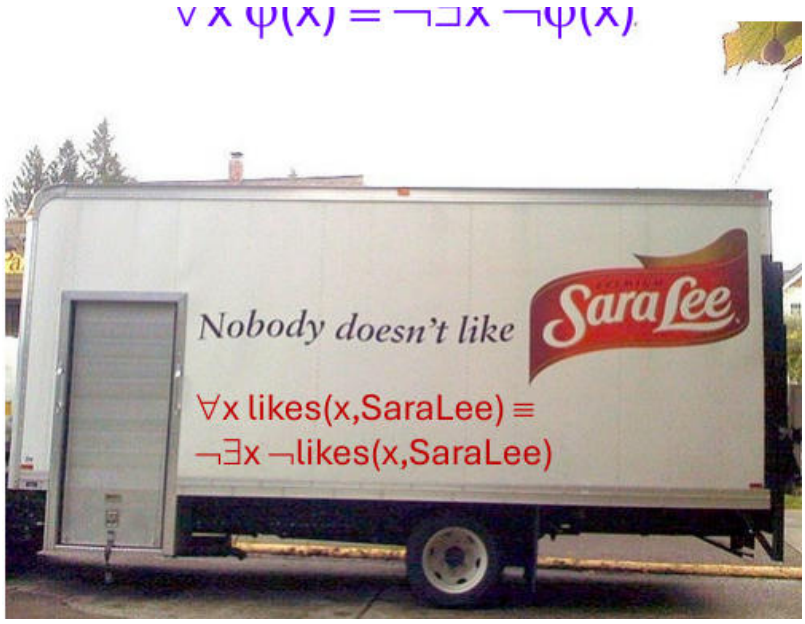
- **Tuple calculus**
  $ {,t:\text{theater} \mid \exists s \in \text{schedule};\exists m \in \text{movie}; [,t(\text{theater}) = s(\text{theater}) ,\wedge, s(\text{title}) = m(\text{title}) ,\wedge, m(\text{director}) = \text{“Bertolucci”},] ,} $

Eliminating Universal Quantifiers

Logical equivalence:

$ \forall x \in R;\phi(x) ;\equiv; \lnot\exists x \in R;\lnot\phi(x) $

Image illustrating the equivalence:



The diagram shows how a universal statement can be rewritten as the negation of an existential violation.

Converting the "Actors in Every Berto Movie" Query

Tuple calculus (with ∀)

$ {,a:\text{actor} \mid \exists y \in \text{movie}; [,a(\text{actor}) = y(\text{actor}) ,\wedge, \forall m \in \text{movie};(m(\text{director}) = \text{“Berto”} \rightarrow \exists t \in \text{movie};(m(\text{title}) = t(\text{title}) ,\wedge, t(\text{actor}) = y(\text{actor})))] ,} $

After eliminating ∀

$ {,a:\text{actor} \mid \exists y \in \text{movie}; [,a(\text{actor}) = y(\text{actor}) ,\wedge, \lnot\exists m \in \text{movie};(m(\text{director}) = \text{“Berto”} ,\wedge, \lnot\exists t \in \text{movie};(m(\text{title}) = t(\text{title}) ,\wedge, t(\text{actor}) = y(\text{actor})))] ,} $

Corresponding SQL (using NOT EXISTS)

```
SELECT y.actor
FROM movie y
WHERE NOT EXISTS (
    SELECT *
    FROM movie m
```

```
    WHERE m.director = 'Berto'
      AND NOT EXISTS (
          SELECT *
          FROM movie t
          WHERE t.title = m.title
            AND t.actor = y.actor
      )
);
```

Alternative SQL (using NOT IN)

```
SELECT actor
FROM movie
WHERE actor NOT IN (
    SELECT s.actor
    FROM movie s, movie m
    WHERE s.title = m.title
      AND m.director = 'Berto'
      AND s.actor NOT IN (
          SELECT t.actor
          FROM movie t
          WHERE t.title = m.title
      )
);
```

- The ability to mix ∃ and ∀ in tuple calculus provides greater expressive flexibility than standard SQL, which must simulate universal quantification via negated existential subqueries.

📊 Database Normalization Overview

Brief Overview

This note covers Database Design and was created from a PDF presentation.
It focuses on **Universal schema**, anomaly identification, **functional dependencies**, and design guidelines for lossless joins.

Key Points

- Identify insertion, deletion, and update anomalies.
- Decompose the universal relation into Supplier, Part, and Shipment tables.
- Apply Armstrong's axioms to determine keys and dependencies.
- Verify lossless-join properties and key minimality.

📦 Universal Schema & Data

> **Universal schema** – a single relation that attempts to store all information about suppliers, parts, and shipments in one table.

Attribute	Meaning
S#	Supplier number
SNAME	Supplier name
SCITY	Supplier city
P#	Part number
PNAME	Part name
PCITY	City where part is stored
QTY	Quantity of shipment

Example tuples

S#	SNAME	SCITY	P#	PNAME	PCITY	QTY
S1	Smith	London	P1	Nut	London	100

S1	Smith	London	P2	Bolt	Paris	150
S2	Adams	Paris	P1	Nut	London	200
S3	Blake	Rome	P3	Screw	Rome	250
S4	Clark	Paris	P2	Bolt	Paris	300

⚠️ Anomalies in the Universal Relation

🚩 Insertion Anomaly

> When a new **supplier** or **part** cannot be added without also inserting a shipment record.

- Adding supplier **S5 (Jones, Berlin)** requires a fake shipment (e.g., QTY = 0) because P# and QTY cannot be null.

🚩 Deletion Anomaly

> Removing the last shipment of a supplier (or part) deletes the entity's information.

- If **Smith (S1)** only supplies P1 and P2 and both rows are deleted, the database loses all data about supplier **S1**.

🚩 Update Anomaly

> Redundant storage leads to inconsistent updates.

- Supplier city for **S1** appears in multiple rows. Changing SCITY from **London** to **Berlin** requires updating every tuple with S# = S1; missing any row creates inconsistency.

📋 Summary of Anomalies

Anomaly	Description	Example in BAD schema

Insertion	Cannot add new supplier/part without a shipment	Must insert fake QTY for S5
Deletion	Deleting last shipment erases entity info	Removing both S1 rows deletes supplier S1
Update	Redundant data causes inconsistent values	SCITY for S1 stored in several rows

---

🗃️ Hidden Conceptual Relations

The universal relation actually contains three logical relations:

Relation	Meaning
S	Supplier information (S#, SNAME, SCITY)
P	Part information (P#, PNAME, PCITY)
SP	Shipment linking supplier and part (S#, P#, QTY)

📏 Design Guidelines

1️⃣ Clear Semantics (Guideline 1)

> Each tuple should represent **one entity** or **one relationship instance**.

- Do not mix attributes of different entities in the same relation.
- Use foreign keys **only** to reference other entities.

2️⃣ Avoid Anomalies (Guideline 2)

> Design schemas that are free from insertion, deletion, and update anomalies.

- If anomalies exist, they must be documented for application handling.

3️⃣ Minimize NULLs (Guideline 3)

> Reduce the occurrence of NULL values by separating frequently-null attributes into their own relations (or weak entities).

4️⃣ Lossless Join & No Spurious Tuples (Guideline 4)

> Decompositions must satisfy the **lossless-join** property; natural joins should never produce extra (spurious) tuples.

🔗 Functional Dependencies (FDs)

> **Functional dependency** $X \rightarrow Y$ holds in relation R if any two tuples agreeing on attributes X also agree on attributes Y .

Formal Statement

- For all tuples $t_1, t_2 \in R$:
  $t_1[X] = t_2[X] ;\Rightarrow; t_1[Y] = t_2[Y]$.

Example Table & Derived FDs

A	B	C	D
a	1	40	100
a	1	40	100
b	3	20	100
b	3	20	100
c	5	30	300
c	5	20	300
c	5	30	300

Derived FDs (non-exhaustive):

- $A \rightarrow B$
- $B \rightarrow A$
- $A \rightarrow C$
- $A \rightarrow D$
- $D \rightarrow A$
- $B \rightarrow D$
- $D B \rightarrow A$

Armstrong's Axioms (Inference Rules)

Rule	Description
Reflexive (IR1)	If $Y \subseteq X$, then $X \rightarrow Y$.
Augmentation (IR2)	If $X \rightarrow Y$, then $XZ \rightarrow YZ$ (where Z is any set of attributes).
Transitive (IR3)	If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.

Additional derived rules:

- **Union**: $X \rightarrow Y$ and $X \rightarrow Z$ $\Rightarrow$ $X \rightarrow YZ$.
- **Decomposition**: $X \rightarrow YZ$ $\Rightarrow$ $X \rightarrow Y$ and $X \rightarrow Z$ (right-hand side only).

Closure Concepts

- **FD closure** F^{+}: all FDs that can be inferred from a set F .
- **Attribute closure** X^{+} (w.r.t. F): all attributes functionally determined by X .
- Compute X^{+} by repeatedly applying IR1–IR3; linear-time algorithm exists.

🔑 Keys & Superkeys

> **Superkey** – a set of attributes S such that no two distinct tuples share the same S values.

> **Key** – a *minimal* superkey; removing any attribute from a key destroys the uniqueness property.

- For any key K of relation R, the FD $K \rightarrow R$ holds.

🔀 Decomposition & Lossless Join

Spurious Tuple Example

Decomposition:

1. $R_1(S#, SNAME, SCITY, QTY)$
2. $R_2(P#, PNAME, PCITY, QTY)$

Natural join of R_1 and R_2 yields extra rows because **QTY** is not a reliable join attribute.

Lossless-Join Property

- A decomposition is **lossless** if joining the projected relations recreates the original relation without extra tuples.
- Preserve functional dependencies whenever possible; losslessness is mandatory, while FD preservation can be relaxed.

Decomposition Algorithm (Sketch)

1. **Unfold** each projection into an atom with fresh variables for non-projected attributes.
2. **Apply** all known FDs: propagate equalities across atoms that share determining attributes.
3. **Remove** redundant atoms whose attributes are fully determined by others.
4. **Read off** the remaining atom(s); if a single atom remains, the decomposition is lossless.

📚 Minimal Cover (Canonical Cover) of FDs

> A **minimal set of FDs** satisfies:
>
> 1. Right-hand side of each FD is a single attribute.
> 2. No FD can be removed without changing the closure.

> 3. No left-hand attribute can be removed while preserving equivalence.

- Every FD set has at least one equivalent minimal cover (canonical cover).
- Multiple minimal covers may exist; computing one is non-trivial and not covered here.

🧩 Example Problems

Determining Keys

Relation $R(E, F, G, H, I, J, K, L, M, N)$ with FDs:

1. $E \rightarrow G$
2. $F \rightarrow I J$
3. $H \rightarrow L$
4. $K \rightarrow M$
5. $L \rightarrow N$

Key search (illustrative steps):

- Compute closures, e.g., $(E, F)^{+} = {E, F, G, I, J}$; adding H, K, L expands to all attributes, giving a candidate key ${E, F, H, K, L}$.

Equivalence of FD Sets

Two FD sets F and G are **equivalent** iff $F^{+} = G^{+}$.

- F **covers** G when $G^{+} \subseteq F^{+}$.
- Equivalence ⇔ each covers the other.

📚🗃️ Normalization Deep Dive

Brief Overview

This note covers database normalization and was created from a 70-page PDF presentation. It discusses functional dependencies, lossless-join testing, dependency preservation, BCNF, and 4NF.

Key Points

- Understand **functional dependencies** and how they drive decomposition.
- Learn practical tests for lossless-join and dependency preservation.
- Follow step-by-step algorithms for moving schemas into BCNF, 3NF, and 4NF/5NF.
- See real-world examples that illustrate partial, transitive, and multivalued dependencies.

Dependencies and Decomposition 📚

- **Lossless join** – the original relation can be reconstructed exactly from the decomposed sub-relations.
- **Dependency preservation** – every functional dependency (FD) in the original set **F** can be enforced by checking constraints in at least one of the sub-relations, without recomputing joins.
- **Local dependency** – a dependency (functional, multivalued, etc.) that involves only attributes of a single sub-relation in a decomposition.

> **Definition:** A decomposition ρ = (R_1,...,R_k) of a relation **R** is *lossless* if the natural join of all R_i yields exactly **R**.

Example:
Relation **R(A,B,C,D)** with FDs **F = {A → B, B → C, C → D}**.

- Decomposition **R_1(A,B,C), R_2(C,D)** preserves the dependencies locally.
- Decomposition **R_1(A,B,C), R_2(A,D)** does **not** preserve them locally.

Dependency-Preserving Decompositions ✅

Formal definition:
Let ρ = (R_1,...,R_k) be a decomposition of **R** and **F** a set of FDs over **R**. ρ *preserves* **F** iff

$$ \bigcup_{i=1}^{k}\pi_{R_i}(F^{+}) \equiv F $$

In words, the closure of the projection of **F** onto each sub-relation must imply all dependencies in **F**.

Testing Preservation of Dependencies 🔍

Naïve method (impractical):

1. Compute the full closure F^+ (size can be exponential).
2. Form $G = \bigcup_{i=1}^{k} \pi_{R_i}(F^+)$.
3. Verify that $F \subseteq G^+$.

Improved method (polynomial-time):

- For each FD $X \to Y$ in **F**:
    1. Initialise $Z := X$.
    2. While **Z** changes:
        - For each sub-relation R_i:
            - Update $Z := Z \cup ((Z \cap R_i)^{+} \cap R_i)$ (the "+" is taken w.r.t. **F**).
    3. If $Y \not\subseteq Z$, output "no" (dependency not preserved).
- If all FDs pass, output "yes".

Full Example of Dependency Preservation

Relation: $R = {A,B,C,D}$
Decomposition: ρ = {AB, BC, CD}
FD set: $F = {A → B,; B → C,; C → D,; D → A}$

- Local FDs $A → B$, $B → C$, $C → D$ are trivially preserved.
- To test $D → A$:
    1. Start with $Z = D$.
    2. Using CD, add C (since $D → C$ is local).
    3. Using BC, add B (now $C → B$).

4. Using AB, add A (now $B → A$).

A is reached, so $D → A$ is preserved.

Normalization Overview 🧩

- **Normalization** – the process of decomposing "bad" relations into smaller, well-structured relations.
- **Normal form** – a condition, expressed in terms of keys and functional (or multi-valued) dependencies, that a relation schema must satisfy.

Normal Forms Summary 📊

Normal Form	Key Requirement	Dependency Type
1NF	All attribute values are atomic	–
2NF	Every non-prime attribute fully depends on **the whole** candidate key	Full functional dependencies
3NF	No non-prime attribute is transitively dependent on a candidate key	Transitive FDs
BCNF	For every non-trivial FD $X → Y$, **X** is a superkey	–
4NF	No non-trivial multi-valued dependency with a non-superkey left side	MVDs
5NF	No non-trivial join dependency with a non-superkey left side	JDs

Additional design goals: **lossless join** and **dependency preservation**.

Keys, Superkeys, and Attributes 🔑

> **Superkey:** A set of attributes **S** ⊆ **R** such that no two distinct tuples in any legal state of **R** agree on all attributes of **S**.

> **Key:** A minimal superkey; removing any attribute from a key destroys the superkey property.

- **Candidate key** – any key of the relation.
- **Primary key** – arbitrarily chosen candidate key.
- **Prime attribute** – an attribute that belongs to at least one candidate key.
- **Non-prime attribute** – an attribute that belongs to no candidate key.

Functional Dependency Concepts 📈

> **Full functional dependency (FFD):** An FD $Y → Z$ is *full* if removing any attribute from **Y** makes the dependency false.

- **Partial dependency** – a functional dependency where a proper subset of the left-hand side determines the right-hand side.

Example:

- ${SSN, PNUMBER} → HOURS$ is a full FD (neither $SSN → HOURS$ nor $PNUMBER → HOURS$ holds).
- ${SSN, PNUMBER} → ENAME$ is a partial dependency because $SSN → ENAME$ also holds.

First Normal Form (1NF) 🗃️

> **1NF:** Every cell contains an *atomic* (indivisible) value; no repeating groups or nested relations.

- No grouping of information inside a cell.
- No duplicate rows (theoretical).
- Every cell must contain a value (no nulls).

Non-1NF example: a table with a **Phone** column that stores multiple phone numbers in a single cell.

Second Normal Form (2NF) 📚

> **2NF:** A relation is in 2NF if it is in 1NF **and** every non-prime attribute is **fully** functionally dependent on **every** candidate key.

Identifying 2NF Violations

- Check each FD $X → Y$: if **X** is a *proper* subset of a candidate key and **Y** is non-prime, the relation violates 2NF.

Example: Electric Toothbrush Manufacturers

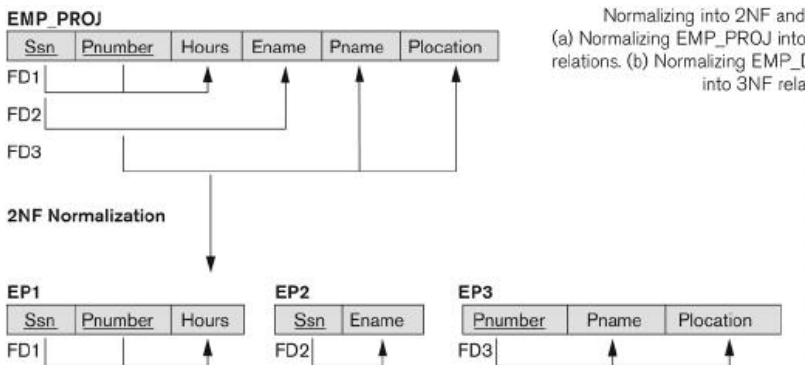
Manufacturer	Model	Model Full Name	Country
Forte	X-Prime	Forte X-Prime	Italy
Forte	Ultraclean	Forte Ultraclean	Italy
Dent-o-Fresh	EZbrush	Dent-o-Fresh EZbrush	USA
...	...	...	...

- Candidate key: **{Manufacturer, Model}**.
- Dependency **Manufacturer → Country** is a partial dependency (left side is only part of the key).

Decomposition into 2NF:

1. **Manufacturer** (**Manufacturer**, **Country**) – now **Country** fully depends on the whole key (**Manufacturer** is a key here).
2. **Model** (**Manufacturer**, **Model**, **ModelFullName**) – all non-prime attributes depend on the full composite key.

Visual Aid



Normalizing into 2NF and
(a) Normalizing EMP_PROJ into
relations. (b) Normalizing EMP_[
into 3NF rela

The diagram shows the original EMP_PROJ table being split into three 2NF tables (EP1, EP2, EP3) to eliminate partial dependencies.

Third Normal Form (3NF) 📘

> **Transitive functional dependency:** An FD $X → Z$ that can be derived from $X → Y$ and $Y → Z$.

> **3NF:** A relation is in 3NF if it is in 2NF **and** no non-prime attribute is transitively dependent on any candidate key.

Example: Music Genres

Relation **R(MusicId, Title, Singer, GenreId, Genre)** with FD **GenreId → Genre**.

- **GenreId** is not a key, and **Genre** (non-prime) depends on **GenreId** via a transitive path, violating 3NF.
- **Decomposition:**
  1. **R_1(MusicId, Title, Singer, GenreId)**
  2. **R_2(GenreId, Genre)** – both now satisfy 3NF.

Example: Tournament Winners

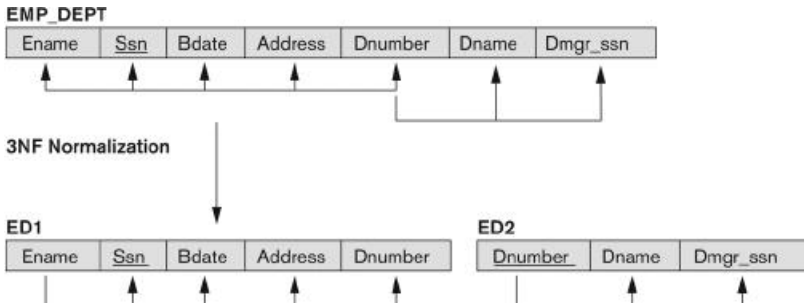
Tournament	Year	Winner	WinnerDoB
Indiana Invitational	1998	Al Fredrickson	21 July 1975
Cleveland Open	1999	Bob Albertson	28 Sept 1968
...	...	...	...

- Candidate key: **{Tournament, Year}**.
- FD **Winner → WinnerDoB** is a transitive dependency (Winner is non-prime).

Decomposition into 3NF:

1. **WinnersInfo(Winner, WinnerDoB)**
2. **TournamentResults(Tournament, Year, Winner)**

Visual Aid



The diagram illustrates how EMP_DEPT is split into ED1 and ED2, removing the transitive dependency between Dnumber and Dmgr_ssn.

Boyce-Codd Normal Form (BCNF) 🏆

> **BCNF:** A relation **R** is in BCNF if for every non-trivial FD $X → Y$ that holds in **R**, **X** is a superkey.

Example: favBeer

Relation **Drinkers(name, addr, beersLiked, manf)** with FDs:

- $name → addr$
- $name → beersLiked$
- $beersLiked → manf$

Only key: **{name, beersLiked}**.

- Left side of each FD is **not** a superkey ⇒ violation of BCNF.

BCNF Decomposition Steps

1. Pick a violating FD (e.g., $name → addr$).
2. Compute $name^{+}$ → {name, addr, beersLiked, manf}.
3. Create:
    - **R_1(name, addr, manf)**
    - **R_2(name, beersLiked)**
4. Check each new relation for BCNF; further decompose if needed (e.g., $beersLiked → manf$ in **R_2** leads to additional tables).

Resulting schemas:

- **Drinkers1(name, addr, manf)** – BCNF.
- **Drinkers3(beersLiked, manf)** – BCNF.
- **Drinkers4(name, beersLiked)** – BCNF.

Key Insight: Every BCNF relation is automatically in 3NF, but the converse is not true.

Algorithms for Lossless-Join Decomposition ⚙️

1. **Start** with the set ρ = {R}.
2. **Iteratively** select a relation **S** in ρ that violates BCNF.
3. Find a violating FD $X → A$ where X is not a superkey of **S**.
4. **Decompose S** into:
    - $S_1 = X^{+}$ (attributes reachable from **X**).
    - $S_2 = S - (X^{+} - X)$ (remaining attributes).
5. **Replace S** by S_1 and S_2 in ρ.
6. **Repeat** until all schemas are in BCNF.

The decomposition is guaranteed to be **lossless**, though it may **not** preserve all original dependencies.

Example of BCNF Decomposition & Dependency Preservation Issue

Relation **R(C, T, H, R, S, G)** with FDs:

- $C → T$
- $CS → G$

- $HR → C$
- $HS → R$
- $TH → R$
- **Key** computed: **HS** (only key).

Decomposition steps produce schemas: **CHS**, **CSG**, **CHR**, **CT**.

- The FD $TH → R$ is **not** preserved in any of the resulting BCNF relations, illustrating that BCNF decomposition can lose dependency preservation.

3NF Decomposition Procedure 📐

1. **Canonical cover** – eliminate extraneous attributes and redundant FDs.
2. **First-cut decomposition** – for each FD $X → A$ in the canonical cover, create a relation $R_i = X ∪ {A}$. Add any attribute that never appears on the right-hand side as a singleton relation.
3. **Ensure lossless join** – if no relation already contains a key for **R**, add a relation consisting of a candidate key.

The resulting set of relations is **dependency-preserving**, **lossless**, and each relation is in **3NF**.

Additional Remarks 🗒️

- **Non-uniqueness:** Different sequences of decomposition steps can yield different sets of relations.
- **Complexity:** Deciding BCNF membership is NP-complete; finding a BCNF decomposition that also preserves dependencies is generally impossible.
- **Practical design:** Designers often stop at 3NF or BCNF (occasionally 4NF) and may deliberately **denormalize** for performance reasons.

Lossless-Join Test with Added Key 🔎

> **Lossless-join property:** A decomposition ρ of relation **R** is lossless if the natural join of the sub-relations in ρ yields exactly **R** (no spurious tuples).

- Original FD set: **F = {A → C, BC → D, AD → E}**
- Initial decomposition: ρ = {**AC**, **BCD**, **ADE**}

Observation:

- None of **AC**, **BCD**, **ADE** is a superkey; their closures are
  - $AC^{+} = A C$ (does not contain all attributes)
  - $BCD^{+} = B C D$
  - $ADE^{+} = A D E$

Solution: add the key **AB** to the decomposition.

- **AB** is a key because $AB^{+} = A B C D E$ (using the three FDs in order).
- New decomposition: ρ′ = {**AC**, **BCD**, **ADE**, **AB**}.

Chase Test (illustrative):

Row	A	B	C	D	E
AC	a	–	c	–	–
BCD	–	b	c	d	–
ADE	a	–	–	–	e
AB	a	b	–	–	–

Apply FDs in the order **A → C**, **BC → D**, **AD → E**:

1. From row **AB**, $A → C$ adds **C** → row becomes **a b c – –**.
2. From rows **AB** and **BCD**, $BC → D$ adds **D** → **a b c d –**.
3. From rows **AB** and **ADE**, $AD → E$ adds **E** → **a b c d e**.

All rows converge to the same tuple **(a,b,c,d,e)**, confirming a **lossless join** for ρ′.

Schema Design Workflow Using Functional Dependencies 🛠️

> **Design process:** Choose attributes, specify functional dependencies, then decompose into normal-form schemas that are lossless-join and (if possible) dependency-preserving.

1. **Select attributes** for the target relation **R**.
2. **Specify the FD set F** (use *Armstrong relations* to validate that the chosen FDs are exactly those that hold).
3. **Compute a decomposition** that:
    - Is **lossless-join** (e.g., via the chase test).
    - **Preserves** all dependencies (if possible).
    - Places each sub-relation in a desired normal form:
        - Prefer **BCNF** when achievable.
        - Otherwise aim for **3NF** (still dependency-preserving).

Attributes vs. Data: Modeling Choices 📊

Design	Layout	Example Row	Typical Queries
Option 1 (single table)	**EmpName	Dept**	Joe
Option 2 (cross-tab)	**EmpName	Candy	PCs

Considerations

- **Stability of departments:** If department list changes often, a separate table (Option 2) may become cumbersome.
- **Sparsity vs. density:** Option 2 can be sparse (many null/false entries) leading to wasted space.
- **Query ease:** Simple look-ups (Option 1) vs. set-based queries (Option 2).

Armstrong Relations: Verifying Exact FD Sets ✔

> **Armstrong relation:** A relation instance that satisfies *exactly* the functional dependencies in F^+ (no more, no fewer).

- **FD set: F = {theater → title}**
- **Armstrong relation instance:**

Schedule	theater	title
1	Paloma	Casablanca
2	Hillcrest	Casablanca

- **Satisfies:** theater → title (each theater appears with a single title).
- **Violates:** title → theater (the same title “Casablanca” appears under two different theaters).

Thus the instance is an **Armstrong relation** for **F**.

Beyond Functional Dependencies: Multi-Valued Dependencies (MVDs) 🌐

> **Multi-valued dependency (MVD):** $X \twoheadrightarrow Y$ holds when, for a given value of **X**, the set of **Y** values is independent of the set of other attributes.

Illustrative Example

Movie	Director	Actor
M1	D1	A1
M1	D2	A1
M1	D1	A2
M1	D2	A2

- A movie can have **multiple directors** *and* **multiple actors**.
- No non-trivial **FDs** exist (the relation is already in **BCNF**).
- **Redundancy** remains because each director-actor pair is repeated.

Improved Design Using MVDs

Decompose into two separate relations that capture the independent multi-valued information:

1. **Directors**(Title, Director)
2. **Actors**(Title, Actor)

Both tables are in **4th Normal Form (4NF)**, which extends BCNF by eliminating non-trivial MVDs whose left side is not a superkey.

Normal Forms Recap (Reference) 📚

Normal Form	Condition on FDs / MVDs	Key Requirement
BCNF	Every non-trivial **FD** $X → Y$ has **X** as a superkey	—
4NF	Every non-trivial **MVD** $X \twoheadrightarrow Y$ has **X** as a superkey	—
5NF	Every non-trivial **JD** (join dependency) has a superkey left side	—

As discussed earlier, BCNF guarantees lossless join but may sacrifice dependency preservation; 4NF addresses the additional redundancy caused by MVDs.

🚀 Query Planning Overview

Brief Overview

This note covers **query planning** and was created from 18-page PDF slides. It covers an **example SQL query**, logical and physical plans, cost estimation, statistics, access paths, and data layout.

Key Points

- Understand how SQL queries translate into logical and physical plans.
- Learn the roles of different physical operators and their cost considerations.
- Explore how statistics drive optimization decisions.
- Get a visual sense of how data is stored in slotted pages.

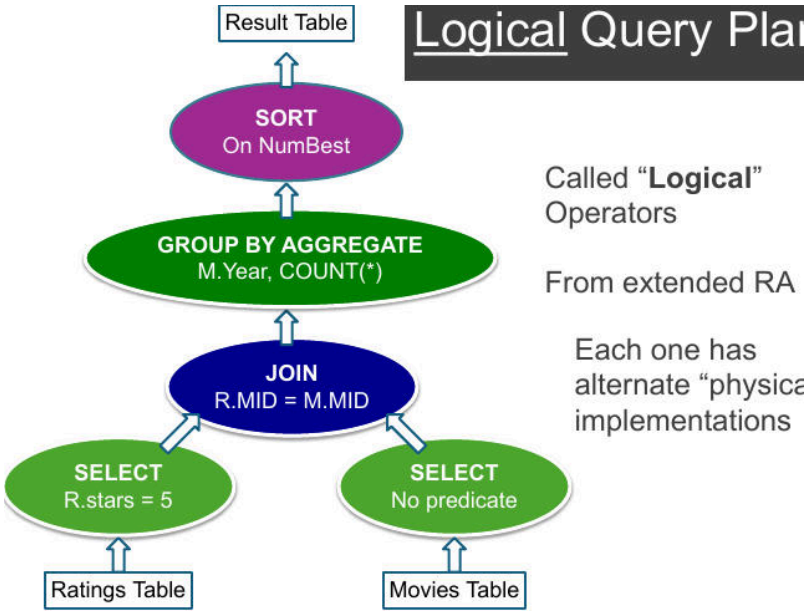
📊 Example SQL Query

> **SQL Statement**
> SELECT M.Year, COUNT(*) AS NumBest FROM Ratings R, Movies M WHERE R.MID = M.MID AND R.Stars = 5 GROUP BY M.Year ORDER BY NumBest DESC;

- Retrieves the number of 5-star ratings per movie release year.
- Results are sorted by the count (NumBest) in descending order.

🧩 Logical Query Plan

> **Logical Query Plan** – A tree of relational-algebra operators that describe *what* must be done, without specifying *how*.

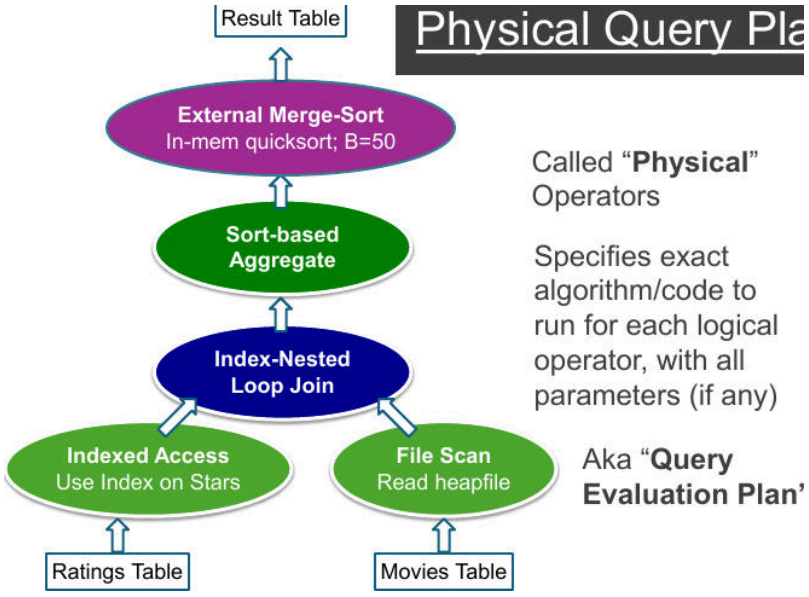


*The diagram visualizes each logical step: selecting rows from **Ratings** and **Movies**, joining on MID, grouping by Year, aggregating with COUNT(*), and sorting on NumBest.*

- **SELECT** (two green ovals) – filter R.Stars = 5 and pass through **Movies** rows.
- **JOIN** – inner join on R.MID = M.MID.
- **GROUP BY AGGREGATE** – group by M.Year and compute COUNT(*).
- **Result Table** – final sort on NumBest.

🏎️ Physical Query Plan

> **Physical Query Plan** – A tree of concrete operators that specify the exact algorithms the DBMS will execute.



The flowchart annotates each node with the chosen algorithm (e.g., "Index-Nested Loop Join") and access method (e.g., "Indexed Access Use Index on Stars").

Key operators in this plan:

Logical Operator	Physical Implementation	Notes
Select (filter on Stars)	Indexed Access (use index on Stars)	Fast lookup via index leaf pages
Join	Index-Nested Loop Join	Uses the index on Stars for the outer relation
Aggregate	Sort-based Aggregate (or Hash-based in alternative plan)	Groups after sorting
Sort (final ordering)	External Merge-Sort with in-memory quicksort (B=50)	Handles large intermediate results
Table Scan	File Scan (heapfile) for Movies	Full sequential read

Alternative physical plan (shown later) replaces the sort-based aggregate with a **Hash-based Aggregate** and uses a **Hash Join**.

⚙️ Components of a Physical Query Plan

> **Physical Operator** – A low-level database routine (scan, join, sort, etc.) with a concrete algorithm, estimated cost, output cardinality, and physical properties.

- **Leaves** – Access methods (SeqScan, IndexScan, BitmapScan).
- **Internal nodes** – Relational operators (Hash Join, Merge Join, Nested Loop Join, Sort, Hash Aggregate, Projection, Filter).

Each operator defines:

- **Algorithm** (e.g., quicksort, hash partitioning).
- **Estimated cost** (I/O + CPU + memory).
- **Output cardinality** (expected number of rows).
- **Physical properties** (ordering, partitioning, uniqueness, tuple format).

📐 Physical Properties of Operators

> **Physical Property** – Traits of an operator's output that influence downstream choices.

- **Output Ordering** – Is the result sorted? (e.g., merge join preserves order, hash join does not).
- **Partitioning / Hashing** – Does the output consist of hash buckets? Useful for subsequent hash-based operators.
- **Uniqueness** – Are duplicate rows retained?
- **Tuple Format / Row Layout** – Column order, presence of system columns, materialization vs. streaming.

Understanding these properties lets the optimizer chain compatible operators and avoid unnecessary re-ordering.

📈 Cost Model Overview

> **Cost Model** – Formula that predicts the total execution expense of a plan.

$ \text{Cost} = \text{I/O cost} + \text{CPU cost} + \text{Memory cost} $

- **I/O Cost** – Page reads/writes, index traversals, sequential scan page counts, spill-to-disk for sorts/hashes.
- **CPU Cost** – Predicate evaluation, hash computations, tuple comparisons, join condition checks.
- **Memory Cost** – Size of hash tables, sort buffers, and whether spills to disk are required.

The optimizer selects the plan with the lowest estimated total cost.

📏 Cardinality & Selectivity Estimation

> **Cardinality Estimation** – Predicting the number of rows produced by each operator.

- Uses **statistics**: total rows, NDV (number of distinct values), histograms, most-common-value (MCV) lists, null fractions.
- Example: predicate age > 40 → consult histogram of age to estimate percentage.
- For joins, estimate using NDV of the join attribute:

$ \text{Join size} \approx \frac{|R| \times |S|}{\text{NDV}(A)} $

- Accurate estimates reduce memory usage (smaller hash tables, fewer loop iterations) and improve overall plan quality.

📊 Statistics Utilized by the Optimizer

Statistic Type	Example Data	Impact on Planning
Base	Row count, page count, tuple width, dead vs. live rows	Determines scan costs; dead rows inflate cost if not vacuumed.
Column	NDV, histograms, MCVs, null fraction, correlation with physical order	Guides index selection, selectivity of predicates, and ordering benefits.
Index	Tree height, leaf page count, duplicate fraction, correlation with heap order	Influences choice between index scan vs. heap scan; predicts locality.

Multivariate	Multi-column MCVs, joint histograms, functional dependencies	Helps avoid independence assumptions; improves join size estimates.

Regular **VACUUM** updates base statistics, keeping cost estimates realistic.

🔍 Access Path Selection

> **Access Path** – The concrete method used to retrieve rows from a table.

Path	Typical Cost Estimate	Best Use Cases
Sequential Scan	O(#pages)	Non-selective predicates, small tables, no suitable index.
Index Scan	O(log N) + #tuples·random I/O	Equality lookups, small ranges, moderate selectivity.
Index-Only Scan	O(log N) (no heap reads)	All required columns are in the index.
Bitmap Index Scan	Combines multiple indexes	OR conditions, low-selectivity ranges.

The optimizer weighs estimated selectivity against these costs to choose the optimal path.

📚 Summary of Physical Planning Steps

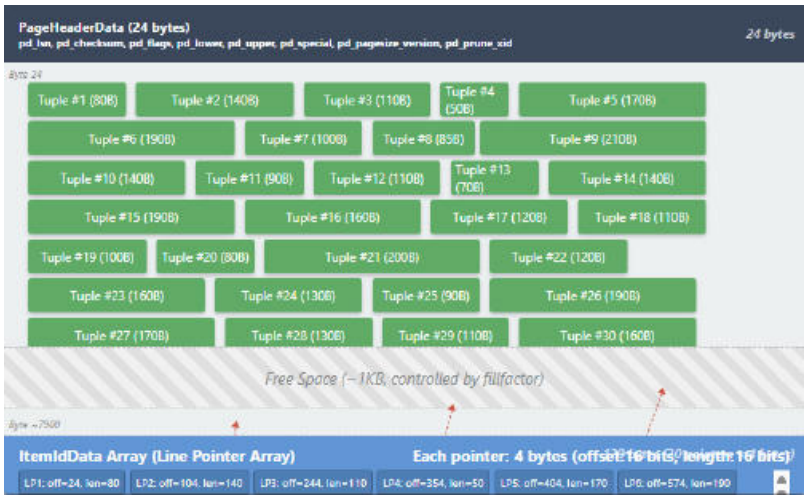
1. **Statistics lookup** – Gather table, column, and index metrics.
2. **Cardinality estimation** – Predict output sizes for each predicate and join.
3. **Access-path selection** – Choose scans or index accesses.
4. **Join ordering** – Dynamic programming / memoization to explore alternatives.
5. **Operator choice** – Pick join (hash, merge, NLJ), aggregation (hash vs. sort), and sorting methods.
6. **Physical operator tree generation** – Assemble a rooted tree of concrete operators.

7. **Execution** – Engine runs the tree.

📁 Data Layout: Slotted Pages

> **Slotted Page** – On-disk page format that stores variable-length records with a slot directory.

- **Page Header** – Metadata (page ID, free-space pointers, record count).
- **Data Records** – Variable-length tuples stored from the top downward.
- **Slot Directory** – Array of pointers at the bottom upward, each entry containing: offset, length, and optional status flags.



The image visualizes the header, tuple slots, and the ItemIdData array (line-pointer array) that maps TIDs to physical record locations.

Behavior of Slotted Pages

- **Delete a tuple** – Slot entry marked dead; space becomes reclaimable by **VACUUM**.
- **Update a tuple**
  - If space permits, a new version is written in-place (HOT update).
  - Otherwise, the tuple moves to another page; the old version stays dead, potentially causing bloat.
- **In-page movement** does **not** break indexes because indexes store TIDs (block number + offset), not absolute byte locations.

📖 Physical Operators Cheat Sheet

Category	Operators	Typical Use
Access Paths	Sequential Scan, Index Scan, Index-Only Scan, Bitmap Scan	Retrieve rows from base tables.
Join Methods	Nested Loop, Index-Nested Loop, Hash Join, Merge Join, Grace Hash Join (disk-based)	Combine rows from two relations.
Aggregation & Sorting	Sort, Hash Aggregate, Sort-Merge Union	Grouping, ordering, set operations.
Materialization	Materialize, Limit	Cache intermediate results, early stopping.
Parallelism	Parallel Seq Scan, Parallel Index Scan, Parallel Hash Join, Gather Merge, Parallel Append	Exploit multiple CPUs for large workloads.

These operators are combined according to the optimizer's cost model to form the final execution plan.

📚 Query Optimization Flow

Brief Overview

This note covers **SQL query optimization** and was created from a 22-page PDF presentation. It outlines the end-to-end flow from parsing to runtime, key logical rewrite rules, physical plan selection, and practical examples of join minimization.

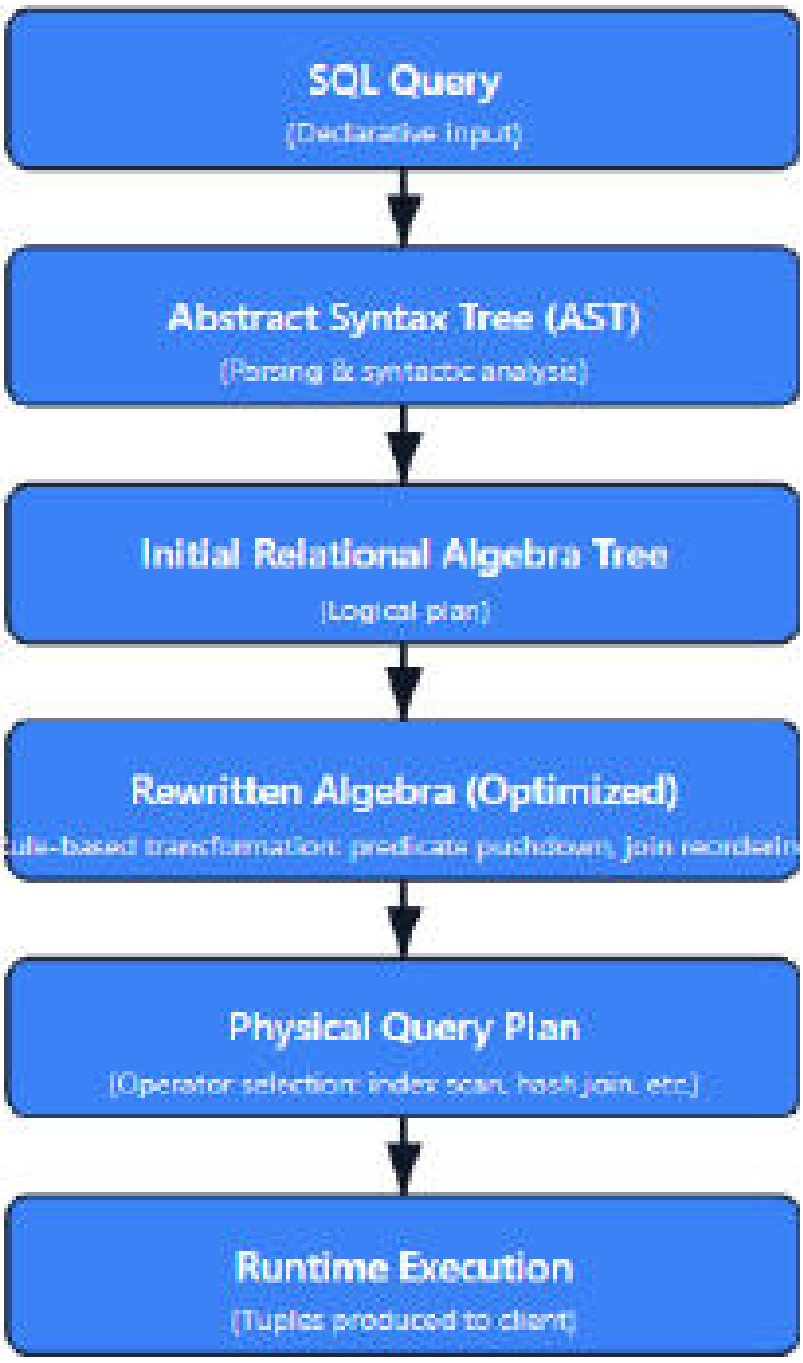
Key Points

- Understand the six-stage query lifecycle
- Master predicate push-down and join re-ordering
- Learn how logical rewrites influence physical plans
- See concrete examples with semi-joins and join minimization

---

📊 Overall Query Processing Flow

The flowchart below visualizes the six stages a SQL query goes through from textual input to result delivery.



The diagram shows a top-down progression:

1. **SQL Query** – declarative input.
2. **Abstract Syntax Tree (AST)** – parsing & syntactic analysis.
3. **Initial Relational Algebra Tree** – logical plan.
4. **Rewritten Algebra (Optimized)** – clause-based transformations (predicate push-down, join re-ordering).
5. **Physical Query Plan** – operator selection (index scan, hash-join, etc.).

6. **Runtime Execution** – tuples produced to the client.

🧩 Parsing & Semantic Analysis

- **SQL Query example**: SELECT name FROM Customer WHERE city = 'Paris';
- **Parsing → AST**

> The SQL string is tokenized and parsed into a tree representing grammatical components (SELECT, FROM, WHERE). Syntax errors are caught before semantic analysis.

- **Semantic Analysis → Initial Relational Algebra**

> The AST is translated into a logical algebra tree whose nodes are abstract operators (σ = selection, π = projection, ⋈ = join). This captures the query's meaning independently of physical storage.

📐 Logical Algebra Generation

- **Logical operators** represent *what* to do, not *how*:
  - **σ** (selection) – filter rows.
  - **π** (projection) – keep required columns.
  - **⋈** (join) – combine relations.

The resulting tree is the **initial logical plan**.

🔄 Logical Rewriting (Optimization)

Principles

> **Equivalence Preservation** – every rewrite must return exactly the same result as the original query.
> **Relational Algebra as the Internal Language** – rewrites are applied at the algebra

> level.
> **Reduce Before You Join** – push selections (σ) and projections (π) as early as possible.
> **Local Transformations** – most rules modify small sub-trees.
> **Iterative Refinement** – apply rules repeatedly until a fixpoint is reached.

Heuristic Goals

- **Push selections down** → fewer rows enter joins.
- **Push projections down** → narrower tuples, lower memory use.
- **Reorder joins** using commutativity/associativity for cheaper execution orders.
- **Replace expensive operators** (e.g., nested subqueries → semi-joins).
- **Exploit constraints & keys** (PK/FK, uniqueness, nullability).
- **Remove redundant operators** (merge cascaded selections, drop unnecessary projections).

Common Rewrites

- Predicate push-down
- Join re-ordering
- Eliminate redundant projections/selections
- Convert IN / EXISTS subqueries to **semi-joins**

📚 Equivalence Rules

Rule Type	Example (Algebra)	Description
Commutativity	$\sigma_1 \sigma_2 R \equiv \sigma_2 \sigma_1 R$	Order of selections does not matter.
	$R \bowtie S \equiv S \bowtie R$	Join is symmetric.
	$R \cup S \equiv S \cup R$	Union is symmetric.
Associativity	$(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$	Join grouping can be rearranged.
	$(R \cup S) \cup T \equiv R \cup (S \cup T)$	Union grouping can be rearranged.
Selection Rules	$\sigma_{c_1 \wedge c_2}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(R))$	Cascade of selections.

	$\sigma_c(\pi_{L}(R)) \equiv \pi_{L}(\sigma_c(R))$ if attrs(c)⊆L	Move selection above projection when attributes match.
	$\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$ if c references only R	Predicate push-down through join.
Projection Rules	$\pi_{L_1}(\pi_{L_2}(R)) \equiv \pi_{L_1}(R)$ if $L_1 \subseteq L_2$	Cascade of projections.
	$\pi_{L_1 \cup L_2}(R \bowtie S) \equiv \pi_{L_1}(R) \bowtie \pi_{L_2}(S)$	Push projection below join (when safe).
Join-Selection Interaction	$R \bowtie_{c}(S) \equiv \sigma_c(R \times S)$	Theta-join equals selection over Cartesian product.
Set Operation Rules	$\sigma_c(R \cup S) \equiv \sigma_c(R) \cup \sigma_c(S)$	Selection distributes over union.
	$\pi_L(R \cup S) \equiv \pi_L(R) \cup \pi_L(S)$	Projection distributes over union.
Rename (ρ) Rules	$\rho_{A \to B}(\sigma_c(R)) \equiv \sigma_{c'}(\rho_{A \to B}(R))$	Rename commutes with selection (attributes in c renamed).
	$\rho_{A \to B}(\pi_L(R)) \equiv \pi_{L'}(\rho_{A \to B}(R))$	Rename commutes with projection.

🛠️ Physical Plan Selection

Logical operators are mapped to concrete algorithms:

Logical Operator	Physical Alternatives
Scan	*Seq Scan* (full table) vs. *Index Scan* (use B-tree/bitmap index)
Join	*Nested Loop Join* (row-by-row), *Hash Join* (build hash table), *Merge Join* (sorted inputs)
Aggregation	*Hash-aggregate*, *Sort-based aggregate*

Sorting	*External merge sort*, *In-memory quicksort*

The chosen algorithms form the **physical query plan**, a blueprint for execution.

🚀 Execution Engine (Runtime)

- Operators run as **iterators**, pulling tuples from child nodes.
- Intermediate results stream **bottom-up** (leaf → root).
- Final rows are returned to the client application.

🔗 Logical ↔ Physical Optimization Connection

- **Logical rewriting shrinks the search space** for physical plans by removing unnecessary operators and reducing intermediate cardinalities.
- Early **selection/projection push-down** → smaller intermediate results → fewer physical join candidates → lower I/O & memory use.
- A **poor logical plan** (e.g., bad join order) forces the physical optimizer to consider expensive join strategies; a **good logical plan** enables cost-based choices (hash, merge, index-nested-loop) on a simpler tree.

📈 Semi-Join Concept

> A **semi-join** R ⋉ S returns all tuples of R that have at least one matching tuple in S; it does **not** append columns from S.

Key properties

- Produces only the left-hand relation's attributes.
- Implements existence checks (IN, EXISTS).
- Reduces tuple width and intermediate result size.
- Enables join reordering and selection push-down across subquery boundaries.

🔧 Join Minimization

Goal: Eliminate joins that do not affect the result.

- **Redundant join**: when the information it supplies is already guaranteed by other joins or constraints (PK/FK, functional dependencies).
- **Benefits:** fewer intermediate results → faster execution.

Typical sources of redundancy

- Hand-written queries or auto-generated SQL.
- Unfolded view definitions.
- Schemas with strong constraints.

Pattern folding (homomorphism) – mapping multiple variables to a single one without altering semantics.

📝 End-to-End Query Processing Examples

Example 1 – Customer / Order

SQL

```
SELECT cname
FROM Customer c
WHERE EXISTS (
    SELECT 1
    FROM OrderInfo o
    WHERE o.cid = c.cid
      AND o.total_value > (
          SELECT AVG(o2.total_value)
          FROM Customer c2
          JOIN OrderInfo o2 ON c2.cid = o2.cid
          WHERE c2.city = c.city
      )
);
```

Logical Rewrite Highlights

- **Selection push-down** on OrderInfo (filter by cid).
- **Projection push-down** (keep only needed columns).
- **Join re-ordering** to compute city-wise average before applying the outer filter.
- **Subquery to semi-join** (EXISTS → semi-join).
- **Removal of unused attributes** (e.g., columns not needed for final cname).

Resulting logical plan (simplified):

```
π_cname ( σ_{ o.total_value > AvgCity } ( Customer ⋈ OrderInfo ) )
```

where AvgCity is computed via an aggregation on Customer ⋈ OrderInfo grouped by city.

Example 2 – Student / Course / Dept

SQL

```
SELECT s.sname
FROM Student s
JOIN Enroll e ON s.sid = e.sid
JOIN Course c ON e.cid = c.cid
WHERE c.dept IN (
    SELECT d.dept FROM Dept d WHERE d.chair LIKE '^Dr\.'
)
AND c.credits > (
    SELECT AVG(c2.credits) FROM Course c2 WHERE c2.dept = c.dept
);
```

Logical Rewrite Highlights

- **Selection push-down** on Dept (chair LIKE '^Dr\.').
- **Projection push-down** (retain only dept).
- **IN → semi-join**: c.dept ⋉ Dept_filtered.
- **Join minimization** – join Course with the filtered Dept first, then with Enroll, and finally with Student.
- **Aggregation push-down** – compute AVG(credits) per department before applying the credit filter.

Resulting join order (optimal):

```
((Course ⋉ Dept_filtered) ⋈ Enroll) ⋈ Student
```

📊 Summary Tables

Logical vs. Physical Optimizations

Aspect	Logical Optimization	Physical Optimization
Focus	Transform algebraic expression while preserving semantics.	Choose concrete algorithms for each operator.
Typical Techniques	Predicate push-down, projection push-down, join re-ordering, semi-join conversion.	Index scan vs. sequential scan, hash vs. merge join, parallel execution.
Effect on Search Space	Reduces size of algebra tree → fewer physical alternatives.	Explores cost-based choices on the reduced tree.
Primary Goal	Lower cardinalities & tuple widths.	Minimize I/O, CPU, and memory usage.

📚🗃️ Relational Data & SQL Basics

Brief Overview

This note covers **Relational Databases** and was created from a 27-page PDF presentation. It includes a quick recap of relational model, Airbnb listings example, SQL query basics, data types, and constraint enforcement.

Key Points

- Understand key relational model terminology (attributes, tuples, domains).
- Apply basic SELECT queries and learn common constraints (uniqueness, primary keys).
- Explore the Airbnb dataset to practice filtering, joining, and type casting.

📊 Quick Recap – Relational Data Model

> A **relational data model** stores data in *relations* (tables).
>
> - **Attributes** → columns
> - **Tuples** → rows (records)
> - **Domain** → set of all possible values for an attribute

Attribute	Example Values
CustomerID	1, 2, 3
CustomerName	“Google”, “Amazon”, “Apple”
Status	“Active”, “Inactive”

❌ Bad Example of Relational Data

> Missing clear domain definitions for columns C2, C3, C4 leads to ambiguity.

C1	C2	C3	C4
100	John	2022-01-04	15.3
63.85	21.999	Fred	2021-12-30

🏡 Our First Example – Airbnb Listings (Washington, DC)

- **Attributes:** 62
- **Records:** 867 (each row = one Airbnb listing)

Sample rows (truncated for brevity):

id	listing_url	name	space	room_type	price
1328195	https://airbnb.com/rooms/1328195	“Fabulous Furnished JBR/IBA”	–	Entire home/apt	$160
1422933	https://airbnb.com/rooms/1422933	“Fabulous furnished Jedcoom St!”	–	Entire home/apt	$?
1449253	…	“Efficiency Dedroom”	–	Private room	$?

(Only a few rows shown; full table contains many listings.)

🗃️ Operations in a Relational Model

- **Retrieve whole table** → SELECT * FROM ;
- **Get a specific record** → SELECT * FROM WHERE id = 1625869;
- **Project a column** → SELECT column_name FROM ;
- **Select rows with condition** → e.g., *price < $100* and *room_type = 'Entire home'*

🔒 Constraints in a Relational Model

> **Constraint:** logical rule that restricts permissible data.

- **Attribute domain constraint** – e.g., ID cannot be negative.
- **Uniqueness constraint** – e.g., listing_url must be unique.
- **Key constraint** – combination of attributes uniquely identifies a row.

Example CREATE TABLE (simplified)

```
CREATE TABLE "Airbnb_listings" (
    listing_id   INTEGER PRIMARY KEY,
    listing_url  TEXT    UNIQUE,
    name         VARCHAR(256) NOT NULL,
    summary      TEXT,
    space        TEXT,
    description  TEXT,
    neighborhood_overview TEXT,
    host_id      INTEGER,
    host_name    VARCHAR(128),
    host_since   TEXT,
    city         VARCHAR(256),
    state        CHAR(2) NOT NULL,
    zipcode      INTEGER,
    price        TEXT,
    weekly_price TEXT,
    monthly_price TEXT,
    -- other columns omitted for brevity
);
```

📚 Standard Data Types

Category	Types
Numeric	INTEGER, NUMERIC (floating-point)
Textual	VARCHAR(n), TEXT
Temporal	DATE, TIME, DATETIME
Geometric	POINT, LINE, POLYGON, CIRCLE
Binary	BYTEA (binary array)

> **Note:** Operations are type-specific (e.g., addition works on numbers, not on circles).

🛠️ Basic SQL Construct

```
SELECT
FROM
WHERE  ;
```

- Result of a query is itself a **relation** (table).
- Tuple order in the result is **unspecified** unless ORDER BY is used.

📋 Example Queries

1️⃣ Find basic listing info

```
SELECT id, name, space, room_type, price
FROM "Airbnb_listings";
```

Returns columns: id, name, space, room_type, price.

2️⃣ Private rooms only, hide null spaces

```
SELECT id, name, space, price
FROM "Airbnb_listings"
WHERE room_type = 'Private room' AND space IS NOT NULL;
```

3️⃣ Private rooms in DC or MD, also show city & state

```
SELECT id, name, space, city, state, price
FROM "Airbnb_listings"
WHERE room_type = 'Private room'
  AND space IS NOT NULL
  AND (state = 'MD' OR state = 'DC');
```

❗ Common mistake (missing parentheses)

Incorrect:

```
SELECT id, name, space, city, state, price
FROM "Airbnb_listings"
WHERE room_type = 'Private room' AND space IS NOT NULL
  AND (state = 'MD' OR state = 'DC';
```

Missing closing parenthesis leads to syntax error.

4️⃣ Limit results to 10 rows

```
SELECT id, name, space, city, price
FROM "Airbnb_listings"
WHERE room_type = 'Private room'
LIMIT 10;
```

🧮 Data-Type Issue – Comparing Text to Number

Attempt:

```
SELECT id, name, space, city, price
FROM "Airbnb_listings"
WHERE room_type = 'Private room' AND space IS NOT NULL AND price <
```

Error: *operator does not exist: text = integer* → price is stored as **TEXT**; must cast:

```
SELECT id, name, space, city, price
FROM "Airbnb_listings"
WHERE room_type = 'Private room'
  AND space IS NOT NULL
  AND ltrim(regexp_replace(price, '[\$,]', '', 'g'))::numeric < 100
```

🔤 String Functions & Type Casting

Function	Description
`	
bit_length(str)	Number of bits
char_length(str)	Number of characters
lower(str)	Lower-case conversion
trim([leading	trailing
substring(str FROM pattern)	Extracts substring matching a POSIX regex
regexp_replace(str, pattern, replace)	Regex-based replace

ltrim(str, chars)	Remove leading characters
::type	Cast to type (e.g., ::numeric)

📦 Sets, Multisets & Deduplication

- **Set:** collection with **no duplicates**.
- **Multiset (Bag):** collection that may contain **multiple instances** of the same element.

Distinct values

```
SELECT DISTINCT room_type
FROM "Airbnb_listings";
```

Distinct on multiple columns

```
SELECT DISTINCT room_type, property_type
FROM "Airbnb_listings";
```

🔍 Comparing Strings & Substrings

Equality / Inequality

```
WHERE room_type = 'Private room'
WHERE room_type != 'Private room'
```

LIKE – pattern matching

- % → any sequence of characters
- _ → any single character

Example: find listings with “luxury” in the name (case-insensitive)

```
SELECT id, name, city, price
FROM "Airbnb_listings"
```

```
WHERE lower(name) LIKE '%luxury%';
```

Equivalent using ILIKE (PostgreSQL specific):

```
WHERE name ILIKE '%luxury%';
```

Complex LIKE usage

```
SELECT id, description, notes, city, price
FROM "Airbnb_listings"
WHERE (description ILIKE '%bedroom%' OR description ILIKE '%br%')
  AND (description ILIKE '%smoking%' OR notes ILIKE '%smoking%');
```

Substring comparison – “walk … museum”

```
SELECT id, description, notes, city, price
FROM "Airbnb_listings"
WHERE description ILIKE '%walk%museum%';
```

Using LIKE to filter formatted prices (no arithmetic)

```
SELECT id, description, weekly_price
FROM "Airbnb_listings"
WHERE weekly_price LIKE '$%';
```

🖌️ Expressions in the SELECT Clause (Renaming)

> Build a custom textual report for listings that have daily, weekly, and monthly prices.

```
SELECT
    'Listing ID ' || id ||
    ', located at ' || street ||
    ', has daily rate of ' || price ||
    ', weekly rate of ' || weekly_price ||
```

```
    ' and monthly rate of ' || monthly_price AS report
FROM "Airbnb_listings"
WHERE weekly_price IS NOT NULL
  AND monthly_price IS NOT NULL;
```

Result column report contains a human-readable sentence for each qualifying listing.

📚 Database Fundamentals Overview

Brief Overview

This note covers Database Fundamentals and was created from a PDF presentation, 26 pages. Key concepts include superkeys, **primary keys**, aggregate functions, set operations, and foreign keys.

Key Points

- Definition and distinction between superkeys, candidate keys, and primary keys.
- SQL syntax for creating tables with primary key and foreign key constraints.
- Aggregate functions and set operations like UNION, INTERSECT, and EXCEPT.
- Grouping with GROUP BY, filtering with HAVING, and the role of WHERE clauses.
- Referential integrity rules and how foreign keys enforce relationships.

🔑 Superkeys & Keys

📚 Definition

> **Superkey** – a set of attributes $S \subseteq R$ such that no two distinct tuples in any legal state of R have the same values on S (i.e., $t_1[S] \neq t_2[S]$).

> **Key (Candidate Key)** – a **minimal** superkey; removing any attribute from it would break the uniqueness property.

📂 Example 1 – STUDENT(StudentID, Name, Major, Year)

Tuple	StudentID	Name	Major	Year
1	101	Alice	CS	2024
2	102	Bob	EE	2023
3	103	Carol	CS	2024

- **Superkeys**: {StudentID}, {StudentID, Name}, {StudentID, Major, Year} (all guarantee uniqueness).
- **Keys** (minimal): {StudentID} only. {StudentID, Name} is **not** a key because {StudentID} alone already suffices.

📂 Example 2 – COURSE(CourseID, Title, Room, Time)

CourseID	Title	Room	Time
C101	DB Systems	220	9AM
C102	???	240	10AM
C103	Algorithms	250	9AM

- When **Room + Time** repeats, it is **not** a key.
- With the revised data where each (Room, Time) pair is unique, we have:
  - **Candidate keys**: {CourseID} and {Room, Time}.
  - {CourseID, Room} is a **superkey** but not minimal (since {CourseID} alone works).

📂 Example 3 – ENROLL(StudentID, CourseID, Semester, Grade)

StudentID	CourseID	Semester	Grade
101	C101	Fall2024	A
101	C102	Fall2024	B
102	C101	Fall2024	A

- **Key**: {StudentID, CourseID, Semester} (unique for each enrollment).
- {StudentID, CourseID} is **not** a key because the same course can appear in different semesters.
- Adding Grade yields a **superkey** but not minimal.

🗝️ Primary Key & Candidate Keys

📌 Definitions

> **Candidate key** – any minimal superkey of a relation.
> **Primary key** – the candidate key chosen by the designer to uniquely identify tuples.

> **Prime attribute** – an attribute that belongs to *some* candidate key.
> **Non-prime attribute** – an attribute that does **not** belong to any candidate key.

✅ Primary-Key Properties (presented in a table)

Property	Description
Uniqueness	No two tuples share the same primary-key value.
Minimality	Removing any attribute destroys uniqueness.
Non-nullability	Every tuple must contain a non-NULL primary-key value.
Stability	Primary-key values should rarely change; they act as permanent identifiers.

🛠️ SQL Example – Creating Tables with Primary Keys

```
CREATE TABLE student (
    studentid BIGSERIAL PRIMARY KEY,
    name TEXT,
    major TEXT,
    year INT
);

CREATE TABLE enroll (
    studentid INT REFERENCES student(studentid),
    courseid TEXT REFERENCES course(courseid),
    semester TEXT,
    grade CHAR(2),
    PRIMARY KEY (studentid, courseid, semester)
);
```

Alternative syntax to add a primary-key constraint later:

```
ALTER TABLE enroll
ADD CONSTRAINT enroll_pk PRIMARY KEY (studentid, courseid, semester);
```

📊 Aggregate Functions & Set Operations

🧱 Aggregate Functions

- **Numeric**: sqr(), sqrt(), abs(), round().
- **String**: trim(), lower(), upper().
- **Casting**: e.g., ::numeric.
- **Date**: date '2001-09-28', intervals.
- **User-defined**: e.g., conv_price().
- **Aggregate** (operate on collections): COUNT, SUM, AVG, STDDEV, MAX, MIN.

♾️ Set Operations (SQL)

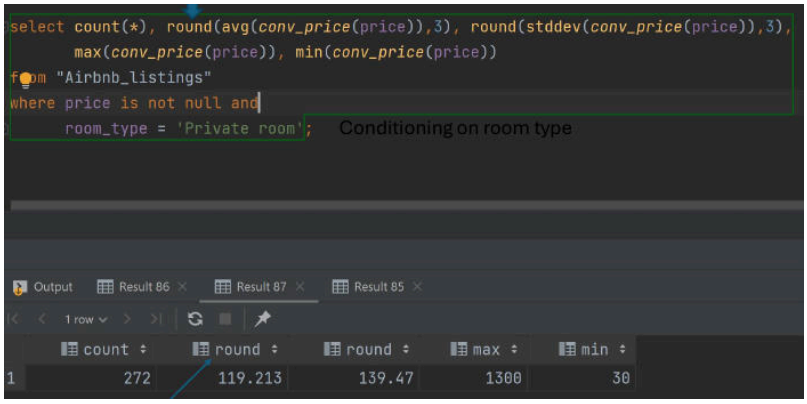
Operation	Symbol	Meaning
Union	UNION	Combines distinct rows from two queries (schemas must be compatible).
Difference	EXCEPT	Rows in the first query that are **not** in the second.
Intersection	INTERSECT	Rows common to both queries (can be expressed as A – (A – B)).

📈 Example – Price Statistics (using UNION)

> The query aggregates price, weekly_price, and monthly_price from the Airbnb_listings table, producing one-row result sets that are then united.

PriceType	count	avg	stddev	max	min
monthly price	349	3341.5186	2436.5140	19000	495
price	867	198.6148	238.7743	2822	10
weekly price	408	936.2843	826.7600	8877	165

🖼️ Visual Aid



The screenshot shows a SQL query that computes count, average, standard deviation, maximum, and minimum for the price column (filtered by room_type = 'Private room'). The result table below the query illustrates the calculated statistics.

📈 GROUP BY, HAVING, WHERE

📂 GROUP BY

- Extracts unique values of the **grouping attributes**.
- Computes aggregate functions **independently** for each group.
- **SELECT** clause may contain only grouping attributes and aggregates (no other columns).

```
SELECT city, property_type, room_type,
       COUNT(*) AS cnt,
       ROUND(AVG(conv_price(price)), 3) AS avg_price,
       ROUND(STDDEV(conv_price(price)), 3) AS std_price,
       MAX(conv_price(price)) AS max_price,
       MIN(conv_price(price)) AS min_price
FROM Airbnb_listings
WHERE price IS NOT NULL
GROUP BY city, property_type, room_type
ORDER BY city ASC, property_type DESC;
```

🔎 HAVING vs. WHERE

Clause	When it applies	What it can reference
WHERE	Before grouping; filters raw rows.	Any column, even if not in SELECT.
HAVING	After grouping; filters groups.	Only aggregate expressions or grouping columns.

📂 Example – HAVING with Group Count

```
SELECT city, property_type, room_type,
       COUNT(*) AS cnt,
       ROUND(AVG(conv_price(price)), 3) AS avg_price,
       ROUND(STDDEV(conv_price(price)), 3) AS std_price,
       MAX(conv_price(price)) AS max_price,
       MIN(conv_price(price)) AS min_price
FROM Airbnb_listings
WHERE price IS NOT NULL
GROUP BY city, property_type, room_type
HAVING COUNT(*) > 5
ORDER BY city ASC, property_type DESC;
```

Result excerpt:

city	property_type	room_type	cnt	avg_price	std_price	max_price	min_price
Washington	Townhouse	Entire home/apt	1	338.25	202.011	720	720
Washington	Townhouse	Private room	1	115.22	113.053	399	399
...	...	...	...	...	...	...	...

🔗 Foreign Keys & Referential Integrity

📌 Definitions

> **Foreign key** – a set of attributes in a *child* table that references the **primary key** of a *parent* table, enforcing a link between the two tables.

> **Referential integrity** – the rule that a foreign-key value must either be **NULL** or match an existing primary-key value in the referenced table.

- **Child table** – contains the foreign-key column(s).
- **Parent (referenced) table** – contains the primary-key being referenced.

🛠️ SQL Example – Defining Foreign Keys

```
CREATE TABLE customers (
    customer_id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    customer_name VARCHAR(255) NOT NULL
);

CREATE TABLE contacts (
    contact_id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    customer_id INT,
    contact_name VARCHAR(255) NOT NULL,
    phone VARCHAR(15),
    email VARCHAR(100),
    CONSTRAINT fk_customer
        FOREIGN KEY (customer_id)
        REFERENCES customers (customer_id)
);
```

- GENERATED ALWAYS AS IDENTITY creates sequential integer IDs; attempts to insert a manual value raise an error.
- GENERATED BY DEFAULT AS IDENTITY would allow manual insertion if needed (not shown in the transcript).

📊 Simple Referential-Integrity Illustration

Persons		
PersonID	LastName	FirstName
1	Hansen	Ola
2	Svendson	Tove

3	Pettersen	Kari
Orders		
OrderID	OrderNumber	PersonID
77895	44678	1
22456	24562	2

An order cannot exist without a matching PersonID in Persons.

📚 PostgreSQL Integrity & Querying

Brief Overview

This note covering **Database Integrity & Querying** was created from a PDF presentation with 35 pages. It focuses on how PostgreSQL enforces referential integrity through foreign keys and their various *ON DELETE/ ON UPDATE* actions, practical ways to retrieve schema information, and common query patterns such as joins, nested queries, and EXISTS clauses.

Key Points

- Detailed explanation of the five *ON DELETE* options (NO ACTION, RESTRICT, CASCADE, SET NULL, SET DEFAULT) and how they affect related rows.
- PostgreSQL **foreign keys** and the use of information_schema queries to list schemas, tables, columns, constraints, and indexes.
- Overview of **JOIN operations** (INNER, LEFT, RIGHT, outer-mixing) and how they shape result sets.
- Techniques for nested queries and EXISTS/NOT EXISTS to filter and correlate data across tables.

🔗 Referential Integrity

> **Definition:** A *foreign key* creates a link between two tables, ensuring that a value in the child table corresponds to an existing value in the parent (referenced) table. PostgreSQL can automatically enforce consistency when the referenced row is updated or deleted.

ON DELETE / ON UPDATE Options

Action	Description
NO ACTION	Rejects the operation if dependent rows exist (default).
RESTRICT	Same as *NO ACTION* but checked immediately (not deferred).
CASCADE	Propagates the change automatically to dependent rows.
SET NULL	Sets the foreign-key column to **NULL**.
SET DEFAULT	Sets the foreign-key column to its **DEFAULT** value.

Example: ON DELETE CASCADE

```
CREATE TABLE enroll (
    studentid INT REFERENCES student(studentid) ON DELETE CASCADE,
    courseid  TEXT REFERENCES course(courseid)   ON DELETE CASCADE,
    semester  TEXT,
    grade     CHAR(2),
```

```
    PRIMARY KEY (studentid, courseid, semester)
);
```

Effect: Deleting a **student** or a **course** automatically removes the related enrollment rows.

Example: ON DELETE SET NULL

```
CREATE TABLE teaches (
    profid    INT REFERENCES professor(profid) ON DELETE SET NULL,
    courseid  TEXT REFERENCES course(courseid),
    semester  TEXT,
    PRIMARY KEY (courseid, semester)
);
```

Effect: Deleting a professor sets profid to **NULL** while keeping the course record.

Example: ON DELETE SET DEFAULT

```
INSERT INTO professor (profid, profname, department)
VALUES (999, 'TBD', 'Administration');

CREATE TABLE teaches (
    profid    INT DEFAULT 999 REFERENCES professor(profid) ON DELETE SET DEFAULT,
    courseid  TEXT REFERENCES course(courseid),
    semester  TEXT,
    PRIMARY KEY (profid, courseid, semester)
);
```

Effect: Deleting a professor reassigns their courses to the placeholder professor (profid = 999).

Example: ON DELETE RESTRICT

```
CREATE TABLE enroll (
    studentid INT REFERENCES student(studentid),
    courseid  TEXT REFERENCES course(courseid) ON DELETE RESTRICT,
    semester  TEXT,
    grade     CHAR(2),
    PRIMARY KEY (studentid, courseid, semester)
);
```

Effect: A course cannot be deleted while enrollments referencing it still exist.

Example: ON DELETE NO ACTION (default, deferred)

```
CREATE TABLE enroll (
    studentid INT REFERENCES student(studentid),
    courseid  TEXT REFERENCES course(courseid) ON DELETE NO ACTION,
```

```
    semester  TEXT,
    grade     CHAR(2),
    PRIMARY KEY (studentid, courseid, semester)
);
```

Effect: The check is performed at transaction commit; the delete is rejected if dependent rows remain.

🗂️ Getting Schema Information (PostgreSQL)

- **List all schemas**

```
SELECT schema_name FROM information_schema.schemata;
```

- **List tables in a schema**

```
SELECT table_name FROM information_schema.tables
WHERE table_schema = 'public';
```

- **List columns for a specific table**

```
SELECT column_name, data_type, is_nullable
FROM information_schema.columns
WHERE table_schema = 'public'
  AND table_name   IN ('airbnb_listings','airbnb_reviews','mutualfunds','usnewspape
```

- **List primary keys**

```
SELECT kc.column_name
FROM information_schema.table_constraints tc
JOIN information_schema.key_column_usage kc
  ON kc.constraint_name = tc.constraint_name
WHERE tc.constraint_type = 'PRIMARY KEY'
  AND tc.table_schema = 'public';
```

- **List foreign keys**

```
SELECT tc.constraint_name, kcu.column_name, ccu.table_name AS foreign_table,
       ccu.column_name AS foreign_column
FROM information_schema.table_constraints tc
JOIN information_schema.key_column_usage kcu
  ON kcu.constraint_name = tc.constraint_name
JOIN information_schema.constraint_column_usage ccu
```

```
  ON ccu.constraint_name = tc.constraint_name
WHERE tc.constraint_type = 'FOREIGN KEY';
```

- List unique constraints

```
SELECT constraint_name, column_name
FROM information_schema.table_constraints tc
JOIN information_schema.key_column_usage kcu
  ON kcu.constraint_name = tc.constraint_name
WHERE tc.constraint_type = 'UNIQUE';
```

- List indexes

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

- List check constraints

```
SELECT conname, pg_get_constraintdef(oid)
FROM pg_constraint
WHERE contype = 'c';
```

✖ Cartesian Product

> **Definition:** The Cartesian product R × S returns every possible concatenated pair where x is a tuple from R and y is a tuple from S.

Illustration:
If R = {John, Fred, Liz} and S = {115 1st St, 823 Main St, 13921 Genesee Ave.}, the product contains 9 rows such as John – 115 1st St, Fred – 823 Main St, etc.

🔄 Join Operations

Inner Join Example (Top 10 Holdings)

```
SELECT m.fund_symbol, n.top10_holdings
FROM mutualfunds m
JOIN mfnews n
  ON m.fund_symbol = n.fund_symbol
WHERE n.src ILIKE 'marketwatch%';
```

SPJ (Select-Project-Join) Query

```
SELECT DISTINCT m.fund_symbol, n.management_name, u.src,
       u.publishdate, u.title
FROM mutualfunds m
JOIN mfnews n ON m.fund_symbol = n.fund_symbol
JOIN usnewspaper u ON n.newsid = u.id
WHERE u.collectiondate >= DATE '2021-01-01'
  AND u.collectiondate < DATE '2021-01-01' + INTERVAL '10 days'
ORDER BY u.publishdate, m.fund_symbol;
```

Left Outer Join

> **Definition:** Returns all rows from the left table; rows without a match in the right table have NULL for right-table columns.

```
SELECT DISTINCT m.fund_symbol, n.newsid
FROM mutualfunds m
LEFT JOIN mfnews n ON m.fund_symbol = n.fund_symbol
ORDER BY n.newsid NULLS LAST;
```

Right Outer Join

> **Definition:** Returns all rows from the right table; unmatched left-table columns become NULL.

```
SELECT DISTINCT m.fund_symbol, n.newsid
FROM mutualfunds m
RIGHT JOIN mfnews n ON m.fund_symbol = n.fund_symbol
ORDER BY n.newsid NULLS LAST;
```

Mixing Outer and Inner Joins (Exercise)

```
-- Query A
SELECT m.fund_symbol, n.newsid, u.title
FROM mfnews n
RIGHT OUTER JOIN mutualfunds m ON m.fund_symbol = n.fund_symbol
INNER JOIN usnewspaper u ON n.newsid = u.id;

-- Query B
SELECT m.fund_symbol, n.newsid, u.title
FROM usnewspaper u
RIGHT OUTER JOIN mfnews n ON n.newsid = u.id
INNER JOIN mutualfunds m ON m.fund_symbol = n.fund_symbol;
```

Key difference: The order of joins determines which table's rows are preserved when no match exists.

📦 Nested Queries

General Pattern

1. **Break** the problem into smaller subqueries.
2. **Combine** them using SELECT, FROM, WHERE, or WITH.
3. Use **EXISTS**, **IN**, or scalar subqueries as needed.

Example 1 – Highest PE Ratio

```
-- Find the maximum PE ratio
SELECT MAX(fund_price_earning_ratio) AS max_pe
FROM mutualfunds;
```

```
-- Retrieve the fund(s) with that ratio
SELECT fund_symbol, fund_long_name, fund_price_earning_ratio
FROM mutualfunds
WHERE fund_price_earning_ratio = (
    SELECT MAX(fund_price_earning_ratio) FROM mutualfunds
);
```

Example 2 – Conditional WHERE Placement

```
SELECT fund_symbol, fund_long_name, fund_price_earning_ratio
FROM mutualfunds
WHERE fund_price_earning_ratio > (
    SELECT AVG(fund_price_earning_ratio) FROM mutualfunds
) AND total_net_assets >= 50000000;
```

Example 3 – Family-Specific PE Comparison

```
SELECT fund_symbol, fund_long_name, fund_price_earning_ratio
FROM mutualfunds
WHERE fund_price_earning_ratio > 1.2 * (
    SELECT AVG(fund_price_earning_ratio) FROM mutualfunds
)
AND fund_family ILIKE '%American Century%';
```

WITH (Common Table Expression)

```
WITH miniquery AS (
    SELECT DISTINCT n.fund_symbol, u.title
    FROM mfnews n
    JOIN usnewspaper u ON n.newsid = u.id
)
SELECT m.fund_symbol, mq.title, m.management_name
```

```
FROM mutualfunds m
LEFT JOIN miniquery mq ON m.fund_symbol = mq.fund_symbol;
```

Finding Median PE Ratio per Family

```
SELECT fund_family,
       PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY fund_price_earning_ratio) AS median_pe
FROM mutualfunds
GROUP BY fund_family
ORDER BY median_pe DESC
LIMIT 10;
```

Nested Subquery in SELECT Clause

```
SELECT fund_symbol,
       fund_price_earning_ratio /
       (SELECT AVG(fund_price_earning_ratio) FROM mutualfunds) AS relative_pe
FROM mutualfunds
WHERE fund_family = 'Aberdeen'
ORDER BY relative_pe DESC NULLS LAST;
```

✅ EXISTS & NOT EXISTS

EXISTS – “Is there at least one matching row?”

```
SELECT DISTINCT name
FROM Airbnb_listings al
WHERE EXISTS (
    SELECT 1
    FROM Airbnb_reviews ar
    WHERE ar.listing_id = al.id
);
```

Result: Lists Airbnb listings that have **at least one** review.

NOT EXISTS – “No matching rows”

```
SELECT al.*
FROM Airbnb_listings al
WHERE NOT EXISTS (
    SELECT 1
    FROM Airbnb_reviews ar
    WHERE ar.listing_id = al.id
);
```

Result: Returns listings **without** any reviews.

Correlated Subqueries

> **Definition:** A subquery that references columns from the outer query, creating a row-by-row evaluation (loop-inside-loop).

```
SELECT DISTINCT al.name
FROM Airbnb_listings al
WHERE EXISTS (
    SELECT 1
    FROM Airbnb_reviews ar
    WHERE ar.listing_id = al.id
);
```

The outer query scans Airbnb_listings; for each row it runs the inner query to test for a matching review.

📊 Miscellaneous Queries

Counting Rows in mutualfunds

```
SELECT COUNT(*) FROM mutualfunds;
```

Selecting Distinct Fund Symbols and Titles (Joined Tables)

```
SELECT DISTINCT m.fund_symbol, u.title, m.management_name
FROM mutualfunds m
LEFT JOIN mfnews n ON m.fund_symbol = n.fund_symbol
JOIN usnewspaper u ON n.newsid = u.id
ORDER BY m.fund_symbol, u.publishdate DESC;
```

📚 Null Values in SQL

Brief Overview

This note covers **NULL** values in SQL and was created from a 49-page PDF presentation. It covers the NULL marker, testing with **IS NULL**, handling unknown data, and query examples.

Key Points

- Understanding NULL semantics
- Using **IS NULL** in WHERE clauses
- Dealing with NULL in aggregates
- Common pitfalls and best practices

Null Values in SQL 🐞

> **NULL** – a special marker used to indicate that a data value does not exist or is unknown.

- **Testing for NULL**
  - IS NULL

📚 Transaction Basics

Brief Overview

This note covers **Transaction Systems** and was created from a 21-page PDF presentation. It covers transaction definition, server architecture, concurrency control, and serializability analysis.

Key Points

- Definition of a transaction and its lifecycle states (Active, Committed, Aborted, etc.).
- Server architecture: server processes, lock manager, log writer, and checkpointing.
- Concurrency control concepts: serializability, conflict, and view equivalence.
- Locking mechanisms: **mutex vs semaphore** and the lock table.

📘 What Is a Transaction?

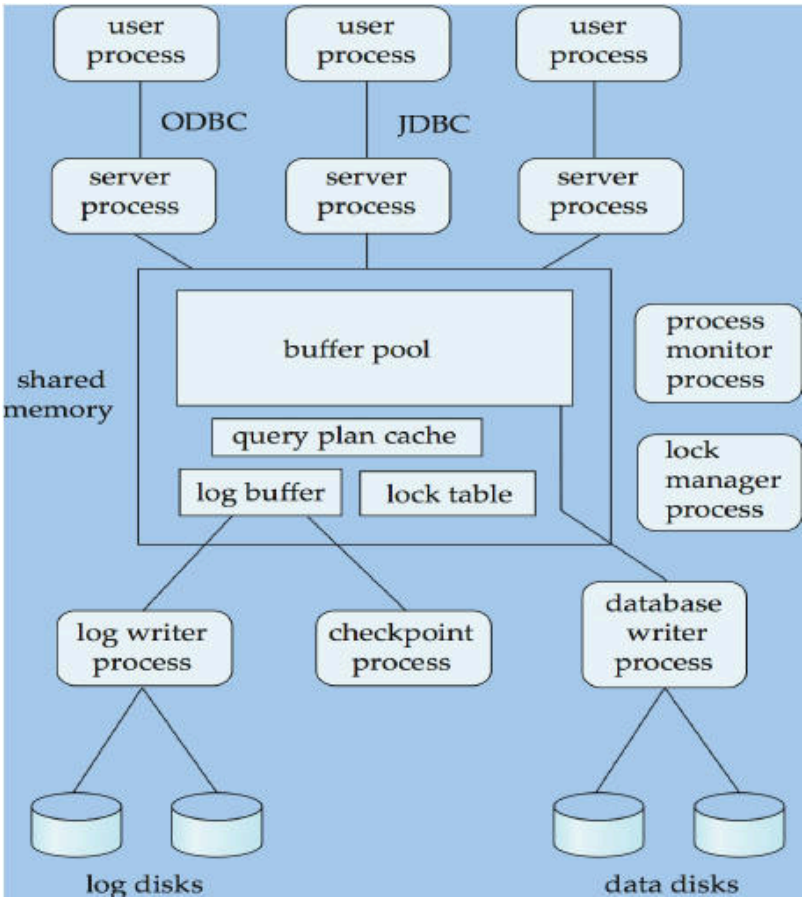
- **Definition**

> *A transaction is a unit of program execution that accesses and possibly updates various data items.*

- **Example: Transfer $50 from account A to account B**
    1. read(A) → A
    2. $A := A - 50$
    3. write(A)
    4. read(B) → B
    5. $B := B + 50$
    6. write(B)
- **Key Issues**
    - Failures (hardware crashes, system errors)
    - Concurrent execution of multiple transactions

🏗️ Basic Architecture of a Transaction Server

- **Server Processes** – receive user queries (transactions), execute them, and return results.
    - Often **multithreaded** so a single process can handle several queries concurrently.
- **Lock Manager Process** – coordinates lock acquisition; used for deadlock detection.
- **Database Writer Process** – continuously writes modified buffer blocks to disk.

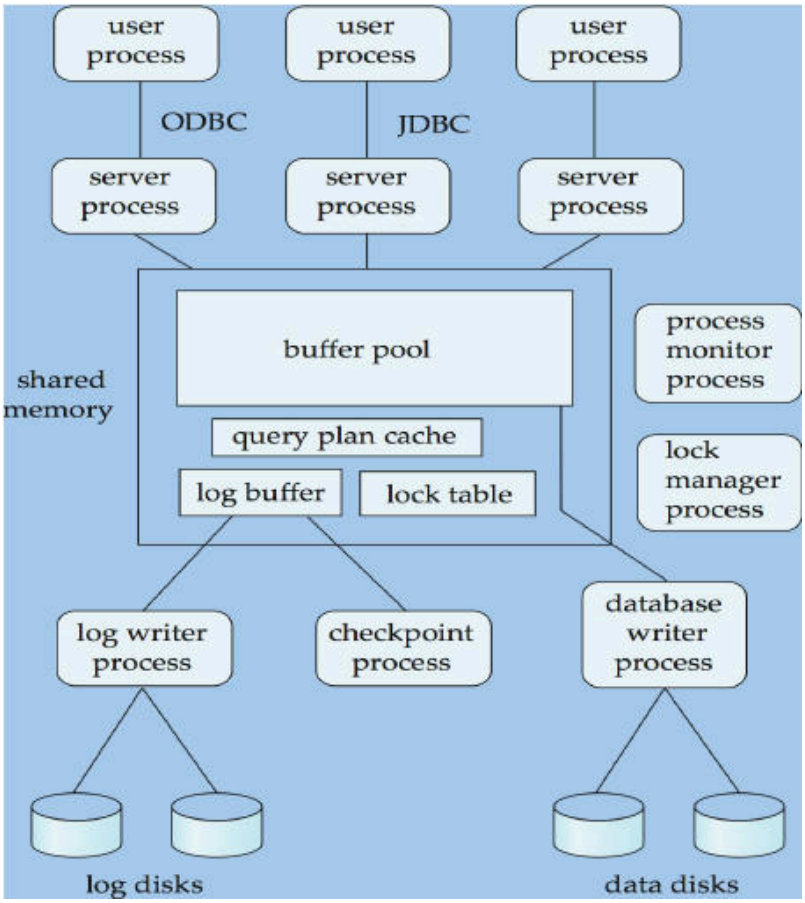


The diagram shows user processes communicating via ODBC/JDBC to server processes, which share a buffer pool containing a query-plan cache, log buffer, and lock table. Background processes (lock manager, log writer, checkpoint, etc.) interact with the buffer pool and storage devices.

Additional Server Components

Component	Role

Log Writer Process	Buffers log records and writes them to stable storage
Checkpoint Process	Performs periodic checkpoints to limit recovery time
Process Monitor	Detects failures, aborts/restarts affected transactions



Illustrates shared memory (buffer pool, lock table, log buffer) and background processes that maintain consistency and durability.

🔒 Transaction-Related Processes & Shared Memory

- **Shared Memory Structures**

    - **Buffer Pool** – caches data pages.
    - **Lock Table** – records current lock owners.
    - **Log Buffer** – holds log records before they are flushed.
    - **Cached Query Plans** – reused for identical queries.
- **Mutual Exclusion**
    - Implemented via **OS semaphores** or **atomic instructions** (e.g., test-and-set).
    - Each process manipulates the lock table directly to avoid IPC overhead; a separate lock manager handles deadlock detection.

🔐 Mutex vs. Semaphore

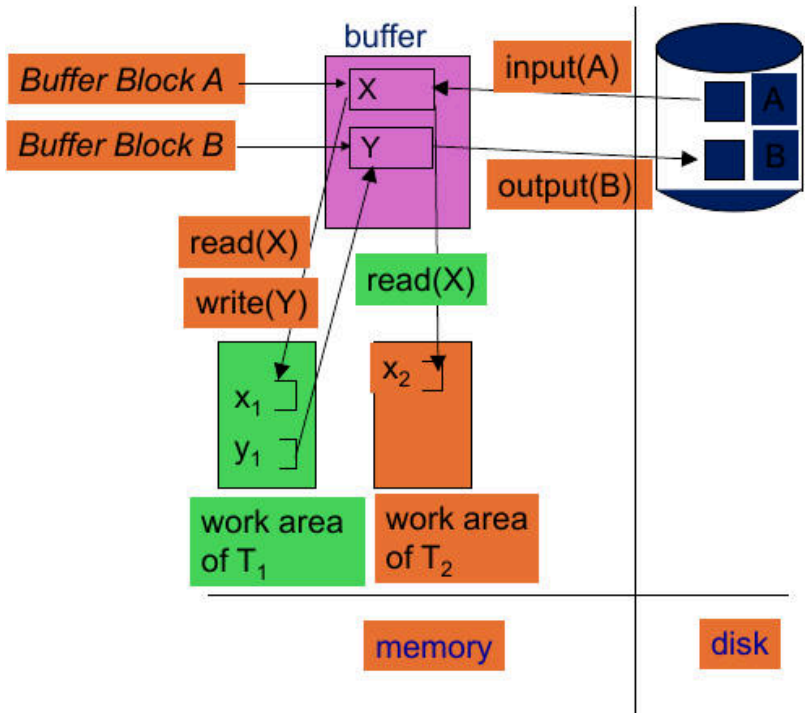
> *A mutex (binary semaphore) provides mutual exclusion for a single thread, while a semaphore allows a preset number of threads to enter a critical region.*

- **Mutex**
    - Only one thread may acquire it at a time.
    - Used for **mutual exclusion**.
- **Semaphore**
    - Allows up to *N* concurrent acquisitions.
    - Used for **flow control**.

Both must be **released** after the protected code region finishes.

📂 Transactions Have Their Own Work Areas

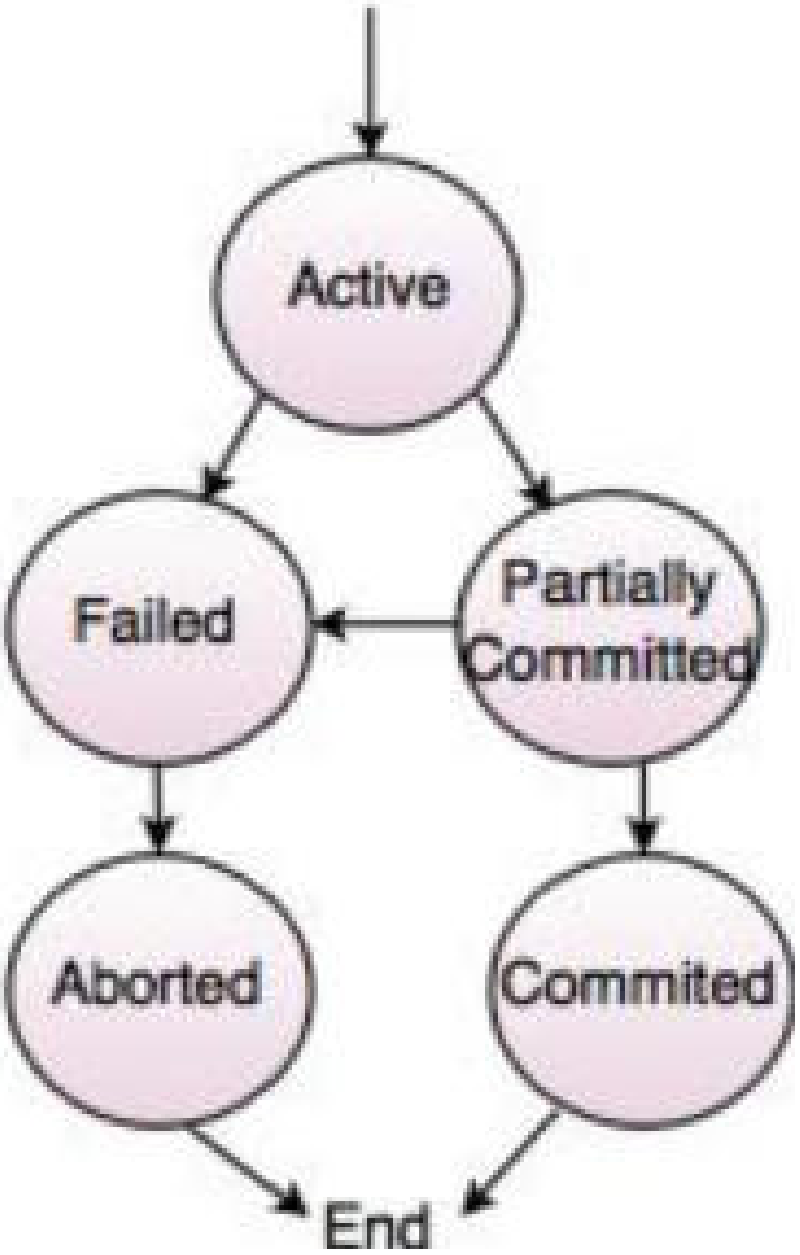
- Each transaction maintains a **private workspace** (e.g., x1, y1 for T1) while reading/writing through the shared buffer.



The figure shows two transactions (T1 in green, T2 in orange) interacting with a shared buffer (pink) and underlying disk storage. Each transaction reads from the buffer, updates its private workspace, and later writes back.

🔄 Transaction States

- **Active** – execution in progress.
- **Partially Committed** – final statement executed, but not yet durable.
- **Failed** – cannot continue normal execution.
- **Aborted** – rolled back to the state before the transaction started; may be restarted or killed.
- **Committed** – changes are permanent.



Flowchart visualizing transitions among the five transaction states.

✅ Requirements for Transactions (Why ACID Matters)

1. **Atomicity** – either all operations take effect or none do.
2. **Consistency** – integrity constraints (primary/foreign keys, sum of balances) must hold after commit.
3. **Isolation** – intermediate results are invisible to other concurrent transactions.

4. **Durability** – once committed, changes survive crashes.

- Example of inconsistency: if step 3 fails after debiting A but before crediting B, the total $A + B$ changes, violating the "sum unchanged" rule.
- Isolation can be trivially achieved by **serial execution**, but concurrency yields higher throughput.

🔁 Concurrent Executions & Serializability

- **Benefits**: higher CPU/disk utilization, reduced response time, increased throughput.
- **Serializability** – a schedule is **serializable** if its effect equals some **serial schedule** (transactions run one after another).

Schedule Types

Type	Description
Serial Schedule	Transactions execute consecutively, no interleaving.
Non-serial Schedule	Operations from different transactions interleave.
Serializable Schedule	Non-serial but yields the same final state as a serial schedule.
Non-serializable Schedule	No equivalent serial order; may cause anomalies.

Example Analysis

- **Given Interleaving** (A and B are data items):
  - Final A = $2A_0 + 200$ (matches T1→T2 order)
  - Final B = $2B_0 + 100$ (matches T2→T1 order)
  - Since the two results correspond to **different** serial orders, the schedule is **non-serializable**.

⚔️ Conflict Serializability

- **Conflict** occurs when two operations:
  1. Access the same data item, **and**

2. At least one is a **write**.

- **Rules**
  - Read-Read → not a conflict.
  - Read-Write or Write-Read → conflict; order cannot be swapped.
  - Write-Write → conflict but can be ordered arbitrarily (they don't read each other's value).
- Conflict serializability is a **sufficient** condition for overall serializability.

👁️ View Serializability

Two schedules are **view-equivalent** if they satisfy:

1. **Initial Read** – each data item is first read by the same transaction in both schedules.
2. **Final Write** – the transaction that performs the last write on each item is the same.
3. **Update Read** – any read of a value written by another transaction occurs after that write in both schedules.

📊 Example Schedules Comparison

Schedule	Operations (simplified)	Serializability
Schedule 1	T1: R(A), W(A); T2: R(A), W(A)	**Conflict-serializable** (non-conflicting reads can be reordered)
Schedule 2	T1: R(A), W(B); T2: R(B), W(A)	**Non-serializable** (conflicting writes interleaved)

🔄 A Serialization Scheme Example

1. **Step 1** – T1 updates X to 2000.
2. **Step 2** – Switch to T2; T2 updates X to 4000, then back to T1.
3. **Step 3** – T1 finishes; T2 updates Y to 4000 (by multiplying 2000).

Result: $X + Y = 4000$, same as the original total, thus the schedule is **equivalent to a serial schedule** and maintains consistency.

📈 Concurrency Control Overview

- **Goal**: Allow many users to perform operations simultaneously **without** compromising data integrity.
- **Advantages**
  - Improves system **throughput**.
  - Reduces **waiting time** for short transactions.
- **Risks Without Control** – lost updates, dirty reads, and other anomalies that break consistency.
