Database Fundamentals: Relational Algebra, PostgreSQL, and Normalization

Yehor KOROTENKO

May 5, 2025

Contents

Chapter 1

Introduction

Welcome, second-year computer science students, to the exciting world of databases! In this course, we'll delve into the fundamental concepts and practical applications of database management systems. Due to the current circumstances, this book will serve as your primary guide to understanding relational algebra, PostgreSQL, and database normalization. We'll explore these topics in detail, providing you with the knowledge and skills to design, implement, and manage effective and efficient databases.

This book is designed to be self-contained, with clear explanations, illustrative examples, and helpful diagrams. We'll start with the theoretical foundations of relational algebra, then move on to the practical implementation of databases using PostgreSQL, and finally, learn how to design robust and well-structured databases using normalization techniques.

Let's embark on this journey together!

Chapter 2

Relational Algebra

Relational algebra is a theoretical query language that provides a formal foundation for manipulating relations (tables) in a relational database. It consists of a set of operations that take one or more relations as input and produce a new relation as output. Understanding relational algebra is crucial for understanding how databases work internally and for optimizing database queries.

2.1 Core Relational Algebra Operations

The core operations of relational algebra are fundamental building blocks for constructing more complex queries. Let's examine each of them in detail:

2.1.1 Selection (σ)

The selection operation, denoted by σ, filters rows (tuples) from a relation based on a specified condition. It selects tuples that satisfy a given predicate.

Syntax: $\sigma_{condition}(R)$

where:

- σ is the selection operator.
- *condition* is a predicate (Boolean expression) that evaluates to true or false for each tuple.
- R is the relation (table) being filtered.

Example:

Consider a relation `Students` with the following attributes: `StudentID`, `Name`, `Major`, `GPA`.

To select all students majoring in Computer Science:

$\sigma_{Major=\text{'Computer Science'}}(Students)$

The result would be:

Conditions:

Conditions can involve comparison operators (e.g., =, <, >, <=, >=, !=) and logical operators (e.g., $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT)).

Example using AND ($\wedge$):

$\sigma_{Major=\text{'Computer Science'} \wedge GPA>3.8}(Students)$

Result:

StudentID	Name	Major	GPA
101	Alice	Computer Science	3.8
102	Bob	Mathematics	3.5
103	Charlie	Computer Science	3.9
104	David	Physics	3.2
105	Eve	Biology	3.7

Table 2.1: The `Students` Relation

StudentID	Name	Major	GPA
101	Alice	Computer Science	3.8
103	Charlie	Computer Science	3.9

Table 2.2: Result of selecting Computer Science students

2.1.2 Projection (π)

The projection operation, denoted by π, selects specific columns (attributes) from a relation. It creates a new relation containing only the specified attributes.

Syntax: $\pi_{attribute1,\ attribute2,\ ...,\ attributeN}(R)$
where:

- π is the projection operator.
- *attribute1, attribute2, ..., attributeN* are the attributes to be selected.
- R is the relation (table).

Example:
Using the `Students` relation from Table 2.1, to project only the `Name` and `Major` attributes:
$\pi_{Name,\ Major}(Students)$
The result would be:
Duplicate Elimination:
Projection automatically eliminates duplicate tuples in the result. This is a key characteristic of relational algebra; results are always sets, not multisets. If you want to preserve duplicates (which is uncommon in relational algebra), you'd need to use variations found in implementations, which we will discuss later within the context of SQL.

2.1.3 Union ($\cup$)

The union operation, denoted by $\cup$, combines the tuples from two relations, producing a new relation containing all tuples from both relations.

StudentID	Name	Major	GPA
103	Charlie	Computer Science	3.9

Table 2.3: Result of selecting CS students with GPA > 3.8

Name	Major
Alice	Computer Science
Bob	Mathematics
Charlie	Computer Science
David	Physics
Eve	Biology

Table 2.4: Result of projecting Name and Major from Students

Syntax: $R \cup S$
where:

- R and S are relations (tables).

Conditions:
For the union operation to be valid, the relations R and S must be *union-compatible*. This means:

1. They must have the same number of attributes.

2. The corresponding attributes must have compatible data types (domains).

Example:
Consider two relations: `Undergraduates` and `Graduates`. Both have the attributes `StudentID` and `Name`.

StudentID	Name
201	John
202	Jane
203	Mike

Table 2.5: The `Undergraduates` Relation

StudentID	Name
301	Sarah
302	David
202	Jane

Table 2.6: The `Graduates` Relation

Undergraduates $\cup$ *Graduates*
The result would be:
Note that the duplicate tuple (202, Jane) appears only once in the result due to the set semantics of relational algebra.

StudentID	Name
201	John
202	Jane
203	Mike
301	Sarah
302	David

Table 2.7: Result of *Undergraduates* ∪ *Graduates*

2.1.4 Set Difference (−

The set difference operation, denoted by −, returns tuples that are present in the first relation but not in the second relation.

Syntax: $R - S$

where:

- R and S are relations (tables).

Conditions:

Like the union operation, set difference requires the relations R and S to be union-compatible.

Example:

Using the `Undergraduates` (Table 2.5) and `Graduates` (Table 2.6) relations from the previous example:

Undergraduates − *Graduates*

The result would be:

StudentID	Name
201	John
203	Mike

Table 2.8: Result of *Undergraduates* − *Graduates*

This is because (202, Jane) is present in both `Undergraduates` and `Graduates`.

2.1.5 Cartesian Product (×)

The Cartesian product operation, denoted by ×, combines each tuple from the first relation with each tuple from the second relation. It creates a new relation containing all possible pairs of tuples from the input relations.

Syntax: $R \times S$

where:

- R and S are relations (tables).

Example:

Consider two relations: `Departments` and `Courses`.

Departments × *Courses*

The result would be:

DeptID	DeptName
1	Computer Science
2	Mathematics

Table 2.9: The `Departments` Relation

CourseID	CourseName
CS101	Intro to Programming
MA101	Calculus

Table 2.10: The `Courses` Relation (for Cartesian Product)

Notice how each row from `Departments` is paired with each row from `Courses`. The number of rows in the resulting table is the product of the number of rows in the input tables. In this case, 2 x 2 = 4.

Important Note: The Cartesian product is often used conceptually as the basis for joins (followed by a selection). Directly using the Cartesian product without a subsequent selection can often lead to very large and potentially meaningless results and is inefficient. It's usually best to use specific join operations.

2.1.6 Rename (ρ)

The rename operation, denoted by ρ, renames a relation or its attributes. It allows you to change the name of a table or column, which is especially useful for self-joins or when combining results.

Syntax:

- $\rho_{NewName}(R)$ (Renames the relation R to NewName)
- $\rho_{NewAttr1,\ NewAttr2,\ ...}(R)$ (Renames attributes of R in order)
- $\rho_{NewName(NewAttr1,\ NewAttr2,\ ...)}(R)$ (Renames relation R and its attributes)

where:

- ρ is the rename operator.
- *NewName* is the new name for the relation.
- *NewAttr1, NewAttr2, ...* are the new names for the attributes (must match the arity and be in order).

DeptID	DeptName	CourseID	CourseName
1	Computer Science	CS101	Intro to Programming
1	Computer Science	MA101	Calculus
2	Mathematics	CS101	Intro to Programming
2	Mathematics	MA101	Calculus

Table 2.11: Result of *Departments* $\times$ *Courses*

- R is the relation (table).

Example:
Using the `Students` relation from Table 2.1:
To rename the relation to `EnrolledStudents`: $\rho_{EnrolledStudents}(Students)$
To rename the attributes `StudentID` to `ID` and `Name` to `StudentName`: $\rho_{ID,\ StudentName,\ Major,\ GPA}(Stude$
To rename the relation to `EnrolledStudents` and attributes `StudentID` to `ID` and `Name` to `StudentName`: $\rho_{EnrolledStudents(ID,\ StudentName,\ Major,\ GPA)}(Students)$

2.2 Derived Relational Algebra Operations

Derived operations are operations that can be expressed in terms of the core operations. They are often included for convenience and to simplify query expressions.

2.2.1 Intersection ($\cap$)

The intersection operation, denoted by $\cap$, returns tuples that are present in both relations.
Syntax: $R \cap S$
where:

- R and S are union-compatible relations (tables).

Derivation using Core Operations:
$R \cap S = R - (R - S)$ (Also equivalent to $S - (S - R)$).
This means: The intersection of R and S is equal to R minus (R minus S). We're removing from R all those tuples that *aren't* in S, leaving us with only the tuples that *are* common to both R and S.
Example:
Using the `Undergraduates` (Table 2.5) and `Graduates` (Table 2.6) relations:
$Undergraduates \cap Graduates$
The result would be:

StudentID	Name
202	Jane

Table 2.12: Result of $Undergraduates \cap Graduates$

2.2.2 Join ($\bowtie$)

The join operation combines tuples from two relations based on a specified condition. There are several types of joins:

Theta Join ($\bowtie_\theta$)

The theta join, denoted by $\bowtie_\theta$, combines tuples from two relations that satisfy a given condition θ.
Syntax: $R \bowtie_\theta S$
where:

- R and S are relations (tables).
- θ is a condition (predicate) involving attributes from both R and S.

Derivation using Core Operations:
$R \bowtie_\theta S = \sigma_\theta(R \times S)$
This means: The theta join is equivalent to taking the Cartesian product of R and S and then selecting only the tuples that satisfy the condition θ.
Example:
Consider two relations: `Employees` and `Departments2`.

EmpID	EmpName	DeptID
1	Alice	10
2	Bob	20
3	Charlie	10

Table 2.13: The `Employees` Relation (for Join)

DeptID	DeptName	Location
10	Sales	New York
20	Marketing	London
30	Engineering	San Francisco

Table 2.14: The `Departments2` Relation (for Join)

To join `Employees` and `Departments2` where `Employees.DeptID` = `Departments2.DeptID`:
$Employees \bowtie_{Employees.DeptID=Departments2.DeptID} Departments2$
The result would be:

EmpID	EmpName	Employees.DeptID	Departments2.DeptID	DeptName	Location
1	Alice	10	10	Sales	New York
2	Bob	20	20	Marketing	London
3	Charlie	10	10	Sales	New York

Table 2.15: Result of Theta Join on `DeptID`

Note: It's often useful to use rename (ρ) to avoid duplicate attribute names like 'DeptID' if not needed.

Equijoin

An equijoin is a special case of the theta join where the condition θ only involves equality comparisons (=). The previous example is an equijoin.

Natural Join ($\bowtie$)

The natural join, denoted by $\bowtie$, is a special case of the equijoin. It automatically joins on all attributes that have the same name in both relations and eliminates duplicate attributes from the result.

Syntax: $R \bowtie S$
where:

- R and S are relations (tables) with one or more common attribute names.

Example:
Using the `Employees` (Table 2.13) and `Departments2` (Table 2.14) relations:
Employees $\bowtie$ *Departments2*
The common attribute is `DeptID`. The result would be:

EmpID	EmpName	DeptID	DeptName	Location
1	Alice	10	Sales	New York
2	Bob	20	Marketing	London
3	Charlie	10	Sales	New York

Table 2.16: Result of Natural Join on `DeptID`

Notice that only one `DeptID` column remains.

Outer Joins

Outer joins extend the results of an inner join (like Theta or Natural Join) by preserving tuples from one or both relations that do not have matching tuples in the other relation, filling in missing attributes with null values. Relational algebra doesn't have standard widely-adopted symbols for outer joins, so they are often described textually or defined using extensions. SQL provides explicit syntax for them.

Left Outer Join: Preserves all tuples from the left relation (R). If a tuple in R has no match in S based on the join condition, it still appears in the result, with attributes from S set to NULL.

Right Outer Join: Preserves all tuples from the right relation (S). If a tuple in S has no match in R, it still appears in the result, with attributes from R set to NULL.

Full Outer Join: Preserves all tuples from both relations (R and S). Tuples without matches in the other relation appear with NULLs for the attributes from that other relation.

Example:
Consider the `Employees` and `Departments2` relations again, but let's add a department without employees and assume we want to join on `DeptID`.

Left Outer Join Result (Conceptual): (`Employees` LEFT OUTER JOIN `Departments2` ON `Employees.DeptID` = `Departments2.DeptID`)

(This output preserves all rows from `Employees` *and finds matching* `Departments2`*. If there were an employee with a DeptID not in Departments2, they would appear here with NULLs for DeptName and Location.)*

Right Outer Join Result (Conceptual): (`Employees` RIGHT OUTER JOIN `Departments2` ON `Employees.DeptID` = `Departments2.DeptID`)

EmpID	EmpName	DeptID
1	Alice	10
2	Bob	20
3	Charlie	10

Table 2.17: The `Employees` Relation (for Outer Join)

DeptID	DeptName	Location
10	Sales	New York
20	Marketing	London
30	Engineering	San Francisco
40	HR	Chicago

Table 2.18: The `Departments2` Relation (with extra Dept for Outer Join)

EmpID	EmpName	Employees.DeptID	Departments2.DeptID	DeptName	Location
1	Alice	10	10	Sales	New York
2	Bob	20	20	Marketing	London
3	Charlie	10	10	Sales	New York

Table 2.19: Result of Left Outer Join (Employees LEFT JOIN Departments2)

EmpID	EmpName	Employees.DeptID	Departments2.DeptID	DeptName	Location
1	Alice	10	10	Sales	New York
2	Bob	20	20	Marketing	London
3	Charlie	10	10	Sales	New York
NULL	NULL	NULL	30	Engineering	San Francisco
NULL	NULL	NULL	40	HR	Chicago

Table 2.20: Result of Right Outer Join (Employees RIGHT JOIN Departments2)

(This output preserves all rows from `Departments2`*. Departments 30 and 40 had no matching employees, so employee attributes are NULL.)*

Full Outer Join Result (Conceptual): (`Employees` FULL OUTER JOIN `Departments2` ON `Employees.DeptID` = `Departments2.DeptID`)

EmpID	EmpName	Employees.DeptID	Departments2.DeptID	DeptName	Location
1	Alice	10	10	Sales	New York
2	Bob	20	20	Marketing	London
3	Charlie	10	10	Sales	New York
NULL	NULL	NULL	30	Engineering	San Francisco
NULL	NULL	NULL	40	HR	Chicago

Table 2.21: Result of Full Outer Join (Employees FULL JOIN Departments2)

(This output preserves all rows from both tables. Unmatched rows from either side appear with NULLs for the attributes of the other side.)

2.2.3 Division (÷)

The division operation, denoted by ÷, is used for queries that involve the concept of "for all". It returns tuples from one relation that are associated with *all* tuples in another relation. It's often used to answer queries like "Find all students who have taken *all* required courses."

Syntax: $R(A, B) \div S(B)$
where:

- R is a relation with attributes A and B (where A and B can represent sets of attributes).
- S is a relation with attributes B.
- The attributes B in S must be a subset of the attributes in R.
- The result relation has attributes A (i.e., attributes in R but not in S).

Derivation using Core Operations:
$R \div S = \pi_A(R) - \pi_A((\pi_A(R) \times S) - R)$
Let's break this down:

1. $\pi_A(R)$: Find all possible values (or combinations) for the attributes A present in R.
2. $\pi_A(R) \times S$: Create all possible combinations of A values from step 1 with the B values from S. This represents all the pairs (a, b) that *should* exist if a is associated with every b.
3. $(\pi_A(R) \times S) - R$: Find the combinations from step 2 that are *not* present in the original relation R. These are the "missing" associations.
4. $\pi_A((\pi_A(R) \times S) - R)$: Project onto A to find which A values have at least one missing association from S.

5. $\pi_A(R) - \pi_A(\dots)$: Subtract the A values with missing associations (step 4) from the set of all possible A values (step 1). The result is the set of A values that are associated with *all* B values in S.

Example:
Consider two relations: `Enrolled` and `RequiredCourses`.

StudentID	CourseID
101	CS101
101	CS201
102	CS101
102	CS201
102	MA101
103	CS101

Table 2.22: The `Enrolled` Relation

CourseID
CS101
CS201

Table 2.23: The `RequiredCourses` Relation

$Enrolled(StudentID, CourseID) \div RequiredCourses(CourseID)$
This operation finds all students (StudentID) who are enrolled in *all* the courses listed in `RequiredCourses`.
The result would be:

StudentID
101
102

Table 2.24: Result of $Enrolled \div RequiredCourses$

Student 101 and 102 are enrolled in both CS101 and CS201. Student 103 is only enrolled in CS101, so they are not included in the result.

2.3 Complex Queries with Relational Algebra

Relational algebra operations can be combined sequentially or nested to express complex queries.

Example 1:
Find the names of all students who are majoring in Computer Science and have a GPA greater than 3.7.
Using the `Students` relation from Table 2.1:
$\pi_{Name}(\sigma_{Major=\text{'Computer Science'} \wedge GPA>3.7}(Students))$

This query first selects the students who meet the specified criteria (Computer Science major and GPA > 3.7) using σ, and then projects only their names using π.

The result would be:

Name
Charlie

Table 2.25: Result of finding names of top CS students

Example 2:

Find the names of all employees who work in the 'Sales' department.

Using the `Employees` relation (Table 2.13) and a simplified `Departments` relation (different from `Departments2` used earlier).

EmpID	EmpName	DeptID
1	Alice	10
2	Bob	20
3	Charlie	10

Table 2.26: The `Employees` Relation (for Complex Query)

DeptID	DeptName
10	Sales
20	Marketing

Table 2.27: Simplified `Departments` Relation (for Complex Query)

$\pi_{EmpName}(Employees \bowtie (\sigma_{DeptName='Sales'}(Departments)))$

This query first selects the 'Sales' department row(s) from the `Departments` relation (σ). Then, it performs a natural join ($\bowtie$) between the `Employees` relation and the result of the selection (which will match on the common attribute `DeptID`). Finally, it projects (π) the `EmpName` from the result of the join.

The result would be:

2.4 Limitations of Relational Algebra

While relational algebra provides a powerful formal foundation for database queries, it has some limitations in its pure form:

- **Lack of Aggregation:** Relational algebra does not include built-in operators for aggregate functions such as `SUM`, `AVG`, `MIN`, `MAX`, and `COUNT`. These operations require extensions or are handled directly in query languages like SQL.
- **No Ordering:** Relational algebra operates on sets (or bags in some variations), which are inherently unordered. It does not define or guarantee any specific order for the tuples in a result relation. Ordering is typically specified externally (e.g., using `ORDER BY` in SQL).

EmpName
Alice
Charlie

Table 2.28: Result of finding names of Sales employees

- **No Computational Capabilities:** Relational algebra primarily focuses on set-oriented data retrieval and manipulation based on existing values. It lacks general computational capabilities beyond basic comparisons used in selection and join conditions.
- **Limited Expressiveness for Certain Queries:** Some types of queries, particularly recursive queries (e.g., finding all subordinates of a manager in an organizational hierarchy), are difficult or impossible to express using only standard relational algebra operations. SQL provides extensions like recursive CTEs for such cases.

Despite these limitations, relational algebra remains fundamental for understanding query processing, optimization, and the logical structure of relational database operations.

Chapter 3

PostgreSQL

PostgreSQL is a powerful, open-source object-relational database system (ORDBMS). It has earned a strong reputation for reliability, feature robustness, data integrity, and adherence to SQL standards. PostgreSQL supports a wide range of features, including complex queries, foreign keys, triggers, views, transactional integrity, multi-version concurrency control (MVCC), advanced data types, sophisticated locking mechanisms, and support for stored procedures and functions in various languages. It is a popular choice for applications ranging from small projects to large, mission-critical systems. We will cover the fundamentals of using PostgreSQL through its implementation of SQL.

3.1 Installation and Setup

Detailed, platform-specific instructions for installing PostgreSQL can be found on the official PostgreSQL website: https://www.postgresql.org/download/. Installation typically involves downloading the appropriate installer or package for your operating system (Windows, macOS, Linux variants) and following the setup prompts. During installation, you will usually be asked to set a password for the default superuser, typically named `postgres`, and configure network settings (like the port, usually 5432) and locale settings.

3.1.1 Connecting to PostgreSQL

Once PostgreSQL is installed, running, and configured, you can connect to it using various client tools. Some common options include:

- **psql:** The interactive command-line terminal for PostgreSQL. It's a powerful and versatile tool included with the standard distribution, suitable for scripting and direct interaction.
- **pgAdmin:** A popular open-source graphical administration and development tool for PostgreSQL. It provides a user-friendly interface for managing databases, schemas, tables, users, permissions, and executing queries.
- **Third-party GUI tools:** Many other graphical database clients support PostgreSQL, such as DBeaver (universal), DataGrip (commercial, by JetBrains), Azure Data Studio (with PostgreSQL extension), etc.

- **Programming Language Connectors/Drivers:** Libraries for languages like Python (psycopg2, asyncpg), Java (JDBC driver), Node.js (node-postgres), etc., allow applications to connect and interact with the database.

For this book, we will primarily use `psql` to demonstrate SQL commands.

To connect to a locally running PostgreSQL server using `psql` from your terminal or command prompt, you might use a command like this:

```
# Connect as user 'postgres' to the database 'postgres' on
    localhost
psql -U postgres -d postgres -h localhost
```

Listing 3.1: Connecting via psql

This command attempts to connect to the PostgreSQL server running on `localhost` as the user `postgres` and targets the default database also named `postgres`. You will likely be prompted for the password you set for the `postgres` user during installation. If connecting to a different database or as a different user, adjust the `-d` and `-U` options accordingly.

Inside `psql`, you can use meta-commands (starting with a backslash \) for database operations. For example, to list databases, use `\l`; to list tables in the current database, use `\dt`; to connect to a different database, use `\c dbname`. Use `\q` to quit `psql`.

3.2 Basic SQL Commands

SQL (Structured Query Language) is the standard language for defining, manipulating, and querying data in relational databases. PostgreSQL implements a large part of the SQL standard along with its own extensions.

3.2.1 Creating a Database

To create a new database within the PostgreSQL instance, connect first (e.g., to the default `postgres` database) and then use the `CREATE DATABASE` command:

```
CREATE DATABASE mydatabase;
```

Listing 3.2: Creating a Database

This command creates a new, empty database named `mydatabase`. Database names should follow identifier rules (typically starting with a letter or underscore, followed by letters, numbers, or underscores).

After creating the database, you can connect directly to it using `psql`'s `\c` command:

```
\c mydatabase
```

3.2.2 Creating a Table

Once connected to the desired database, use the `CREATE TABLE` command to define the structure of a table:

```sql
CREATE TABLE Students (
    StudentID SERIAL PRIMARY KEY,
    Name VARCHAR(255) NOT NULL,
    Major VARCHAR(255),
    GPA DECIMAL(3, 2), -- Precision 3, Scale 2 (e.g., 9.99)
    EnrollmentDate DATE DEFAULT CURRENT_DATE
);
```

Listing 3.3: Creating the Students Table

This command creates a table named `Students` with the following columns:

- `StudentID`: An integer column that automatically increments for each new row (using the `SERIAL` pseudo-type, which creates a sequence). It's also designated as the `PRIMARY KEY`, meaning values must be unique and not null, serving to uniquely identify each row.
- `Name`: A variable-length string column with a maximum length of 255 characters. The `NOT NULL` constraint ensures this column must have a value.
- `Major`: A variable-length string column, allowing NULL values.
- `GPA`: A decimal number column suitable for exact fractional values. `DECIMAL(3,2)` allows up to 3 total digits, with 2 digits after the decimal point.
- `EnrollmentDate`: A date column. The `DEFAULT CURRENT_DATE` clause automatically sets the value to the current date if no value is provided during insertion.

Common PostgreSQL Data Types:
PostgreSQL supports a rich set of data types, including:

- Numeric Types: `INTEGER` (or `INT`), `SMALLINT`, `BIGINT`, `DECIMAL` (or `NUMERIC`), `REAL`, `DOUBLE PRECISION`, `SERIAL`, `BIGSERIAL`.
- Character Types: `VARCHAR(n)`, `CHAR(n)`, `TEXT`.
- Date/Time Types: `DATE`, `TIME`, `TIMESTAMP` (with or without time zone), `INTERVAL`.
- Boolean Type: `BOOLEAN` (can store TRUE, FALSE, or NULL).
- Geometric Types: `POINT`, `LINE`, `POLYGON`, etc.
- Network Address Types: `INET`, `CIDR`, `MACADDR`.
- UUID Type: `UUID`.
- JSON Types: `JSON`, `JSONB` (binary, indexed format).
- Array Types: Any data type can have an array version (e.g., `INTEGER[]`, `TEXT[]`).

3.2.3 Inserting Data

To add rows (tuples) into a table, use the `INSERT INTO` command:

```
-- Specify columns and corresponding values
INSERT INTO Students (Name, Major, GPA) VALUES ('Alice', 'Computer
    Science', 3.8);

-- Can insert multiple rows at once
INSERT INTO Students (Name, Major, GPA, EnrollmentDate) VALUES
    ('Bob', 'Mathematics', 3.5, '2023-09-01'),
    ('Charlie', 'Computer Science', 3.9, '2022-09-05');

-- If providing values for all columns in order, column list can
   be omitted
-- (though explicit is usually better). StudentID is SERIAL, so it
   's auto-generated.
-- We provide NULL for GPA for the last student.
INSERT INTO Students VALUES (DEFAULT, 'David', 'Physics', NULL, '
   2023-01-15');
```

Listing 3.4: Inserting data into Students

These commands insert four rows into the `Students` table. The `StudentID` is automatically generated for each row due to the `SERIAL` type. The `EnrollmentDate` for Alice will use the default (current date at time of insert).

3.2.4 Selecting Data

To retrieve data from a table, use the `SELECT` command:

```
-- Select all columns (*) from all rows
SELECT * FROM Students;

-- Select specific columns
SELECT Name, Major FROM Students;

-- Filter rows using a WHERE clause
SELECT StudentID, Name, GPA FROM Students WHERE Major = 'Computer
   Science';

-- Filter with multiple conditions
SELECT Name, GPA FROM Students WHERE Major = 'Computer Science'
   AND GPA > 3.8;

-- Order the results using ORDER BY (ASC is default, DESC for
   descending)
SELECT Name, GPA FROM Students ORDER BY GPA DESC;

-- Limit the number of rows returned
SELECT Name, GPA FROM Students ORDER BY GPA DESC LIMIT 2;
```

Listing 3.5: Selecting data from Students

3.2.5 Updating Data

To modify existing data in a table, use the `UPDATE` command with a `WHERE` clause to specify which rows to change:

```
-- Update Bob's GPA (assuming his StudentID is 2, check first!)
UPDATE Students
SET GPA = 3.6, Major = 'Applied Mathematics'
WHERE Name = 'Bob'; -- It's safer to use the Primary Key if known

-- Example using StudentID (let's assume Alice's ID is 1)
UPDATE Students
SET GPA = 3.85
WHERE StudentID = 1;

-- Update GPA for all Math majors
UPDATE Students
SET GPA = GPA * 1.05 -- Give a 5% GPA boost (example calculation)
WHERE Major = 'Mathematics';
```

Listing 3.6: Updating data in Students

Caution: Omitting the `WHERE` clause in an `UPDATE` statement will modify *all* rows in the table!

3.2.6 Deleting Data

To remove rows from a table, use the `DELETE FROM` command, typically with a `WHERE` clause:

```
-- Delete the student named David
DELETE FROM Students
WHERE Name = 'David';

-- Delete all students with NULL GPA
DELETE FROM Students
WHERE GPA IS NULL;
```

Listing 3.7: Deleting data from Students

Caution: Omitting the `WHERE` clause in a `DELETE` statement will remove *all* rows from the table!

3.2.7 Dropping Tables and Databases

To permanently remove a table and all its data:

```
DROP TABLE Students;
```

Listing 3.8: Dropping a Table

To permanently remove an entire database and all its contents (tables, views, functions, etc.):

```
-- Must be connected to a DIFFERENT database (e.g., postgres)
-- Ensure no active connections to 'mydatabase' exist
DROP DATABASE mydatabase;
```

Listing 3.9: Dropping a Database

Extreme Caution: `DROP TABLE` and `DROP DATABASE` are irreversible operations. Use them with extreme care, especially in production environments. Ensure you have backups.

3.3 Advanced SQL Concepts

Now that we've covered the basics (often called CRUD: Create, Read, Update, Delete), let's explore some more advanced SQL concepts frequently used in PostgreSQL.

3.3.1 Joins

Joins are fundamental for combining data from multiple related tables based on common columns (usually foreign keys linking to primary keys).

Let's set up example tables: `Students` and `Enrollments`.

```
CREATE TABLE Students (
    StudentID SERIAL PRIMARY KEY,
    Name VARCHAR(255) NOT NULL,
    Major VARCHAR(255)
);

CREATE TABLE Courses (
    CourseID VARCHAR(10) PRIMARY KEY, -- e.g., 'CS101'
    CourseName VARCHAR(255) NOT NULL,
    Credits INTEGER
);

CREATE TABLE Enrollments (
    EnrollmentID SERIAL PRIMARY KEY,
    StudentID INTEGER REFERENCES Students(StudentID), -- Foreign Key
    CourseID VARCHAR(10) REFERENCES Courses(CourseID), -- Foreign Key
    Grade CHAR(1) -- e.g., 'A', 'B'
);

-- Sample Data
INSERT INTO Students (Name, Major) VALUES
    ('Alice', 'Computer Science'), ('Bob', 'Mathematics'), ('Charlie', 'Physics');
-- StudentIDs will likely be 1, 2, 3

INSERT INTO Courses (CourseID, CourseName, Credits) VALUES
    ('CS101', 'Intro to Programming', 3),
    ('MA101', 'Calculus I', 4),
    ('PH101', 'Physics I', 4);
```

```
29
30 INSERT INTO Enrollments (StudentID, CourseID, Grade) VALUES
31     (1, 'CS101', 'A'), -- Alice takes CS101
32     (1, 'MA101', 'B'), -- Alice takes MA101
33     (2, 'MA101', 'A'), -- Bob takes MA101
34     (1, 'PH101', NULL); -- Alice takes PH101, grade pending
35 -- Charlie is not enrolled in anything yet
```

Listing 3.10: Setup for Join Examples

Inner Join: Returns only rows where the join condition is met in *both* tables.

```
1 -- Get student names and the names of courses they are enrolled in
2 SELECT s.Name, c.CourseName, e.Grade
3 FROM Students s
4 INNER JOIN Enrollments e ON s.StudentID = e.StudentID
5 INNER JOIN Courses c ON e.CourseID = c.CourseID;
```

Listing 3.11: Inner Join Example

(This will show Alice's and Bob's enrollments. Charlie won't appear as he has no entries in Enrollments. Courses not taken won't appear.)

Left Outer Join (or LEFT JOIN): Returns all rows from the *left* table (the one listed first, `Students` here) and matching rows from the *right* table (`Enrollments`). If a row in the left table has no match in the right table, the columns from the right table will be NULL.

```
1 -- List all students and the courses they are enrolled in,
     including students not enrolled in any course
2 SELECT s.Name, c.CourseName, e.Grade
3 FROM Students s
4 LEFT JOIN Enrollments e ON s.StudentID = e.StudentID
5 LEFT JOIN Courses c ON e.CourseID = c.CourseID;
```

Listing 3.12: Left Outer Join Example

(This will show Alice's and Bob's enrollments, and *it will show Charlie with NULLs for CourseName and Grade, because he exists in Students but has no matching Enrollments.)*

Right Outer Join (or RIGHT JOIN): Returns all rows from the *right* table and matching rows from the *left*. If a row in the right table has no match, columns from the left table will be NULL. (Less common than LEFT JOIN, often can be rewritten as a LEFT JOIN by swapping table order).

Full Outer Join (or FULL JOIN): Returns all rows from *both* tables. If a row in one table has no match in the other, the columns from the unmatched table will be NULL.

```
1 -- Show all students and all their enrollments,
2 -- and also show students with no enrollments,
3 -- and also show enrollments potentially referencing non-existent
     students (if possible)
4 SELECT s.Name, c.CourseName
5 FROM Students s
6 FULL OUTER JOIN Enrollments e ON s.StudentID = e.StudentID
7 FULL OUTER JOIN Courses c ON e.CourseID = c.CourseID;
```

Listing 3.13: Full Outer Join Example

(This is useful for finding orphans or seeing the complete picture from both sides.)

3.3.2 Subqueries

A subquery (or inner query) is a query nested inside another SQL query (the outer query). Subqueries can appear in various clauses:

Subquery in `WHERE` clause: Often used with operators like `IN`, `NOT IN`, `EXISTS`, `NOT EXISTS`, or comparison operators (=, >, <, etc.) when the subquery returns a single value.

```
-- Find students enrolled in 'Calculus I'
SELECT Name
FROM Students
WHERE StudentID IN (SELECT StudentID FROM Enrollments WHERE
   CourseID = 'MA101');

-- Find courses that have at least one enrollment
SELECT CourseName
FROM Courses
WHERE EXISTS (SELECT 1 FROM Enrollments WHERE Enrollments.CourseID
    = Courses.CourseID);
```

Listing 3.14: Subquery in WHERE Clause

Subquery in `SELECT` clause (Scalar Subquery): Must return a single value (one row, one column). Often correlated with the outer query.

```
-- Show each student's name and the count of courses they are
   enrolled in
SELECT
    s.Name,
    (SELECT COUNT(*) FROM Enrollments e WHERE e.StudentID = s.
       StudentID) AS CourseCount
FROM Students s;
```

Listing 3.15: Subquery in SELECT Clause

(Note: Correlated subqueries in the SELECT list can sometimes be inefficient compared to joins or window functions.)

Subquery in `FROM` clause (Derived Table): The result of the subquery acts as a temporary table that the outer query can select from. Must be given an alias.

```
-- Calculate the average grade value (assuming A=4, B=3 etc.) for
   CS101
SELECT AVG(GradeValue)
FROM (
    SELECT
        CASE Grade
            WHEN 'A' THEN 4.0
            WHEN 'B' THEN 3.0
            WHEN 'C' THEN 2.0
            WHEN 'D' THEN 1.0
```

```
            ELSE 0.0
        END AS GradeValue
    FROM Enrollments
    WHERE CourseID = 'CS101' AND Grade IS NOT NULL
) AS GradesForCS101; -- Alias is required
```

Listing 3.16: Subquery in FROM Clause

3.3.3 Aggregate Functions

Aggregate functions perform calculations on a set of rows and return a single summary value. They are often used with the `GROUP BY` clause.

Common aggregate functions:

- `COUNT(*)` or `COUNT(column)`: Counts rows or non-NULL values.
- `SUM(column)`: Calculates the sum of values.
- `AVG(column)`: Calculates the average of values.
- `MIN(column)`: Finds the minimum value.
- `MAX(column)`: Finds the maximum value.
- `STRING_AGG(column, delimiter)`: Concatenates strings.
- `ARRAY_AGG(column)`: Aggregates values into an array.

```
-- Count the total number of students
SELECT COUNT(*) FROM Students;

-- Calculate the average credits for all courses
SELECT AVG(Credits) FROM Courses;

-- Find the highest grade achieved in MA101
SELECT MAX(Grade) FROM Enrollments WHERE CourseID = 'MA101';

-- Count distinct majors
SELECT COUNT(DISTINCT Major) FROM Students;
```

Listing 3.17: Aggregate Function Examples

3.3.4 GROUP BY Clause

The `GROUP BY` clause groups rows that have the same values in one or more specified columns into a summary row. Aggregate functions are then applied to each group independently.

```
-- Calculate the number of students enrolled in each course
SELECT CourseID, COUNT(*) AS NumberOfStudents
FROM Enrollments
GROUP BY CourseID;
```

```
5
6  -- Calculate the average grade per course (more complex, requires
       grade mapping)
7  -- Let's find the number of 'A' grades per course instead
8  SELECT CourseID, COUNT(*) AS A_GradeCount
9  FROM Enrollments
10 WHERE Grade = 'A'
11 GROUP BY CourseID;
12
13 -- Find the number of students in each major
14 SELECT Major, COUNT(*) AS StudentCount
15 FROM Students
16 GROUP BY Major;
```

Listing 3.18: GROUP BY Example

Important Rule: When using `GROUP BY`, any column in the `SELECT` list that is *not* an aggregate function must be included in the `GROUP BY` clause.

3.3.5 HAVING Clause

The `HAVING` clause filters the results *after* the `GROUP BY` clause has been applied and aggregate functions have been computed. It's like a `WHERE` clause for groups.

```
1  -- Find majors with more than 5 students
2  SELECT Major, COUNT(*) AS StudentCount
3  FROM Students
4  GROUP BY Major
5  HAVING COUNT(*) > 5; -- Filter based on the aggregated count
6
7  -- Find courses with an average grade better than 'B' (conceptual)
8  -- Let's find courses with more than 1 enrollment
9  SELECT CourseID, COUNT(*) AS EnrollmentCount
10 FROM Enrollments
11 GROUP BY CourseID
12 HAVING COUNT(*) > 1;
```

Listing 3.19: HAVING Clause Example

3.3.6 Common Table Expressions (CTEs)

A CTE (Common Table Expression), defined using the `WITH` clause, creates a temporary, named result set that you can reference within a single SQL statement (SELECT, INSERT, UPDATE, DELETE). CTEs improve readability and modularity, especially for complex queries, and are essential for recursive queries.

```
1  -- Find students enrolled in courses with 4 credits
2  WITH HighCreditCourses AS (
3    SELECT CourseID
4    FROM Courses
5    WHERE Credits = 4
6  ),
```

```sql
StudentEnrollmentsInHighCreditCourses AS (
  SELECT DISTINCT e.StudentID -- Use DISTINCT if student could
     enroll multiple times
  FROM Enrollments e
  JOIN HighCreditCourses hcc ON e.CourseID = hcc.CourseID
)
SELECT s.Name
FROM Students s
JOIN StudentEnrollmentsInHighCreditCourses se ON s.StudentID = se.
   StudentID;
```

Listing 3.20: CTE Example

This query finds courses with 4 credits (first CTE), then finds students enrolled in those courses (second CTE), and finally selects the names of those students.

3.3.7 Window Functions

Window functions perform calculations across a set of table rows that are somehow related to the current row. Unlike aggregate functions used with `GROUP BY`, window functions do not collapse rows; they return a value for *each* row based on a "window" of related rows defined by the `OVER()` clause.

The `OVER()` clause specifies:

- `PARTITION BY column(s)`: Divides rows into partitions (groups). The window function is applied independently to each partition. (Optional)
- `ORDER BY column(s)`: Defines the order of rows within each partition, which is crucial for ranking and row-comparison functions. (Optional, but often required)
- `ROWS` or `RANGE` frame clause: Specifies the subset of rows within the partition to include in the window relative to the current row (e.g., `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`). (Optional)

Common window functions:

- Ranking: `ROW_NUMBER()`, `RANK()`, `DENSE_RANK()`, `NTILE(n)`
- Aggregate as Window: `SUM() OVER (...)`, `AVG() OVER (...)`, `COUNT() OVER (...)`, etc.
- Value Comparison: `LAG(col, offset, default)`, `LEAD(col, offset, default)`, `FIRST_VALUE(col)`, `LAST_VALUE(col)`

```sql
-- Rank students within each major based on GPA (assuming we add
   GPA to Students)
-- First, let's add GPA to our existing Students table for this
   example
ALTER TABLE Students ADD COLUMN GPA DECIMAL(3, 2);
UPDATE Students SET GPA = 3.8 WHERE StudentID = 1;
UPDATE Students SET GPA = 3.5 WHERE StudentID = 2;
-- Let's assume Charlie's ID is 3
UPDATE Students SET GPA = 3.9 WHERE StudentID = 3;
```

```sql
-- Let's add another CS student
INSERT INTO Students (Name, Major, GPA) VALUES ('Diane', 'Computer Science', 3.8);

-- Now the query
SELECT
    Name,
    Major,
    GPA,
    RANK() OVER (PARTITION BY Major ORDER BY GPA DESC NULLS LAST) AS RankInMajor,
    DENSE_RANK() OVER (PARTITION BY Major ORDER BY GPA DESC NULLS LAST) AS DenseRankInMajor,
    ROW_NUMBER() OVER (PARTITION BY Major ORDER BY GPA DESC NULLS LAST) AS RowNumInMajor
FROM Students;
```

Listing 3.21: Window Function Example (Ranking)

- `RANK()` gives the same rank for ties, skips next rank.
- `DENSE_RANK()` gives the same rank for ties, does *not* skip next rank.
- `ROW_NUMBER()` gives unique numbers even for ties.
- `PARTITION BY Major` calculates ranks independently for each major.
- `ORDER BY GPA DESC` determines the ranking criteria (highest GPA first). `NULLS LAST` handles potential NULL GPAs.

```sql
-- Show each student's GPA and the average GPA for their major
SELECT
    Name,
    Major,
    GPA,
    AVG(GPA) OVER (PARTITION BY Major) AS AvgMajorGPA
FROM Students;
```

Listing 3.22: Window Function Example (Aggregation)

(This calculates the average GPA for each major (`PARTITION BY Major`) and displays it alongside each student's individual GPA without collapsing rows.)

3.4 Transactions

A transaction is a sequence of one or more SQL operations executed as a single logical unit of work. Transactions are crucial for maintaining data integrity, especially in multi-user environments or when performing multi-step operations where all steps must succeed or fail together.

PostgreSQL provides ACID guarantees for transactions:

- **Atomicity:** Ensures that all operations within a transaction complete successfully or none of them are applied. If any part fails, the entire transaction is rolled back.
- **Consistency:** Guarantees that a transaction brings the database from one valid state to another, respecting all defined constraints (like primary keys, foreign keys, check constraints).
- **Isolation:** Ensures that concurrent transactions do not interfere with each other, making them appear to run sequentially. PostgreSQL uses Multi-Version Concurrency Control (MVCC) to achieve high levels of isolation with good performance. Different isolation levels (`READ COMMITTED`, `REPEATABLE READ`, `SERIALIZABLE`) can be set.
- **Durability:** Ensures that once a transaction is successfully committed, its changes are permanent and will survive subsequent system failures (e.g., crashes, power outages). This is typically achieved through Write-Ahead Logging (WAL).

Basic Transaction Control Commands:

- `BEGIN;` or `START TRANSACTION;`: Starts a new transaction.
- `COMMIT;`: Makes all changes within the transaction permanent and ends the transaction.
- `ROLLBACK;`: Discards all changes made within the transaction and ends the transaction.
- `SAVEPOINT name;`: Creates a point within the transaction to which you can later roll back.
- `ROLLBACK TO SAVEPOINT name;`: Rolls back changes to a specific savepoint.
- `RELEASE SAVEPOINT name;`: Removes a savepoint.

```
-- Example: Transfer $100 from Account 1 to Account 2
BEGIN; -- Start the transaction

-- Step 1: Deduct from Account 1
UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1;

-- Step 2: Add to Account 2
UPDATE Accounts SET Balance = Balance + 100 WHERE AccountID = 2;

-- (Check for errors or conditions here in application logic)

-- If all steps are successful:
COMMIT; -- Make the changes permanent

-- If any step failed or condition not met (in application logic):
-- ROLLBACK; -- Discard all changes made since BEGIN
```

Listing 3.23: Basic Transaction Example

In `psql`, auto-commit is typically ON by default outside an explicit `BEGIN`. Inside a `BEGIN...COMMIT/ROLLBACK` block, auto-commit is off. Application code using database drivers often controls transaction boundaries explicitly.

3.5 Indexes

Indexes are database structures associated with tables or views that speed up data retrieval operations (primarily `SELECT` queries with `WHERE` clauses or joins). They work much like an index in the back of a book, allowing the database engine to find specific rows quickly without scanning the entire table (full table scan).

PostgreSQL automatically creates indexes for primary key and unique constraints. You can create additional indexes on other columns to optimize specific query patterns.

Creating an Index:

```
-- Create an index on the Major column of the Students table
CREATE INDEX idx_students_major ON Students (Major);

-- Create a multi-column index
CREATE INDEX idx_enrollments_student_course ON Enrollments (
    StudentID, CourseID);

-- Create a unique index (enforces uniqueness in addition to
    speeding lookups)
CREATE UNIQUE INDEX idx_courses_coursename ON Courses (CourseName)
    ;
```

Listing 3.24: Creating an Index

When to Use Indexes:

- Columns frequently used in `WHERE` clauses.
- Columns used in `JOIN` conditions (especially foreign keys).
- Columns frequently used in `ORDER BY` clauses.
- Columns involved in primary key or unique constraints (usually automatic).

When to Avoid or Be Cautious with Indexes:

- **Small Tables:** The overhead of using the index might exceed the time saved compared to a full table scan.
- **Columns with Low Cardinality:** Columns with very few distinct values (e.g., a boolean flag) might not benefit much from a standard B-tree index (though specialized indexes like Bitmap indexes, not detailed here, might help in some systems, or BRIN in Postgres for correlated data).
- **Tables with Heavy Write Loads:** Indexes slow down `INSERT`, `UPDATE`, and `DELETE` operations because the index structure also needs to be updated. Over-indexing can hurt write performance significantly.
- **Columns Rarely Queried:** Indexing columns not used in query conditions wastes space and adds write overhead.

Types of Indexes in PostgreSQL: PostgreSQL offers several index types, each suited for different data types and query patterns:

- **B-tree:** The default and most common type. Excellent for equality (=) and range (<, >, <=, >=, BETWEEN) queries, and supports sorting (ORDER BY).
- **Hash:** Optimized only for equality (=) comparisons. Can be faster than B-tree for simple equality but has limitations (e.g., not WAL-logged before PostgreSQL 10, meaning not crash-safe until then; now generally usable but still less versatile than B-tree).
- **GiST (Generalized Search Tree):** A framework for building indexes over complex data types like geometric data (points, polygons) or for full-text search. Supports nearest-neighbor searches (<-> operator) and other specialized operators.
- **SP-GiST (Space-Partitioned GiST):** An extension of GiST suitable for non-balanced data structures like quadtrees, k-d trees, radix trees. Useful for certain geometric or network address types, or prefix searches.
- **GIN (Generalized Inverted Index):** Optimized for indexing composite values where elements within the value are queried, such as arrays (& &, <@, @>), JSONB documents (?, ?&, ?|, @>), or full-text search lexemes (@@). Very efficient for checking if an element exists within a composite type.
- **BRIN (Block Range Index):** Stores summary information (min/max, potentially others) for ranges of table blocks (pages). Very small footprint and low maintenance cost, extremely effective for large tables where values have a strong physical correlation with their storage location (e.g., timestamp columns that increase monotonically with inserts). Only useful for certain query types.

You can specify the index type using the USING clause:

```
CREATE INDEX idx_name ON table USING GIN (jsonb_column);
```

Listing 3.25: Specifying Index Type

Dropping an Index:

```
DROP INDEX idx_students_major;
```

Listing 3.26: Dropping an Index

3.6 Views

A view is a virtual table based on the result set of a stored SQL query. Views do not store data themselves (unless they are materialized views); they are essentially named queries that can be referenced like regular tables.

Creating a View:

```
-- Create a view showing only Computer Science students and their
   GPAs
CREATE VIEW CS_Student_View AS
SELECT Name, GPA
FROM Students
WHERE Major = 'Computer Science';

```

```sql
-- Create a more complex view joining tables
CREATE VIEW Student_Course_Grades AS
SELECT s.Name AS StudentName, c.CourseName, e.Grade
FROM Students s
JOIN Enrollments e ON s.StudentID = e.StudentID
JOIN Courses c ON e.CourseID = c.CourseID;
```

Listing 3.27: Creating a View

Querying a View: You query a view exactly like you query a table:

```sql
-- Select all data from the CS student view
SELECT * FROM CS_Student_View WHERE GPA > 3.8;

-- Query the complex view
SELECT * FROM Student_Course_Grades WHERE StudentName = 'Alice';
```

Listing 3.28: Querying a View

When you query a view, PostgreSQL essentially substitutes the view's definition into your query (though optimizations occur).

Benefits of Using Views:

- **Simplifies Complex Queries:** Encapsulate complex joins or logic into a simple view, making queries against it easier to write and understand.
- **Improves Security / Access Control:** Grant users access to a view that exposes only specific columns or rows, hiding sensitive data in the underlying tables.
- **Provides Logical Data Independence:** The underlying table structure can change, but the view definition can sometimes be modified to provide a consistent interface to applications, minimizing application code changes.
- **Readability and Reusability:** Give meaningful names to common query structures.

Updatable Views: Simple views (typically based on a single table, without aggregates, `GROUP BY`, `DISTINCT`, window functions, or complex expressions) can sometimes be directly updatable using `INSERT`, `UPDATE`, `DELETE`. However, complex views are generally not directly updatable. PostgreSQL has rules defining view updatability. You can also use `INSTEAD OF` triggers to define custom update logic for views.

Materialized Views: PostgreSQL also supports `CREATE MATERIALIZED VIEW`. Unlike regular views, materialized views physically store their result set. They are useful for caching the results of complex, expensive queries that don't need real-time data. You need to explicitly `REFRESH MATERIALIZED VIEW` to update their stored data.

Dropping a View:

```sql
DROP VIEW CS_Student_View;
DROP VIEW IF EXISTS Student_Course_Grades; -- Avoids error if view
    doesn't exist
```

Listing 3.29: Dropping a View

Chapter 4

Database Normalization

Database normalization is the process of structuring a relational database in accordance with a series of so-called normal forms in order to reduce data redundancy and improve data integrity. It involves organizing the columns (attributes) and tables (relations) of a database to ensure that database integrity constraints can be properly enforced through functional dependencies. This is typically achieved by decomposing large tables into smaller, less redundant tables and defining explicit relationships (foreign keys) between them.

4.1 Goals of Normalization

The primary objectives of database normalization are:

- **Minimizing Data Redundancy:** Storing the same piece of information in multiple places leads to wasted storage and potential inconsistencies (update anomalies). Normalization aims to store each logical piece of data only once.
- **Improving Data Integrity:** Reducing redundancy makes it easier to maintain consistency. When data is updated, it only needs to be changed in one place. This avoids insertion, update, and deletion anomalies.
- **Simplifying Data Modification:** Normalized schemas make `INSERT`, `UPDATE`, and `DELETE` operations more straightforward and less prone to error, as dependencies are clearly defined and data isn't unnecessarily duplicated.
- **Providing a Better Foundation for Future Growth:** A well-normalized database is generally easier to modify and extend as application requirements evolve.
- **Making Queries More Predictable (often):** While normalization can sometimes lead to more joins (which might impact performance), it often makes the relationships between data clearer, leading to more logical and maintainable queries.

4.2 Functional Dependencies

Understanding functional dependencies (FDs) is key to normalization. A functional dependency is a constraint between two sets of attributes in a relation.

Definition: An attribute B is functionally dependent on an attribute A (or a set of attributes A) if, for every valid instance of the relation, the value of A uniquely determines the value of B. This is denoted as $A \rightarrow B$.

In simpler terms: If you know the value(s) in A, there can only be one corresponding value (or set of values) for B in any given row.

Example: Consider a relation `Employees(EmpID, Name, DeptID, DeptName, Salary)`. Assume the following rules hold:

- Each employee has a unique ID (`EmpID`).
- Each employee has one name (`Name`) and one salary (`Salary`).
- Each employee belongs to exactly one department (`DeptID`).
- Each department ID (`DeptID`) corresponds to exactly one department name (`DeptName`).

Based on these rules, we have the following FDs:

- `EmpID` $\rightarrow$ `Name` (An employee ID determines the employee's name)
- `EmpID` $\rightarrow$ `Salary`
- `EmpID` $\rightarrow$ `DeptID` (An employee ID determines their department ID)
- `DeptID` $\rightarrow$ `DeptName` (A department ID determines the department's name)

From these, we can also infer:

- `EmpID` $\rightarrow$ `DeptName` (This is derived via `EmpID` $\rightarrow$ `DeptID` and `DeptID` $\rightarrow$ `DeptName`)

However, `DeptName` $\rightarrow$ `DeptID` might *not* hold if multiple departments could potentially have the same name (though assigned different IDs). Also, `DeptID` $\rightarrow$ `Salary` does not hold, as different employees in the same department can have different salaries.

Types of Functional Dependencies:

- **Trivial FD:** $A \rightarrow B$ where $B \subseteq A$. (e.g., `{EmpID, Name}` $\rightarrow$ `EmpID`). Always true.
- **Non-Trivial FD:** $A \rightarrow B$ where $B \not\subseteq A$. (e.g., `EmpID` $\rightarrow$ `Name`). These are the interesting ones for normalization.
- **Full Functional Dependency:** $A \rightarrow B$ is fully functional if removing any attribute from A invalidates the dependency. This is relevant when A is a composite key (multiple attributes). B must depend on *all* parts of A, not just a subset.
- **Partial Functional Dependency:** $A \rightarrow B$ where B depends on only a *part* (a proper subset) of the composite key A. Violates Second Normal Form (2NF).
- **Transitive Functional Dependency:** An indirect dependency where $A \rightarrow B$ and $B \rightarrow C$, resulting in $A \rightarrow C$, but B is *not* part of the primary key and B does not functionally determine A. Violates Third Normal Form (3NF). In our example, `EmpID` $\rightarrow$ `DeptID` and `DeptID` $\rightarrow$ `DeptName` leads to the transitive dependency `EmpID` $\rightarrow$ `DeptName` (where `DeptID` is the intermediate non-key attribute).

4.3 Normal Forms

Normal forms (NFs) are criteria for structuring relations based on their functional dependencies. Higher normal forms have stricter requirements.

4.3.1 First Normal Form (1NF)

A relation is in 1NF if and only if all underlying domains contain atomic values only. This means:

1. Each cell (intersection of a row and column) must contain a single, indivisible value.
2. There are no repeating groups (e.g., multiple phone numbers stored in a single `PhoneNumbers` column).
3. Each row is unique (implicitly required by the definition of a relation, usually enforced by a primary key).

Example (Not in 1NF): A table storing student courses like this:

StudentID	Name	CoursesTaken
101	Alice	'CS101', 'MA101'
102	Bob	'PH101'

Table 4.1: Student Courses Relation (Not in 1NF)

The `CoursesTaken` attribute contains a set/list of values, violating atomicity.

Example (In 1NF): To achieve 1NF, we decompose the table. The standard way is to create a separate row for each student-course combination:

StudentID	Name
101	Alice
102	Bob

Table 4.2: Students Relation (1NF)

StudentID	CourseID
101	CS101
101	MA101
102	PH101

Table 4.3: Enrollments Relation (1NF)

Now, each cell contains a single value. The primary key for `Enrollments` would likely be `(StudentID, CourseID)`.

4.3.2 Second Normal Form (2NF)

A relation is in 2NF if and only if:

1. It is already in 1NF.
2. Every non-key attribute is fully functionally dependent on the *entire* primary key.

This means there are no partial dependencies. If the primary key is a single attribute, the relation is automatically in 2NF if it's in 1NF. 2NF is only relevant for relations with composite primary keys.

Example (Not in 2NF): Consider an `Enrollments` table where we store student and course details directly: Primary Key: `(StudentID, CourseID)`

StudentID	CourseID	StudentName	CourseName	Grade
101	CS101	Alice	Intro to Programming	A
101	MA101	Alice	Calculus I	B
102	MA101	Bob	Calculus I	A

Table 4.4: Enrollments Relation (Assumed PK: (StudentID, CourseID) - Not in 2NF)

Here:

- `StudentName` depends only on `StudentID` (partial dependency).
- `CourseName` depends only on `CourseID` (partial dependency).
- `Grade` depends on the full key `(StudentID, CourseID)`.

The partial dependencies on `StudentName` and `CourseName` violate 2NF. This leads to redundancy (Alice's name stored twice, Calculus I name stored twice) and update anomalies (if Alice changes her name, it needs updating in multiple rows).

Example (In 2NF): Decompose into tables where non-key attributes depend on the whole key or exist in tables with single-attribute keys:

StudentID	StudentName
101	Alice
102	Bob

Table 4.5: Students Relation (2NF)

CourseID	CourseName
CS101	Intro to Programming
MA101	Calculus I

Table 4.6: Courses Relation (2NF)

Now, `Grade` depends on the full key `(StudentID, CourseID)` in the `Enrollments` table. `StudentName` depends on `StudentID` (the key) in the `Students` table. `CourseName` depends on `CourseID` (the key) in the `Courses` table. There are no partial dependencies.

StudentID	CourseID	Grade
101	CS101	A
101	MA101	B
102	MA101	A

Table 4.7: Enrollments Relation (PK: (StudentID, CourseID) - Now in 2NF)

4.3.3 Third Normal Form (3NF)

A relation is in 3NF if and only if:

1. It is already in 2NF.
2. There are no transitive functional dependencies of non-key attributes on the primary key.

This means no non-key attribute should be functionally dependent on another non-key attribute.

Example (Not in 3NF): Consider an `Employees` table with department information: Primary Key: `EmployeeID`

EmployeeID	Name	DeptID	DeptName	DeptLocation
1	Alice	10	Sales	New York
2	Bob	20	Marketing	London
3	Charlie	10	Sales	New York

Table 4.8: Employees Relation (PK: EmployeeID - Not in 3NF)

Here:

- `EmployeeID` → `DeptID` (Employee determines their department ID)
- `DeptID` → `DeptName` (Department ID determines its name)
- `DeptID` → `DeptLocation` (Department ID determines its location)

Because `EmployeeID` → `DeptID` and `DeptID` → `DeptName` (and `DeptLocation`), we have transitive dependencies: `EmployeeID` → `DeptName` and `EmployeeID` → `DeptLocation` via the non-key attribute `DeptID`. This violates 3NF. It causes redundancy (Sales/New York stored multiple times) and update anomalies (changing the Sales department location requires updating multiple employee rows).

Example (In 3NF): Decompose to remove the transitive dependency:

EmployeeID	Name	DeptID
1	Alice	10
2	Bob	20
3	Charlie	10

Table 4.9: Employees Relation (PK: EmployeeID - Now in 3NF)

DeptID	DeptName	DeptLocation
10	Sales	New York
20	Marketing	London

Table 4.10: Departments Relation (PK: DeptID - Also in 3NF)

Now, the `Employees` table only contains attributes directly dependent on `EmployeeID`. The `Departments` table contains attributes dependent on `DeptID`. There are no transitive dependencies within either table.

4.3.4 Boyce-Codd Normal Form (BCNF)

BCNF (Boyce-Codd Normal Form) is a slightly stricter version of 3NF. A relation is in BCNF if and only if:

1. It is already in 3NF.
2. For every non-trivial functional dependency $X \rightarrow Y$, X must be a superkey. (A superkey is a set of attributes that uniquely identifies a row; a candidate key is a minimal superkey).

Essentially, every determinant (the left side of an FD) must be a candidate key. BCNF handles certain rare anomalies not addressed by 3NF, typically involving relations with multiple overlapping candidate keys.

Example (Not in BCNF, but in 3NF): Consider a relation `AdvisorAssignments(StudentID, AdvisorID, MajorID)` representing that a student can have multiple advisors for potentially different aspects of their major, an advisor advises for only one major, and a student has only one major they are associated with via any specific advisor. Assume these FDs:

- `(StudentID, AdvisorID)` $\rightarrow$ `MajorID` (A specific student-advisor pairing relates to one major)
- `MajorID` $\rightarrow$ `AdvisorID` (Let's assume, perhaps unrealistically, that each major has only *one* specific lead advisor assigned, identified by `AdvisorID`)

Candidate Keys:

- `(StudentID, AdvisorID)` (Determines `MajorID`)
- `(StudentID, MajorID)` (Because `MajorID` $\rightarrow$ `AdvisorID`, knowing the student and the major tells you the unique lead advisor for that major, thus determining the full tuple).

Functional Dependencies:

1. `(StudentID, AdvisorID)` $\rightarrow$ `MajorID` (Determinant `(StudentID, AdvisorID)` is a candidate key - OK for BCNF)
2. `MajorID` $\rightarrow$ `AdvisorID` (Determinant `MajorID` is *not* a candidate key (it's only part of one) - Violates BCNF!)

This relation *is* in 3NF because `AdvisorID` (the right side of the violating FD) is part of a candidate key `(StudentID, AdvisorID)`. 3NF allows dependencies where the determinant is not a candidate key if the dependent attribute is part of *some* candidate key. BCNF forbids this.

Example (In BCNF): Decompose to satisfy BCNF, separating the problematic dependency:

StudentID	AdvisorID
S101	A20
S101	A25
S102	A20

Table 4.11: Student_Advisors Relation (PK: (StudentID, AdvisorID) - BCNF)

AdvisorID	MajorID
A20	CS
A25	MATH

Table 4.12: Advisor_Majors Relation (PK: AdvisorID - BCNF)

Now, in `Student_Advisors`, the only FD is the trivial one from the key. In `Advisor_Majors`, the FD is `AdvisorID` -> `MajorID`, and `AdvisorID` is the key. Both are in BCNF.

BCNF ensures the highest level of integrity related to functional dependencies but sometimes requires decompositions that might make certain queries require more joins.

4.4 Denormalization

While normalization is generally desirable for data integrity and reducing redundancy, there can be performance drawbacks, primarily due to the increased number of joins required to retrieve data spread across multiple tables. Denormalization is the *controlled* process of introducing redundancy back into a database schema, typically by combining tables or adding pre-calculated fields, specifically to improve read performance for critical queries.

Reasons for Denormalization:

- **Improving Query Performance:** Reducing the number of joins is the most common reason. Joins can be computationally expensive, especially with large tables.
- **Simplifying Queries:** Some queries become much simpler to write and understand against a denormalized structure.
- **Reporting Requirements:** Data warehouses and reporting databases are often denormalized (using star or snowflake schemas) to facilitate slicing, dicing, and aggregation for business intelligence.
- **Pre-computation:** Storing derived or calculated values (e.g., `OrderTotal` in an `Orders` table instead of calculating it from `OrderItems` every time).

Example: In our 3NF example (Tables **??** and **??**), retrieving an employee's name and their department's name requires a join. If this query is extremely frequent and performance-critical, we might denormalize by adding `DeptName` back into the `Employees` table:

EmployeeID	Name	DeptID	DeptName
1	Alice	10	Sales
2	Bob	20	Marketing
3	Charlie	10	Sales

Table 4.13: Denormalized Employees Table (for performance)

Drawbacks of Denormalization:

- **Increased Data Redundancy:** Duplicates `DeptName`.
- **Increased Storage Space:** Requires more disk space.
- **Update Anomalies Risk:** If the 'Sales' department name changes, it must be updated in *all* relevant employee rows, increasing complexity and the risk of inconsistency if not done carefully (often requires triggers or application logic to manage).
- **More Complex Data Modification Logic:** Inserts, updates, and deletes become more complex to keep redundant data synchronized.

When to Consider Denormalization: Denormalization should be considered a targeted optimization technique, applied cautiously *after* achieving a reasonable level of normalization (usually 3NF/BCNF) and *after* identifying specific performance bottlenecks that cannot be adequately resolved through other means (like proper indexing, query tuning, or caching).

- High read-to-write ratio for the affected data.
- Performance requirements cannot be met otherwise.
- Application logic can reliably manage the consistency of redundant data.

It's a trade-off: sacrificing some normalization purity for performance gains.

4.5 Choosing the Right Normal Form

The most common target for application databases (OLTP - Online Transaction Processing systems) is **3NF**. It provides a good balance between data integrity, reduced redundancy, and reasonable performance without the potential complexities of achieving BCNF in all cases or the performance overhead of higher normal forms (4NF, 5NF, DKNF, which deal with multi-valued dependencies and join dependencies, not covered here).

- **1NF:** Absolutely essential baseline.

- **2NF:** Automatically achieved if the table is in 1NF and has no composite keys. Otherwise, important to eliminate partial dependencies.
- **3NF:** The standard goal for most OLTP databases. Eliminates transitive dependencies, significantly reducing redundancy and anomalies.
- **BCNF:** Stricter than 3NF. Desirable if achievable without sacrificing crucial relationships or performance, but achieving it might sometimes lead to less intuitive schemas or require more joins for common queries.
- **Denormalization:** A deliberate step back from normalization, used selectively for performance optimization when the benefits clearly outweigh the integrity risks and maintenance costs.

The choice depends on the specific application requirements, query patterns, performance needs, and the acceptable level of data redundancy and update complexity. Start with 3NF/BCNF as the goal, and only denormalize strategically if necessary and well-justified.

Chapter 5

Conclusion

Congratulations on reaching the end of this introductory guide to database fundamentals! We've journeyed through the theoretical underpinnings of the relational model with Relational Algebra, explored the practical implementation and querying capabilities of a powerful open-source database system, PostgreSQL, using SQL, and delved into the crucial principles of Database Normalization for designing robust and efficient database schemas.

You now have a foundational understanding of:

- How data can be formally manipulated using relational algebra operations ($\sigma, \pi, \cup, -, \times, \rho, \cap, \bowtie, \div$).
- How to create, populate, query, and manage databases and tables using SQL in PostgreSQL (`CREATE`, `INSERT`, `SELECT`, `UPDATE`, `DELETE`, joins, subqueries, aggregates, CTEs, window functions, transactions, indexes, views).
- The importance of normalization (1NF, 2NF, 3NF, BCNF) for minimizing redundancy and improving data integrity, and the concept of denormalization as a performance tuning technique.

These concepts are the bedrock upon which modern data management systems are built. Mastering them provides you with essential skills for designing, developing, and interacting with databases effectively. Remember that database design and querying are often iterative processes; experience will refine your ability to choose appropriate data types, design efficient schemas, write optimized queries, and make informed decisions about normalization trade-offs.

Keep practicing with PostgreSQL, experiment with different query structures, and explore more advanced topics like database security, performance tuning, replication, and NoSQL alternatives as you continue your computer science journey. The world of data is vast and constantly evolving – embrace the learning process!

Good luck with your future database endeavors!