Notes
CSC-361
Database System Design

Dr. Qixin Deng

January 2026

Contents

1 Introduction

1.1 What is a Database?

A database is a structured collection of related data that models some aspect of the real world. Rather than being a random set of files, a database captures an abstract representation of an application domain.

- Data are typically organized into records
- Relationships between records are explicitly represented
- Data are stored persistently and accessed by many users and programs

This course focuses on Database Management Systems (DBMSs): software systems designed to create, store, manipulate, and retrieve data efficiently and safely. A DBMS acts as an intermediary between applications and data. Applications do not directly access files; instead, they issue requests to the DBMS, which enforces correctness, security, and performance guarantees.

1.2 Why Study Databases?

Databases are everywhere: university registration systems, learning management platforms, payroll services, research data repositories, and alumni records all rely on DBMS technology. Beyond ubiquity, databases have significant real-world impact:

- Core infrastructure for enterprises, governments, and research labs
- Central to analytics, reporting, and decision making
- Essential knowledge for building scalable software systems

Understanding databases equips you with concepts that extend well beyond any single system or application.

1.3 A College Management Example

To understand why DBMSs are necessary, we begin with a concrete example from a college environment.

Suppose you are responsible for designing an information system for a medium-sized college. The institution has thousands of students, hundreds of faculty and staff members, and multiple administrative offices. Many activities occur simultaneously, and accurate data sharing is essential. The college needs to store and maintain information about:

- Students: personal data, enrollment status, majors, transcripts
- Faculty: appointments, teaching assignments, advising roles
- Courses: schedules, prerequisites, capacity limits
- Departments and programs
- Administrative records: tuition payments, financial aid, payroll
- Campus services: housing, dining, transportation

Different groups require different access privileges. For example:

- Students may view their own grades but not others'
- Faculty can manage course rosters but not payroll data
- Administrators require broad access for reporting and compliance

1.3.1 A First Attempt: Using Files

Imagine implementing this system using only a file system.

1. **Data representation:** Decide what entities exist and which attributes to store (e.g., student ID, name, major).
2. **Storage:** Create files such as `students.txt`, `courses.txt`, `faculty.txt`, each containing one record per line.
3. **Access control:** Attempt to restrict file access at the operating system level.
4. **Data access:** Write separate programs to scan files, parse records, and update information.
5. **Performance:** Searching for students who have not paid tuition requires scanning large files repeatedly.

Soon, serious problems emerge.

1.3.2 Fundamental Challenges

1. When a student drops a course, enrollment counts and billing records must be updated together.
2. Course capacity constraints must never be violated.
3. Hundreds of students may register simultaneously without overwriting each other's actions.
4. Once grades are submitted, they must not be lost due to crashes or power failures.

1.3.3 ACID Properties

Transactions in a database management system are governed by four fundamental properties, collectively known as **ACID**. These properties ensure correctness, reliability, and robustness in the presence of concurrent execution and system failures.

- **Atomicity** guarantees that a transaction is treated as a single, indivisible unit of work. Either all operations within the transaction are successfully completed, or none of them are applied to the database. As a result, the database never reflects partial or incomplete results of a transaction.
- **Consistency** ensures that a transaction transforms the database from one valid state to another valid state. All integrity constraints, such as primary key constraints, foreign key constraints, and domain constraints, must be preserved before and after the transaction executes.
- **Isolation** guarantees that concurrently executing transactions do not interfere with one another. Each transaction behaves as if it were executed in isolation, even though multiple transactions may be processed simultaneously by the database system.
- **Durability** ensures that once a transaction has been committed, its effects persist permanently in the database. The committed changes will survive system crashes, power failures, or restarts.

These guarantees allow application developers to reason about database correctness without having to manage low-level concerns such as concurrency control and failure recovery.

1.4 Core Database Concepts

1.4.1 Data Model

A **data model** defines how data are conceptually organized in a database system. It specifies the structure of data, the relationships among data items, and the operations that can be performed on them. In other words, a data model provides an abstract framework for describing real-world information in a form that a database system can manage.

Different data models emphasize different ways of representing relationships and constraints. Common data models include:

- **Relational Model**: Data are represented as tables (relations), where each row corresponds to a record and each column corresponds to an attribute. Relationships between tables are expressed using keys. This model has a strong mathematical foundation and is the most widely used in practice.
- **Hierarchical Model**: Data are organized in a tree structure, where each child record has exactly one parent. This model is efficient for strictly hierarchical relationships but lacks flexibility for representing complex associations.
- **Graph-Based Model**: Data are represented as nodes and edges, making it natural to model many-to-many relationships. This model is commonly used in applications such as social networks, recommendation systems, and knowledge graphs.
- **Object-Oriented Model**: Data are stored as objects that encapsulate both state and behavior, supporting concepts such as inheritance and polymorphism. This model aligns closely with object-oriented programming but is less common in large-scale database systems.

In this course, we primarily focus on the **relational model** due to its simplicity, expressive power, and widespread adoption in modern database management systems.

1.4.2 Schema

A database can be described at multiple levels of abstraction. Two of the most important levels are the **logical schema** and the **physical schema**.

The **logical schema** describes the structure of the database using a chosen data model. It specifies:

- The tables in the database

- The attributes of each table
- Data types
- Primary keys and foreign keys

The logical schema defines what data are stored and how they are logically related, without considering how the data are physically implemented.

The **physical schema**, in contrast, describes how data are actually stored on disk. It includes details such as:

- File organization
- Index structures
- Page layout
- Storage locations

These details are managed internally by the database management system (DBMS).

Most users and application developers interact only with the logical schema, while the DBMS is responsible for translating logical operations into efficient physical actions. This separation allows database systems to optimize storage and performance without requiring changes to application-level queries.

1.4.3 Transactions

Transactions greatly simplify application development in concurrent and failure-prone environments. A transaction groups multiple database operations into a single logical unit of work.

Example:

Enroll a student in a course

Update enrollment table

Possibly update seat availability

These operations must behave as if they were one action, not several independent ones.

1.4.4 Structured Query Language (SQL)

Database Management Systems provide declarative query languages, such as SQL. You specify what data you want. You do not specify how to retrieve it

Let us focus on a simplified subset:

- Students (`student_id`, name, major)
- Courses (`course_id`, title, credits)
- Enrollment (`student_id`, `course_id`, grade)

Using the relational model, we represent each entity and relationship as a table. Good database design minimizes redundancy while preserving all required information. Examples of common queries:

- Find all courses taken by a given student

```sql
SELECT course_id
FROM Enrollment
WHERE student_id = 'S123';
```

- Retrieve email addresses of students enrolled in a specific course

```sql
SELECT s.email
FROM Student s, Enrollment e
WHERE s.student_id = e.student_id
AND e.course_id = 'CSC211';
```

Updates can be executed transactionally:

```sql
BEGIN;
INSERT INTO Enrollment VALUES ('S123', 'CSC211', NULL);
COMMIT;
```

1.5 Summary

Database systems provide principled solutions to data organization, access, concurrency, and reliability. Through formal data models, declarative querying, and transactional guarantees, DBMSs enable applications—such as college management systems—to scale safely and efficiently. The remainder of this course will explore these concepts in depth, focusing on both theoretical foundations and practical system design.

2 Preparation

2.1 MySQL Server and MySQL Workbench

MySQL is an open-source relational database management system widely used in academic and industrial applications. This tutorial guides you through installing MySQL on macOS and verifying that the system works correctly.

2.1.1 Install MySQL 8.0 Server

Install MySQL Server version 8.0 using Homebrew:

```
brew install mysql@8.0
```

After installation, add MySQL binaries to the shell `PATH`, you can see the command in your terminal:

```
echo 'export PATH="/opt/homebrew/opt/mysql@8.0/bin:$PATH"' >> ~/.zshrc
source ~/.zshrc
```

Verify the installation:

```
mysql --version
```

Start MySQL as a background service:

```
brew services start mysql@8.0
```

Run the built-in security configuration script:

```
mysql_secure_installation
```

Recommended answers are **Yes** for all security options.

Connect to the MySQL server as the root user:

```
mysql -u root -p
```

After entering the password, you should see the MySQL prompt:

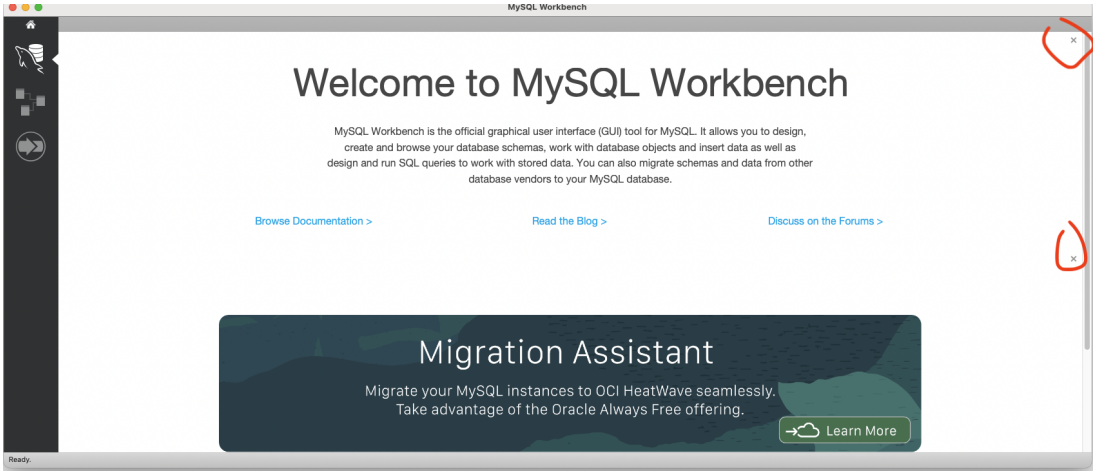
```
mysql>
```

Exit MySQL:

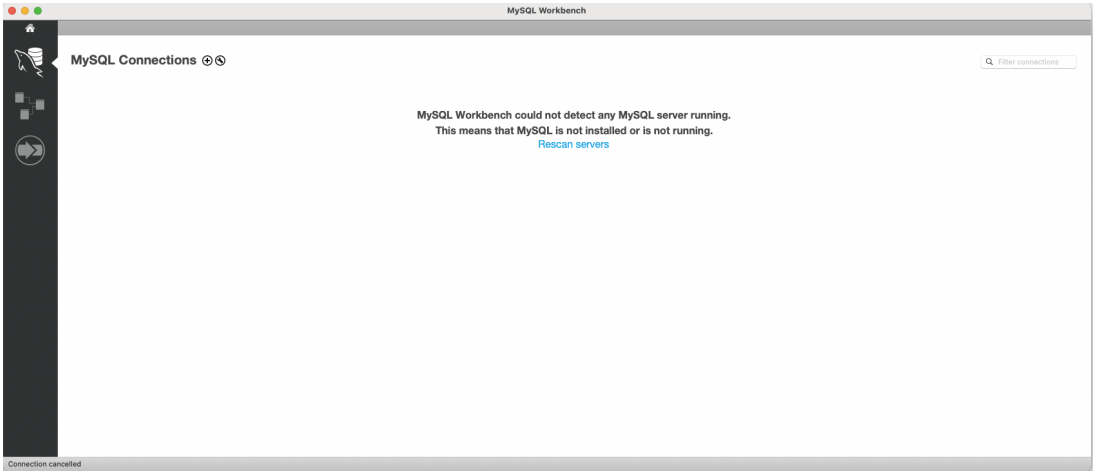
```
mysql>exit;
```

2.1.2 MySQL Workbench

MySQL Workbench is a graphical client and must connect to a running MySQL server. Open a web browser and navigate to `https://dev.mysql.com/downloads/workbench/`. Download the MySQL Workbench DMG file, and skip creating an Oracle account if prompted. Open the downloaded `.dmg` file and drag MySQL Workbench into the Applications folder to complete the installation. Open MySQL Workbench from the Applications folder and confirm that the application launches without errors. Close the welcome pages:



On the home screen, locate the `MySQL Connections` panel and click the `+` button to create a new connection.



In the connection setup window, set **Connection Name** to `Local MySQL`, set **Hostname** to `localhost`, set **Port** to `3306`, set **Username** to `root`, store your password in **Keychain**, and click **Test Connection**. A successful window will show up, close it and click OK. Double-click the newly created connection to open the SQL Editor interface in MySQL Workbench.

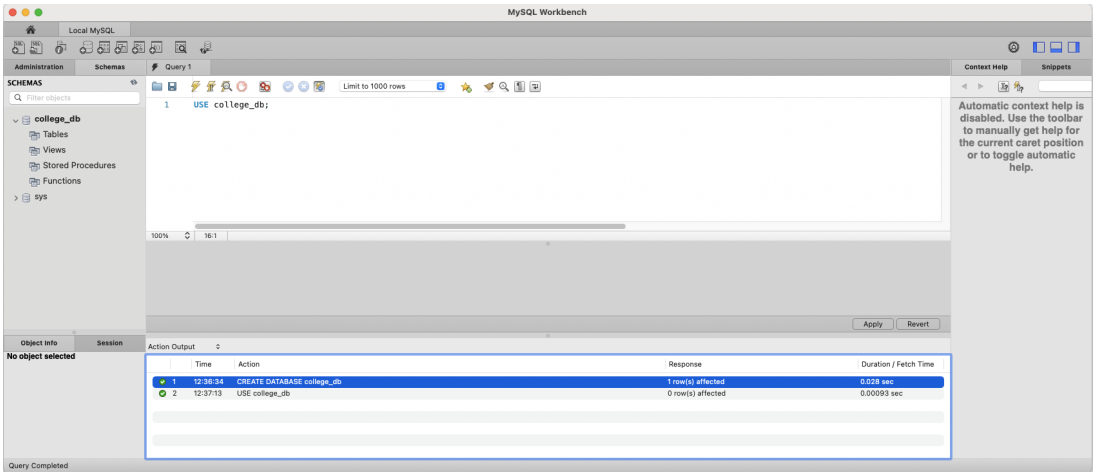
In the SQL Editor, create a new database by executing the following SQL statement using the lightning bolt (Execute) button:

```
1 CREATE DATABASE college_db;
```

Set the newly created database as the active schema by executing:

```
USE college_db;
```

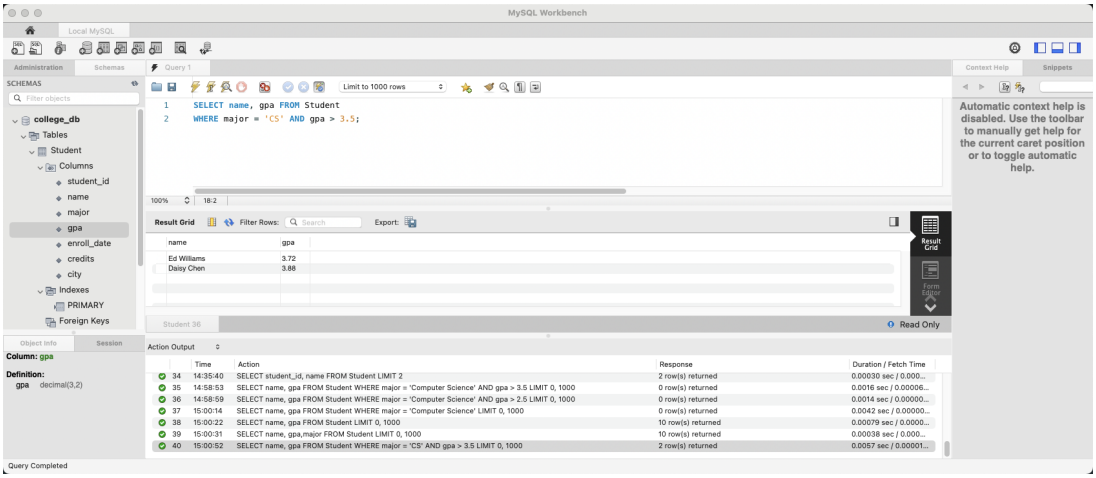
MySQL Workbench does NOT auto-refresh schemas. In the Schemas panel (left side), right-click anywhere then click `Refresh All`.



Create a sample table named `Student` by executing the following SQL statement in MySQL Workbench:

```
CREATE TABLE Student (
    student_id  INT PRIMARY KEY,
    name        VARCHAR(80) NOT NULL,
    major       VARCHAR(50),
    gpa         DECIMAL(3,2),
    enroll_date DATE NOT NULL,
    credits     INT NOT NULL DEFAULT 0,
    city        VARCHAR(60)
);
```

After executing the statement, refresh the schema view in MySQL Workbench. You should see the newly created `Student` table listed under the `college_db` database.



Insert a sample record into the `Student` table. Because some columns are optional or have default values, we can explicitly list the columns we want to insert:

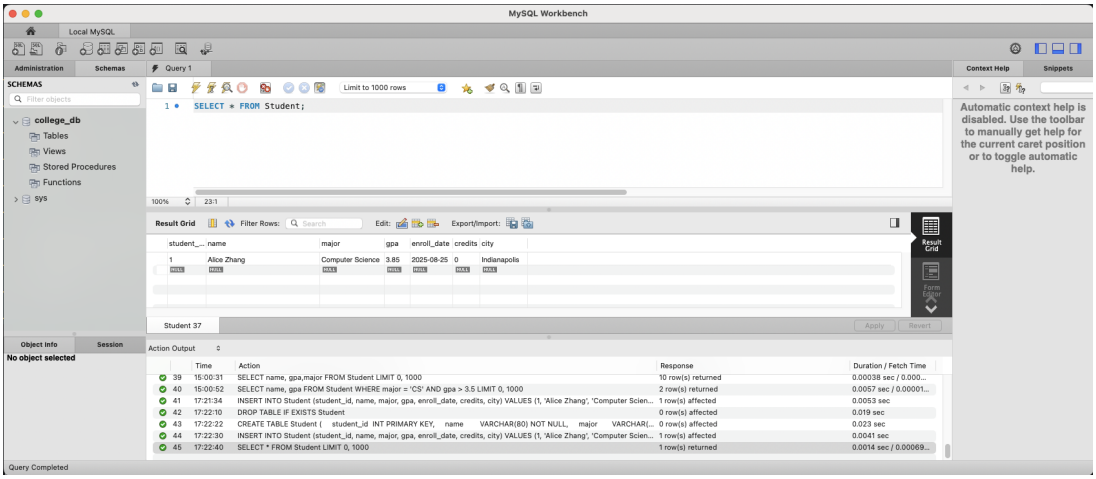
```
INSERT INTO Student
(student_id, name, major, gpa, enroll_date, credits, city)
VALUES
(1, 'Alice Zhang', 'Computer Science', 3.85, '2025-08-25', 0, 'Indianapolis');
```

Alternatively, if we omit optional columns such as `major`, `gpa`, or `city`, MySQL will store `NULL` for those fields, and the `credits` column will automatically use its default value of `0`.

Verify the table contents by executing:

```
SELECT * FROM Student;
```

Confirm that the query results appear in the Result Grid at the bottom of the MySQL Workbench window, showing all columns defined in the table schema.



After completing the verification, you may close MySQL Workbench.

The database and its tables are stored on disk as binary files managed by MySQL. For example, on a macOS system using Homebrew, the database directory may be located at:

```
ls /opt/homebrew/var/mysql/college_db/
```

You should see a file corresponding to the `Student` table, such as:

```
student.ibd
```

This binary file contains the physical storage for the `Student` table. Its contents are managed entirely by MySQL and should never be edited directly by users.

2.2 Sample Python Project

This project is a minimal full-stack CRUD demo (CRUD refers to the four fundamental operations—Create, Read, Update, and Delete—that define how applications interact with persistent data.) for a **college student database**:

- A **client** web page (`index.html`) renders a student table and provides forms to add/delete students. We will add more functions as class going on.
- A **server** (`main.py, db.py`) implemented with **FastAPI** provides HTTP routes.
- A **database** (`college_db`) implemented with `MySQL Server 8.0`.

The code is intentionally small so that you can clearly see the **client–server–database** flow. We will add more functions as the class goes on.

Layer	Responsibility
Client (HTML)	UI forms, displaying the student list, sending HTTP requests via standard form submissions.
Server (FastAPI)	Routing, validating form fields, calling database access functions, rendering templates, redirecting after mutations.
Database (MySQL)	Persistent storage; enforces primary key uniqueness for `student_id`.

A typical layout for this project is:

```
project_root/
  main.py
  db.py
  templates/
    index.html
```

The required dependencies are:

```
fastapi==0.115.0
uvicorn==0.30.6
mysql-connector-python==9.0.0
jinja2==3.1.4
python-dotenv==1.0.1
python-multipart
```

2.2.1 Database Logic: db.py

The file `db.py` defines environment-driven database configuration and creates a **connection pool**. `Autocommit` means that every SQL statement that modifies data (such as INSERT, UPDATE, or DELETE) is automatically committed to the database as soon as it executes, without requiring an explicit COMMIT. In practice, this makes the code simpler for a teaching demo, but it also means you cannot roll changes back (ROLLBACK) later. All of the information listed below is the same information we used when setting up the server and creating the database.

```python
import os
import mysql.connector
import mysql.connector.pooling

# Read database settings from environment variables.
# If a variable is missing, use the default value shown.
DB_CONFIG = {
    "host": os.getenv("DB_HOST", "127.0.0.1"),
    "port": int(os.getenv("DB_PORT", "3306")),
    "database": os.getenv("DB_NAME", "college_db"),
    "user": os.getenv("DB_USER", "root"),
    "password": os.getenv("DB_PASSWORD", "YOUR PASSWORD HERE"),
    "autocommit": True,
}
```

When developing database-backed applications, database credentials should never be hard-coded in source files. PyCharm provides a simple and secure mechanism to inject secrets at runtime using **Run Configuration environment variables**, which integrates seamlessly with Python virtual environments.

In PyCharm, navigate to:

`Run` → `Edit Configurations...`

Click **Environment variables** and add the following key–value pair:

`DB_PASSWORD=your_real_password`

These variables are injected into the runtime process automatically. The application retrieves credentials using environment variables:

```python
DB_CONFIG = {
    "host": os.getenv("DB_HOST", "127.0.0.1"),
    "port": int(os.getenv("DB_PORT", "3306")),
    "database": os.getenv("DB_NAME", "college_db"),
    "user": os.getenv("DB_USER", "root"),
    "password": os.getenv("DB_PASSWORD"),
    "autocommit": True,
}
```

No sensitive information appears in the source code or version control system.

Why use a connection pool? Opening a new MySQL connection for every request can be slow. A pool keeps a small number of ready-to-use connections so each request can borrow one quickly and return it when done. In Python, ** is used to unpack a dictionary so that: (1) dictionary keys become parameter names (2) dictionary values become argument values.

```
_pool = mysql.connector.pooling.MySQLConnectionPool(
    pool_name="college_pool",
    pool_size=5,
    **DB_CONFIG,
)
```

Each database function borrows a connection, runs a query, then closes the connection to return it to the pool. `conn.cursor()` creates a cursor object from an existing database connection. The cursor is used to execute SQL statements and retrieve results from the database, since SQL commands cannot be run directly on the connection itself. `dictionary=True` creates a cursor object that returns query results as dictionaries instead of tuples, allowing columns to be accessed by name. After execution, calling `cur.fetchall()` retrieves all rows from the result set at once.

In our updated full-stack project, the `Student` table has the following schema:

Column	Data Type / Constraint	Description
student_id	INT, PRIMARY KEY	Unique identifier for each student
name	VARCHAR(80), NOT NULL	Student name
major	VARCHAR(50), NULL	Academic major (optional)
gpa	DECIMAL(3,2), CHECK (0.0–4.0)	Grade point average (optional)
enroll_date	DATE, NOT NULL	Date the student enrolled
credits	INT, NOT NULL, DEFAULT 0	Credits earned by the student
city	VARCHAR(60), NULL	City of residence (optional)

Therefore, the database logic must select and insert all columns in a consistent way. One special observation is %s is a placeholder, not a string type, different from the definition in Python. Below are the core helper functions for listing students, inserting a student, and deleting a student using the above schema.

```
from typing import Optional

def fetch_all_students():
    conn = _pool.get_connection()
    try:
        cur = conn.cursor(dictionary=True)
        cur.execute(
            """
            SELECT student_id, name, major, gpa, enroll_date, credits, city
            FROM Student
            ORDER BY student_id DESC
            """
        )
        return cur.fetchall()
    finally:
        conn.close()


def insert_student(
    student_id: int,
    name: str,
    major: Optional[str],
    gpa: Optional[float],
    enroll_date: str,
    credits: Optional[int] = None,
    city: Optional[str] = None,
):
    """
    Insert a student record into the Student table.

    - major, gpa, city may be None (stored as SQL NULL)
    - credits may be None, in which case the DEFAULT value (0) is used
    - gpa validity (0.0{4.0) is enforced by the database CHECK constraint
```

```
    """
    # Normalize empty strings to None
    major = major.strip() if isinstance(major, str) and major.strip() else None
    city = city.strip() if isinstance(city, str) and city.strip() else None

    conn = _pool.get_connection()
    try:
        cur = conn.cursor()

        if credits is None:
            # Let the database apply DEFAULT 0
            cur.execute(
                """
                INSERT INTO Student
                (student_id, name, major, gpa, enroll_date, city)
                VALUES (%s, %s, %s, %s, %s, %s)
                """,
                (student_id, name, major, gpa, enroll_date, city),
            )
        else:
            cur.execute(
                """
                INSERT INTO Student
                (student_id, name, major, gpa, enroll_date, credits, city)
                VALUES (%s, %s, %s, %s, %s, %s, %s)
                """,
                (student_id, name, major, gpa, enroll_date, credits, city),
            )
    finally:
        conn.close()


def delete_student(student_id: int):
    conn = _pool.get_connection()
    try:
        cur = conn.cursor()
        cur.execute(
            "DELETE FROM Student WHERE student_id = %s",
            (student_id,),
        )
    finally:
        conn.close()
```

While a `try` block is used, the primary concern in this context is not exception handling but guaranteeing that the database connection is properly closed. Therefore, the `finally` block is essential.

2.2.2 Web Logic: main.py

The server file `main.py` builds a `FastAPI` app and loads `Jinja2` templates. `Jinja2` is a server-side templating engine that allows developers to generate dynamic `HTML` by combining static `HTML` templates with data provided by Python code. In a `FastAPI` application, `Jinja2` processes a template (such as `index.html`) on the server, replaces variables and control structures with actual data, and produces a final static `HTML` document. This `HTML` is then sent to the browser, which handles the visual rendering for the user.

In our full-stack project, the `Student` table contains the following columns: `student_id`, `name`, `major`, `gpa`, `enroll_date`, `credits`, and `city`. Therefore, the web-layer logic must parse these values from the HTML form and pass them to the database-layer function `insert_student`.

```
from typing import Optional

from fastapi import FastAPI, Request, Form
from fastapi.responses import HTMLResponse, RedirectResponse
from fastapi.templating import Jinja2Templates

from db import fetch_all_students, insert_student, delete_student

```

```
app = FastAPI(title="College DB Demo (Student table)")

templates = Jinja2Templates(directory="templates")
```

Do you still remember **decorators** in Python? In Python, a `decorator` is a special language feature that allows a function to be wrapped or augmented with additional behavior without modifying the function's source code. Syntactically, a decorator is written using the `@` symbol placed directly above a function definition. Conceptually, a decorator takes a function as input (higher-order function), adds extra functionality to it, and returns a new function. This mechanism is widely used in frameworks because it cleanly separates what a function does from how and when the function is invoked. In FastAPI, `@app.get("/")` is implemented using a `decorator factory`. Although FastAPI uses Python decorators syntactically, they are primarily used for route registration and event linkage, not for wrapping function behavior as in traditional decorators. The decorator runs once, at startup, then every time FastAPI router sees `"GET /"`, `index()` is called.

The code below defines a FastAPI route that handles HTTP `GET` requests sent to the root URL (`/`). Whenever a browser issues a `"GET /"` request—such as when a user types the website URL into the address bar, refreshes the page, clicks a link pointing to `/`, or is redirected back to `/` after submitting a form—FastAPI invokes the `index` function. Inside this function, the server calls `fetch_all_students()` to retrieve the current list of students from the database, including all columns in the updated schema. It then passes this data, along with the required `request` object, to Jinja2 via `TemplateResponse`. Jinja2 renders the `index.html` template by replacing template variables and control structures with actual data, producing a final static HTML page. FastAPI sends this HTML as the HTTP response, and the browser is responsible for visually rendering the page for the user.

In plain Python, parameter types are optional, but in FastAPI they are essential because FastAPI uses type annotations to decide how to parse, validate, and inject request data (HTTP request → parsing → validation → function call).

```
@app.get("/", response_class=HTMLResponse) # response_class is JSON by default
def index(request: Request): # templates need access to the HTTP request object to produce new html
    students = fetch_all_students()
    return templates.TemplateResponse(
        "index.html",
        {"request": request, "students": students},
    )
```

The `@app.post("/students")` decorator defines a FastAPI route that responds to HTTP `POST` requests sent to the `/students` endpoint. The function parameters are declared using `Form(...)` together with Python type annotations. This instructs FastAPI to extract values from the submitted HTML form and automatically validate and convert them to the specified types. Thank means all of the datatype annotations here are necessary! An HTML form is just a structured way to send key–value strings; FastAPI uses types to turn them into real data safely.

In our schema, `student_id`, `name`, and `enroll_date` are required, while `major`, `gpa`, and `city` are optional. The `credits` column is `NOT NULL` but has a database-level `DEFAULT 0`. Therefore, in the web layer we may either require `credits` in the form or allow it to be optional and let the database fill in the default value. In the implementation below, we treat `credits` as optional: if it is missing, we pass `None` so that the database uses `DEFAULT 0`.

After inserting the new record, the function returns a `RedirectResponse` with status code `303`. This response instructs the browser to issue a new `GET` request to the root URL (`/`). This pattern, known as Post–Redirect–Get, prevents duplicate submissions if the user refreshes the page and ensures that the updated list of students is displayed.

```
@app.post("/students")
def add_student(
    student_id: int = Form(...),
    name: str = Form(...),
    major: Optional[str] = Form(None),
    gpa: Optional[float] = Form(None),
    enroll_date: str = Form(...),
    credits: Optional[int] = Form(None),
    city: Optional[str] = Form(None),
):
    insert_student(student_id, name, major, gpa, enroll_date, credits, city)
    return RedirectResponse("/", status_code=303)
```

The decorator `@app.post("/students/{student_id}/delete")` registers the function `remove_student` as the handler for POST requests whose URL matches the specified path pattern, where `student_id` is a dynamic path parameter.

When such a request is received, FastAPI extracts the value of `student_id` from the URL, converts it to an integer according to the function's type annotation, and passes it into the function. The function then invokes the database-layer deletion operation to remove the corresponding student record and returns a `RedirectResponse` with status code `303`, instructing the browser to issue a new GET request to the root URL so that the updated student list can be rendered and displayed.

```
@app.post("/students/{student_id}/delete")
def remove_student(student_id: int):
    delete_student(student_id)
    return RedirectResponse("/", status_code=303)
```

The conditional block `if __name__ == "__main__"` ensures that the enclosed code is executed only when the Python file is run directly, and not when it is imported as a module. Inside this block, the `uvicorn` server is imported and used to launch the FastAPI application specified by `"main:app"`, which refers to the `app` object defined in the `main.py` file. The server is configured to listen on the local host address `127.0.0.1` at port `3000`, making the application accessible through a web browser at that address. The `reload=True` option enables automatic server restarts whenever the source code changes, which is particularly useful during development and testing.

```
if __name__ == "__main__":
    import uvicorn
    uvicorn.run("main:app", host="127.0.0.1", port=3000, reload=True)
```

2.2.3 Web Template: index.html

This file is an HTML page with Jinja2 template syntax. The HTML provides the structure and styling of the page, while Jinja2 directives (e.g., `{% for ... %}` and `{{ ... }}`) allow the server to inject dynamic student data at render time. The page contains two major user-facing features: (1) a form to add a new student and (2) a table that lists existing students and provides a delete action.

```
<!doctype html>
<html>
```

- `<!doctype html>` tells the browser to interpret the file as **HTML5**, enabling modern parsing rules.
- `<html>` is the root element that contains the entire document.

```
<head>
  <meta charset="utf-8">
  <title>College DB (FastAPI)</title>
  <style>
    body { font-family: Arial, sans-serif; margin: 2rem; }
    .card { border: 1px solid #ddd; border-radius: 10px; padding: 1rem; margin-bottom: 1rem; }
    table { border-collapse: collapse; width: 100%; }
    th, td { border-bottom: 1px solid #ddd; padding: 0.5rem; text-align: left; }
    input, select { padding: 0.4rem; margin-right: 0.4rem; }
    button { padding: 0.45rem 0.8rem; cursor: pointer; }
  </style>
</head>
```

- `<head>` contains information about the webpage, not the webpage itself.
- `<meta charset="utf-8">` sets the character encoding to UTF-8, allowing international characters.
- `<title>` defines the text shown in the browser tab.
- The embedded `<style>` block defines a simple and readable layout:
  - `body`: sets a clean sans-serif font and adds spacing.
  - `.card`: creates visually grouped sections.
  - `table`: formats student data in a standard tabular layout.
  - `input`, `select`, and `button`: improve usability and spacing.

```
<body>

<h1>College DB Demo (FastAPI + MySQL)</h1>
```

- The `<body>` element contains all visible content.
- The `<h1>` element provides a top-level heading for the application.

```
<div class="card">
  <h2>Add Student</h2>
  <form method="post" action="/students">
    <input name="student_id" type="number" placeholder="Student ID" required>
    <input name="name" placeholder="Name" required>
    <input name="major" placeholder="Major (optional)">
    <input name="gpa" type="number" step="0.01" min="0" max="4.0" placeholder="GPA (0.0{4.0)">
    <input name="enroll_date" type="date" required>
    <input name="credits" type="number" placeholder="Credits (default 0)">
    <input name="city" placeholder="City (optional)">
    <button type="submit">Add</button>
  </form>
  <p style="color:#666;margin-top:0.5rem;">
    Note: <code>student_id</code> is the primary key. Fields marked optional may be left blank.
  </p>
</div>
```

- This `div` uses the `.card` class to present the form as a grouped panel.
- The `<form>` collects input values and submits them to the server.
- `method="post"` indicates that the form modifies database state.
- `action="/students"` corresponds to the FastAPI route `@app.post("/students")`.
- Each `name` attribute matches a parameter in the FastAPI handler:
  - `student_id`, `name`, `enroll_date` are required.
  - `major`, `gpa`, and `city` are optional.
  - `credits` may be omitted to allow the database default value to apply.

```
<div class="card">
  <h2>Students</h2>
  <table>
    <thead>
      <tr>
        <th>Student ID</th>
        <th>Name</th>
        <th>Major</th>
        <th>GPA</th>
        <th>Enroll Date</th>
        <th>Credits</th>
        <th>City</th>
        <th>Action</th>
      </tr>
    </thead>
```

- The table header `<thead>` defines column labels corresponding to the database schema.
- `<tr>` means table row and `<th>` represents table header cell
- The `Action` column is reserved for row-level operations.

```
    <tbody>
      {% for s in students %}
      <tr>
        <td>{{ s.student_id }}</td>
```

```
        <td>{{ s.name }}</td>
        <td>{{ s.major }}</td>
        <td>{{ s.gpa }}</td>
        <td>{{ s.enroll_date }}</td>
        <td>{{ s.credits }}</td>
        <td>{{ s.city }}</td>
        <td>
          <form method="post" action="/students/{{ s.student_id }}/delete" style="display:inline;">
            <button type="submit">Delete</button>
          </form>
        </td>
      </tr>
      {% else %}
      <tr>
        <td colspan="8">No students yet.</td>
      </tr>
      {% endfor %}
    </tbody>
  </table>
</div>
```

- `<tbody>` contains the table's data rows.
- `{% for s in students %}` iterates over the list of student records.
- Jinja2 expressions such as `{{ s.gpa }}` are replaced with actual values at render time.
- Each row includes a delete form that submits a POST request to:

  `/students/{{ s.student_id }}/delete`

- The `{% else %}` block handles the empty-table case, displaying a single message row.
- `colspan="8"` ensures the message spans all table columns.

```
</body>
</html>
```

These closing tags mark the end of the document body and the HTML document. After Jinja2 rendering, the browser receives plain HTML and is responsible for visual presentation.

3 Queries

Relational databases organize data into **tables** (also called **relations**). Each table has a name and a fixed schema consisting of **columns** (also called **attributes**). Each row in the table represents a single record and is called a **tuple**. The `Student` table in `college_db` is listed below:

student_id	name	major	gpa	enroll_date	credits	city
1001	Alice Zhang	Computer Science	3.72	2025-08-25	32	San Diego
1002	Brian Lee	Mathematics	3.10	2025-08-25	18	Santa Ana
1003	Carla Gomez	NULL	NULL	2024-08-26	28	Fresno
1004	David Kim	Computer Science	3.88	2025-01-10	36	San Jose

The columns of this table are `student_id`, `name`, `major`, `gpa`, `enroll_date`, `credits`, and `city`. Each row corresponds to one student enrolled in the college.

3.1 Data Definition and Modification

3.1.1 Data Types

Every column in a relational database table must be assigned a **data type**, which determines:

- what kind of values the column can store,
- how much storage space is required,
- what operations and comparisons are allowed on the column.

Choosing appropriate data types is essential for both data integrity and query performance. In this course, we focus on a small but commonly used subset of SQL data types, illustrated using the `Student` table.

- Integer types store whole numbers. They are commonly used for identifiers and counts. `INT`: stores a standard 32-bit integer. In our schema, `student_id` and `credits` use the `INT` type.
- Character string types store textual data. `VARCHAR(n)`: stores a variable-length string with a maximum length of `n` characters. The `VARCHAR` type is preferred over fixed-length types because it saves space when strings vary in length. In the `Student` table, several columns use `VARCHAR`, such as `name`, `major`, `city`.
- Numeric types are used for values that may include fractional components. `DECIMAL(p, s)`: stores a fixed-point number with `p` total digits, `s` of which appear after the decimal point. In the `Student` table, the `gpa` column uses `gpa DECIMAL(3,2)`, this definition allows values from `0.00` to `9.99`, which is sufficient for representing grade point averages while avoiding floating-point rounding errors.
- SQL provides specialized data types for representing dates and times. `DATE`: stores a calendar date in the format `YYYY-MM-DD`. In our schema, the `enroll_date` column records when a student enrolled. Using a date type enables meaningful comparisons, sorting, and range queries, such as filtering students who enrolled within a specific semester.
- By default, a column may contain the special value `NULL`, which represents missing or unknown data. Constraints are used to control whether `NULL` values are permitted.
- Some other popular data types such as FLOAT, TIME, TIME-STAMP, etc

3.1.2 Data Type Constraints

In addition to specifying data types, SQL allows us to define **constraints** on columns. Constraints are rules enforced by the database system to maintain the **correctness**, **consistency**, and **integrity** of the data. Unlike data types, which describe the kind of values a column can store, constraints describe the conditions that values must satisfy.

- `NOT NULL`: By default, a column may contain the special value `NULL`, representing missing or unknown information. The `NOT NULL` constraint prevents a column from storing `NULL` values.
- `PRIMARY KEY`: The `PRIMARY KEY` constraint uniquely identifies each row in a table. A primary key enforces two rules automatically:
  - values must be **unique**,
  - values must be **not NULL**.
- `UNIQUE`: The `UNIQUE` constraint ensures that all values in a column are distinct, but unlike `PRIMARY KEY`, it typically allows `NULL` values. A table may contain multiple `UNIQUE` constraints, but only one `PRIMARY KEY`. `NULL` is not considered equal to `NULL` in SQL's three-valued logic. Therefore, multiple rows may contain `NULL` in a `UNIQUE` column.

- `DEFAULT`: The `DEFAULT` constraint provides a default value when no value is supplied during insertion.
- `CHECK`: The `CHECK` constraint restricts values based on a logical condition.
- Multiple constraints may be applied to a single column.
- When an `INSERT` or `UPDATE` statement violates a constraint, the database rejects the operation and raises an error. This guarantees that invalid data cannot be stored, even if application-level checks are missing.

3.1.3 PRIMARY KEY AND FOREIGN KEY

Primary Key A primary key is a column or a set of columns that uniquely identifies each row in a table, enforcing uniqueness and non-nullability, and providing a stable identifier that other tables can safely reference.

Foreign Key A foreign key is a constraint declared on a column or group of columns in one table that requires every value in those columns to already exist as a primary key or unique key value in another table, thereby maintaining referential integrity.

Basic Foreign Key Declaration A foreign key is declared during table creation using a table-level constraint that specifies the referencing column and the referenced primary key column.

```
CREATE TABLE Enrollment (
    student_id INT,
    course_id  VARCHAR(20),
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id)
        REFERENCES Student(student_id)
);
```

Because a primary key is unique and non-null, the database system can guarantee that each foreign key value refers to exactly one row in the referenced table, eliminating ambiguity. When a foreign key constraint exists, the database rejects any insertion into the child table if the referenced primary key value does not already exist in the parent table, preventing orphan records.

Deletion and Update Rules Foreign keys may specify referential actions such as `ON DELETE CASCADE`, `ON DELETE RESTRICT`, `ON DELETE SET NULL`, or `ON UPDATE CASCADE`, which determine how changes to primary key rows propagate.

```
FOREIGN KEY (student_id)
REFERENCES Student(student_id)
ON DELETE CASCADE
ON UPDATE CASCADE
```

The `ON DELETE CASCADE` rule instructs the database to automatically delete all child table rows whose foreign key values reference a primary key row that is being deleted, which is appropriate when child records have no meaning without their parent.

```
FOREIGN KEY (student_id)
REFERENCES Student(student_id)
ON DELETE CASCADE
```

The `ON DELETE RESTRICT` and `NO ACTION` rules prevent deletion of a primary key row if any child table rows still reference it, forcing the user to explicitly resolve dependent records and thereby avoiding accidental data loss.

```
FOREIGN KEY (student_id)
REFERENCES Student(student_id)
ON DELETE RESTRICT
```

ON DELETE SET NULL The `ON DELETE SET NULL` rule sets the foreign key column in the child table to `NULL` when the referenced primary key row is deleted, which is only valid when the foreign key column allows null values and is useful when the relationship is optional rather than mandatory.

```
FOREIGN KEY (advisor_id)
REFERENCES Professor(professor_id)
ON DELETE SET NULL
```

ON UPDATE CASCADE The `ON UPDATE CASCADE` rule automatically propagates updates of primary key values to all referencing foreign key rows, ensuring consistency in schemas where primary key values may change, though such updates are generally discouraged in well-designed systems.

```
FOREIGN KEY (student_id)
REFERENCES Student(student_id)
ON UPDATE CASCADE
```

In practice, `CASCADE` is commonly used for strong ownership relationships, for example, in an `Order`–`OrderItem` relationship, each order item has no meaning without its parent order, so using `ON DELETE CASCADE` ensures that when an order is deleted, all associated order items are automatically removed, preventing orphan records. `RESTRICT` for safety-critical data, for example, in a `Student`–`Transcript` or `Employee`–`Payroll` relationship, `ON DELETE RESTRICT` is appropriate because deleting a student or employee while historical academic or payroll records still exist would violate legal, financial, or institutional requirements. `SET NULL` for weak or optional associations, while frequent primary key updates should generally be avoided in favor of stable surrogate keys, for example, in a `Professor`–`Student` advising relationship, a student may temporarily have no advisor, so using `ON DELETE SET NULL` allows the advisor reference to be cleared when a professor leaves without deleting or blocking the student record.

Composite Foreign Keys When the referenced table uses a composite primary key, the foreign key must reference the same set of columns in the same order to preserve tuple-level correspondence.

```
FOREIGN KEY (course_id, section_id)
REFERENCES CourseSection(course_id, section_id)
```

Post-Creation Declaration Foreign keys can be added after table creation using `ALTER TABLE` together with `ADD CONSTRAINT`, and explicitly naming constraints improves schema readability and debugging.

```
ALTER TABLE Enrollment
ADD CONSTRAINT enrollment_student
FOREIGN KEY (student_id)
REFERENCES Student(student_id);
```

3.2 Data Definition Language (DDL) and Data Manipulation Language (DML)

SQL provides commands to **define**, **modify**, and **remove** database objects and records. These commands are commonly categorized into:

- **Data Definition Language (DDL)**: `CREATE`, `DROP`, `ALTER`
- **Data Manipulation Language (DML)**: `INSERT`, `DELETE`, `UPDATE`

In this section, we illustrate these commands using the `Student` table in the `college_db` database.

The `CREATE` statement is used to create new database objects such as databases and tables. For example, the following statement creates the `Student` table.

```
CREATE TABLE Student (
    student_id  INT PRIMARY KEY,
    name        VARCHAR(80) NOT NULL,
    major       VARCHAR(50),
    gpa         DECIMAL(3,2)
                    CHECK (gpa >= 0.0 AND gpa <= 4.0),
    enroll_date DATE NOT NULL,
    credits     INT NOT NULL DEFAULT 0,
    city        VARCHAR(60)
);
```

Each column is defined with a data type, and constraints such as `PRIMARY KEY` and `NOT NULL` are used to enforce data integrity.

The `DROP` statement permanently removes a database object. For example, the following statement deletes the `Student` table.

```
DROP TABLE Student;
```

Because `DROP` irreversibly deletes both the table structure and all stored data, it should be used with caution. To avoid errors when the table may not exist, the `IF EXISTS` clause is often used.

```
DROP TABLE IF EXISTS Student;
```

The `ALTER` statement modifies the schema of an existing table. Common uses include adding, removing, or changing columns.

For example, to add a new column `email` to the `Student` table, we can write:

```
ALTER TABLE Student ADD COLUMN email VARCHAR(120);
```

The new attribute will have NULLs in all the tuples of the relation right after the command is executed; hence, the `NOT NULL` constraint is not allowed for such an attribute.

To remove the column later:

```
ALTER TABLE Student DROP COLUMN email;
```

Schema changes affect all existing rows and should therefore be carefully planned.

The `INSERT` statement adds new rows to a table. To insert a single student record, we specify values for each column.

```
INSERT INTO Student
(student_id, name, major, gpa, enroll_date, credits, city)
VALUES
(1011, 'Emily Wong', 'Mathematics', 3.65, '2025-08-25', 30, 'Chicago');
```

SQL allows multiple rows to be inserted using a single `INSERT` statement. This is both more efficient and more readable than issuing many individual inserts.

```
INSERT INTO Student (student_id, name, major, enroll_date, credits)
VALUES
(1011, 'Emily Wong', 'Mathematics', 3.65, '2025-08-25', 30, 'Chicago'),
(1011, 'Ivan Patel', 'Mathematics', 3.75, '2025-08-25', 30, 'Crawfordsville');
```

Each parenthesized tuple represents one row. Bulk inserts offer several advantages:

- Fewer round-trips between application and database
- Better performance for large datasets
- Atomic execution (either all rows insert or none, if wrapped in a transaction)

This is especially important when loading initial data or importing files.

In SQL, a column may be defined with a `DEFAULT` value or as an `AUTO_INCREMENT` column. When inserting a new row, such columns do not need to be explicitly listed in the `INSERT` statement. This design greatly simplifies insertion logic and helps maintain data consistency.

Let's add `AUTO_INCREMENT` constraint to our `student_id` :

```
ALTER TABLE Student
MODIFY student_id INT AUTO_INCREMENT;
```

The column keeps its primary key constraint, existing values are preserved, and new rows will auto-generate IDs.

- `student_id` is a surrogate primary key generated automatically.
- `credits` defaults to `0` if not provided.
- `gpa` defaults to `NULL`.

When a column has a default value or is auto-generated, it may be safely omitted from the `INSERT` statement:

```
INSERT INTO Student (name, gpa, enroll_date)
VALUES ('Grace Kim', 3.5, '2025-08-25');
```

This approach:

- Reduces errors caused by manually assigning IDs
- Improves schema evolution (adding defaults later does not break old inserts)
- Is standard practice in real-world database systems

Using `AUTO_INCREMENT` is most common for surrogate keys, which:

- Have no business meaning
- Never change
- Are used only to uniquely identify rows

In contrast, natural keys (such as email or SSN) may change and are generally avoided as primary keys.

The `DELETE` statement removes rows from a table. For example, to delete a student with a specific ID:

```
DELETE FROM Student WHERE student_id = 1012;
```

If the `WHERE` clause is omitted, **all rows** in the table will be deleted.

```
DELETE FROM Student;
```

For this reason, it is considered best practice to always include a `WHERE` clause when using `DELETE`.

The `UPDATE` statement modifies existing rows. For example, to update the GPA of a specific student:

```
UPDATE Student
SET gpa = 3.85
WHERE student_id = 1001;
```

Multiple columns can be updated simultaneously.

```
UPDATE Student
SET major = 'Computer Science',
    city = 'San Mateo'
WHERE name = 'Emily Wong';
```

As with `DELETE`, omitting the `WHERE` clause will update **every row** in the table.

```
UPDATE Student
SET credits = credits + 3;
```

This statement increases the `credits` value for all students.

Both `DELETE` and `UPDATE` without `WHERE` are considered unsafe, the database with safe mode enabled will not allow you do this. You can disable the safe mode in settings and restart the connect if you want to test SQLs.

3.3 SELECT

Relational databases organize data into **tables** (also called **relations**). Each table has a name and a fixed schema consisting of **columns** (also called **attributes**). Each row in the table represents a single record and is called a **tuple**. The `Student` table in `college_db` is listed below:

student_id	name	major	gpa	enroll_date	credits	city
1001	Alice Zhang	Computer Science	3.72	2025-08-25	32	San Diego
1002	Brian Lee	Mathematics	3.10	2025-08-25	18	Santa Ana
1003	Carla Gomez	NULL	NULL	2024-08-26	28	Fresno
1004	David Kim	Computer Science	3.88	2025-01-10	36	San Jose

```sql
INSERT INTO Student (student_id, name, major, gpa, enroll_date, credits, city)
VALUES
  (1001, 'Alice Zhang', 'Computer Science', 3.72, '2025-08-25', 32, 'San Diego'),
  (1002, 'Brian Lee', 'Mathematics', 3.10, '2025-08-25', 18, 'Santa Ana'),
  (1003, 'Carla Gomez', NULL, NULL, '2024-08-26', 28, 'Fresno'),
  (1004, 'David Kim', 'Computer Science', 3.88, '2025-01-10', 36, 'San Jose');
```

The columns of this table are `student_id`, `name`, `major`, `gpa`, `enroll_date`, `credits`, and `city`. Each row corresponds to one student enrolled in the college.

The most fundamental SQL query has the following structure.

```sql
SELECT <columns> FROM <table>;
```

The `FROM` clause specifies which table we are querying, and the `SELECT` clause specifies which columns we want to display.

For example, to retrieve each student's name, major, and GPA from the `Student` table, we can write the following query.

```sql
SELECT name, major, gpa FROM Student;
```

The result would be a table such as the following.

name	major	gpa
Alice Zhang	Computer Science	3.72
Brian Lee	Mathematics	3.10
Carla Gomez	NULL	NULL
David Kim	Computer Science	3.88

The wildcard `*` can be used to select all columns from a table.

```sql
SELECT * FROM Student;
```

This query returns every column for every row in the `Student` table. While useful for quick inspection or debugging, `SELECT *` is generally discouraged in production code. Explicitly listing column names improves readability, reduces unnecessary data transfer, and avoids unexpected behavior if the table schema changes.

The `AS` keyword allows columns or expressions in the result to be renamed.

```sql
SELECT name AS student_name, gpa AS GPA FROM Student;
```

Aliases only affect the output of the query and do not change the underlying table schema.

student_name	GPA
Alice Zhang	3.72
Brian Lee	3.10
Carla Gomez	NULL
David Kim	3.88

The `SELECT` clause may contain expressions, not just column names. These expressions are evaluated once per row.

```sql
SELECT name, credits * 3 AS estimated_hours FROM Student;
```

Computed columns exist only in the query result and are not stored in the database.

name	estimated_hours
Alice Zhang	96
Brian Lee	54
Carla Gomez	84
David Kim	108

SQL provides functions to replace missing values in query results. In MySQL, the `IFNULL` function can be used to substitute a default value when an attribute is `NULL`.

```
SELECT name, IFNULL(major, 'GEN') AS major_display
FROM Student;
```

name	major_display
Alice Zhang	Computer Science
Brian Lee	Mathematics
Carla Gomez	GEN
David Kim	Computer Science

Replacing `NULL` values in a query result does not modify the underlying table. The original `major` attribute for Carla Gomez remains `NULL` in the database.

The `DISTINCT` keyword removes duplicate rows from the output. For example, if we want to see which majors are represented in the database, we can use the following query.

```
SELECT DISTINCT major FROM Student;
```

major
Computer Science
Mathematics
NULL

This query returns each major exactly once, even if many students share the same major. The `DISTINCT` keyword can apply to multiple columns, and it means the entire row:

```
SELECT DISTINCT major, city FROM Student;
```

major	city
Computer Science	San Diego
Mathematics	Santa Ana
NULL	Fresno
Computer Science	San Jose

In SQL, the order of rows in a query result is not guaranteed unless an `ORDER BY` clause is explicitly specified. The `ORDER BY` clause may include multiple attributes.

```
SELECT name, major, gpa FROM Student ORDER BY gpa DESC;
```

name	major	gpa
David Kim	Computer Science	3.88
Alice Zhang	Computer Science	3.72
Brian Lee	Mathematics	3.10
Carla Gomez	NULL	NULL

You can set different orders on different columns, the example below shows that rows are first ordered by major, and ties are broken by GPA.

```
SELECT name, major, gpa
FROM Student
ORDER BY major ASC, gpa DESC;
```

name	major	gpa
David Kim	Computer Science	3.88
Alice Zhang	Computer Science	3.72
Brian Lee	Mathematics	3.10
Carla Gomez	NULL	NULL

Often, we are only interested in a subset of records. The `WHERE` clause allows us to **filter rows** based on a condition called a **predicate**.

```
SELECT <columns> FROM <table> WHERE <predicate>;
```

For example, to retrieve only students who major in Computer Science, we can write the following query.

```
SELECT student_id, name FROM Student WHERE major = 'Computer Science';
```

student_id	name
1001	Alice Zhang
1004	David Kim

More complex conditions can be constructed using the boolean operators `NOT`, `AND`, and `OR`. For example, to find students whose GPA is above 3.5 and who are majoring in Computer Science, we can write:

```
SELECT name, gpa FROM Student
WHERE major = 'Computer Science' AND gpa > 3.8;
```

name	gpa
David Kim	3.88

SQL supports a special value called `NULL`, which represents missing or unknown information. Any comparison involving `NULL` evaluates to `NULL`. To test for missing values explicitly, the `IS NULL` and `IS NOT NULL` operators must be used.

```
SELECT name FROM Student WHERE major IS NULL;
```

name
Carla Gomez

The `BETWEEN` operator checks whether a value lies within an inclusive range.

```
SELECT name, gpa
FROM Student
WHERE gpa BETWEEN 3.0 AND 3.8;
```

name	gpa
Alice Zhang	3.72
Brian Lee	3.10

The `IN` operator tests membership in a finite set of values.

```
SELECT name
FROM Student
WHERE city IN ('San Diego', 'San Jose');
```

name
Alice Zhang
David Kim

SQL provides built-in aggregate functions such as `COUNT`, `AVG`, `MIN`, `MAX`, and `SUM`. For example, to compute the average GPA of all students:

```
SELECT AVG(gpa) FROM Student;
```

AVG(gpa)
3.57

When summaries are needed per category, the `GROUP BY` clause is used. For example, to count how many students belong to each major:

```sql
SELECT major, COUNT(*) AS num_students
FROM Student
GROUP BY major;
```

major	num_students
Computer Science	2
Mathematics	1
NULL	1

The `HAVING` clause filters groups after aggregation. For example, to display only majors with more than one student:

```sql
SELECT major, COUNT(*) AS num_students
FROM Student
GROUP BY major
HAVING COUNT(*) > 1;
```

major	num_students
Computer Science	2

The `LIMIT` clause restricts the number of rows returned.

```sql
SELECT student_id, name FROM Student LIMIT 1;
```

student_id	name
1001	Alice Zhang

`OFFSET` skips a specified number of rows.

```sql
SELECT name FROM Student ORDER BY gpa DESC LIMIT 2 OFFSET 1;
```

name
Alice Zhang
Brian Lee

3.4 Logical Evaluation Order of SQL Queries

Although SQL queries are written in a particular syntactic order, the database system logically evaluates the clauses in a different order. Understanding this evaluation order is essential for correctly using `WHERE`, `GROUP BY`, `HAVING`, etc. For a query of the form:

```sql
SELECT <columns>
FROM <table>
WHERE <row predicate>
GROUP BY <columns>
HAVING <group predicate>
ORDER BY <columns>
LIMIT <n>;
```

SQL evaluates the clauses in the following logical order:

Step	Clause Evaluated
1	`FROM`
2	`WHERE`
3	`GROUP BY`
4	Aggregate functions (`COUNT`, `AVG`, ...)
5	`HAVING`
6	`SELECT`
7	`ORDER BY`
8	`LIMIT`

Each step operates on the result produced by the previous step.

3.4.1 Row Filtering vs. Group Filtering

The key distinction between `WHERE` and `HAVING` lies in what kind of data they are allowed to filter.

- `WHERE` filters **individual rows** before grouping.
- `HAVING` filters **groups of rows** after aggregation.

Consider the `Student` table. If we want to restrict the input to only Computer Science students, we use `WHERE`:

```sql
SELECT name, gpa
FROM Student
WHERE major = 'Computer Science';
```

1. `FROM`: start with all rows in `Student`.
2. `WHERE`: keep only rows where `major = 'Computer Science'`.
3. `SELECT`: output the requested columns.

At the time `WHERE` is evaluated, each row still represents a single student.

Now consider grouping students by major:

```sql
SELECT major, COUNT(*) AS num_students
FROM Student
GROUP BY major;
```

After the `GROUP BY` step, the intermediate result conceptually looks like:

major	num_students
Computer Science	2
Mathematics	1
NULL	1

At this point, each row represents an entire group, not an individual student.

Suppose we want to keep only majors with more than one student. This condition depends on `COUNT(*)`, which is computed after grouping.

```sql
SELECT major, COUNT(*) AS num_students
FROM Student
GROUP BY major
HAVING COUNT(*) > 1;
```

The `FROM Student` clause defines the initial input relation containing all rows in the table.

student_id	name	major	gpa	enroll_date	credits	city
1001	Alice Zhang	Computer Science	3.72	2025-08-25	32	San Diego
1002	Brian Lee	Mathematics	3.10	2025-08-25	18	Santa Ana
1003	Carla Gomez	NULL	NULL	2024-08-26	28	Fresno
1004	David Kim	Computer Science	3.88	2025-01-10	36	San Jose

The `WHERE gpa IS NOT NULL` clause removes rows whose GPA value is `NULL` before grouping.

student_id	name	major	gpa	enroll_date	credits	city
1001	Alice Zhang	Computer Science	3.72	2025-08-25	32	San Diego
1002	Brian Lee	Mathematics	3.10	2025-08-25	18	Santa Ana
1004	David Kim	Computer Science	3.88	2025-01-10	36	San Jose

The `GROUP BY major` clause partitions the remaining rows into groups sharing the same major.

major	students in group
Computer Science	Alice Zhang, David Kim
Mathematics	Brian Lee
NULL	Carla Gomez

Rows no longer exist individually; Each group becomes a single logical unit.

The aggregate function `COUNT(*)` computes the number of rows within each group.

major	COUNT(*)
Computer Science	2
Mathematics	1
NULL	1

The `HAVING COUNT(*) > 1` clause filters groups based on aggregate values rather than individual rows.

major	COUNT(*)
Computer Science	2

SELECT The `SELECT` clause projects the final attributes and assigns an alias to the aggregate result.

major	num_students
Computer Science	2

The `WHERE` clause is evaluated before grouping occurs, so aggregate functions such as `COUNT(*)` do not yet exist.

```
WHERE COUNT(*) > 1   -- illegal
```

The `WHERE` clause filters individual rows rather than groups, and aggregate values are only defined after the `GROUP BY` stage.

The `HAVING` clause is intended for filtering groups and should not be used for row-level conditions. In a grouped query, any column referenced in `HAVING` must either be part of the `GROUP BY` list or be wrapped in an aggregate function.

```
HAVING gpa IS NOT NULL   -- illegal
```

The below one is correct:

```
HAVING COUNT(*) > 1 AND AVG(gpa) IS NOT NULL;
```

even though the below SQL is syntactically correct, but is conceptually misleading:

```
HAVING major IS NOT NULL   -- conceptual error
```

3.5 Pattern Match

In many SQL queries, exact string matching using the equality operator (`=`) is too restrictive. Instead, we often want to retrieve rows whose text values follow a certain pattern, such as names that start with a particular letter or cities that share a common prefix.

In MySQL, pattern matching is primarily supported through the `LIKE` operator. MySQL also provides more powerful, regular-expression-based pattern matching using the `REGEXP` operator.

3.5.1 Pattern Matching with `LIKE`

The `LIKE` operator is the standard mechanism for pattern matching in MySQL. It is used in the `WHERE` clause to compare a string value against a pattern.

- % matches zero or more characters
- _ matches exactly one character

Prefix Matching:

```sql
SELECT name
FROM Student
WHERE name LIKE 'A%';
```

name
Alice Zhang

Suffix Matching:

```sql
SELECT name
FROM Student
WHERE name LIKE '%e';
```

name
Brian Lee

Substring Matching:

```sql
SELECT name
FROM Student
WHERE name LIKE '%an%';
```

name
Alice Zhang
Brian Lee

Single-Character Wildcard:

```sql
SELECT name
FROM Student
WHERE name LIKE '_____ Lee';
```

name
Brian Lee

In MySQL, the behavior of `LIKE` depends on the column collation. Pattern matching is **case-insensitive.** As a result, patterns such as `'alice%'` still match `Alice Zhang`.

To match literal wildcard characters, MySQL supports the `ESCAPE` keyword.

```sql
SELECT name
FROM Student
WHERE name LIKE 'A\_%' ESCAPE '\\';
```

`\_ -> literal _`

The escape character itself appears inside a SQL string literal and therefore must be escaped. Within a SQL string literal, a backslash must be written as \\ so that the resulting string value contains a single backslash.

```sql
'\\'   -- string whose value is \
```

If a single backslash is used, the SQL parser misinterprets it as escaping the closing quote.

```sql
ESCAPE '\'
```

The sequence \' is treated as an escaped quote, resulting in a syntax error. The literal '\\' is required to represent a single backslash character inside a SQL string.

If you want to avoid this confusion for students, you can choose a different escape character, such as !.

```sql
SELECT name
FROM Student
WHERE name LIKE 'A!_%' ESCAPE '!';
```

3.5.2 Regular Expression Matching with `REGEXP`

MySQL also supports pattern matching using regular expressions via the `REGEXP` operator. This mechanism is more powerful than `LIKE` but is MySQL-specific and typically slower.

- `^` — beginning of string
- `$` — end of string
- `[abc]` — any one of the listed characters
- `[^abc]` — any character except those listed
- `+` — one or more repetitions
- `|` — alternation (logical OR)
- Inside [...], a hyphen - defines a range of characters based on their ordering.

Names Starting with the Letter A:

```sql
SELECT name
FROM Student
WHERE name REGEXP '^A';
```

name
Alice Zhang

Names Ending with Lee or Kim:

```sql
SELECT name
FROM Student
WHERE name REGEXP '(Lee|Kim)$';
```

name
Brian Lee
David Kim

Names with Exactly One Space:

```sql
SELECT name
FROM Student
WHERE name REGEXP '^[A-Za-z]+ [A-Za-z]+$';
```

`^`	start of the string
`[A-Za-z]+`	first word (letters only)
`(space)`	exactly one space
`[A-Za-z]+`	second word (letters only)
`$`	end of the string

name
Alice Zhang
Brian Lee
Carla Gomez
David Kim

Names Not Starting with A:

```sql
SELECT name
FROM Student
```

```
WHERE name REGEXP '^[^A]';
```

name
Brian Lee
Carla Gomez
David Kim

- `LIKE` is simpler, portable, and usually more efficient
- `REGEXP` is more expressive but MySQL-specific
- Most applications should prefer `LIKE` unless complex patterns are required

3.6 Aggregation

Aggregation in SQL allows us to summarize multiple rows of data into meaningful numerical results using aggregate functions, and in MySQL these operations form the foundation of reporting, analytics, and data-driven decision making.

MySQL provides five core aggregate functions: `AVG`, `SUM`, `MIN`, `MAX`, and `COUNT`, all of which operate on a set of rows and return a single value. To demonstrate aggregation clearly, we use a simplified table that records student grades across multiple courses, which naturally supports grouping and summarization.

```
CREATE TABLE Grade (
    student_id INT NOT NULL,
    course_id  VARCHAR(20) NOT NULL,
    score      INT,
    PRIMARY KEY (student_id, course_id)
);
```

student_id	course	score
1001	CSC101	88
1001	CSC102	92
1002	CSC101	75
1002	CSC102	80
1003	CSC101	NULL
1003	CSC102	85

```
INSERT INTO Grade (student_id, course_id, score)
VALUES
    (1001, 'CSC101', 88),
    (1001, 'CSC102', 92),
    (1002, 'CSC101', 75),
    (1002, 'CSC102', 80),
    (1003, 'CSC101', NULL),
    (1003, 'CSC102', 85);
```

3.6.1 Summary Queries

A summary query applies an aggregate function to an entire table and returns a single summarized result.

`AVG`:

```
SELECT AVG(score) AS avg_score FROM Grade;
```

avg_score
84.0

`SUM`

```
SELECT SUM(score) AS total_score FROM Grade;
```

total_score
420

`MIN and MAX`. MySQL allows multiple aggregate functions to appear in the same `SELECT` clause, enabling richer summary statistics in a single query.

```
SELECT MIN(score) AS min_score, MAX(score) AS max_score FROM Grade;
```

min_score	max_score
75	92

The `COUNT` function supports `*` and column counting, and in MySQL `COUNT(*)` counts all rows while `COUNT(column)` counts only non-`NULL` values.

```
SELECT COUNT(*) AS total_rows FROM Grade;
```

total_rows
6

```
SELECT COUNT(score) AS non_null_scores FROM Grade;
```

non_null_scores
5

The `WHERE` clause filters individual rows before aggregation occurs, which is useful when you want the summary to reflect only a subset of rows.

```
SELECT AVG(score) AS avg_score
FROM Grade
WHERE score > 85;
```

avg_score
90.0

The `GROUP BY` clause partitions rows into groups so that aggregate functions are applied separately to each group.

```
SELECT student_id, AVG(score) AS avg_score
FROM Grade
GROUP BY student_id;
```

student_id	avg_score
1001	90.0
1002	77.5
1003	85.0

When `GROUP BY` is used, every column in the `SELECT` list must either appear in the `GROUP BY` clause or be wrapped in an aggregate function.

The `HAVING` clause filters groups after aggregation and cannot be replaced by `WHERE` when aggregate conditions are involved.

```
SELECT student_id, AVG(score) AS avg_score
FROM Grade
GROUP BY student_id
HAVING AVG(score) > 85;
```

student_id	avg_score
1001	90.0

The `WHERE` clause filters rows before grouping, while the `HAVING` clause filters groups after aggregation, and understanding this distinction is essential for correct query design.

MySQL allows multiple aggregate conditions to be combined using logical operators in the `HAVING` clause to express more complex group-level constraints.

```
SELECT course, COUNT(*) AS count_students, AVG(score) AS avg_score
FROM Grade
```

```
GROUP BY course
HAVING COUNT(*) > 1 AND AVG(score) > 80;
```

course	count_students	avg_score
CSC101	3	81.5
CSC102	3	85.7

The `WITH ROLLUP` modifier adds summary rows to grouped query results, allowing MySQL to compute subtotals and grand totals in the same result set.

```
SELECT course, COUNT(*) AS num_records
FROM Grade
GROUP BY course WITH ROLLUP;
```

course	num_records
CSC101	3
CSC102	3
NULL	6

3.6.2 Window Functions

Window functions extend aggregation by allowing aggregate values to be computed without collapsing rows. Unlike `GROUP BY`, which reduces multiple rows into a single output row per group, a window function computes an aggregate for each row while still preserving the original row-level detail. This is achieved through the `OVER` clause, which defines a window: the set of rows that participate in the calculation for each output row.

The `OVER` clause controls how a window function operates and may include up to three components:

- `PARTITION BY` — divides rows into independent groups (similar to `GROUP BY`, but without collapsing rows),
- `ORDER BY` — defines the order of rows within each partition,
- a frame specification (`ROWS` or `RANGE`) — determines which rows relative to the current row are included in the calculation.

Conceptually, a window function answers the question: "For this row, which other rows should be considered when computing the aggregate?"

To illustrate window functions clearly, we use the following `Sales` table, which contains multiple customers and duplicate ordering values so that the effects of partitioning, ordering, and frame specifications are visible.

```
CREATE TABLE Sales (
    sale_id   INT PRIMARY KEY,
    customer  VARCHAR(50) NOT NULL,
    sale_date DATE NOT NULL,
    amount    INT NOT NULL
);
```

sale_id	customer	sale_date	amount
1	Alice	2024-01-01	100
2	Alice	2024-01-01	50
3	Alice	2024-01-02	70
4	Bob	2024-01-01	40
5	Bob	2024-01-02	60

```
INSERT INTO Sales (sale_id, customer, sale_date, amount)
VALUES
    (1, 'Alice', '2024-01-01', 100),
    (2, 'Alice', '2024-01-01', 50),
    (3, 'Alice', '2024-01-02', 70),
    (4, 'Bob',   '2024-01-01', 40),
    (5, 'Bob',   '2024-01-02', 60);
```

Window Without `PARTITION BY`: If no `PARTITION BY` clause is specified, the window consists of all rows in the table, so the aggregate value is computed over the entire dataset and repeated for every row.

```sql
SELECT sale_id, customer, sale_date, amount,
       AVG(amount) OVER() AS overall_avg
FROM Sales;
```

sale_id	customer	sale_date	amount	overall_avg
1	Alice	2024-01-01	100	64.0
2	Alice	2024-01-01	50	64.0
3	Alice	2024-01-02	70	64.0
4	Bob	2024-01-01	40	64.0
5	Bob	2024-01-02	60	64.0

Window With `PARTITION BY`: The `PARTITION BY` clause divides rows into groups, and the aggregate is computed separately within each group while still returning one row per original record.

```sql
SELECT sale_id, customer, sale_date, amount,
       AVG(amount) OVER(PARTITION BY customer) AS customer_avg
FROM Sales;
```

sale_id	customer	sale_date	amount	customer_avg
1	Alice	2024-01-01	100	73.33
2	Alice	2024-01-01	50	73.33
3	Alice	2024-01-02	70	73.33
4	Bob	2024-01-01	40	50.0
5	Bob	2024-01-02	60	50.0

Ordered Windows and Running Totals: When `ORDER BY` is included in the `OVER` clause, the window function becomes order-sensitive and can compute cumulative or running aggregates.

```sql
SELECT sale_id, customer, sale_date, amount,
       SUM(amount) OVER(PARTITION BY customer ORDER BY sale_date) AS running_total
FROM Sales;
```

sale_id	customer	sale_date	amount	running_total
1	Alice	2024-01-01	100	150
2	Alice	2024-01-01	50	150
3	Alice	2024-01-02	70	220
4	Bob	2024-01-01	40	40
5	Bob	2024-01-02	60	100

Explicit Frame: `ROWS`: The `ROWS` frame defines the window in terms of physical row positions, so the running total increases one row at a time even when ordering values repeat.

```sql
SELECT sale_id, customer, sale_date, amount,
       SUM(amount) OVER (
         PARTITION BY customer
         ORDER BY sale_date
         ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
       ) AS running_total_rows
FROM Sales;
```

`ROWS` means the window frame is defined by row positions, not by values. `UNBOUNDED PRECEDING` means: Start from the very first row in the partition. `CURRENT ROW` means: Stop at this row in the ordered sequence.

Alice's Running Total:

sale_date	amount	rows included	running_total_rows
2024-01-01	100	row 1	100
2024-01-01	50	rows 1–2	150
2024-01-02	70	rows 1–3	220

Bob's Running Total:

sale_date	amount	rows included	running_total_rows
2024-01-01	40	row 1	40
2024-01-02	60	rows 1–2	100

sale_id	customer	sale_date	amount	running_total_rows
1	Alice	2024-01-01	100	100
2	Alice	2024-01-01	50	150
3	Alice	2024-01-02	70	220
4	Bob	2024-01-01	40	40
5	Bob	2024-01-02	60	100

Explicit Frame: `RANGE`: The `RANGE` frame defines the window based on ordering values rather than physical rows, so all rows with the same ordering value as the current row are included together.

```
SELECT sale_id, customer, sale_date, amount,
       SUM(amount) OVER (
         PARTITION BY customer
         ORDER BY sale_date
         RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
       ) AS running_total_range
FROM Sales;
```

`RANGE` groups rows with the same ORDER BY value and treats them as happening at the same time.

sale_id	customer	sale_date	amount	running_total_range
1	Alice	2024-01-01	100	150
2	Alice	2024-01-01	50	150
3	Alice	2024-01-02	70	220
4	Bob	2024-01-01	40	40
5	Bob	2024-01-02	60	100

Named windows allow window definitions to be declared once and reused within a query, which improves readability, reduces repetition, and makes complex window queries easier to maintain. A named window does not change the semantics of the query; it simply gives a descriptive name to a window specification that can be referenced by multiple window functions.

Sharing a Named Window Across Multiple Window Functions: Named windows are especially useful when several window functions use the same partitioning or ordering logic. In this case, the window definition is written once and then reused, ensuring consistency across all window calculations.

Using the `Sales` table, we define a named window that partitions rows by customer. We then compute multiple window aggregates over the same window, including the average sale amount, total sales, and the number of sales per customer.

```
SELECT sale_id, customer, sale_date, amount,
       AVG(amount)   OVER customer_window AS avg_amount,
       SUM(amount)   OVER customer_window AS total_amount,
       COUNT(*)      OVER customer_window AS num_sales
FROM Sales
WINDOW customer_window AS (PARTITION BY customer);
```

sale_id	customer	sale_date	amount	avg_amount	total_amount	num_sales
1	Alice	2024-01-01	100	73.33	220	3
2	Alice	2024-01-01	50	73.33	220	3
3	Alice	2024-01-02	70	73.33	220	3
4	Bob	2024-01-01	40	50.0	100	2
5	Bob	2024-01-02	60	50.0	100	2

The `WINDOW` clause defines a named window called `customer_window` that partitions rows by customer. Each window function (`AVG`, `SUM`, and `COUNT`) is evaluated independently but over the same set of rows defined by the named window, producing consistent per-customer statistics that are attached to every row. Why This Is Better Than Repeating the Window? Without a named window, the `PARTITION BY customer` clause would need to be repeated for each window function, increasing verbosity and the risk of inconsistencies if the query is later modified. Named windows make analytical queries shorter, clearer, and easier to maintain. This query is logically equivalent to repeating `OVER (PARTITION BY customer)` for each window function, but the named-window version scales better when queries become more complex. There is no obviously performance improvement compare to the SQL below:

```sql
SELECT sale_id, customer, sale_date, amount,
       AVG(amount)   OVER (PARTITION BY customer) AS avg_amount,
       SUM(amount)   OVER (PARTITION BY customer) AS total_amount,
       COUNT(*)      OVER (PARTITION BY customer) AS num_sales
FROM Sales;
```

3.7 Revising Server

Users are deliberately prevented from writing raw SQL queries. This decision is made not only for security reasons, but also to improve usability. Just as a user does not need to be a computer scientist to shop on Amazon, a user should not need to know SQL syntax to explore a database. Instead, the goal is to design a universal platform that understands user intent and automatically translates it into correct SQL queries on the server side.

Rather than expressing queries in SQL, users interact with the system through structured interface elements such as dropdown menus, checkboxes, and toggles. Each of these UI components captures a specific piece of information about what the user wants to retrieve, such as which table to query, which columns to display, or whether duplicate rows should be removed. This information is collected in a controlled and predictable format and sent to the server as structured data, rather than as free-form text.

On the server side, the application acts as an interpreter between the user and the database. It takes the structured input provided by the user interface, validates it against the actual database schema, and then constructs a corresponding SQL statement programmatically. Because the server generates the SQL itself, it can enforce important constraints, such as restricting queries to read-only operations, ensuring that only existing tables and columns are referenced, and applying limits on the number of returned rows.

Once the SQL query has been safely constructed and executed, the raw database results are transformed into a format that is easy for users to understand. Rows are returned as dictionaries with meaningful column names, and special values such as SQL `NULL` are handled explicitly and displayed clearly in the web interface. The final output is presented in a readable table, allowing users to focus on understanding the data rather than interpreting low-level database behavior.

This approach reflects a broader principle in database and web application design: users should express what they want, not how to compute it. The platform is responsible for translating user intent into correct, safe, and efficient queries. By separating query formulation from query execution, the system becomes more secure, more robust, and easier to extend, while also providing a better learning experience for users who are new to databases.

3.7.1 Revised db.py

This section explains the updated database layer in `db.py`. Compared with the earliest "minimal" version (which only contained a small set of CRUD helpers), the revised `db.py` adds infrastructure that makes the full-stack project more robust and easier to learn. In particular, the current design contains the following key components:

- `_pool`: a module-level connection pool object (created lazily).
- `_bootstrap_create_db`: ensures the database `college_db` exists before using it.
- `_ensure_pool`: creates the connection pool on demand (so the rest of the code can assume it exists).
- `init_db`: creates the `Student` table (optionally dropping it first) and inserts seed data.
- `run_select`: executes read-only `SELECT` queries safely and returns rows as dictionaries.
- `get_schema`: retrieves schema metadata from `INFORMATION_SCHEMA` for UI generation.

All of these functions bridge the FastAPI server and the MySQL database. Conceptually, they answer three categories of questions:

1. **Connection management:** How do we create and reuse MySQL connections efficiently? (`_pool`, `_ensure_pool`)
2. **Initialization:** How do we reliably create the database and tables for a reproducible demo? (`_bootstrap_create_db`, `init_db`)
3. **Safe exploration:** How do we support query building and schema-driven UI without allowing arbitrary SQL? (`run_select`, `get_schema`)

Configuration and the Pool Handle The database settings are stored in `DB_CONFIG`. These values come from environment variables (with defaults), allowing the same code to run on different machines without modification. In the revised design, `_pool` is stored as a module-level variable and may start as `None`.

```python
_pool = None  # created on demand
```

`_bootstrap_create_db`: Ensure the Database Exists Before a connection pool can be created for a particular database (e.g., `college_db`), that database must exist. The function `_bootstrap_create_db` connects to MySQL without specifying a database, then runs:

```
CREATE DATABASE IF NOT EXISTS college_db
```

```python
def _bootstrap_create_db() -> None:
    cfg = dict(DB_CONFIG)
    dbname = cfg.pop("database")

    conn = mysql.connector.connect(**cfg)  # connect without a DB
    try:
        cur = conn.cursor()
        cur.execute(f"CREATE DATABASE IF NOT EXISTS `{dbname}`")
    finally:
        conn.close()
```

`_ensure_pool`: Lazy Pool Initialization The function `_ensure_pool` guarantees that `_pool` exists before any query is executed. If `_pool` is `None`, it first calls `_bootstrap_create_db` and then creates a `MySQLConnectionPool`. After `_ensure_pool` runs once, the rest of the module can safely borrow connections.

```python
def _ensure_pool() -> mysql.connector.pooling.MySQLConnectionPool:
    global _pool
    if _pool is None:
        _bootstrap_create_db()
        _pool = mysql.connector.pooling.MySQLConnectionPool(
            pool_name="college_pool",
            pool_size=5,
            **DB_CONFIG,
        )
    return _pool
```

`init_db`: Create/Reset the `Student` Table The function `init_db` prepares the schema for the demo. In teaching, it is often convenient to start from a clean state. Therefore, `init_db` may optionally drop the `Student` table first, then recreate it.

The current `Student` schema is:

```sql
CREATE TABLE Student (
    student_id  INT PRIMARY KEY,
    name        VARCHAR(80) NOT NULL,
    major       VARCHAR(50),
    gpa         DECIMAL(3,2)
                    CHECK (gpa >= 0.0 AND gpa <= 4.0),
    enroll_date DATE NOT NULL,
    credits     INT NOT NULL DEFAULT 0,
    city        VARCHAR(60)
);
```

This schema includes `PRIMARY KEY`, `NOT NULL`, a `DEFAULT` value for `credits`, and a `CHECK` constraint for `gpa`. The initialization function may also insert a small set of seed rows so that queries have meaningful data immediately.

```python
def init_db(reset: bool = False, seed: bool = True) -> None:
    pool = _ensure_pool()
    conn = pool.get_connection()
    try:
        cur = conn.cursor()
```

```
        if reset:
            cur.execute("DROP TABLE IF EXISTS Student")

        cur.execute(
            """
            CREATE TABLE IF NOT EXISTS Student (
                student_id  INT PRIMARY KEY,
                name        VARCHAR(80) NOT NULL,
                major       VARCHAR(50),
                gpa         DECIMAL(3,2)
                                CHECK (gpa >= 0.0 AND gpa <= 4.0),
                enroll_date DATE NOT NULL,
                credits     INT NOT NULL DEFAULT 0,
                city        VARCHAR(60)
            )
            """
        )

        if seed:
            cur.execute("SELECT COUNT(*) FROM Student")
            (n,) = cur.fetchone()
            if n == 0:
                cur.executemany(
                    """
                    INSERT INTO Student
                    (student_id, name, major, gpa, enroll_date, credits, city)
                    VALUES (%s, %s, %s, %s, %s, %s, %s)
                    """,
                    [
                        (1001, "Alice Zhang", "Computer Science", 3.82, "2025-08-25", 30, "Los Angeles"),
                        (1002, "Brian Lee", "Mathematics", 3.40, "2025-08-25", 28, "Chicago"),
                        (1003, "Carla Gomez", "Biology", None, "2025-08-25", 0,  "Phoenix"),
                        (1004, "David Kim", "Computer Science", 3.10, "2025-08-25", 16, "Seattle"),
                        (1005, "Eva Patel", None, 3.95, "2025-08-25", 32, None),
                        (1006, "Frank Miller", "History", 2.75, "2025-08-25", 12, "Boston"),
                        (1007, "Grace Chen", "Economics", 3.60, "2025-08-25", 24, "San Diego"),
                        (1008, "Hank Wilson", "Physics", 3.05, "2025-08-25", 20, "Austin"),
                        (1009, "Ivy Nguyen", "Art", None, "2025-08-25", 8,  "Portland"),
                        (1010, "Jack Brown", "Chemistry", 3.25, "2025-08-25", 18, "Denver"),
                    ],
                )
    finally:
        conn.close()
```

`run_select`: Safe Read-Only Query Execution The function `run_select` answers the question: "How do we safely execute a `SELECT` query and return rows to the Python server?" The design goal is that the query builder subsystem can retrieve data without allowing arbitrary database modifications. Therefore, `run_select` is restricted to `SELECT` queries.

```
def run_select(sql: str, params: Optional[tuple] = None):
    pool = _ensure_pool()

    # Defensive restriction: this helper is for SELECT only.
    if not sql.lstrip().upper().startswith("SELECT"):
        raise ValueError("run_select only accepts SELECT queries.")

    conn = pool.get_connection()
    try:
        cur = conn.cursor(dictionary=True)
        cur.execute(sql, params or ())
        return cur.fetchall()
    finally:
        conn.close()
```

The purpose of `run_select` is to return query results as a list of dictionaries. This makes downstream code simpler because each row is self-describing. For example, a row may look like:

```
{
  "student_id": 1001,
  "name": "Alice Zhang",
  "major": "Computer Science",
  "gpa": 3.82,
  "enroll_date": "2025-08-25",
  "credits": 30,
  "city": "Los Angeles"
}
```

Both `run_select` and `get_schema` use the same resource management pattern:

1. borrow a connection from the pool,
2. create a cursor (dictionary cursor for readable rows),
3. execute SQL,
4. fetch results (if any),
5. always close the connection in a `finally` block.

Even though we call `conn.close()`, in a pooled setup this typically means "return the connection to the pool" rather than destroying it.

`get_schema`: Discover Tables and Columns for the UI The function `get_schema` answers the question: "How can our server discover what tables and columns exist in the database so the web client can build a user interface automatically?" Rather than hard-coding table names and column names in HTML or JavaScript, the client can ask the server for schema metadata and then generate dropdowns and checkboxes dynamically.

The schema information is stored in MySQL's system database `INFORMATION_SCHEMA`.

In particular, `information_schema.COLUMNS` contains one row per column for every table in every database.

```
def get_schema():
    pool = _ensure_pool()
    conn = pool.get_connection()
    try:
        cur = conn.cursor(dictionary=True)
        cur.execute(
            """
            SELECT
              TABLE_NAME   AS table_name,
              COLUMN_NAME  AS column_name,
              DATA_TYPE    AS data_type,
              IS_NULLABLE  AS is_nullable
            FROM information_schema.COLUMNS
            WHERE TABLE_SCHEMA = %s
            ORDER BY TABLE_NAME, ORDINAL_POSITION
            """,
            (DB_CONFIG["database"],),
        )

        schema = {}
        for row in cur.fetchall():
            t = row["table_name"]
            schema.setdefault(t, []).append({
                "column": row["column_name"],
                "type": row["data_type"],
                "nullable": row["is_nullable"],
            })
        return schema
    finally:
        conn.close()
```

The SQL query inside `get_schema` selects four important attributes:

- `TABLE_NAME`: the name of the table the column belongs to,
- `COLUMN_NAME`: the name of the column,
- `DATA_TYPE`: the column's data type (such as `int`, `varchar`, `decimal`, or `date`),
- `IS_NULLABLE`: whether the column accepts `NULL` values.

The clause `WHERE TABLE_SCHEMA = %s` restricts the search to the current database (for example, `college_db`). The database name is passed as a parameter:

```
(DB_CONFIG["database"],)
```

which is safer than manually embedding strings into SQL.

Finally, the output of `get_schema` is a dictionary mapping table names to lists of column descriptors. For our updated `Student` table, the structure is:

```
{
  "Student": [
    {"column": "student_id",  "type": "int",     "nullable": "NO"},
    {"column": "name",        "type": "varchar", "nullable": "NO"},
    {"column": "major",       "type": "varchar", "nullable": "YES"},
    {"column": "gpa",         "type": "decimal", "nullable": "YES"},
    {"column": "enroll_date", "type": "date",    "nullable": "NO"},
    {"column": "credits",     "type": "int",     "nullable": "NO"},
    {"column": "city",        "type": "varchar", "nullable": "YES"}
  ]
}
```

This is exactly the kind of JSON format that is convenient for the web client: JavaScript can list tables in a dropdown and generate one checkbox per column automatically. As with `run_select`, the `try/finally` ensures that connections are always returned to the pool, preventing resource leaks when the page loads and schema endpoints are called repeatedly.

3.7.2 Revised main.py

This section explains how `main.py` safely converts structured user input into executable SQL queries. The core idea of the project remains the same: users never write SQL directly. Instead, the server interprets user intent, validates it against the database schema, constructs a well-formed SQL statement, executes it, and returns readable results to the client.

Dependencies and Application Setup The server imports FastAPI components (`FastAPI`, `Request`, `Form`, `Body`), response classes (`HTMLResponse`, `RedirectResponse`, `JSONResponse`), and the template engine `Jinja2Templates`. It also mounts a `static` folder so the browser can load `style.css` and `app.js`. :contentReference[oaicite:1]index=1

```
from typing import Any, Dict, List, Optional

from fastapi import FastAPI, Request, Form, Body
from fastapi.responses import HTMLResponse, RedirectResponse, JSONResponse
from fastapi.templating import Jinja2Templates
from fastapi.staticfiles import StaticFiles

from db import (
    fetch_all_students,
    insert_student,
    delete_student,
    get_schema,
    run_select,
    init_db,
)

app = FastAPI(title="College DB Demo (Student table)")
app.mount("/static", StaticFiles(directory="static"), name="static")

templates = Jinja2Templates(directory="templates")
```

`@app.on_event("startup")`: Database Initialization On server startup, the application initializes the database by calling `init_db(seed=True, reset=True)`. This ensures that:

- the required tables exist,
- the demo is reproducible (seed data is inserted),
- an optional reset can drop and recreate tables so the schema is always up-to-date.

This is triggered automatically when FastAPI starts. Note that FastAPI prints a warning that `on_event` is deprecated and recommends using lifespan handlers; however, `on_event("startup")` is still widely used and is sufficient for this teaching project.

```python
@app.on_event("startup")
def startup():
    init_db(seed=True, reset=True)
```

To allow the browser to build the query-builder UI dynamically, the server provides `/api/schema`. This endpoint returns the output of `get_schema()`, which contains table/column/type/nullability metadata. The JavaScript client uses this data to populate the table dropdown and generate checkboxes for each column.

```python
@app.get("/api/schema")
def api_schema():
    return get_schema()
```

The helper function `_qident` quotes SQL identifiers (table names and column names) using MySQL backticks. This prevents syntax errors if an identifier conflicts with reserved keywords or contains special characters. Importantly, `_qident` is used only after the server validates that the identifier appears in the schema; it is not used to quote values.

```python
def _qident(x: str) -> str:
    return f"`{x}`"
```

The function `build_min_query` is the heart of the minimal query builder. It takes:

- `spec`: a structured JSON-like object describing user intent,
- `schema`: schema metadata returned by `get_schema`.

It validates the requested table and columns against the schema and then constructs a safe SQL `SELECT` statement. Supported features include selecting specific columns (or `*`), applying `DISTINCT`, querying exactly one table, and enforcing a fixed `LIMIT 200`.

The function begins by checking that the table exists. It then builds a set of valid columns and validates each selected column. Finally, it assembles the SQL statement : it builds `SELECT` with an optional `DISTINCT`, quotes identifiers using `_qident`, and applies `LIMIT 200`.

```python
def build_min_query(spec: Dict[str, Any], schema: Dict[str, List[Dict[str, str]]]) -> str:

    table = spec.get("table")
    if not table or table not in schema:
        raise ValueError("Invalid table.")

    colset = {c["column"] for c in schema[table]}
    distinct = bool(spec.get("distinct", False))
    select_cols: List[str] = spec.get("select") or []

    for c in select_cols:
        if c not in colset:
            raise ValueError(f"Invalid select column: {c}")

    select_part = ", ".join(_qident(c) for c in select_cols) if select_cols else "*"

    sql = "SELECT "
    if distinct:
        sql += "DISTINCT "
    sql += select_part
    sql += f" FROM {_qident(table)}"
    sql += " LIMIT 200"
    return sql
```

```
24
```

The endpoint `/api/min_query` receives structured JSON from the web client (the query builder). Its workflow is:

1. retrieve schema metadata using `get_schema()`,
2. build SQL using `build_min_query(payload, schema)`,
3. execute the query using `run_select(sql)`,
4. return `columns`, `rows`, and `row_count` as JSON.

The endpoint handles invalid user requests (unknown table/column) by returning an HTTP 400 error with a clear message, and unexpected database failures by returning an HTTP 500 response.

```python
@app.post("/api/min_query")
def api_min_query(payload: Dict[str, Any] = Body(...)):
    schema = get_schema()
    try:
        sql = build_min_query(payload, schema)
        rows = run_select(sql)
        columns = list(rows[0].keys()) if rows else []
        return {"columns": columns, "rows": rows, "row_count": len(rows)}
    except ValueError as e:
        return JSONResponse(status_code=400, content={"error": str(e)})
    except Exception as e:
        return JSONResponse(status_code=500, content={"error": f"Database error: {e}"})
```

3.8 Revising the Web Client

In this project, we built a small web interface for interacting with a MySQL database using FastAPI. In the earliest version, the HTML structure, CSS styles, and JavaScript behavior were all written in a single HTML file. While this approach can work for a quick demo, it is **not a good practice** for real projects and it also makes the code harder to understand and maintain.

To improve clarity, maintainability, and learning value, we follow the modern web-development principle of **Separation of Concerns** and split the client into **three separate files**:

- **HTML** (`templates/index.html`) — structure and content
- **CSS** (`static/style.css`) — appearance and layout
- **JavaScript** (`static/app.js`) — behavior and interaction

After splitting, our project structure looks like:

```
project/
|-- main.py
|-- db.py
|-- templates/
|   |-- index.html
|-- static/
    |-- style.css
    |-- app.js
```

FastAPI serves:

- HTML templates from the `templates/` directory (rendered by Jinja2),
- static assets (CSS/JS) from the `static/` directory (mounted by FastAPI).

3.8.1 Purpose of HTML

HTML defines the structure and content of a web page: headings, forms, tables, buttons, and placeholders for dynamic content. It does not control appearance in detail (that is CSS), and it does not fetch schema data or execute queries (that is JavaScript + the server).

Because our app now uses separate static files, the template includes a stylesheet link near the top:

```
<link rel="stylesheet" href="/static/style.css">
```

This tells the browser to request `/static/style.css` from the server.

At the bottom of the HTML template, we load JavaScript:

```
<script src="/static/app.js"></script>
```

Loading JavaScript at the end is a common pattern: it ensures the DOM (HTML elements) already exists before JavaScript tries to access elements by `id`.

Our database table is now:

```
Student(student_id, name, major, gpa, enroll_date, credits, city)
```

Therefore, the HTML form for inserting a student must provide values that match the database constraints:

- `student_id` is required and must be unique (`PRIMARY KEY`),
- `name` is required (`NOT NULL`),
- `enroll_date` is required (`NOT NULL`),
- `credits` is required but has a database default of 0 (`NOT NULL DEFAULT 0`),
- `major`, `gpa`, `city` are optional and may be stored as `NULL`.

In particular, `credits` caused a common beginner bug: if the input field is left blank, the browser sends an empty string `""`, and FastAPI cannot parse `""` as an integer. For this reason, the form should either (1) require `credits`, or (2) provide a client-side default such as `value="0"`. The same logic applies to `gpa` if we want a default display value.

A simplified form structure might look like:

```
<form method="post" action="/students">
  <input name="student_id" type="number" placeholder="Student ID" required>
  <input name="name" placeholder="Name" required>
  <input name="major" placeholder="Major (optional)">
  <input name="gpa" type="number" step="0.01" placeholder="GPA (optional)">
  <input name="enroll_date" type="date" required>
  <input name="credits" type="number" value="0" required>
  <input name="city" placeholder="City (optional)">
  <button type="submit">Add</button>
</form>
```

Jinja2 is a server-side template engine. It lets FastAPI render `index.html` by inserting database rows into the HTML before the page is sent to the browser. A loop such as:

```
{% for s in students %}
  {{ s.name }}
{% endfor %}
```

means: for each student record `s` returned from the database, substitute its fields into the HTML.

3.8.2 Understanding `id` and JavaScript DOM Access

Many parts of the page are interactive and depend on retrieving elements by `id`. For example, the minimal query builder uses:

```
const container = $("qbSelectCols");
```

This does not create an element. It retrieves an existing HTML element that was already defined in the template, such as:

```
<div id="qbSelectCols" class="chips"></div>
```

When the browser loads the page, it builds the Document Object Model (DOM). The DOM stores all elements in memory. JavaScript then accesses those elements by their unique `id` values.

3.8.3 Purpose of CSS

CSS controls the visual appearance of the page: fonts, margins, spacing, borders, and layout. CSS does not query the database, does not generate HTML content, and does not handle user actions. It only changes how existing elements look.

In our project, CSS defines reusable classes such as `card` and `chips`. For example, the `card` class creates the boxed UI sections:

```
.card {
  border: 1px solid #ddd;
  border-radius: 10px;
  padding: 1rem;
  margin-bottom: 1rem;
}
```

Any element with `class="card"` will receive this consistent style.

When the Student table grew to include more columns, horizontal overflow became more likely. Therefore, the CSS includes (or should include) a wrapper such as:

```
.table-wrap { overflow-x: auto; }
```

This ensures that wide tables remain usable on smaller screens by allowing horizontal scrolling instead of breaking layout.

3.8.4 Purpose of JavaScript

JavaScript provides interactivity and dynamic behavior. In our project, `static/app.js` implements the minimal query builder, which allows users to explore database tables without typing SQL. The user selects a table and columns, optionally chooses `DISTINCT`, and the browser sends a structured JSON request to the server. The server then validates the request and generates SQL on behalf of the user.

On page load, the client requests schema metadata:

```
const resp = await fetch("/api/schema");
SCHEMA = await resp.json();
```

The server returns a structure like:

```
{
  "Student": [
    {"column":"student_id","type":"int","nullable":"NO"},
    {"column":"name","type":"varchar","nullable":"NO"},
    {"column":"major","type":"varchar","nullable":"YES"},
    {"column":"gpa","type":"decimal","nullable":"YES"},
    {"column":"enroll_date","type":"date","nullable":"NO"},
    {"column":"credits","type":"int","nullable":"NO"},
    {"column":"city","type":"varchar","nullable":"YES"}
  ]
}
```

This is why the query builder automatically shows the new columns: it does not hard-code them. It discovers them from the database.

After schema is loaded, JavaScript fills in the dropdown menu and generates column checkboxes inside:

```
<div id="qbSelectCols" class="chips"></div>
```

The function `renderColumnCheckboxes()` performs DOM manipulation: it creates `<label>` and `<input type="checkbox">` elements programmatically. Each checkbox stores its column name in `dataset.col`, which can be retrieved later when the user clicks `Apply`.

When the user clicks `Apply`, the client constructs a structured payload:

```
const payload = {
  table: $("qbTable").value,
  distinct: $("qbDistinct").checked,
  select: collectSelectedColumns()
};
```

Then it sends this payload to:

```
fetch("/api/min_query", {
  method: "POST",
  headers: { "Content-Type":"application/json" },
  body: JSON.stringify(payload)
});
```

The server validates the table/column names against the live schema and then constructs a safe SQL query with `LIMIT 200`. Because users never send raw SQL, the system remains safer and more predictable.

The server responds with:

- `columns`: the output column names,
- `rows`: a list of dictionaries (each dictionary is a row),
- `row_count`: the number of rows returned.

JavaScript renders this into an HTML table using `renderTable(columns, rows)`. During rendering, SQL `NULL` values are displayed explicitly as the string `NULL`, making missing data visible to learners.

Finally, modify `main.py` to expose local static files (CSS/JS) under the "/static" URL path:

```
app.mount("/static", StaticFiles(directory="static"), name="static")
```

A useful way to remember the web-client control flow is:

$$\texttt{Page loads} \rightarrow \texttt{init() fetches schema} \rightarrow \texttt{dropdown filled} \rightarrow \texttt{checkboxes generated}$$

Later:

$$\texttt{User clicks Apply} \rightarrow \texttt{applyBuilder() sends JSON} \rightarrow \texttt{server builds SQL} \rightarrow \texttt{browser renders results}$$

Overall, the revised web client behaves like a small database exploration tool: users describe what they want using UI controls, the client sends structured intent to the server, and the server translates that intent into safe SQL and returns readable results.

4 Joins & SubQueries

4.1 Joins

In previous chapter, we only queried data from a single table. In practice, however, the information needed to answer a real question is often distributed across multiple tables. To combine related data from two tables, we use a **join**.

Throughout this section, we will use our Student table, and to demonstrate joins, we also introduce a second table that records course enrollments:

```
CREATE TABLE Enrollment (
    student_id INT,
    course_id  VARCHAR(20),
    PRIMARY KEY (student_id, course_id)
);
```

What does `PRIMARY KEY (student_id, course_id)` mean? This declaration means that the combination of `student_id` and `course_id` uniquely identifies each row in the table. Neither column by itself is required to be unique; uniqueness is enforced only on the pair together. Such a key is called a **composite primary key** (also known as a **compound primary key**).

- A student can take many courses.
- A course can have many students.
- A student can enroll in the same course only once.

The rule that a student may enroll in a given course only once is enforced by the composite primary key `(student_id, course_id)`.Below is a sample `Enrollment` table:

```
INSERT INTO Enrollment (student_id, course_id)
VALUES
  (1001, 'CSC211'),
  (1002, 'CSC241'),
  (9999, 'CSC400');
```

4.1.1 Cross Join

The simplest type of join is the **cross join**, also known as a **Cartesian product**. A cross join combines every row from the left table with every row from the right table, without applying any matching condition.

A cross join can be written by listing both tables in the `FROM` clause:

```
SELECT *
FROM Student, Enrollment;
```

Suppose the `Student` table contains:

student_id	name	major	gpa	enroll_date	credits	city
1001	Alice Zhang	Computer Science	3.82	2025-08-25	30	Los Angeles
1002	Brian Lee	Mathematics	3.40	2025-08-25	28	Chicago
1003	Carla Gomez	Biology	NULL	2025-08-25	0	Phoenix

And the `Enrollment` table contains:

student_id	course_id
1001	CSC211
1002	CSC241
9999	CSC400

The result of the cross join is:

student_id	name	major	gpa	enroll_date	credits	city	student_id	course_id
1001	Alice Zhang	Computer Science	3.82	2025-08-25	30	Los Angeles	1001	CSC211
1001	Alice Zhang	Computer Science	3.82	2025-08-25	30	Los Angeles	1002	CSC241
1001	Alice Zhang	Computer Science	3.82	2025-08-25	30	Los Angeles	9999	CSC400
1002	Brian Lee	Mathematics	3.40	2025-08-25	28	Chicago	1001	CSC211
1002	Brian Lee	Mathematics	3.40	2025-08-25	28	Chicago	1002	CSC241
1002	Brian Lee	Mathematics	3.40	2025-08-25	28	Chicago	9999	CSC400
1003	Carla Gomez	Biology	NULL	2025-08-25	0	Phoenix	1001	CSC211
1003	Carla Gomez	Biology	NULL	2025-08-25	0	Phoenix	1002	CSC241
1003	Carla Gomez	Biology	NULL	2025-08-25	0	Phoenix	9999	CSC400

This Cartesian product contains many rows that do not make sense conceptually, because information from different students is combined arbitrarily. To ensure that each row refers to the same student, we add a join condition that matches rows with the same `student_id`.

```
SELECT *
FROM Student, Enrollment
WHERE Student.student_id = Enrollment.student_id;
```

This produces:

student_id	name	major	gpa	enroll_date	credits	city	student_id	course_id
1001	Alice Zhang	Computer Science	3.82	2025-08-25	30	Los Angeles	1001	CSC211
1002	Brian Lee	Mathematics	3.40	2025-08-25	28	Chicago	1002	CSC241

Notice that Carla Gomez does not appear in the result because she has no matching row in the `Enrollment` table, and the enrollment `(9999, CSC400)` does not appear because there is no matching row in the `Student` table.

4.1.2 Inner Join

The **INNER JOIN** syntax makes the join condition explicit and easier to read by placing it in the `ON` clause.

```
SELECT *
FROM Student
INNER JOIN Enrollment
ON Student.student_id = Enrollment.student_id;
```

This query is logically equivalent to the previous query. An inner join returns only rows that satisfy the join condition in both tables.

4.1.3 Left Outer Join

A **left outer join** guarantees that every row from the left table appears in the result. If there is no matching row in the right table, the missing values are filled with `NULL`.

```
SELECT *
FROM Student
LEFT OUTER JOIN Enrollment
ON Student.student_id = Enrollment.student_id;
```

Result:

student_id	name	major	gpa	enroll_date	credits	city	student_id	course_id
1001	Alice Zhang	Computer Science	3.82	2025-08-25	30	Los Angeles	1001	CSC211
1002	Brian Lee	Mathematics	3.40	2025-08-25	28	Chicago	1002	CSC241
1003	Carla Gomez	Biology	NULL	2025-08-25	0	Phoenix	NULL	NULL

4.1.4 Right Outer Join

A **right outer join** guarantees that every row from the right table appears in the result. If there is no matching row in the left table, the missing values are filled with `NULL`.

```
SELECT *
FROM Student
RIGHT OUTER JOIN Enrollment
```

```
ON Student.student_id = Enrollment.student_id;
```

In this query, all rows from the `Enrollment` table (the table on the right side of the join) are preserved.

Result:

student_id	name	major	gpa	enroll_date	credits	city	student_id	course_id
1001	Alice Zhang	Computer Science	3.82	2025-08-25	30	Los Angeles	1001	CSC211
1002	Brian Lee	Mathematics	3.40	2025-08-25	28	Chicago	1002	CSC241
NULL	NULL	NULL	NULL	NULL	NULL	NULL	9999	CSC400

4.1.5 Full Outer Join

A **full outer join** returns all rows from both tables. Rows without matches in the other table still appear, with missing attributes filled with `NULL`.

```
SELECT *
FROM Student
FULL OUTER JOIN Enrollment
ON Student.student_id = Enrollment.student_id;
```

Result:

student_id	name	major	gpa	enroll_date	credits	city	student_id	course_id
1001	Alice Zhang	Computer Science	3.82	2025-08-25	30	Los Angeles	1001	CSC211
1002	Brian Lee	Mathematics	3.40	2025-08-25	28	Chicago	1002	CSC241
1003	Carla Gomez	Biology	NULL	2025-08-25	0	Phoenix	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	9999	CSC400

4.1.6 Why Do We Still Use `INNER JOIN`?

At first glance, it may appear that an `INNER JOIN` is unnecessary, since the same result can be obtained by placing the join condition in the `WHERE` clause:

```
SELECT *
FROM Student, Enrollment
WHERE Student.student_id = Enrollment.student_id;
```

Result:

student_id	name	major	gpa	enroll_date	credits	city	student_id	course_id
1001	Alice Zhang	Computer Science	3.82	2025-08-25	30	Los Angeles	1001	CSC211
1002	Brian Lee	Mathematics	3.40	2025-08-25	28	Chicago	1002	CSC241

For inner joins only, this query is logically equivalent to the explicit join syntax:

```
SELECT *
FROM Student
INNER JOIN Enrollment
ON Student.student_id = Enrollment.student_id;
```

Result:

student_id	name	major	gpa	enroll_date	credits	city	student_id	course_id
1001	Alice Zhang	Computer Science	3.82	2025-08-25	30	Los Angeles	1001	CSC211
1002	Brian Lee	Mathematics	3.40	2025-08-25	28	Chicago	1002	CSC241

Although the two queries above return the same result, `INNER JOIN` is still necessary and preferred for several important reasons.

- the `ON` clause specifies how tables are related (join logic),
- the `WHERE` clause specifies which rows to keep (filtering logic).

This separation becomes critical once additional conditions or outer joins are introduced.

Consider the following query:

```sql
SELECT *
FROM Student
LEFT OUTER JOIN Enrollment
ON Student.student_id = Enrollment.student_id
WHERE Enrollment.course_id = 'CSC211';
```

Result:

student_id	name	major	gpa	enroll_date	credits	city	student_id	course_id
1001	Alice Zhang	Computer Science	3.82	2025-08-25	30	Los Angeles	1001	CSC211

Although this query uses a `LEFT OUTER JOIN`, the condition in the `WHERE` clause removes rows where `Enrollment.course_id` is `NULL`. As a result, students without enrollments are eliminated, and the query behaves like an inner join.

If the intended meaning is to keep all students and show only their enrollment in `CSC211` when it exists, the condition must be placed in the `ON` clause instead:

```sql
SELECT *
FROM Student
LEFT OUTER JOIN Enrollment
ON Student.student_id = Enrollment.student_id
AND Enrollment.course_id = 'CSC211';
```

Result:

student_id	name	major	gpa	enroll_date	credits	city	student_id	course_id
1001	Alice Zhang	Computer Science	3.82	2025-08-25	30	Los Angeles	1001	CSC211
1002	Brian Lee	Mathematics	3.40	2025-08-25	28	Chicago	NULL	NULL
1003	Carla Gomez	Biology	NULL	2025-08-25	0	Phoenix	NULL	NULL

This query preserves the intended left outer join semantics.

4.1.7 Name Conflicts and Aliases

Up to this point, our tables have used different column names, so there was no ambiguity when referring to attributes. However, it is very common in practice for different tables to contain columns with the same name.

In our schema, both the `Student` table and the `Enrollment` table contain a column named `student_id`. If we write a query that refers to `student_id` without qualification, SQL cannot determine which table we mean.

To resolve this ambiguity, we must qualify the column name using the table name, followed by a period.

Qualified column names. For example, the following inner join explicitly specifies which `student_id` column belongs to which table:

```sql
SELECT *
FROM Student
INNER JOIN Enrollment
ON Student.student_id = Enrollment.student_id;
```

Result:

student_id	name	major	gpa	enroll_date	credits	city	student_id	course_id
1001	Alice Zhang	Computer Science	3.82	2025-08-25	30	Los Angeles	1001	CSC211
1002	Brian Lee	Mathematics	3.40	2025-08-25	28	Chicago	1002	CSC241

Table aliases. Repeatedly writing full table names can be verbose, especially in queries involving many tables. SQL allows us to assign a **table alias**, which acts as a short name for the table within the query.

```sql
SELECT *
FROM Student AS s
INNER JOIN Enrollment AS e
```

```
ON s.student_id = e.student_id;
```

This query is logically equivalent to the previous one, but is easier to read and write.

Result:

student_id	name	major	gpa	enroll_date	credits	city	student_id	course_id
1001	Alice Zhang	Computer Science	3.82	2025-08-25	30	Los Angeles	1001	CSC211
1002	Brian Lee	Mathematics	3.40	2025-08-25	28	Chicago	1002	CSC241

Renaming output columns. Aliases can also be used in the `SELECT` clause to rename columns in the output, which is especially useful when different tables contain columns with the same name.

```
SELECT
    s.student_id AS sid,
    s.name,
    e.student_id AS eid,
    e.course_id
FROM Student AS s
INNER JOIN Enrollment AS e
ON s.student_id = e.student_id;
```

Result:

sid	name	eid	course_id
1001	Alice Zhang	1001	CSC211
1002	Brian Lee	1002	CSC241

4.1.8 Natural Join

In many relational schemas, the columns used for joining have the same name in both tables. SQL provides a **natural join**, which automatically performs an equijoin on all columns with identical names.

In our case, both tables contain the column `student_id`, so the following query is equivalent to an explicit inner join on that column:

```
SELECT *
FROM Student NATURAL JOIN Enrollment;
```

Here, the join condition `Student.student_id = Enrollment.student_id` is implicit.

Result:

student_id	name	major	gpa	enroll_date	credits	city	course_id
1001	Alice Zhang	Computer Science	3.82	2025-08-25	30	Los Angeles	CSC211
1002	Brian Lee	Mathematics	3.40	2025-08-25	28	Chicago	CSC241

Although natural joins are concise, they are rarely used in practice. Because the join condition is implicit, adding or renaming columns in a table can silently change the behavior of the query, making it harder to read, debug, and maintain.

4.2 UNION

The SQL `UNION` clause combines the results of multiple `SELECT` queries into **one single result set** by **stacking rows**. Each `SELECT` is executed independently, and then the outputs are merged vertically.

- `UNION` **removes duplicates** (like `DISTINCT`).
- `UNION ALL` **keeps duplicates**.

For `UNION` to work:

1. **Same number of columns** in each `SELECT`.
2. **Compatible data types** column-by-column (by position).
3. **Column names** in the final output come from the **first** `SELECT`.

Sometimes we want to see every ID that appears either in `Student` or in `Enrollment`.

```sql
SELECT student_id
FROM Student
UNION
SELECT student_id
FROM Enrollment;
```

student_id
1001
1002
1003
1004
9999

Explanation: `1001` and `1002` appear in both tables, but `UNION` returns them only once.

Using `UNION ALL` to keep duplicates):

```sql
SELECT student_id
FROM Student
UNION ALL
SELECT student_id
FROM Enrollment;
```

student_id
1001
1002
1003
1004
1001
1002
9999

Explanation: now the duplicates `1001` and `1002` show up twice.

A very common pattern is to union two lists that share a common schema and add a label to show where each row came from.

```sql
SELECT student_id, name, 'StudentTable' AS source
FROM Student
UNION
SELECT student_id, NULL AS name, 'EnrollmentTable' AS source
FROM Enrollment;
```

Why does this work?

- Both `SELECT`s output exactly **3 columns**.
- Column 2 is text in both outputs (`name` vs `NULL` which is compatible).
- The final column names are: `student_id`, `name`, `source` (from the first `SELECT`).

student_id	name	source
1001	Alice Zhang	StudentTable
1002	Brian Lee	StudentTable
1003	Carla Gomez	StudentTable
1004	David Kim	StudentTable
1001	NULL	EnrollmentTable
1002	NULL	EnrollmentTable
9999	NULL	EnrollmentTable

Note: `UNION` only removes duplicates if **all columns match**. Here, `1001, Alice Zhang, StudentTable` is not the same as `1001, NULL, EnrollmentTable`, so both remain.

Important rule: **only one** `ORDER BY` is allowed, and it must appear at the **end**.

```sql
SELECT student_id
```

```
FROM Student
UNION ALL
SELECT student_id
FROM Enrollment
ORDER BY student_id;
```

student_id
1001
1001
1002
1002
1003
1004
9999

Each `SELECT` can have its own `WHERE` clause.

```
SELECT student_id
FROM Student
WHERE major = 'Computer Science'
UNION
SELECT student_id
FROM Enrollment
WHERE course_id = 'CSC400';
```

student_id
1001
1004
9999

Again, `UNION` stacks rows. If you want to match students to their enrollments, you need a `JOIN`, not `UNION`.

4.3 Add Inner Join to Project

4.3.1 Add Enrollment Table

In this section, we extend the full-stack project to display and manage the `Enrollment` table on the web interface, in the same way that the `Student` table is currently handled. Inside your database Layer (`db.py`):

Add the following function right after the existing student query helpers (`Right after def fetch_all_students()`):

```
def fetch_all_enrollments():
    conn = _pool.get_connection()
    try:
        cur = conn.cursor(dictionary=True)
        cur.execute(
            """
            SELECT student_id, course_id
            FROM Enrollment
            ORDER BY student_id DESC, course_id ASC
            """
        )
        return cur.fetchall()
    finally:
        conn.close()
```

Add this function near `insert_student`:

```
def insert_enrollment(student_id: int, course_id: str):
    conn = _pool.get_connection()
    try:
        cur = conn.cursor()
        cur.execute(
            """
            INSERT INTO Enrollment (student_id, course_id)
            VALUES (%s, %s)
            """,
            (student_id, course_id),
        )
    finally:
```

```
        conn.close()
```

Add this function near `delete_student`:

```
def delete_enrollment(student_id: int, course_id: str):
    conn = _pool.get_connection()
    try:
        cur = conn.cursor()
        cur.execute(
            "DELETE FROM Enrollment WHERE student_id=%s AND course_id=%s",
            (student_id, course_id),
        )
    finally:
        conn.close()
```

Move to the backend Routes (`main.py`):

Extend the import list at the top of `main.py`:

```
from db import (
    fetch_all_students,
    fetch_all_enrollments,
    insert_enrollment,
    delete_enrollment,
)
```

Modify the `index` route so it sends both students and enrollments to the HTML template:

```
@app.get("/", response_class=HTMLResponse)
def index(request: Request):
    students = fetch_all_students()
    enrollments = fetch_all_enrollments()
    return templates.TemplateResponse(
        "index.html",
        {"request": request, "students": students, "enrollments": enrollments},
    )
```

Add a route for inserting enrollment records:

```
@app.post("/enrollments")
def add_enrollment(
    student_id: int = Form(...),
    course_id: str = Form(...),
):
    insert_enrollment(student_id, course_id)
    return RedirectResponse("/", status_code=303)
```

Add a route for deleting enrollment records:

```
@app.post("/enrollments/{student_id}/{course_id}/delete")
def remove_enrollment(student_id: int, course_id: str):
    delete_enrollment(student_id, course_id)
    return RedirectResponse("/", status_code=303)
```

Now let's move to frontend Template (`index.html`)

Insert the following card after the existing "Students" card to add Enrollment form:

```
<div class="card">
  <h2>Add Enrollment</h2>

  <form method="post" action="/enrollments">
    <div class="row">
      <label>Student ID</label>
      <input name="student_id" type="number" required>
    </div>

    <div class="row">
      <label>Course ID</label>
      <input name="course_id" placeholder="e.g., CSC211" required>
    </div>
```

```
    <div class="row" style="justify-content: flex-end;">
      <button type="submit">Add</button>
    </div>
  </form>

  <p class="muted small" style="margin-top:0.5rem;">
    Note: \texttt{(student\_id, course\_id)} must be unique because it is the composite primary key.
  </p>
</div>
```

Insert the following card after the existing “Students” table to display Enrollment table:

```
<div class="card">
  <h2>Enrollments</h2>

  <div class="table-wrap">
    <table>
      <thead>
        <tr>
          <th>Student ID</th>
          <th>Course ID</th>
          <th>Action</th>
        </tr>
      </thead>
      <tbody>
        {% for e in enrollments %}
        <tr>
          <td>{{ e.student_id }}</td>
          <td>{{ e.course_id }}</td>
          <td>
            <form method="post"
                  action="/enrollments/{{ e.student_id }}/{{ e.course_id }}/delete"
                  style="display:inline;">
              <button type="submit">Delete</button>
            </form>
          </td>
        </tr>
        {% else %}
        <tr>
          <td colspan="3">No enrollments yet.</td>
        </tr>
        {% endfor %}
      </tbody>
    </table>
  </div>
</div>
```

4.3.2 Inner Join Demo

Move to database Layer: `db.py` to Creat and seed the `Enrollment` table. All changes are made inside the function `init_db(reset: bool = False, seed: bool = True)`.

Locate the following code:

```
if reset:
    cur.execute("DROP TABLE IF EXISTS Student")
```

Replace it with to drop tables on reset:

```
if reset:
    cur.execute("DROP TABLE IF EXISTS Enrollment")
    cur.execute("DROP TABLE IF EXISTS Student")
```

The enrollment table must be dropped first because it depends on student data. Immediately after the `CREATE TABLE Student (...)` block, add:

```
cur.execute(
    """
    CREATE TABLE IF NOT EXISTS Enrollment (
        student_id INT,
        course_id  VARCHAR(20),
```

```
        PRIMARY KEY (student_id, course_id)
    )
    """
)
```

Still inside `init_db`, in the student seeding `if seed:` block, add:

```
if seed:
    cur.execute("SELECT COUNT(*) FROM Enrollment")
    (m,) = cur.fetchone()
    if m == 0:
        cur.executemany(
            """
            INSERT INTO Enrollment (student_id, course_id)
            VALUES (%s, %s)
            """,
            [
                (1001, "CSC211"),
                (1002, "CSC241"),
                (9999, "CSC400"),  # orphan enrollment (no matching student)
            ],
        )
```

Let's move to `main.py`:

Add the following endpoint near the other `/api/*` routes (e.g., below `@app.get("/api/schema")` and before the main entry point):

```
@app.get("/api/inner_join")
def api_inner_join():
    sql = """
    SELECT
      s.student_id,
      s.name,
      s.major,
      s.gpa,
      s.enroll_date,
      s.credits,
      s.city,
      e.course_id
    FROM Student AS s
    INNER JOIN Enrollment AS e
      ON s.student_id = e.student_id
    ORDER BY s.student_id, e.course_id
    LIMIT 200
    """
    rows = run_select(sql)
    columns = list(rows[0].keys()) if rows else []
    return {
        "columns": columns,
        "rows": rows,
        "row_count": len(rows),
    }
```

Move to our HTML file, insert the following block anywhere convenient in the page layout (recommended: immediately before or after the "Minimal Query Builder" card) to add an Inner Join demo card:

```
<div class="card">
  <h2>Inner Join Demo: Student Join Enrollment</h2>
  <p class="muted" style="margin-top:0.3rem;">
    This calls <code>/api/inner_join</code> and displays the joined rows
    (capped at 200).
  </p>

  <div class="row" style="justify-content: flex-end;">
    <button type="button" id="joinApply">Run INNER JOIN</button>
    <span id="joinStatus" class="small muted"></span>
  </div>

  <div style="margin-top:0.7rem;">
    <div id="joinError" class="error"></div>
    <div id="joinMeta" class="small muted"></div>
  </div>

```

```
  <div class="divider"></div>

  <div class="table-wrap">
    <table id="joinResultTable" style="display:none;">
      <thead id="joinResultHead"></thead>
      <tbody id="joinResultBody"></tbody>
    </table>
  </div>
</div>
```

Let's go to `static/app.js`. After the existing `clearResults()` function (or near other utilities), add:

```
function clearJoinResults(){
  $("joinError").textContent = "";
  $("joinMeta").textContent = "";
  $("joinResultHead").innerHTML = "";
  $("joinResultBody").innerHTML = "";
  $("joinResultTable").style.display = "none";
}
```

Add the following helper function to rendering the join table:

```
function renderJoinTable(columns, rows){
  const thead = $("joinResultHead");
  const tbody = $("joinResultBody");
  thead.innerHTML = "";
  tbody.innerHTML = "";

  const trh = document.createElement("tr");
  columns.forEach(c => {
    const th = document.createElement("th");
    th.textContent = c;
    trh.appendChild(th);
  });
  thead.appendChild(trh);

  rows.forEach(r => {
    const tr = document.createElement("tr");
    columns.forEach(c => {
      const td = document.createElement("td");
      const v = r[c];
      td.textContent = (v === null || v === undefined) ? "NULL" : v;
      tr.appendChild(td);
    });
    tbody.appendChild(tr);
  });

  $("joinResultTable").style.display = "table";
}
```

Add the following asynchronous function to triggering the inner join query:

```
async function runInnerJoin(){
  clearJoinResults();
  $("joinStatus").textContent = "Running...";

  try{
    const resp = await fetch("/api/inner_join");
    const data = await resp.json();
    if (!resp.ok) throw new Error(data.error || resp.status);

    $("joinMeta").textContent = `Rows: ${data.row_count || 0}`;
    renderJoinTable(data.columns || [], data.rows || []);
    $("joinStatus").textContent = "Done.";
    setTimeout(() => $("joinStatus").textContent = "", 1000);

  } catch(e){
    $("joinStatus").textContent = "";
    $("joinError").textContent = String(e);
  }
}
```

Inside the existing `DOMContentLoaded` handler, add the line below in your existing

`document.addEventListener("DOMContentLoaded", () => { ... })`

block to wire the button event:

```
$("joinApply").addEventListener("click", runInnerJoin);
```

No changes are required for `static/style.css`. The existing card and table styles automatically apply to the new inner join demo.

4.4 Subqueries

Throughout this section, we use the following sample data.

student_id	name	major	gpa	enroll_date	credits	city
1001	Alice Zhang	Computer Science	3.82	2025-08-25	30	Los Angeles
1002	Brian Lee	Mathematics	3.40	2025-08-25	28	Chicago
1003	Carla Gomez	Biology	NULL	2025-08-25	0	Phoenix
1004	David Kim	Computer Science	3.65	2025-08-25	16	San Jose

student_id	course_id
1001	CSC211
1001	CSC241
1002	CSC241
1002	CSC400
1003	CSC211
9999	CSC400

```
INSERT INTO Enrollment (student_id, course_id)
VALUES
  (1001, 'CSC241'),
  (1002, 'CSC400'),
  (1003, 'CSC211');
```

4.4.1 A Simple Subquery

Suppose we want to find all students who appear in the `Enrollment` table.

```
SELECT student_id, name
FROM Student
WHERE student_id IN (
    SELECT student_id
    FROM Enrollment
);
```

The subquery is evaluated first and produces the following set:

```
SELECT student_id
FROM Enrollment
```

student_id
1001
1001
1002
1002
1003
9999

Then each row in `Student` is checked to see whether its `student_id` belongs to this set. The final result is:

student_id	name
1001	Alice Zhang
1002	Brian Lee
1003	Carla Gomez

4.4.2 Subqueries in the FROM Clause (Derived Tables)

Create a temporary table of course enrollment counts and then filter it.

```sql
SELECT t.course_id, t.num_students
FROM (
    SELECT course_id, COUNT(*) AS num_students
    FROM Enrollment
    GROUP BY course_id
) AS t
WHERE t.course_id = 'CSC241';
```

Intermediate Table: derived table t

course_id	num_students
CSC211	2
CSC241	2
CSC400	2

Final Output

course_id	num_students
CSC241	2

Subqueries in FROM behave like temporary tables and must be given an alias.

4.4.3 Subqueries with Aggregation (HAVING)

Find all courses whose enrollment size is greater than or equal to the average enrollment size across all courses.

```sql
SELECT e.course_id
FROM Enrollment ASe
GROUP BY e.course_id
HAVING COUNT(*) >= (
    SELECT AVG(course_cnt) AS avg_cnt
    FROM (
        SELECT COUNT(*) AS course_cnt
        FROM Enrollment
        GROUP BY course_id
    ) AS course_counts
);
```

Intermediate Table 1: enrollment count per course

```sql
    SELECT COUNT(*) AS course_cnt
    FROM Enrollment
    GROUP BY course_id
);
```

course_cnt
2
2
2

Intermediate Table 2: average enrollment size

```sql
SELECT AVG(course_cnt) AS avg_cnt
FROM (
    SELECT COUNT(*) AS course_cnt
    FROM Enrollment
    GROUP BY course_id
) AS course_counts
```

avg_cnt
2.0

AS course_counts is necessary because every derived table (subquery in the **FROM** clause) must have its own alias. Subqueries in WHERE do not need aliases.

Finally, all courses satisfy the condition:

```sql
SELECT e.course_id
FROM Enrollment ASe
GROUP BY e.course_id
HAVING COUNT(*) >= (
    SELECT AVG(course_cnt) AS avg_cnt
    FROM (
        SELECT COUNT(*) AS course_cnt
        FROM Enrollment
        GROUP BY course_id
    ) AS course_counts
);
```

course_id
CSC211
CSC241
CSC400

4.4.4 Correlated Subqueries with `EXISTS`

Return all students who are enrolled in at least one course.

```sql
SELECT s.student_id, s.name
FROM Student s
WHERE EXISTS (
    SELECT 1
    FROM Enrollment e
    WHERE e.student_id = s.student_id
);
```

Conceptual Intermediate Evaluation 1 is dummy data, which means I am not fetching data. I am only checking whether a row exists. It is slightly more efficient than SELECT *.

student_id	name	Matching rows in Enrollment	EXISTS
1001	Alice Zhang	(1001, CSC211), (1001, CSC241)	TRUE
1002	Brian Lee	(1002, CSC241), (1002, CSC400)	TRUE
1003	Carla Gomez	(1003, CSC211)	TRUE

Final Output

student_id	name
1001	Alice Zhang
1002	Brian Lee
1003	Carla Gomez

A correlated subquery is evaluated once per row of the outer query.

4.4.5 Subquery Factoring with `WITH` (Common Table Expressions (CTEs))

Think of WITH as "define a temporary named result first, then use it". It lets you give a subquery a name, so the main query becomes easier to read, reason about, and sometimes reuse.

Find students who are enrolled in more courses than the average student.

```sql
WITH StudentLoads AS (
    SELECT student_id, COUNT(*) AS num_courses
    FROM Enrollment
    GROUP BY student_id
),
AvgLoad AS (
    SELECT AVG(num_courses) AS avg_courses
    FROM StudentLoads
)
SELECT s.student_id, s.name, sl.num_courses
FROM Student s
JOIN StudentLoads sl ON s.student_id = sl.student_id
WHERE sl.num_courses > (SELECT avg_courses FROM AvgLoad);
```

Intermediate Table 1: `StudentLoads`

student_id	num_courses
1001	2
1002	2
1003	1
9999	1

Intermediate Table 2: `AvgLoad`

avg_courses
1.5

Intermediate Table 3: Join with `Student`

student_id	name	num_courses
1001	Alice Zhang	2
1002	Brian Lee	2
1003	Carla Gomez	1

Final Output

student_id	name	num_courses
1001	Alice Zhang	2
1002	Brian Lee	2

Common Table Expressions (CTEs) make intermediate results explicit, readable, and reusable.

4.5 What is an SQL Function?

An SQL function is an expression that takes input values and returns a computed output. In `SELECT`, you often use functions to:

- clean or transform text (string functions),
- compute derived numeric values (numeric functions),
- compute or format dates and times (date/time functions),
- implement logic inside queries (`CASE`),
- compute ranks and running totals (window/ranking functions).

4.5.1 String / Substring Functions

String functions operate on `CHAR`/`VARCHAR`/`TEXT` columns. We will use `name`, `major`, and `city`.

Extract first name and last name using `SUBSTRING_INDEX`. `SUBSTRING_INDEX(str, delim, count)` splits by a delimiter and returns part of the string.

- If `count` is positive, return from the left.
- If `count` is negative, return from the right.

```
SELECT
  student_id,
  name,
  SUBSTRING_INDEX(name, ' ', 1)  AS first_name,
  SUBSTRING_INDEX(name, ' ', -1) AS last_name
FROM Student
ORDER BY student_id;
```

`SUBSTRING_INDEX(name, ' ', 1)` means "Give me everything before the FIRST space".

Extract a substring using `SUBSTRING`. `SUBSTRING(str, start, length)` extracts characters by position (1-indexed in MySQL). Extract the first 3 characters of the city.

```
SELECT
  student_id,
  city,
  SUBSTRING(city, 1, 3) AS city_prefix
FROM Student
```

```
ORDER BY student_id;
```

Standard text cleanup: `UPPER`, `LOWER`, `TRIM`, `CONCAT`.

```
SELECT
  student_id,
  UPPER(name) AS name_upper,
  LOWER(city) AS city_lower,
  CONCAT(TRIM(name), ' (', COALESCE(major, 'Undeclared'), ')') AS display_label
FROM Student
ORDER BY student_id;
```

- UPPER(name) – converts the student's name to all uppercase letters.
- LOWER(city) – converts the city name to all lowercase letters.
- TRIM(name) – removes leading and trailing spaces from the name.
- COALESCE(major, 'Undeclared') – returns the major if it exists; otherwise uses 'Undeclared'.
- CONCAT(...) – joins strings together into a single formatted display label.

Note: `COALESCE(major,'Undeclared')` replaces `NULL` with a readable label.

4.5.2 Numeric Functions

Numeric functions compute derived values from numeric columns such as `gpa` and `credits`.

Rounding and formatting: `ROUND`, `CEIL`, `FLOOR`. Suppose we want to show a rounded GPA and also the ceiling/floor.

```
SELECT
  student_id,
  name,
  gpa,
  ROUND(gpa, 1) AS gpa_round_1,
  CEIL(gpa)     AS gpa_ceiling,
  FLOOR(gpa)    AS gpa_floor
FROM Student
ORDER BY student_id;
```

- ROUND(gpa, 1) – rounds the GPA to one decimal place using standard rounding rules.
- CEIL(gpa) – rounds the GPA up to the nearest integer (ceiling).
- FLOOR(gpa) – rounds the GPA down to the nearest integer (floor).

Compute a derived metric from credits. Example: assume a typical student takes `4` credits per course. Estimate how many courses completed:

```
SELECT
  student_id,
  name,
  credits,
  credits / 4.0 AS est_courses_completed,
  MOD(credits, 4) AS leftover_credits
FROM Student
ORDER BY student_id;
```

- credits / 4.0 – estimates the number of courses completed by dividing total credits by 4 (using 4.0 to force decimal division).
- MOD(credits, 4) – returns the remainder when credits are divided by 4, showing leftover credits not forming a full course.

Note: using `4.0` forces floating-point division.

4.5.3 Date/Time Functions (using `enroll_date`)

Dates are a core DB topic: filtering, grouping, formatting, and computing date differences. Extract year/month/day:

```sql
SELECT
  student_id,
  name,
  enroll_date,
  YEAR(enroll_date)  AS enroll_year,
  MONTH(enroll_date) AS enroll_month,
  DAY(enroll_date)   AS enroll_day
FROM Student
ORDER BY enroll_date, student_id;
```

Date arithmetic: `DATEDIFF` and `DATE_ADD`. `DATEDIFF(a, b)` returns the number of days from `b` to `a`. `DATE_ADD(date, INTERVAL n unit)` shifts a date.

Example: how many days since enrollment (relative to today's date), and compute a 30-day "check-in" date.

```sql
SELECT
  student_id,
  name,
  enroll_date,
  DATEDIFF(CURDATE(), enroll_date) AS days_since_enroll,
  DATE_ADD(enroll_date, INTERVAL 30 DAY) AS checkin_date_30d
FROM Student
ORDER BY student_id;
```

Formatting dates for display: `DATE_FORMAT`.

```sql
SELECT
  student_id,
  name,
  enroll_date,
  DATE_FORMAT(enroll_date, '%b %d, %Y') AS enroll_pretty
FROM Student
ORDER BY student_id;
```

More examples:

- `%Y-%m-%d` → 2026–02–22
- `%b %d, %Y` → Feb 22, 2026
- `%M %e, %Y` → February 22, 2026
- `%Y/%m/%d` → 2026/02/22
- `%W, %b %d, %Y` → Sunday, Feb 22, 2026

4.5.4 `CASE` Expression (SQL "if/else")

`CASE` lets you compute labels and categories inside queries. Categorize students by GPA (handling `NULL`) Rules:

- If GPA is `NULL`, label `'No GPA'`.
- If GPA ≥ 3.7, label `'Honors'`.
- If GPA ≥ 3.0, label `'Good Standing'`.
- Otherwise label `'At Risk'`.

```sql
SELECT
  student_id,
  name,
  gpa,
  CASE
    WHEN gpa IS NULL THEN 'No GPA'
    WHEN gpa >= 3.7 THEN 'Honors'
    WHEN gpa >= 3.0 THEN 'Good Standing'
    ELSE 'At Risk'
  END AS gpa_status
FROM Student
ORDER BY student_id;
```

Compute a custom field using `CASE` with multiple columns. Example: build a short status string based on major and credits.

```sql
SELECT
  student_id,
  name,
  major,
  credits,
  CASE
    WHEN major IS NULL THEN 'Undeclared'
    WHEN major = 'Computer Science' AND credits >= 32 THEN 'CS Upper-Level Track'
    WHEN major = 'Computer Science' THEN 'CS Early Track'
    ELSE 'Non-CS'
  END AS academic_track
FROM Student
ORDER BY student_id;
```

4.5.5 Ranking (Window) Functions

Ranking functions are part of **window functions** (MySQL 8+). They compute values across rows without collapsing rows (unlike `GROUP BY`).

Rank students by GPA using `RANK` and `DENSE_RANK`.

- `RANK()` creates gaps when ties occur (e.g., 1,1,3).
- `DENSE_RANK()` does not create gaps (e.g., 1,1,2).

We typically want to push `NULL` GPAs to the bottom. In MySQL, `ORDER BY gpa DESC` naturally puts `NULL` last in descending order.

```sql
SELECT
  student_id,
  name,
  major,
  gpa,
  RANK()       OVER (ORDER BY gpa DESC) AS gpa_rank,
  DENSE_RANK() OVER (ORDER BY gpa DESC) AS gpa_dense_rank
FROM Student
ORDER BY gpa DESC, student_id;
```

Rank within each major (partitioned ranking). Now we rank within each major using `PARTITION BY`.

```sql
SELECT
  student_id,
  name,
  major,
  gpa,
  RANK() OVER (PARTITION BY major ORDER BY gpa DESC) AS rank_in_major
FROM Student
ORDER BY major, rank_in_major, student_id;
```

A more complicated example: Ranking enrollments by course load (using a join + aggregation + ranking). We can compute "how many courses each student is taking" and then rank students by that count.

Step 1: Count enrollments per student

```sql
SELECT
  e.student_id,
  COUNT(*) AS num_courses
FROM Enrollment e
GROUP BY e.student_id;
```

Step 2: Join to Student and rank by number of courses

```sql
SELECT
  s.student_id,
  s.name,
  COALESCE(x.num_courses, 0) AS num_courses,
  RANK() OVER (ORDER BY COALESCE(x.num_courses, 0) DESC) AS course_load_rank
FROM Student s
LEFT JOIN (
```

```
  SELECT e.student_id, COUNT(*) AS num_courses
  FROM Enrollment e
  GROUP BY e.student_id
) AS x
  ON s.student_id = x.student_id
ORDER BY num_courses DESC, s.student_id;
```

Notes:

- The subquery `x` computes course counts.
- `LEFT JOIN` keeps all students even if they have 0 enrollments.
- `COALESCE` converts `NULL` counts into 0.
- `RANK()` assigns rank based on the computed course count.

5 Disk And Files

Disk Space Management sits at the very bottom of a DBMS software stack. Its job is to manage how database data is laid out on disk and moved between disk and main memory. Typical responsibilities include: (1) translating logical page identifiers into physical disk locations, (2) fetching pages from disk into memory when needed, (3) writing modified pages back to disk safely, and (4) ensuring writes happen reliably and in the correct order when required.

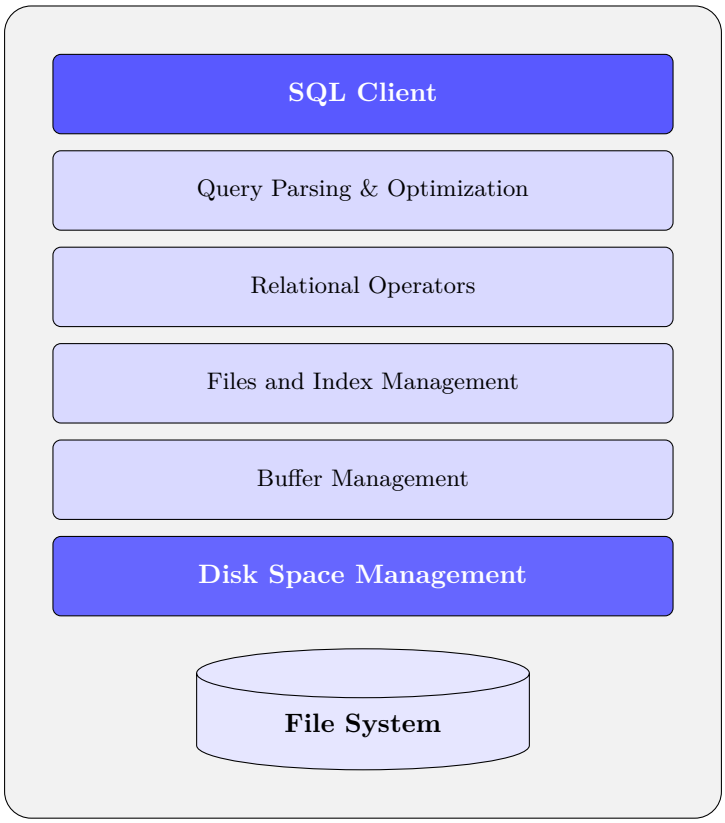


Figure 1: DBMS software stack with emphasis on Disk Space Management.

In a relational database, the smallest meaningful unit of data is a **record** (a row). Records live inside **relations** (tables) and may be created, searched, updated, or deleted during query processing.

On disk, the fundamental unit is the **page**: it is the smallest chunk the DBMS typically reads from disk into memory (and writes back). To store tables on disk, each relation is placed into (at least) one **file**, and the file is divided into pages that hold the records. Depending on the relation's schema and expected access patterns, the DBMS decides:

- what **file organization** to use,
- how **pages** are arranged within the file,
- how **records** are placed within each page,
- and how each **record format** is represented (fixed-length vs. variable-length, etc.).

5.1 Heap File

Two common file organizations are **heap files** and **sorted files**. For each relation, the DBMS selects an organization by estimating I/O cost under the expected workload. Here, **one I/O** typically means **reading one page from disk** or **writing one page back to disk**. The DBMS compares the predicted I/O costs for operations such as inserts, deletes, and scans, and then chooses the file type that yields the lower overall cost for that access pattern.

A **heap file** is a storage organization in which neither the pages in the file nor the records inside each page are maintained in any sorted order. Records are placed wherever free space is available. Heap files are widely used because they support efficient insertion and simple implementation.

Two common implementations of heap files are:

- Linked List Organization
- Page Directory Organization

5.1.1 Linked List Organization

In the linked list organization, every data page stores records along with metadata describing available free space. Each page also maintains pointers (usually byte offsets) to neighboring pages.

A special **header page** acts as the entry point to the file. The header separates data pages into two logical groups:

- Pages that are already full
- Pages that still contain free space

When new space is required, additional empty pages can be allocated and linked into the list of pages with available space. When a page becomes full, it is moved into the full-page list so future insertions avoid scanning unnecessary pages.

Some systems maintain pointers to the end of these lists for faster insertion, but the exact implementation strategy is not critical for understanding heap files.

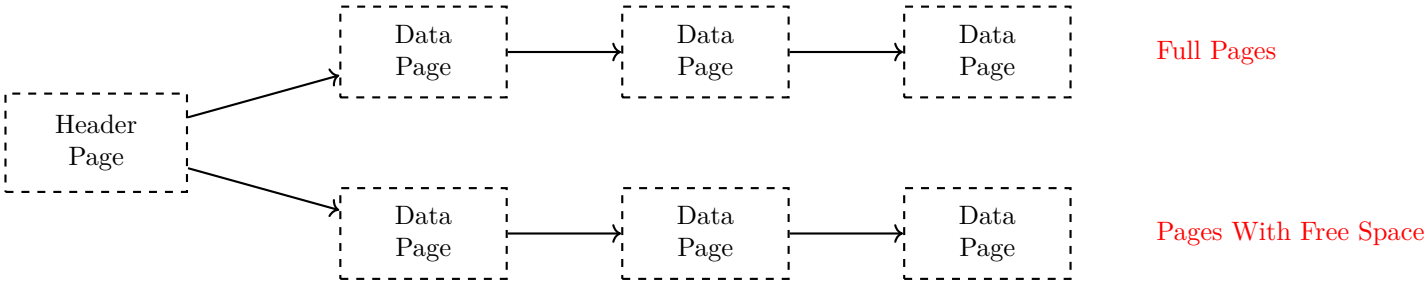


Figure 2: Linked list heap file organization.

5.1.2 Page Directory Organization

The page directory organization improves search efficiency by introducing a directory structure composed only of header pages.

Each header page contains entries that store:

- A pointer to a data page
- The amount of free space available in that page

Header pages themselves are linked together, forming a directory chain. Since free space information is stored in the directory, data pages no longer need to maintain neighbor pointers.

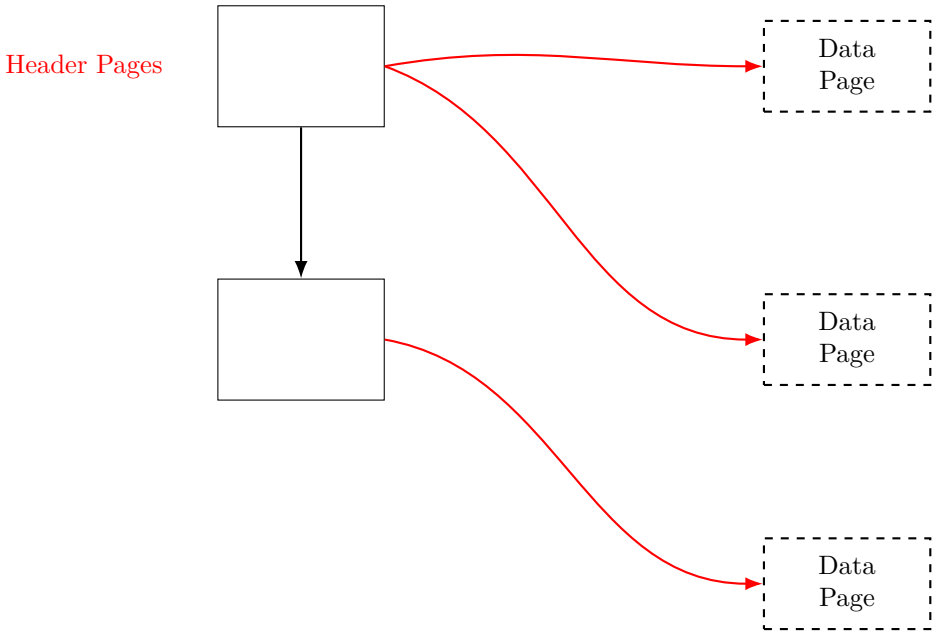


Figure 3: Page directory heap file organization.

5.1.3 Performance Comparison

Page directories typically allow faster insert operations compared to linked list heap files. In a linked list implementation, the system may need to scan multiple pages to locate available space. In contrast, directory entries already track available space, reducing unnecessary page reads. Assume each page has a capacity of 30 bytes, and we insert a 20-byte record.

The numbers below represent the free space remaining in each page.

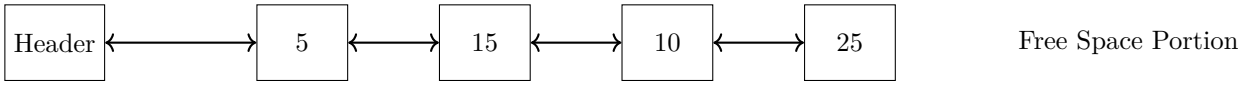


Figure 4: Linked list free space tracking.

After inserting a 20-byte record, the last page is chosen:

Remaining free space: **5 bytes**

I/O Cost:

$$5 \text{ page reads} + 1 \text{ page write} = \textbf{6 I/Os}$$

The figure below represents the file using page directory:

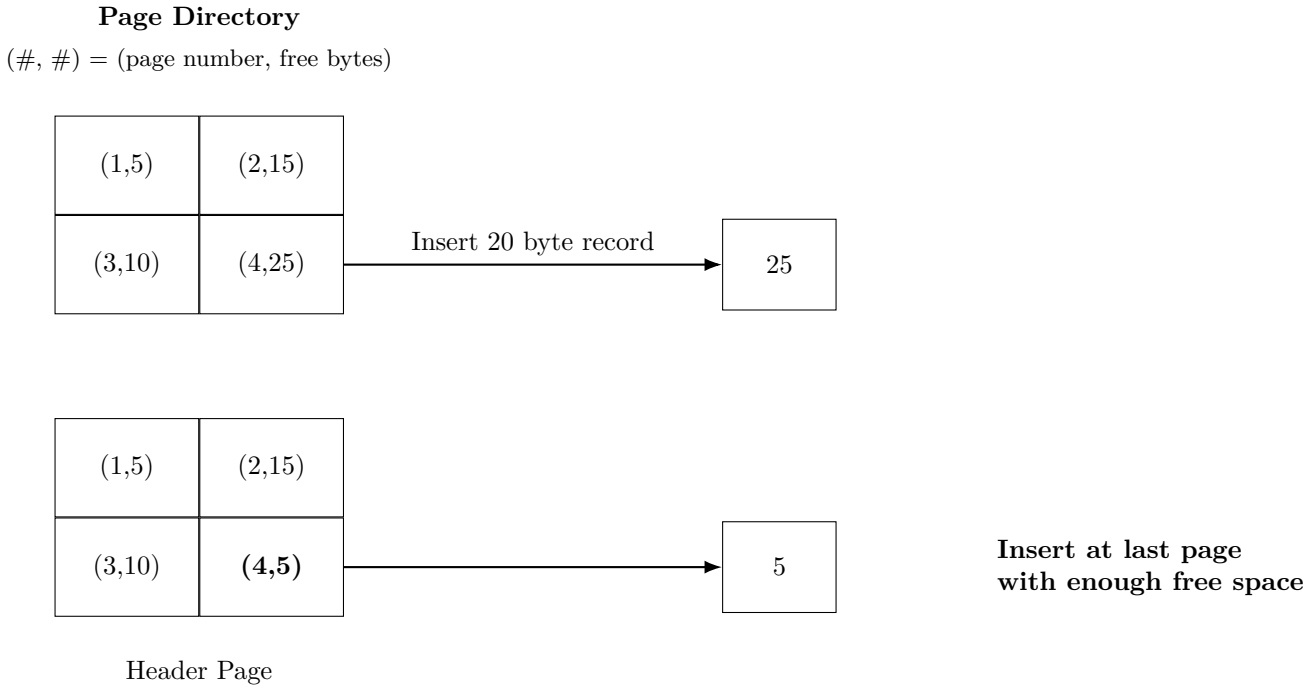


Figure 5: Page directory insertion using stored free-space metadata.

A typical accounting for this example is:

I/O Cost: 1 (read header) + 1 (read data) + 1 (write data) + 1 (write header) = **4 I/Os**.

This is only a small illustration, but it highlights a general trend: as the file grows, insertions that rely on scanning a long free-space list can become increasingly expensive, while a directory that summarizes free space can avoid many unnecessary page reads.

Heap-file tradeoff. Regardless of whether a heap file uses linked lists or page directories, it often supports **fast inserts** because new records can be placed on *any page with available space*. The downside is that **searching** is usually expensive: since records are not ordered, locating a particular record typically requires a **full scan** of pages, leading to a linear number of page reads in the worst case.

5.2 Sorted Files

A **sorted file** maintains an ordering of pages, and records within each page are kept in sorted order by one or more key attributes.

Searching. Because pages are ordered by key range, the DBMS can use **binary search over pages** to identify the page that should contain a key. This typically yields about $\log N$ page reads, where N is the number of pages.

Insertion. Insertion is more costly than in heap files: after locating the correct page (again about $\log N$ reads), inserting a record may force later records to shift forward to preserve sorted order. In the average case, this "push-back" can affect a substantial suffix of pages, so the insertion cost is often modeled as roughly $\log N + N$ I/Os (the $\log N$ to find the target, plus a linear term for shifting and rewriting affected pages).

5.3 Record Types

The internal layout of records is determined by the schema of a relation. Two primary record formats are commonly used:

- **Fixed-Length Records (FLR)**
- **Variable-Length Records (VLR)**

Fixed-length records contain only attributes whose sizes are known in advance, such as integers, booleans, or fixed-width character fields. Every record constructed from the same schema occupies exactly the same number of bytes. This allows the DBMS to directly compute the position of any record using simple arithmetic.

Variable-length records include attributes whose sizes may differ between records, such as `VARCHAR` or text fields. To support these fields, records typically begin with a **record header**. The header stores metadata including pointers that indicate where each variable-length field ends. This allows the DBMS to locate fields even when record sizes differ. (**Read fixed fields first, jump using offsets.**)

Regardless of the record format, each record is uniquely identified using a **record identifier (RID)**. The RID usually contains the page number and the slot number within that page.

5.4 Page Formats

A database page stores multiple records together and contains metadata to help manage them efficiently.

5.4.1 Pages with Fixed-Length Records

Pages storing fixed-length records always include a page header that tracks the number of records stored in the page.

When a page is **packed**, records are stored consecutively without gaps. Insertion is efficient because the position of the next record can be computed using:

$$\text{offset} = \text{number of records} \times \text{record length}.$$

Deletion is slightly more expensive because all records following the deleted record must be shifted upward to maintain a contiguous layout.

When a page is **unpacked**, the page header maintains a **bitmap**. Each bit corresponds to a slot and indicates whether that slot is occupied. This layout avoids shifting records during deletion and simplifies insertion because the DBMS only needs to locate the first free slot.

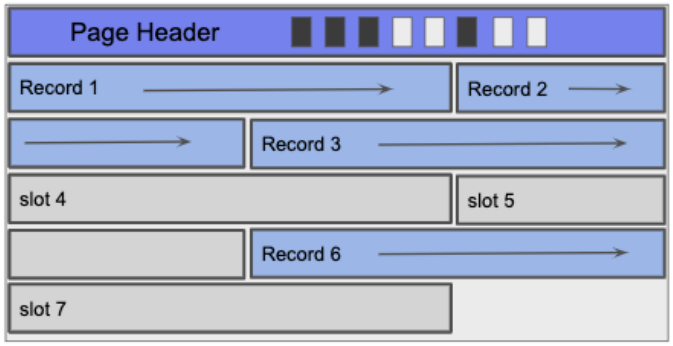


Figure 6: Pages with Fixed-Length Records

Insertion using a bitmap proceeds by locating the first zero bit, placing the new record in the corresponding slot, and updating the bitmap. When deleting a record, the DBMS clears the slot's bit, allowing the space to be reused later.

5.4.2 Pages with Variable-Length Records

With variable-length records, the DBMS cannot assume that all records occupy the same number of bytes. As a result, a page needs extra metadata to find records quickly and to reuse space after deletions.

A common solution is the `slotted-page format`. Instead of storing the directory at the top, the page keeps a `footer` at the bottom that contains a `slot directory`. The directory grows upward from the bottom, while record bytes grow downward from the top. This "two-ended" design leaves a shared region of free space in the middle. Why? **Because the middle approach lets the slot directory and data area naturally compete for space based on actual needs. If you have many small records, the slot directory can grow larger. If you have few large records, the data area gets more room. With fixed separate parts, you might waste space if one part has excess while the other is full.**

The slot directory typically stores:

- a **slot count** (total slots, including empty ones),
- a **free-space pointer** (where the next record can be placed),
- and for each slot: a pair **(record pointer, record length)**.

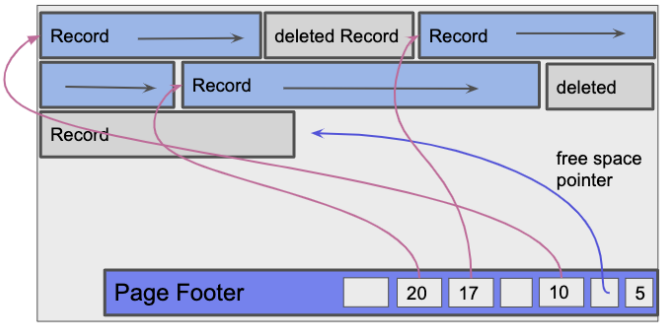


Figure 7: Pages with Variable-Length Records

If the page is **unpacked**, deleting a record usually means marking its slot entry as empty (e.g., setting pointer/length to null). The record bytes may remain in place as a "hole" until a later compaction step. For future insertions, the DBMS writes the new record at the free-space pointer and then fills an available empty slot entry. If no empty entries exist, the slot directory grows by adding a new entry and increasing the slot count. If a page is maintained in a **packed** state, deletions require shifting record bytes to remove gaps and updating affected slot pointers. Compaction is local to the page (records are not shifted across pages).

May a single heap file contains different type of pages? No. Mixing page formats within a single heap file violates the principle that a heap file stores records of a single relation with a consistent schema. Tuples in a single table should have the same schema. If you're storing the same relation/table, all records have the same structure.

5.5 Field Types

Different attribute types consume different amounts of storage on disk. The table below summarizes typical byte sizes used in many DBMS implementations.

Fixed or Variable	Type	Size (B)
Fixed	Integer	4
Fixed	Float	4
Fixed	Boolean	1
Fixed	Byte	1
Fixed	Char(N)	N
Variable	Varchar(N)	$\leq N$
Variable	Text	≥ 0

Figure 8: Common field types and typical storage sizes.

5.6 PageFile Class

A database system never reads or writes individual records directly from disk. Instead, **all I/O is done in fixed-size pages**. This design decision connects the DBMS storage engine to how operating systems and disks actually work.

- Disks and filesystems operate on blocks (typically 4KB).
- OS page cache and DBMS buffer pool both reason in page-sized units.
- Page-based addressing makes random access simple and predictable.

5.6.1 File Format and Page 0

Every DBMS file starts with metadata. In EduDB, this metadata lives entirely in **page 0**. We first define the on-disk format using Python's `struct` module.

```python
FILE_MAGIC = b"EDUDB01"   # 7 bytes -- identifies file type and version.

# < = little-endian
# 7s = 7-byte string, I = uint32
HEADER_FMT  = "<7sI"      # magic, page_size
HEADER_SIZE = struct.calcsize(HEADER_FMT)

META_FMT  = "<I"          # page_count
META_SIZE = struct.calcsize(META_FMT)

DIR_OFFSET = HEADER_SIZE + META_SIZE
DEFAULT_PAGE_SIZE = 4096
```

What Is Endianness? Computers store all data as sequences of **bytes**, not bits. For values that occupy more than one byte (such as 16-bit, 32-bit, or 64-bit integers), the system must decide how those bytes are ordered in memory or on disk. **Endianness specifies the byte order used to represent multi-byte values.**

The number 4096 (32-bit hexadecimal number 0x00001000) requires four bytes:

```
00  00  10  00
```

In big-endian format, the **most significant byte** is stored first. This ordering matches how humans usually write numbers and is often described as "natural reading order."

```
00  00  10  00
```

In little-endian format, the **least significant byte** is stored first. This is the native byte order of most modern CPUs (Intel and AMD x86/x86-64).

```
00  10  00  00
```

Database systems and operating systems store structured binary data on disk. A disk file is nothing more than a sequence of bytes, so the interpretation of those bytes must be unambiguous. Suppose a DBMS writes the integer value 4096 (page size) to disk. If the file is written using one byte order but read assuming another, the decoded value

will be incorrect. One key DBMS principle here: **A file format must explicitly specify its endianness.** This guarantees portability across machines, languages, and architectures.

What Is the `struct` Module? The Python `struct` module provides tools to convert between Python values (integers, strings), and item Raw binary representations (bytes). This is essential for implementing DBMS-style storage formats.

Writing values as text (e.g., `"4096\n"`) is inefficient and ambiguous. Binary representations are: Compact(no control char, no associated attributes such as length, etc); Fast to parse (do not have to decode char using ASCII table, only memory load instruction); Fixed-size (critical for random access, do not have to scan the entire string and to find the terminator and further decode).

5.6.2 Page Identity and Future Buffer Pools

Before we touch disk I/O, the code introduces a small but important abstraction.

```python
@dataclass(frozen=True)
class PageRef:
    """
    Globally-unique identity of a page across all files.

    This becomes useful when you have a *global buffer pool* shared by multiple PageFiles.
    """
    file_path: str
    page_id: int
```

In real DBMSs, a buffer pool caches pages from many files. A page is uniquely identified by:

$$(\text{file}, \text{page\_id})$$

What is a dataclass? A `dataclass` is a Python feature that automatically generates common methods for classes that mainly store data. DBMSs use many small **data-only** objects, `dataclass` is ideal for these. Python automatically creates:

- `__init__` (constructor)
- `__repr__` (pretty printing)
- `__eq__` (value-based comparison)

What Does `frozen=True` Mean? The option `frozen=True` makes the object **immutable**. **Immutable** means: once created, the fields cannot be changed and Python can safely compute a hash for it.

5.6.3 Creating or Opening a Page File

The constructor chooses between initializing a new database file or opening an existing one.

```python
class PageFile:
    """
    A page-based file.

    Page 0 stores the PageFile header (magic/page_size/page_count).
    Data pages start at page_id=1.
    """
    def __init__(self, file_path: str, page_size: int = DEFAULT_PAGE_SIZE):
        self.file_path = file_path
        self.page_size = page_size
        self._fh: Optional[object] = None

        if not os.path.exists(file_path):
            self._create_new()
        else:
            self._open_existing()
```

What `_fh` Stands For? The name `_fh` is short for **file handle**. Therefore, `_fh` is the **Python object that represents an open file**. It is the program's handle for interacting with that file. Once `_fh` exists, it provides all low-level file operations needed by the DBMS:

```
self._fh.seek(offset)   # move file pointer
self._fh.read(n)        # read n bytes
self._fh.write(data)    # write bytes
self._fh.flush()        # push buffers to the OS
self._fh.close()        # release OS resources
```

When a file does not exist, EduDB creates page 0 explicitly.

```python
def _create_new(self) -> None:
    os.makedirs(os.path.dirname(self.file_path) or ".", exist_ok=True)
    with open(self.file_path, "wb") as f:
        page0 = bytearray(self.page_size)
        struct.pack_into(HEADER_FMT, page0, 0, FILE_MAGIC, self.page_size)
        struct.pack_into(META_FMT, page0, HEADER_SIZE, 1)  # page_count = 1
        f.write(page0)
```

If the file exists, read only the minimum bytes needed to learn the true page size, then read page 0 fully with the correct size.

```python
def _open_existing(self) -> None:
    self._fh = open(self.file_path, "r+b")
    self._fh.seek(0)

    # Step 1: read only the minimum bytes needed to parse header+meta
    prefix = self._fh.read(DIR_OFFSET)
    if len(prefix) < DIR_OFFSET:
        raise IOError(
            f"File too small to be an EduDB page file: {self.file_path} "
            f"(needed at least {DIR_OFFSET} bytes, got {len(prefix)})"
        )

    # Step 2: parse magic + real page_size from the prefix
    magic, ps = struct.unpack_from(HEADER_FMT, prefix, 0)
    if magic != FILE_MAGIC:
        raise ValueError(
            f"Not an EduDB page file: {self.file_path}. "
            f"Expected magic={FILE_MAGIC!r}, found={magic!r}"
        )

    # Step 3: trust the on-disk page size (source of truth)
    self.page_size = ps

    # Optional sanity check (recommended)
    if self.page_size < DIR_OFFSET:
        raise ValueError(
            f"Corrupt page_size in header: {self.page_size} "
            f"(must be >= {DIR_OFFSET})"
        )

    # Step 4: now read the full page 0 correctly
    self._fh.seek(0)
    page0 = self._fh.read(self.page_size)
    if len(page0) != self.page_size:
        raise IOError("Short read on page 0")

    # (Optional) If later you want page_count immediately, you can decode it here:
    # (pc,) = struct.unpack_from(META_FMT, page0, HEADER_SIZE)

```

`"r+b"` set open mode to read and write in binary; `seek(0)` sets the file pointer (also called the file offset) to the beginning of the file.

5.6.4 Reading and Writing Page 0

Metadata updates follow a classic read–modify–write pattern.

```python
def _read_page0(self) -> bytearray:
    self._fh.seek(0)
    data = self._fh.read(self.page_size)
    if len(data) != self.page_size:
        raise IOError("Short read on page 0")
    return bytearray(data)
```

`bytearray(data)` is used because DBMS pages must be modified in place, and bytes is immutable while `bytearray` is a mutable, page-like buffer.

```
def _write_page0(self, page0: bytearray) -> None:
    self._fh.seek(0)
    self._fh.write(page0)
    self._fh.flush()
```

`self._fh.flush()` forces Python to push buffered writes to the operating system so that page updates become immediately visible and ordered, even though it does not guarantee physical disk durability.

5.6.5 Managing Page Count

```
def _read_page_count(self) -> int:
    page0 = self._read_page0()
    (pc,) = struct.unpack_from(META_FMT, page0, HEADER_SIZE)
    return pc
```

```
def _write_page_count(self, pc: int) -> None:
    page0 = self._read_page0()
    struct.pack_into(META_FMT, page0, HEADER_SIZE, pc)
    self._write_page0(page0)
```

`_fh` gives you access to the file, not ownership of its contents. Because `page_count` is reconstructed from disk on every call instead of stored in DBMS memory, the design avoids stale metadata at the cost of repeated I/O — favoring correctness over speed.

5.6.6 Allocating New Pages

```
def allocate_page(self) -> int:
    pc = self._read_page_count()
    new_pid = pc

    self._fh.seek(new_pid * self.page_size)
    self._fh.write(b"\x00" * self.page_size)
    self._fh.flush()

    self._write_page_count(pc + 1)
    return new_pid
```

This is a pure **append-only allocator**: (1)no free list (2) no reuse of deleted pages

5.6.7 Reading and Writing Data Pages

```
def read_page(self, page_id: int) -> bytes:
    pc = self._read_page_count()
    if page_id < 0 or page_id >= pc:
        raise IndexError("Page out of range")

    self._fh.seek(page_id * self.page_size)
    data = self._fh.read(self.page_size)
    return data
```

```
def write_page(self, page_id: int, data: bytes) -> None:
    if len(data) != self.page_size:
        raise ValueError("Wrong page size")

    pc = self._read_page_count()
    if page_id < 0 or page_id >= pc:
        raise IndexError("Page out of range")

    self._fh.seek(page_id * self.page_size)
    self._fh.write(data)
    self._fh.flush()
```

The disk layer only reads and writes whole pages.

5.7 SlottedPage Class

Disk pages have a **fixed size** (e.g., 4096 bytes), but database records usually have **variable length**. A slotted page is the classic DBMS solution to this mismatch. **Idea:** Separate record location from record identity. Instead of storing records contiguously, the page stores:

- a **slot directory** (stable record identifiers),
- a **data region** (records packed from the end of the page).

A slotted page is divided into three regions:

1. **Header** (fixed-size)
2. **Slot directory** (grows forward)
3. **Record data** (grows backward)

Free space is always the gap between the slot directory and the record data.

5.7.1 Header Format

The page header stores three unsigned 16-bit integers:

```
"""
edudb.slotted_page

A classic slotted-page layout for variable-length records inside a fixed-size page.

Layout (little-endian):
- Header:
    uint16 slot_count
    uint16 free_start  (start of free space from the beginning)
    uint16 free_end    (end of free space from the end)
- Slot directory grows from header forward:
    for each slot: uint16 offset, int16 length  (length<0 means deleted, length=0 means reclaimed)
- Record data grows from page end backward.
"""

HDR_FMT = "<HHH"
HDR_SIZE = struct.calcsize(HDR_FMT)
```

These fields are:

- `slot_count`: number of slots allocated
- `free_start`: first free byte after the slot directory
- `free_end`: first free byte before the record data

This header is small, fixed-size, and always located at offset 0. That makes it fast to read and update. `H` represents unsigned short int (2 bytes).

Each slot entry stores metadata for one record, `h` represents signed short int (2 bytes):

```
SLOT_FMT = "<Hh"
SLOT_SIZE = struct.calcsize(SLOT_FMT)
```

A slot contains:

- `offset`: where the record starts in the page
- `length`: record length (`0` means deleted)

Slots are compact and fixed-size, which allows constant-time indexing.

5.7.2 Record Identifiers (RID)

```
@dataclass(frozen=True)
class RID:
    page_id: int
    slot_id: int
```

DBMS meaning. An RID uniquely identifies a record as:

$$(\text{page\_id}, \text{slot\_id})$$

Even if the record moves inside the page, the RID remains valid because the slot entry is updated.

5.7.3 The `SlottedPage` Abstraction

```
class SlottedPage:
    def __init__(self, page: bytearray):
        self.page = page
```

This class operates on an **in-memory page buffer**.

```
@staticmethod
def init_empty(page: bytearray) -> None:
    slot_count = 0
    free_start = HDR_SIZE
    free_end = len(page)
    struct.pack_into(HDR_FMT, page, 0, slot_count, free_start, free_end)
```

This establishes the initial invariants: (1) no slots (2) slot directory starts right after the header (3) record data grows from the end of the page.

5.7.4 Reading and Writing the Header

```
def _read_hdr(self) -> Tuple[int, int, int]:
    return struct.unpack_from(HDR_FMT, self.page, 0)
```

```
def _write_hdr(self, slot_count: int, free_start: int, free_end: int) -> None:
    struct.pack_into(HDR_FMT, self.page, 0, slot_count, free_start, free_end)
```

```
def slot_count(self) -> int:
    sc, _, _ = self._read_hdr()
    return sc
```

Metadata is always updated atomically at the page level.Higher layers rely on the header being consistent.

5.7.5 Slot Directory Access

```
def _slot_pos(self, slot_id: int) -> int:
    return HDR_SIZE + slot_id * SLOT_SIZE
```

Slots are laid out contiguously after the header.

```
def _read_slot(self, slot_id: int) -> Tuple[int, int]:
    pos = self._slot_pos(slot_id)
    return struct.unpack_from(SLOT_FMT, self.page, pos)
```

```
def _write_slot(self, slot_id: int, offset: int, length: int) -> None:
    pos = self._slot_pos(slot_id)
    struct.pack_into(SLOT_FMT, self.page, pos, offset, length)
```

```
def free_space(self) -> int:
    sc, free_start, free_end = self._read_hdr()
    return free_end - free_start
```

This gives the exact number of bytes available for: one new slot entry plus one record payload.

5.7.6 Inserting a Record

We first do space check:

```
def insert(self, payload: bytes) -> Optional[int]:
    sc, free_start, free_end = self._read_hdr()

    need = len(payload) + SLOT_SIZE
    if free_end - free_start < need:
        return None
```

Records are packed from the end of the page backward. This avoids fragmentation in the data region.

```
new_end = free_end - len(payload)
self.page[new_end:free_end] = payload
```

The slot ID returned here is stable and becomes part of the record's RID.

```
slot_id = sc
self._write_slot(slot_id, new_end, len(payload))
sc += 1
free_start += SLOT_SIZE
free_end = new_end
self._write_hdr(sc, free_start, free_end)
return slot_id
```

5.7.7 Reading a Record

```
def read(self, slot_id: int) -> Optional[bytes]:
    sc, _, _ = self._read_hdr()
    if slot_id < 0 or slot_id >= sc:
        return None
    offset, length = self._read_slot(slot_id)
    if length <= 0:
        return None
    return bytes(self.page[offset:offset+length])
```

A record is considered deleted if its slot length is zero.

5.7.8 Deleting a Record

```
def delete(self, slot_id: int) -> bool:
    sc, _, _ = self._read_hdr()
    if slot_id < 0 or slot_id >= sc:
        return False
    offset, length = self._read_slot(slot_id)
    if length < 0:
        return False
    self._write_slot(slot_id, offset, -length)
    return True
```

5.7.9 Reclaim Space

Modern database systems rarely reclaim space immediately after a tuple deletion. Instead, they adopt a **lazy compaction strategy**:

- Deletion marks tuples as invalid but does not move data.
- Fragmentation accumulates inside the page.
- Compaction is triggered only when insertion requires contiguous space.

To support this mechanism, we introduce two important functions:

1. `reclaimable_space()` — estimate usable space after compaction
2. `compact()` — physically reorganize tuples inside a page

We encode tuple status using slot length:

Length Value	Meaning
> 0	Live tuple
< 0	Deleted tuple (space reclaimable)
$= 0$	Unused slot

This allows the system to estimate reclaimable storage without immediately moving data. Estimate how much space would be available if the page were compacted. This value is used by the **page directory** to decide whether a page should be considered a candidate for insertion. Total usable space equals:

$$\text{contiguous free space} + \text{space occupied by deleted tuples}$$

```python
def reclaimable_space(self) -> int:
    """
    Space usable after compaction.
    """
    sc, free_start, free_end = self._read_hdr()

    reclaim = free_end - free_start

    for sid in range(sc):
        _, length = self._read_slot(sid)
        if length < 0:  # deleted tuple
            reclaim += -length

    return reclaim
```

Physically reorganize the page so that all live tuples become contiguous, eliminating fragmentation. A critical constraint is:

Slot IDs must remain unchanged

because indexes store Record IDs (RID):

$$RID = (page\_id, slot\_id)$$

Changing slot IDs would invalidate indexes.

1. Collect all live tuples.
2. Rewrite tuples from the end of the page backward.
3. Update slot offsets.
4. Recompute header boundaries.

Before compaction:

```
[A][dead][B][dead][C]
```

After compaction:

```
[A][B][C]
---------------- free space ----------------
```

```python
def compact(self) -> None:
    """
    Repack all live tuples to eliminate fragmentation.
    Slot IDs remain unchanged.
    """

    sc, free_start, free_end = self._read_hdr()

    live = []

    # collect live tuples
    for sid in range(sc):
        off, length = self._read_slot(sid)
        if length > 0:
            payload = bytes(self.page[off:off + length])
            live.append((sid, payload))

    # rewrite from page end
```

```
    new_end = len(self.page)

    for sid, payload in live:
        new_end -= len(payload)
        self.page[new_end:new_end + len(payload)] = payload
        self._write_slot(sid, new_end, len(payload))

    # recompute header
    free_start = HDR_SIZE + sc * SLOT_SIZE
    free_end = new_end

    self._write_hdr(sc, free_start, free_end)

    # after packing live tuples:
    for sid in range(sc):
        off, length = self._read_slot(sid)
        if length < 0:
            # this tombstone's payload space has now been reclaimed by compaction
                self._write_slot(sid, 0, 0)
```

The complete storage policy becomes:

1. DELETE marks slots as dead.
2. Directory records reclaimable space.
3. INSERT selects candidate page.
4. If contiguous space is insufficient:
    - run `compact()`
    - retry insertion

Thus, lazy compaction provides an excellent educational approximation of industrial database storage engines. Deletion only updates the slot directory. The record bytes are left untouched and may be reused later.

5.7.10 Iterating Over Records

```
def iter_rids(self) -> List[int]:
    sc, _, _ = self._read_hdr()
    out = []
    for sid in range(sc):
        _, length = self._read_slot(sid)
        if length > 0:
            out.append(sid)
    return out
```

This scans the slot directory, not the data region. It is fast even if records are fragmented.

6 Buffer Management

Database systems store persistent data on secondary storage devices such as hard disks or SSDs. Although these devices provide durability, they are significantly slower than main memory. To bridge this performance gap, database systems employ a **buffer manager**, whose role is to manage a limited region of main memory called the **buffer pool**.

The buffer manager is responsible for:

- loading disk pages into memory,
- deciding which pages remain resident,
- writing modified pages back to disk, and
- selecting victim pages when memory is full.

All higher-level database components rely on the buffer manager to access data efficiently. A **page** is the fundamental unit of data transfer between disk and memory. Each page has a fixed size (e.g., 4KB) and is uniquely identified by a page identifier (`PageRef` in the implementation).

6.1 Frame Table

A **frame** is a fixed-size slot in the buffer pool (main memory) that can hold exactly one page. Since frames and pages are the same size, a page fits entirely within a single frame. To manage these frames efficiently, the buffer manager maintains an in-memory metadata table – **frame table** that records information about each frame. The is an array indexed by frame identifiers, each frame table entry records metadata about the page currently occupying the frame:

Frame ID	PageRef	Dirty	Pin Count	Ref Bit	Last Used Tick
0	(Student, 12)	0	0	1	42
1	(Student, 7)	1	1	1	45
2	(Enroll, 21)	0	0	0	31
3	(Enroll, 4)	1	2	1	47

Table 1: Frame table storing metadata and replacement state for each buffer frame

- **page_ref**: identifies the disk page loaded in the frame
- **data**: the page contents
- **pin_count**: number of active users of the page
- **dirty**: indicates whether the page has been modified
- **last_used_tick**: logical timestamp for replacement policies
- **ref_bit**: reference bit used by the CLOCK policy

6.2 Page Table

The **page table** maps page identifiers to frame identifiers, this mapping allows the buffer manager to determine whether a page is resident in memory and where it is located:

PageRef	Frame ID
(Student, 12)	0
(Student, 7)	1
(Enroll, 21)	2
(Enroll, 4)	3

Table 2: Page table mapping logical page identifiers to buffer frame identifiers

6.3 Handling Page Requests

When a page is requested from the buffer manager and is already resident in memory, the request results in a page hit. In this case, the buffer manager uses the page table to locate the corresponding frame, increments the page's pin count, updates any relevant replacement metadata, and returns a reference to the in-memory page.

If the requested page is not present in the buffer pool and there is free space available, the buffer manager locates an empty frame and reads the page from disk into that frame. The buffer is then pinned by setting its pin count to 1, and

a reference to the in-memory page is returned to the caller. If the requested page is not resident and no empty frames remain, the buffer manager must apply a replacement policy to select a victim frame for eviction.

The effectiveness of a replacement policy depends strongly on page access patterns and is commonly evaluated using hit statistics. A page hit occurs when a requested page can be served directly from memory, whereas a page miss requires an additional disk I/O to fetch the page. Since disk I/O is significantly more expensive than memory access, selecting an effective eviction policy is critical for performance. The hit rate of an access pattern is defined as

$$\text{Hit Rate} = \frac{\#\text{hits}}{\#\text{hits} + \#\text{misses}} = \frac{\#\text{hits}}{\#\text{accesses}}.$$

If the victim page selected for eviction has its dirty bit set, the buffer manager must write the page back to disk before removing it from the buffer pool in order to preserve in-memory updates. The dirty bit is set when a page is modified while resident in memory and is cleared once the updated contents have been successfully flushed to disk.

After completing its work, the requester is responsible for notifying the buffer manager that it is finished using any pages it previously accessed. When a transaction or query accesses a page, that page is said to be **pinned**; the pin count records how many active users currently depend on the page.

A pinned page:

- cannot be selected as a victim for replacement, and
- guarantees that in-memory references to the page remain valid.

When the caller finishes using a page, it must explicitly unpin it. The caller determines whether the page has been modified and communicates this information to the buffer manager through the `dirty` flag.

6.4 Replacement Policies

When all frames are occupied and a page miss occurs, the buffer manager must select a victim frame according to a **replacement policy**.

6.4.1 LRU Replacement

A commonly used replacement policy is **LRU (Least Recently Used)**. When new pages need to be read into a full buffer pool, the least recently used unpinned page (that is, a page with pin count = 0 and the smallest value in the `Last Used` column) is selected for eviction. To track page usage, a `Last Used` column is added to the metadata table and records the most recent time at which a page's pin count was decremented. **LRU is an effective policy choice because it keeps frequently accessed pages in memory, on the assumption that pages accessed recently are likely to be accessed again soon.**

In this example, frame 3 is selected for eviction because it is unpinned and has the smallest `Last Used` value among all unpinned frames.

Frame ID	Page ID	Dirty Bit	Pin Count	Last Used
0	5	1	3	20
1	3	0	1	32
2	10	1	0	40
3	6	0	0	25
4	1	0	1	15

Table 3: LRU metadata table. Frame 3 is the eviction candidate.

6.4.2 Clock Policy

Implementing LRU directly can be costly, since the buffer manager must search for the minimum value in the `Last Used` column. The **Clock policy** provides an efficient approximation of LRU by using a single **reference bit** per frame and a **clock hand** to track the current frame under consideration.

Using a reference bit significantly reduces both space and time overhead: only one bit per frame is required instead of maintaining a full timestamp. Rather than searching the table, the buffer manager simply advances the clock hand.

The Clock policy treats the frame table as a circular list of frames. When a page is initially loaded into a frame, its reference bit is set to 1. The algorithm proceeds as follows when a page eviction is required:

- Iterate through frames in circular order, skipping pinned pages and wrapping around to frame 0 when the end is reached, until an unpinned frame with `ref bit = 0` is found.

- If the current frame's reference bit is 1, set it to 0 and advance the clock hand to the next frame.
- Upon reaching a frame with `ref bit = 0`, evict the page (writing it to disk first if the dirty bit is set), load the new page into the frame, set its reference bit to 1, and advance the clock hand.

If a page that is already resident in the buffer pool is accessed, the Clock policy sets that page's reference bit to 1 without moving the clock hand.

Assume the clock hand is at frame 2 for the example below:

Frame ID	Page ID	Dirty Bit	Pin Count	Ref Bit
0	5	1	3	1
1	3	0	1	1
2	10	1	0	1
3	6	0	0	1
4	1	0	1	0

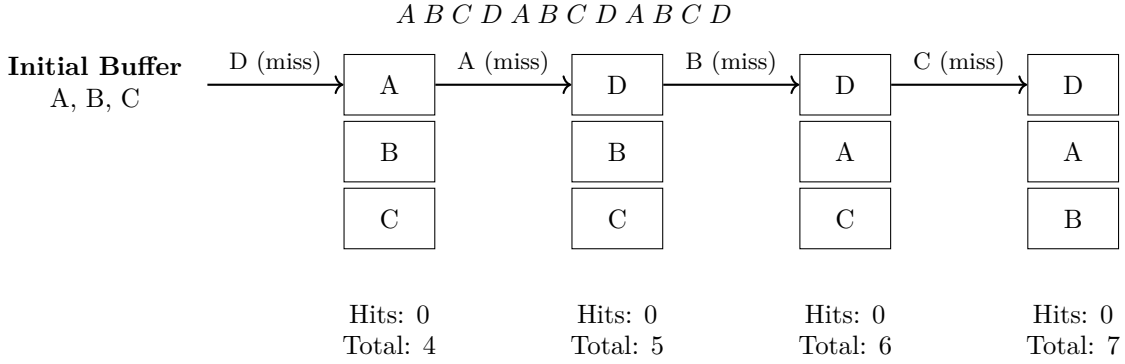
Table 4: Clock policy metadata table with reference bits.

1. The clock hand initially points to **Frame 2**.
2. **Frame 2** is inspected.
   - Pin count = 0 (unpinned).
   - Reference bit = 1.
   - Action: the reference bit is cleared to 0 to give the page a second chance.
   - The clock hand advances to the next frame.
3. **Frame 3** is inspected.
   - Pin count = 0 (unpinned).
   - Reference bit = 1.
   - Action: the reference bit is cleared to 0 to give the page a second chance.
   - The clock hand advances to the next frame.
4. **Frame 4** is inspected.
   - Pin count > 0 (pinned).
   - Action: the frame is skipped.
   - The clock hand advances to the next frame.
5. **Frame 0** is inspected.
   - Pin count > 0 (pinned).
   - Action: the frame is skipped.
   - The clock hand advances to the next frame.
6. **Frame 1** is inspected.
   - Pin count > 0 (pinned).
   - Action: the frame is skipped.
   - The clock hand advances to the next frame.
7. The clock hand returns to **Frame 2**.
   - Pin count = 0 (unpinned).
   - Reference bit = 0.
   - Action: Frame 2 satisfies the eviction conditions and is selected as the victim frame.
8. If the dirty bit of Frame 2 is set, the page is written back to disk. The frame is then reused to load the new page, its reference bit is set to 1, and its pin count is initialized to 1.

6.4.3 Sequential Scanning Performance – LRU

Although LRU performs well in many scenarios, its performance degrades when a set of pages S, where $|S|$ is larger than the buffer pool size, is accessed repeatedly. This access pattern is known as **sequential flooding** and commonly arises in database workloads such as scanning every record in a table.

To illustrate this behavior, consider a buffer pool with three frames using LRU and the following access pattern:

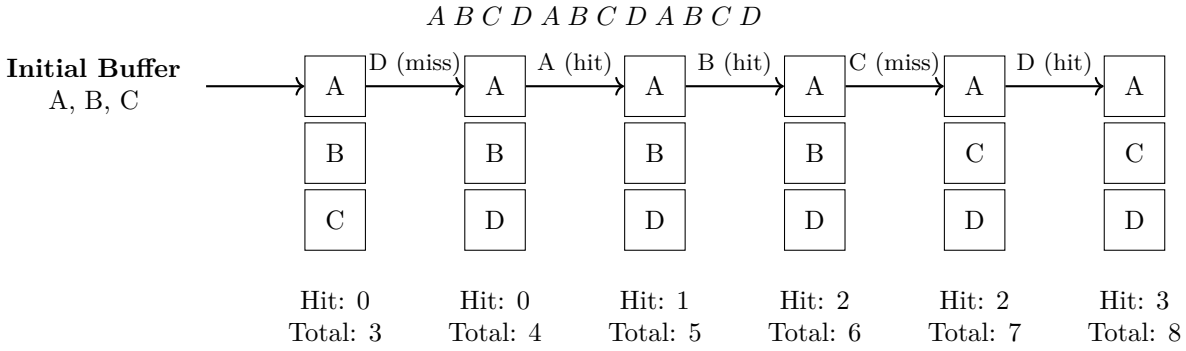


This example demonstrates that LRU can suffer from zero hit rate during sequential scans, as each new page evicts a page that will be needed again shortly.

6.4.4 MRU Replacement

Another commonly used replacement policy is **MRU (Most Recently Used)**. Instead of evicting the least recently used unpinned page, MRU evicts the most recently used unpinned page, as measured by the most recent time at which the page's pin count was decremented. Although MRU may appear counterintuitive, it can perform well under certain access patterns, particularly sequential scans.

To illustrate the behavior of MRU under sequential flooding, consider a buffer pool with three frames using MRU and the following access pattern:



Clearly, MRU significantly outperforms LRU in terms of page hit rate when a sequential flooding access pattern occurs. By evicting the most recently used page, MRU preserves older pages that are more likely to be reused during a sequential scan.

Most real-world database systems use **CLOCK or CLOCK-derived replacement policies**, which efficiently approximate LRU while avoiding the overhead of maintaining exact recency information.

6.5 BufferPool Class

This section presents a detailed, code-driven tutorial for the EduDB `BufferPool` implementation. The buffer pool is **global** and shared across multiple page files. Because page identifiers are only unique within a file, this implementation uses a composite identifier, texttttPageRef(file_path, page_id), to uniquely identify pages in memory.

6.5.1 High-Level Design

The buffer pool is responsible for:

- caching disk pages in main memory,
- tracking page residency,

- managing pin counts and dirty state,
- selecting eviction victims when full, and
- flushing modified pages back to disk.

All table and index operations in EduDB access disk pages only through this buffer pool.

6.5.2 Frame Descriptor

Each slot in the buffer pool is represented by a `Frame` object.

```python
from __future__ import annotations

from dataclasses import dataclass
from typing import Dict, List, Optional, Tuple

from .disk_file import PageFile, PageRef

"""One buffer frame in the buffer pool."""
@dataclass
class Frame:
    frame_id: int
    pref: PageRef
    data: bytearray
    pin_count: int = 0
    dirty: bool = False
    ref_bit: int = 0
    last_used_tick: int = 0
```

- `frame_id` is the physical index of the frame in the buffer pool.
- `pref` identifies the disk page cached in this frame.
- `data` stores the page contents as a mutable byte array.
- `pin_count` records how many active users depend on the page.
- `dirty` indicates whether the page differs from its on-disk version.
- `ref_bit` supports the CLOCK (second-chance) policy.
- `last_used_tick` tracks recency for LRU/MRU and debugging.

Although lecture slides often describe the frame table as "metadata only", this implementation intentionally colocates metadata and page data for clarity and locality.

6.5.3 BufferPool Initialization

The buffer pool constructor initializes all core data structures.

```python
class BufferPool:
    def __init__(self, capacity_pages: int = 256, policy: str = "CLOCK"):
        if capacity_pages <= 0:
            raise ValueError("capacity_pages must be positive")

        self.capacity = capacity_pages

        policy_u = policy.strip().upper()
        if policy_u not in {"CLOCK", "LRU", "MRU"}:
            raise ValueError('policy must be one of: "CLOCK", "LRU", "MRU"')
        self.policy: str = policy_u

        self._frame_table: List[Optional[Frame]] = [None] * self.capacity
        self._page_table: Dict[PageRef, int] = {}
        self._clock_hand: int = 0
        self._tick: int = 0
        self.hits: int = 0
        self.misses: int = 0
        self._pf_registry: Dict[str, PageFile] = {}
```

1. `self.capacity` becomes the global limit used by allocation and eviction logic later.
2. The input policy is normalized by `strip().upper()` so callers can pass `"clock"`, `" CLOCK "`, etc.
3. `_frame_table` is a fixed-size array, each slot is either `None` (empty) or a `Frame` object.

4. `_page_table` maps `PageRef` → `frame_id`. This is the constant-time lookup structure that makes "buffer hit" fast.
5. `_clock_hand` is the moving pointer used during CLOCK eviction scans.
6. `_tick` is a monotonic counter used to timestamp frame usage (`Frame.last_used_tick`). This supports LRU/MRU and is also useful for debugging.
7. `hits` and `misses` counters are updated in `fetch_page()` and allow a clean `hit_rate()` computation later.
8. The pool is global across files, so on eviction/flush it must know which `PageFile` object to call `write_page()` on. `_pf_registry[file_path] = pf` provides that lookup.

After `__init__`, the buffer pool always has:

$$\underbrace{\texttt{\_frame\_table}}_{\text{frame\_id}\to\text{Frame/None}} + \underbrace{\texttt{\_page\_table}}_{\text{PageRef}\to\text{frame\_id}} + \underbrace{\texttt{\_pf\_registry}}_{\text{file\_path}\to\text{PageFile}}$$

So every later operation can be written as "lookup → pin/modify → unpin/flush/evict".

6.5.4 Internal Utilities: Registry, Usage Ticks, Free-Slot Search, and Frame Loading

This section provides the "small building blocks" that the core API will compose.

```python
    # -------------------------
    # Internal utilities
    # -------------------------
    def register_pagefile(self, pf: PageFile) -> None:
        self._pf_registry[pf.file_path] = pf

    def _touch(self, fr: Frame) -> None:
        self._tick += 1
        fr.last_used_tick = self._tick

    def _find_free_frame_id(self) -> Optional[int]:
        """Return an empty frame slot if any, else None."""
        for i, fr in enumerate(self._frame_table):
            if fr is None:
                return i
        return None

    def _load_into_frame(self, frame_id: int, pref: PageRef, data: bytes, *, dirty: bool) -> Frame:
        fr = Frame(
            frame_id=frame_id,
            pref=pref,
            data=bytearray(data),
            pin_count=1,
            dirty=dirty,
            ref_bit=1,  # referenced on load
            last_used_tick=0,
        )
        self._touch(fr)
        self._frame_table[frame_id] = fr
        self._page_table[pref] = frame_id
        return fr
```

6.5.5 Line-by-line walkthrough

1. `register_pagefile` registers the `PageFile` by its `file_path`. This is critical for correctness: eviction/flush later needs the right `PageFile` instance to write dirty pages back.
2. `_touch` advances a global counter and stores it in the frame. This makes "recency" measurable without storing wall-clock time.
3. `_find_free_frame_id` linear scan for a `None` slot. This is called before eviction: if there is empty space, do not evict.
4. `_load_into_frame`: Single constructor path for resident pages:
   - wraps `data` into `bytearray(data)` for in-place mutation,
   - sets `pin_count=1` because fetching/creating returns a pinned page,
   - sets `dirty` based on caller intent,

- sets `ref_bit=1` so CLOCK treats it as recently referenced,
- updates **both** tables: frame table and page table.

Whenever a page becomes resident, the code must update:

$$\texttt{\_frame\_table[frame\_id]} \leftarrow \texttt{Frame} \quad \text{and} \quad \texttt{\_page\_table[pref]} \leftarrow \texttt{frame\_id}.$$

`_load_into_frame` enforces this consistently.

6.5.6 Core Buffer Manager API: `fetch_page`

```python
def fetch_page(self, pf: PageFile, page_id: int) -> Frame:
    """Fetch (and pin) a page into the buffer pool."""
    #Step1:
    self.register_pagefile(pf)
    pref = PageRef(pf.file_path, page_id)

    #Step2: HIT
    frame_id = self._page_table.get(pref)
    if frame_id is not None:
        fr = self._frame_table[frame_id]
        # Defensive: should never be None if page_table is consistent.
        if fr is None:
            raise RuntimeError("Page table points to an empty frame; internal inconsistency")
        fr.pin_count += 1
        fr.ref_bit = 1  # CLOCK: referenced
        self._touch(fr)
        self.hits += 1
        return fr

    #Step3: MISS
    self.misses += 1

    free_id = self._find_free_frame_id()
    if free_id is None:
        free_id = self._evict_one()

    data = pf.read_page(page_id)
    return self._load_into_frame(free_id, pref, data, dirty=False)
```

Step 1: Bind the request to a globally unique key

- `self.register_pagefile(pf)`: because the buffer pool is global across multiple `PageFile`s, it must remember which file object owns a page so it can later flush dirty pages safely.
- `pref = PageRef(pf.file_path, page_id)`: this is the key design decision in a global buffer pool. A `page_id` alone is not unique because every table file has a page 0, page 1, etc. So the cache key must include the file identity.

Step 2: The hit path (fast path)

1. **Lookup in the page table.** `_page_table` maps `PageRef` → `frame_id`. If the key exists, we know exactly which frame holds the bytes.
2. **Defensive consistency check.** `_page_table` and `_frame_table` should agree. If the page table points to an empty slot, the internal state is corrupted, so we raise.
3. **Pin it.** `fr.pin_count += 1` means "one more client is using this page". This is the main correctness rule: pinned frames cannot be evicted.
4. **Mark it referenced (CLOCK).** CLOCK replacement uses a `ref_bit`. On every access we set it to 1, meaning "give this page a second chance" during eviction scanning.
5. **Update recency metadata.** `_touch(fr)` updates `last_used_tick`. Even if CLOCK is default, keeping a tick is useful for debugging and supports LRU/MRU policies.
6. **Update statistics and return.** This is a hit: increment `self.hits` and return the resident frame.

Step 3: The miss path (slow path) If the page is not in `_page_table`, we must read from disk:

1. **Account for the miss.** This makes hit-rate metrics meaningful.

2. **Find an empty slot first.** `_find_free_frame_id()` scans for a `None` slot. If one exists, we avoid eviction entirely.
3. **Evict only if necessary.** If no empty slot exists, `_evict_one()` chooses a victim according to the active policy.
4. **Read from disk and load into the chosen frame.** `pf.read_page(page_id)` returns the raw bytes. `_load_into_frame(..., dirty=False)` makes it resident and pinned.

6.5.7 Core Buffer Manager API: `new_page`

```python
def new_page(self, pf: PageFile) -> Frame:
    """Step1: Allocate a new page on disk, bring it into the buffer pool pinned + dirty."""
    self.register_pagefile(pf)
    new_pid = pf.allocate_page()

    # Step2: allocate_page() updates page 0 on disk; refresh cached page 0 if present
    page0ref = PageRef(pf.file_path, 0)
    page0_fid = self._page_table.get(page0ref)
    if page0_fid is not None:
        fr0 = self._frame_table[page0_fid]
        if fr0 is not None:
            fr0.data[:] = bytearray(pf.read_page(0))
            fr0.dirty = False
            fr0.ref_bit = 1
            self._touch(fr0)

    free_id = self._find_free_frame_id()
    if free_id is None:
        free_id = self._evict_one()

    pref = PageRef(pf.file_path, new_pid)
    # Step3: Newly allocated page is considered dirty until flushed.
    return self._load_into_frame(free_id, pref, b"\x00" * pf.page_size, dirty=True)
```

`fetch_page` is "bring an existing page into memory". `new_page` is "create a new page on disk and then bring it into memory". So `new_page` has two responsibilities:

1. allocate a page ID on disk (`pf.allocate_page()`),
2. return an in-memory frame holding that page, pinned.

Step 1: Allocate on disk This is an important separation of concerns: the `PageFile` owns the file format and directory bookkeeping, so it is the one that decides what the next page ID is.

Step 2: The subtle page-0 refresh PageFile objects store **metadata** in page 0 (directory/header). If `allocate_page()` updates that metadata on disk and page 0 is currently cached, then the cached version becomes stale unless we refresh it.

Your code handles this cleanly:

- only refresh if page 0 is cached,
- overwrite in place via `fr0.data[:]` so any references remain valid,
- mark it clean (`dirty=False`) because it now matches disk again.

Step 3: Load a fresh zeroed page and mark it dirty Two key design choices:

1. **Zero-fill:** the page starts as empty bytes of size `pf.page_size`.
2. **Dirty immediately:** the page is considered "not durable" until flushed. Marking it dirty guarantees it will be written to disk before eviction.

6.5.8 Core Buffer Manager API: `unpin`

```python
def unpin(self, pf: PageFile, page_id: int, dirty: bool = False) -> None:
    pref = PageRef(pf.file_path, page_id)
    frame_id = self._page_table.get(pref)
    if frame_id is None:
        raise KeyError(f"Page {pref} not in buffer")

    fr = self._frame_table[frame_id]
    if fr is None:
```

```
        raise RuntimeError("Page table points to an empty frame; internal inconsistency")
    if fr.pin_count <= 0:
        raise RuntimeError("Unpin called with pin_count=0")

    fr.pin_count -= 1
    fr.dirty = fr.dirty or dirty
    fr.ref_bit = 1  # still counts as recently used
    self._touch(fr)
```

`fr.dirty = fr.dirty or dirty` is a subtle but correct pattern: once a page becomes dirty, it stays dirty until a flush occurs. Passing `dirty=True` is a caller signal: "I modified this page while it was pinned."

Even though we are releasing the page, the fact that we interacted with it suggests it is part of the working set. Marking it referenced supports CLOCK, and touching it supports LRU/MRU recency metadata.

6.5.9 Core Buffer Manager API: `flush_page`

```
def flush_page(self, pf: PageFile, page_id: int) -> None:
    pref = PageRef(pf.file_path, page_id)
    frame_id = self._page_table.get(pref)
    if frame_id is None:
        return

    fr = self._frame_table[frame_id]
    if fr and fr.dirty:
        pf.write_page(page_id, bytes(fr.data))
        fr.dirty = False
```

Flushing is about **durability**: copy the in-memory bytes back to disk so the disk version matches memory. Two deliberate choices appear here:

1. **If the page is not cached, do nothing.** That makes `flush_page` safe to call as a "best effort".
2. **Write only if dirty.** If `dirty=False`, the cached bytes already match disk, so writing again is wasted I/O.

`fr.data` is a mutable `bytearray`. Converting to `bytes` creates an immutable snapshot for I/O. It prevents accidental later modifications from affecting the data being written mid-operation.

6.5.10 Core Buffer Manager API: `flush_all`

```
def flush_all(self) -> None:
    for fr in self._frame_table:
        if fr is None or not fr.dirty:
            continue
        pf = self._pf_registry.get(fr.pref.file_path)
        if pf is None:
            continue
        pf.write_page(fr.pref.page_id, bytes(fr.data))
        fr.dirty = False
```

Notice the two `continue` checks:

- skip empty frames and clean frames,
- skip frames whose `PageFile` is not registered.

6.5.11 Replacement Policy Dispatch: `_evict_one`

```
def _evict_one(self) -> int:
    """Evict one unpinned page according to the active replacement policy."""
    if self.policy == "CLOCK":
        return self._evict_one_clock()
    if self.policy == "LRU":
        return self._evict_one_lru(mru=False)
    if self.policy == "MRU":
        return self._evict_one_lru(mru=True)
    raise RuntimeError(f"Unknown policy: {self.policy!r}")
```

This method is the **single entry point** for eviction. Every caller says: "I need a free frame", and `_evict_one` decides which policy to run.

6.5.12 LRU and MRU Eviction: `_evict_one_lru`

```python
def _evict_one_lru(self, *, mru: bool) -> int:
    """
    Evict one unpinned page using:
        - LRU  if mru=False
        - MRU  if mru=True

    Returns the freed frame_id.
    """

    victim = None

    # Single-pass selection (no temporary list)
    for fr in self._frame_table:
        if fr is None or fr.pin_count > 0:
            continue

        if victim is None:
            victim = fr
        else:
            if mru:
                # MRU: choose most recently used
                if fr.last_used_tick > victim.last_used_tick:
                    victim = fr
            else:
                # LRU: choose least recently used
                if fr.last_used_tick < victim.last_used_tick:
                    victim = fr

    if victim is None:
        raise RuntimeError("Buffer pool full: no unpinned page to evict")

    victim_id = victim.frame_id

    # Flush if dirty
    if victim.dirty:
        pf = self._pf_registry.get(victim.pref.file_path)
        if pf is None:
            raise RuntimeError(
                f"Cannot flush dirty page without PageFile: {victim.pref.file_path}"
            )
        pf.write_page(victim.pref.page_id, bytes(victim.data))
        victim.dirty = False

    # Remove from page table and clear frame slot
    del self._page_table[victim.pref]
    self._frame_table[victim_id] = None

    return victim_id
```

Step 1: filter to legal victims This is the first correctness rule of eviction: *Only unpinned frames are eligible for eviction.* If **no candidates exist**, then all resident pages are pinned, and the buffer pool is "logically full" even if it has many pages — because you are not allowed to evict any of them.

Step 2: choose LRU vs MRU Your `_touch(fr)` method maintains `fr.last_used_tick`. That turns "recency" into a simple numeric comparison.

- **LRU:** evict the smallest tick (oldest access).
- **MRU:** evict the largest tick (most recent access).

Step 3: flush if dirty (write-back) Eviction must preserve durability. If the in-memory bytes differ from disk, we cannot discard them. Why consult `_pf_registry`? The frame stores only `victim.pref.file_path` and `victim.pref.page_id`. To call `write_page`, you need the owning `PageFile` instance. If the registry is missing, you raise because flushing without the correct disk-layer object is unsafe.

Step 4: remove victim from both tables (cache coherence) This is the core table-consistency rule:

$$\texttt{pref} \notin \texttt{\_page\_table} \iff \texttt{frame slot is empty or holds a different page.}$$

Eviction must update **both** structures; otherwise you get stale pointers and"page table points to empty frame" errors.

Step 5: keep the CLOCK hand reasonable even for LRU/MRU Even though LRU/MRU does not use the clock hand, your code keeps it consistent. This is a small engineering detail: if the policy switches later, the clock scan starts from a sensible spot.

6.5.13 CLOCK Eviction: `_evict_one_clock`

```python
def _evict_one_clock(self) -> int:
    """Evict one unpinned page using CLOCK and return its freed frame_id."""
    # If all frames are pinned, we cannot evict.
    if all(fr is not None and fr.pin_count > 0 for fr in self._frame_table):
        raise RuntimeError("Buffer pool full: no unpinned page to evict")

    victim_id = None
    scanned = 0

    while scanned < 2 * self.capacity and victim_id is None:
        fr = self._frame_table[self._clock_hand]

        if fr is None:
            victim_id = self._clock_hand
        else:
            if fr.pin_count == 0:
                if fr.ref_bit == 0:
                    # Found victim
                    victim_id = self._clock_hand
                else:
                    # Second chance
                    fr.ref_bit = 0

        # Move clock hand and increment scan counter
        self._clock_hand = (self._clock_hand + 1) % self.capacity
        scanned += 1

    if victim_id is None:
        raise RuntimeError("Buffer pool full: no unpinned page to evict")

    # --- Evict victim ---
    victim = self._frame_table[victim_id]

    if victim.dirty:
        pf = self._pf_registry.get(victim.pref.file_path)
        if pf is None:
            raise RuntimeError(
                f"Cannot flush dirty page without PageFile: {victim.pref.file_path}"
            )
        pf.write_page(victim.pref.page_id, bytes(victim.data))
        victim.dirty = False

    del self._page_table[victim.pref]
    self._frame_table[victim_id] = None

    return victim_id

```

LRU requires tracking exact recency order (or comparing ticks across all frames). CLOCK approximates LRU with one bit per frame without full sorting:

- `ref_bit=1` means "recently referenced; don't evict me yet."
- `ref_bit=0` means "not referenced recently; evictable if unpinned."

The `_clock_hand` points to the next frame to consider, like the hand of a clock sweeping around a circle.

Step 1: detect the impossible case (all pinned) This is the CLOCK version of the candidate check: if every resident frame is pinned, eviction cannot proceed.

Step 2: scan around the clock Your loop caps scanning to `2 * capacity`. Why?

Because CLOCK may need two passes:

- First pass: clears `ref_bit=1` pages to 0 (giving them a second chance).
- Second pass: eventually finds a `ref_bit=0` victim if one exists.

Step 3: handle an empty slot gracefully In theory, callers only invoke eviction when the pool is full, so empty slots should not appear. But this defensive handling makes the routine robust if it is ever called incorrectly.

Step 4: skip pinned frames Pinned means "actively in use." CLOCK respects the pinning invariant by skipping them.

Step 5: second chance logic This is the heart of CLOCK. If a page was referenced recently, we clear the bit and move on. That does two things:

1. It postpones eviction of hot pages.
2. It ensures that if the page stays hot, later accesses will set `ref_bit=1` again.

Step 6: choose victim (unpinned + ref_bit=0) At this point, CLOCK has found a page that:

- is not in use (`pin_count==0`),
- has not been referenced since its last second chance (`ref_bit==0`).

Step 7: flush-if-dirty and remove from tables The rest is the same eviction contract you used in LRU/MRU:

1. flush if dirty using `_pf_registry`
2. delete from `_page_table`
3. clear frame slot in `_frame_table`
4. advance the hand and return the freed `frame_id`

Step 8:Why the final error exists If at least one frame is unpinned, CLOCK should find a victim within at most two full passes. Reaching this error indicates a serious internal bug (e.g., bookkeeping inconsistency).

6.5.14 Public Helper Methods: `hit_rate`, `stats`, `set_policy`

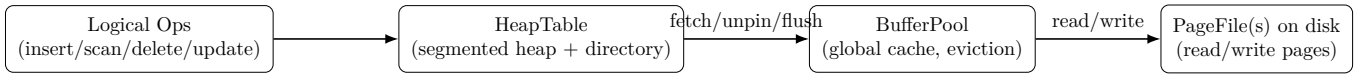
These are small, user-facing helpers that make the buffer pool measurable and configurable.

```
    # -------------------------
    # Public helpers / stats
    # -------------------------
    def hit_rate(self) -> float:
        total = self.hits + self.misses
        return (self.hits / total) if total else 0.0

    def stats(self) -> Dict[str, float]:
        """Return a small stats dict useful for debugging / reporting."""
        return {
            "capacity": float(self.capacity),
            "hits": float(self.hits),
            "misses": float(self.misses),
            "hit_rate": self.hit_rate(),
        }

    def set_policy(self, policy: str) -> None:
        """Switch replacement policy at runtime: CLOCK, LRU, or MRU."""
        policy_u = policy.strip().upper()
        if policy_u not in {"CLOCK", "LRU", "MRU"}:
            raise ValueError('policy must be one of: "CLOCK", "LRU", "MRU"')
        self.policy = policy_u
```

1. `hit_rate` computes:
$$\texttt{hits}/(\texttt{hits}+\texttt{misses})$$
and returns 0.0 when no accesses have occurred (avoids division by zero).
2. `stats`. returns a small dict of numeric values. Note it casts to `float`, which is convenient if you later dump to JSON or compute ratios consistently.
3. textttset_policy allows runtime experiments, e.g., run a workload under CLOCK, then switch to LRU and compare hit rates.

7 HeapTable + BufferPool

7.1 High-level Architecture



The HeapTable decides:

- which segment and which page should receive a record (via directory),
- what bytes to write into a slotted page.

The BufferPool decides:

- whether a requested page is already in memory (**hit**) or must be read from disk (**miss**),
- which page to evict when memory is full (CLOCK/LRU/MRU),
- when and how dirty pages are flushed to disk.

7.2 Data Model in `HeapTable`

Segment Name: Segment 0 keeps its legacy name (often `<table>.pages`). Segment 1+ uses:

`<stem>.segk.pages`

This is implemented by:

- `_segment0_path()`
- `_make_segment_path(seg_id)`
- `_allocate_new_segment()`

Directory Payload: Directory payload (JSON stored after `DIR_OFFSET` in page 0):

```
{
  "free_bytes": {
    "1": 3980,
    "2": 120
  }
}
```

- JSON is just easier for encoding and decoding and more suitable for teaching purposes.
- Keys are strings in JSON; code converts them back to integers.

There are several considerations that why I store all metadata in page0 per segment file:

- locality of metadata. We avoid a separate metadata file, and the segment is self-contained.
- simplicity. Developer can open a segment and inspect page 0 to see exactly how free-space management works.

GlobalRID: Because one table spans multiple segment files, `(page_id, slot_id)` is no longer globally unique. So `HeapTable` returns:

$$\texttt{GlobalRID} = (\texttt{seg\_id}, \texttt{page\_id}, \texttt{slot\_id})$$

This is the simplest correct RID for a segmented heap.

Segmentation with a Max Page Limit: A segment max page limit:

- bounds the size of the directory JSON (page 0 has limited capacity),
- avoids deep complexity like multi-level directory indexing (teaching-friendly).

Lazy Compaction Compaction is expensive. Doing it always would be wasteful. So the design is:

- try insert quickly if the page has enough total free space
- if it fails (no enough contiguous free space), compact and try again.
- if still fails, move on.

This models real systems that avoid aggressive compaction unless needed.

7.3 Interactions with Table

Here I will show the pipeline of a single SQL query, even though many layers we have not yet discussed, but it will give you a big picture and help you better understand the topics we will discuss in the future. Assume the server receives:

```
INSERT INTO Student (student_id, name, gpa)
VALUES (1001, 'Alice', 3.8);
```

We trace every layer of execution.

- **STEP 1 — Server → Database.execute_sql()**

  The server calls:

  ```
  db.execute_sql(sql)
  ```

  Inside `execute_sql()`:

  ```
  parsed = parser.parse(sql)
  ```

- **STEP 2 — Parser**

  The function `_parse_insert()` returns:

  ```
  ("INSERT", table, row_dict)
  ```

  Example:

  ```
  ("INSERT", "Student", {"student_id": 1001,  "name": "Alice",  "gpa": 3.8})
  ```

- **STEP 3 — Lock Acquisition**

  Inside `execute_sql()`:

  ```
  self._acquire_lock(tx, f"table:{table}", "X")
  ```

  The table is locked in exclusive mode.

- **STEP 4 — Call Database.insert()**

  Control moves to:

  ```
  return self.insert(table, row, tx=tx)
  ```

- **STEP 5 — Call HeapTable.insert()**

  Control moves to HeapTable:

  ```
  rid = ti.table.insert(row)
  ```

  Now execution enters the `HeapTable`.

- **STEP 6 — BufferPool.fetch_page()**

  Inside HeapTable.insert():

  ```
  fr = self.bp.fetch_page(pf, 0) # get directory info
  ...
  fr = self.bp.fetch_page(pf, pid) # get a page with sufficient free space
  ```

- **STEP 7 — SlottedPage.insert(payload)**

  Insert the row into the page.

```
slot_id = sp.insert(payload)
```

- **STEP 8 — Mark Page Dirty**

```
self.bp.unpin(pf, pid, dirty=True)
```

- **STEP 9 — Write It Back to Disk**

  Here, we always write the data back to disk. This approach is not efficient in real-world systems, but it is used here for educational purposes and to simplify the code structure.

```
self.bp.flush_page(pf, pid)
```

  Inside BufferPool.flush_page(...):

```
pf.write_page(pid, frame.data)
```

  Actual disk write occurs here.

- **STEP 10 — Update Page Directory**

```
self._set_free_bytes(...)
```

- **STEP 11 — Update Indexes**
- From hash index:

```
idx.insert(key, rid)
```

- From B+Tree index:

```
idx.insert(key, rid)
```

7.4 HeapTable Class

The file `table.py` defines the physical heap-table layer of EduDB. Its job is to store rows as byte payloads inside slotted pages and organize one logical table across multiple physical segment files.

It supports:

- segmented heap-file storage,
- page-directory management,
- direct RID-based row lookup,
- row insertion,
- row deletion,
- physical row update,
- full table scan,
- integration with the buffer pool.

```
"""
edudb.table

Stage 1+2: Segmented heap file + indexes store GlobalRID.

- ONE TABLE = MANY FILE SEGMENTS:
    Student.pages
    Student.seg1.pages
    Student.seg2.pages
    ...

```

```
- Each segment is a normal PageFile:
    * page 0 = PageFile header + HeapTable directory payload (after DIR_OFFSET)
    * pages 1.. = slotted pages

- Directory per segment (in page 0 after DIR_OFFSET):
    {"version": 2, "free_bytes": {"<pid>": <free_space_bytes>, ...}}

GlobalRID:
- (seg_id, page_id, slot_id)
"""
```

This docstring already summarizes the main design. One logical table may span many physical files. Each file segment is a normal `PageFile`. Page 0 of each segment stores metadata, while pages 1 and above are real data pages. Since page IDs are only unique inside one segment, tuple identity must be represented by a `GlobalRID`.

```
Row = Dict[str, Any]
GlobalRID = Tuple[int, int, int]    # (seg_id, page_id, slot_id)
```

A row is represented as a Python dictionary. A tuple identifier is represented as:

$$(\texttt{seg\_id}, \texttt{page\_id}, \texttt{slot\_id})$$

This is necessary because once the table spans multiple segments, `page_id` alone is no longer enough to uniquely identify a tuple.

7.4.1 `HeapTable` Class

```
class HeapTable:
    FLUSH_ON_WRITE = True
    SEGMENT_MAX_PAGES = 1024  # includes page 0
```

This is the main storage class.

- `FLUSH_ON_WRITE = True` means pages are flushed immediately after modification. This is helpful for teaching and debugging because the on-disk state is easier to observe.
- `SEGMENT_MAX_PAGES = 1024` means each segment has a fixed upper limit on the number of pages it may contain, including page 0.

So if one segment becomes full, the table continues growing by allocating a new segment file.

```
def __init__(self, name: str, schema: Schema, segments: List[PageFile], bp: BufferPool):
    self.name = name
    self.schema = schema
    self.segments = segments
    self.bp = bp

    for seg_id in range(len(self.segments)):
        self._ensure_directory_initialized(seg_id)
```

This constructor stores the basic metadata of the heap table:

- the table name,
- the schema used for row encoding and decoding,
- the list of physical segment files,
- the shared buffer pool.

After that, it calls `_ensure_directory_initialized()` for every segment. This guarantees that every segment has a valid page directory in page 0.

Why is this important? Because the table may be reopened from disk. If the directory is missing or stale, the heap table needs to rebuild or rewrite it before normal operations continue.

7.4.2 Directory Helpers

The directory is one of the most important ideas in this file. Instead of scanning every data page to find free space, the heap table stores a free-space summary in page 0 of each segment.

```
def _read_directory(self, seg_id: int) -> Dict[int, int]:
    pf = self.segments[seg_id]
    fr = self.bp.fetch_page(pf, 0)
    try:
        raw = bytes(fr.data[DIR_OFFSET:]).rstrip(b"\x00")
        if not raw:
            return {}

        obj = json.loads(raw.decode("utf-8"))
        if not isinstance(obj, dict):
            return {}

        fb = obj.get("free_bytes", {})
        if not isinstance(fb, dict):
            return {}

        out: Dict[int, int] = {}
        for k, v in fb.items():
            pid = int(k)
            freeb = int(v)
            if pid >= 1 and freeb >= 0:
                out[pid] = freeb
        return out
    finally:
        self.bp.unpin(pf, 0, dirty=False)
```

This method reads the directory of one segment from page 0.

The stored directory is JSON data. Its main content is a mapping:

$$\texttt{page id} \rightarrow \texttt{free bytes}$$

This method:

1. fetches page 0 through the buffer pool,
2. reads the bytes after `DIR_OFFSET`,
3. strips trailing zero bytes,
4. decodes the JSON payload,
5. extracts the `free_bytes` object,
6. converts keys and values back into integers,
7. returns the directory as a Python dictionary.

If anything looks malformed, the method safely returns an empty dictionary.

```
def _write_directory(self, seg_id: int, free_bytes: Dict[int, int]) -> None:
    obj = {
        "version": 2,
        "free_bytes": {str(pid): int(fb) for pid, fb in sorted(free_bytes.items())},
    }
    payload = json.dumps(obj, separators=(",", ":"), ensure_ascii=False).encode("utf-8")

    pf = self.segments[seg_id]
    fr = self.bp.fetch_page(pf, 0)
    try:
        max_len = len(fr.data) - DIR_OFFSET
        if len(payload) > max_len:
            raise ValueError(
                f"Directory too large for page 0 of segment {seg_id}: "
                f"{len(payload)} > {max_len}"
            )
        fr.data[DIR_OFFSET:] = b"\x00" * (len(fr.data) - DIR_OFFSET)
        fr.data[DIR_OFFSET:DIR_OFFSET + len(payload)] = payload
    finally:
        self.bp.unpin(pf, 0, dirty=True)
    if self.FLUSH_ON_WRITE:
```

```
22          self.bp.flush_page(pf, 0)
```

This method writes the directory back into page 0.

First, it builds a JSON object with:

- a version number,
- the `free_bytes` dictionary.

Then it encodes that object into bytes and stores it after `DIR_OFFSET`.

Why zero-fill the region first? Because if the new directory payload is shorter than the old one, old bytes would otherwise remain in page 0 and corrupt the metadata.

```
def _recompute_directory(self, seg_id: int) -> Dict[int, int]:
    pf = self.segments[seg_id]
    out: Dict[int, int] = {}
    for pid in range(1, pf.page_count):
        fr = self.bp.fetch_page(pf, pid)
        try:
            sp = SlottedPage(fr.data)
            out[pid] = sp.free_space()
        finally:
            self.bp.unpin(pf, pid, dirty=False)
    return out
```

This method rebuilds the free-space directory by scanning the actual data pages of a segment.

It is useful when:

- the directory is missing,
- the directory is suspected to be stale,
- the heap table is reopening older files and wants a safe reconstruction path.

```
def _ensure_directory_initialized(self, seg_id: int) -> None:
    fb = self._read_directory(seg_id)
    if not fb and self.segments[seg_id].page_count > 1:
        fb = self._recompute_directory(seg_id)
    self._write_directory(seg_id, fb)
```

This method guarantees that a segment has a usable directory.

If the segment already contains data pages but the directory is empty, the method reconstructs the directory by scanning the pages. Then it writes the final result back into page 0.

```
def _set_free_bytes(self, seg_id: int, pid: int, freeb: int) -> None:
    if pid < 1:
        return
    fb = self._read_directory(seg_id)
    fb[pid] = max(0, int(freeb))
    self._write_directory(seg_id, fb)
```

This updates the directory entry for one page.

If a tuple is inserted or deleted, the free space of that page changes. So the directory must be updated accordingly.

Note. This implementation rewrites the whole directory even when only one page entry changes. That is not the most efficient design, but it is simple and easy to understand.

7.4.3 Segment Naming and Allocation

```
def _segment0_path(self) -> str:
    return self.segments[0].file_path
```

This returns the file path of segment 0. It is used as the naming base for later segments.

```
def _make_segment_path(self, seg_id: int) -> str:
    seg0 = os.path.basename(self._segment0_path())
    dirn = os.path.dirname(self._segment0_path()) or "."
    stem = seg0[:-6] if seg0.endswith(".pages") else seg0
    return os.path.join(dirn, f"{stem}.seg{seg_id}.pages")
```

This constructs the filename for a new segment.

For example, if segment 0 is:

```
Student.pages
```

then later segments become:

```
Student.seg1.pages
Student.seg2.pages
...
```

This keeps the physical organization of one logical table easy to recognize.

```
def _allocate_new_segment(self) -> int:
    seg_id = len(self.segments)
    path = self._make_segment_path(seg_id)
    pf = PageFile(path, page_size=self.segments[0].page_size)
    self.segments.append(pf)
    self._ensure_directory_initialized(seg_id)
    return seg_id
```

This allocates a brand-new segment file. The new segment uses the same page size as segment 0, is appended to the segment list, and has its directory initialized immediately.

```
def _segment_has_capacity(self, pf: PageFile) -> bool:
    return pf.page_count < self.SEGMENT_MAX_PAGES
```

This checks whether a segment can still allocate more pages.

```
def _allocate_data_page(self, seg_id: int) -> int:
    pf = self.segments[seg_id]
    fr = self.bp.new_page(pf)
    pid = fr.pref.page_id
    if pid == 0:
        self.bp.unpin(pf, pid, dirty=False)
        fr = self.bp.new_page(pf)
        pid = fr.pref.page_id
    SlottedPage.init_empty(fr.data)
    freeb = SlottedPage(fr.data).free_space()
    self.bp.unpin(pf, pid, dirty=True)
    if self.FLUSH_ON_WRITE:
        self.bp.flush_page(pf, pid)
    self._set_free_bytes(seg_id, pid, freeb)
    return pid
```

This allocates a fresh data page inside one segment.

Important detail. If the new page turns out to be page 0, the method skips it and allocates again. That is because page 0 is reserved for header and directory metadata, so real tuple storage must begin at page 1.

After allocating the page, the method:

- initializes it as an empty slotted page,
- computes its initial free space,
- flushes it if needed,
- records its free space in the directory.

```
def _data_page_ids(self, seg_id: int) -> Iterable[int]:
```

```
    pf = self.segments[seg_id]
    return range(1, pf.page_count)
```

This returns all valid data-page IDs in one segment. Page 0 is excluded because it is metadata only.

7.4.4 Direct RID Helpers

```
def read_where(self, rid_set: set) -> List[Tuple[GlobalRID, Row]]:
    rows: List[Tuple[GlobalRID, Row]] = []
    for (seg_id, pid, sid) in sorted(rid_set):
        pf = self.segments[seg_id]
        fr = self.bp.fetch_page(pf, pid)
        try:
            sp = SlottedPage(fr.data)
            payload = sp.read(sid)
            if payload is None:
                continue
            rows.append(((seg_id, pid, sid), self.schema.decode(payload)))
        finally:
            self.bp.unpin(pf, pid, dirty=False)
    return rows
```

This method reads rows directly using a set of known RIDs.

Why is this useful? Because if the caller already knows the tuple IDs, perhaps from an index lookup, there is no need to scan the whole table. The heap table can go directly to the requested pages and slots.

Missing or deleted RIDs are skipped.

7.4.5 Heap Operations

```
def insert(self, row: Row) -> GlobalRID:
    payload = self.schema.encode(row)
    need = len(payload) + SLOT_SIZE

    seg_id = 0
    while True:
        if seg_id >= len(self.segments):
            seg_id = self._allocate_new_segment()

        pf = self.segments[seg_id]
        fb = self._read_directory(seg_id)
        candidates = [pid for pid, freeb in fb.items() if 1 <= pid < pf.page_count and freeb >= need]
```

This is the beginning of the insertion algorithm.

First, the row is encoded into bytes using the schema. Then the method computes how much space is required:

$$\texttt{needed space} = \texttt{payload bytes} + \texttt{one slot entry}$$

Then it searches segments in order and uses the page directory to find candidate pages with enough free space.

```
for pid in candidates:
    fr = self.bp.fetch_page(pf, pid)
    try:
        sp = SlottedPage(fr.data)
        sid = sp.insert(payload)
        if sid is None:
            sp.compact()
            sid = sp.insert(payload)
            if sid is None:
                self.bp.unpin(pf, pid, dirty=False)
                self._set_free_bytes(seg_id, pid, sp.free_space())
                continue

        new_free = sp.free_space()
        self.bp.unpin(pf, pid, dirty=True)
        if self.FLUSH_ON_WRITE:
            self.bp.flush_page(pf, pid)
        self._set_free_bytes(seg_id, pid, new_free)
        return (seg_id, pid, sid)
```

```
    except Exception:
        self.bp.unpin(pf, pid, dirty=False)
        raise
```

For each candidate page, the heap table tries to insert the payload.

If insertion fails, it attempts page compaction first and tries again. This is important because a page may have enough total reclaimable space but not enough contiguous free space before compaction.

If insertion succeeds, the page is unpinned, optionally flushed, the directory is updated, and the new `GlobalRID` is returned.

```
if self._segment_has_capacity(pf):
    pid = self._allocate_data_page(seg_id)
    fr = self.bp.fetch_page(pf, pid)
    try:
        sp = SlottedPage(fr.data)
        sid = sp.insert(payload)
        if sid is None:
            self.bp.unpin(pf, pid, dirty=False)
            raise RuntimeError("Insert failed on fresh page (record too large).")

        new_free = sp.free_space()
        self.bp.unpin(pf, pid, dirty=True)
        if self.FLUSH_ON_WRITE:
            self.bp.flush_page(pf, pid)
        self._set_free_bytes(seg_id, pid, new_free)
        return (seg_id, pid, sid)
    except Exception:
        self.bp.unpin(pf, pid, dirty=False)
        raise

seg_id += 1
```

If no existing page can hold the tuple, the heap table tries to allocate a fresh data page in the current segment.

If even a fresh page cannot hold the record, then the record is too large for one page and the method raises an error.

If the segment itself is full, insertion continues in the next segment.

So the full insertion policy is:

1. try candidate pages from the directory,
2. compact if needed,
3. if no page fits, allocate a fresh page,
4. if the segment is full, move to the next segment,
5. if needed, create a new segment.

```
def delete_where(self, rid_set: set) -> List[Tuple[GlobalRID, Row]]:
    deleted_rows: List[Tuple[GlobalRID, Row]] = []
    for (seg_id, pid, sid) in sorted(rid_set):
        pf = self.segments[seg_id]
        fr = self.bp.fetch_page(pf, pid)
        try:
            sp = SlottedPage(fr.data)
            payload = sp.read(sid)
            if payload is None:
                continue
            row = self.schema.decode(payload)
            if not sp.delete(sid):
                continue
            usable = sp.reclaimable_space()
            self._set_free_bytes(seg_id, pid, usable)
            deleted_rows.append(((seg_id, pid, sid), row))
        finally:
            self.bp.unpin(pf, pid, dirty=True)
        if self.FLUSH_ON_WRITE:
            self.bp.flush_page(pf, pid)
    return deleted_rows
```

This method deletes rows directly using a set of RIDs.

For each target RID, it:

- fetches the page,
- reads the tuple payload,
- decodes the row for reporting,
- deletes the slot,
- computes reclaimable space,
- updates the directory entry,
- appends the deleted row to the output list.

Why use `reclaimable_space()` instead of `free_space()`? Because after deletion, some bytes may be reclaimable but not yet compacted into one contiguous free region. This is an important detail of slotted-page behavior.

```python
def update_where(self, rid_set: set, updates: dict):
    """
    Physical RID-based update helper.

    This method only performs storage-level replacement. It does not enforce
    primary-key uniqueness, foreign-key existence, or cascading actions.
    Those rules belong in the database layer.

    Returns a list of:
        (old_rid, new_rid, old_row, new_row)
    """
    updated = []
    old_rows = self.read_where(rid_set)
    for old_rid, old_row in old_rows:
        new_row = dict(old_row)
        new_row.update(updates)
        new_row = self.schema.validate_row(new_row)

        new_rid = self.insert(new_row)
        deleted = self.delete_where({old_rid})
        if not deleted:
            self.delete_where({new_rid})
            continue
        updated.append((old_rid, new_rid, old_row, new_row))
    return updated
```

This method performs a physical update by:

1. reading the old rows by RID,
2. building new row versions,
3. validating the new rows through the schema,
4. inserting the new rows,
5. deleting the old rows,
6. returning both old and new RIDs.

This means update is implemented as:

`insert new tuple + delete old tuple`

That is a common physical design because the new row may have a different payload length and may not fit into the old location.

Very important. The RID may change after update. That is why the method returns both `old_rid` and `new_rid`.

Also very important. This method only performs storage replacement. It does not enforce primary-key uniqueness, foreign-key existence, or cascade/restrict rules. Those must be enforced in `db.py`.

```
def scan(self) -> Iterable[Tuple[GlobalRID, Row]]:
    for seg_id, pf in enumerate(self.segments):
        for pid in self._data_page_ids(seg_id):
            fr = self.bp.fetch_page(pf, pid)
            try:
                sp = SlottedPage(fr.data)
                items: List[Tuple[int, bytes]] = []
                for sid in sp.iter_rids():
                    payload = sp.read(sid)
                    if payload is not None:
                        items.append((sid, bytes(payload)))
            finally:
                self.bp.unpin(pf, pid, dirty=False)

            for sid, payload in items:
                yield (seg_id, pid, sid), self.schema.decode(payload)
```

This method performs a full table scan.

It iterates through:

- every segment,
- every data page in that segment,
- every live slot in that page.

Then it yields:

$$(\texttt{GlobalRID}, \texttt{decoded row})$$

Why collect `items` first? Because the method wants to unpin the page before decoding rows. That reduces how long the page stays pinned in the buffer pool.

```
def close(self) -> None:
    for pf in self.segments:
        pf.close()
```

This closes all physical segment files of the table.

Since one logical table may span many segment files, closing the table means closing every segment.

What this heap-table layer enforces

- segmented physical storage,
- page-directory-based free-space management,
- direct RID lookup,
- slotted-page insertion,
- slotted-page deletion,
- physical update by insert-plus-delete,
- full heap scan,
- correct use of the buffer pool.

8 Indexing

Indexing is a fundamental technique in database systems that improves query performance by providing efficient access paths to tuples without requiring a full table scan. Instead of examining every row in a relation, an index maintains an auxiliary structure that maps search-key values to the physical locations of matching records, allowing the DBMS to locate data much more quickly. In EduDB, indexing plays an important role in connecting the storage layer and the query execution layer, since the optimizer can choose an appropriate index to accelerate selections, range queries, and other operations. In this section, we introduce two common indexing methods used in EduDB—Hash indexing and B+Tree indexing—and explain how each supports different types of query workloads.

8.1 Hash Index

A hash index is an index structure designed primarily for equality search. Its main purpose is to accelerate queries of the form

$$A = c$$

where A is an indexed attribute and c is a constant search key. Instead of scanning every tuple in a table, the DBMS applies a hash function to the search key, uses the result to locate a bucket, and then searches only within that bucket. In this way, a hash index provides a direct access path from a key value to the tuple locations associated with that value.

In EduDB, a hash index is best understood as a separate auxiliary structure maintained alongside the heap table. The heap table stores the actual tuples, while the hash index stores a mapping from indexed key values to tuple identifiers. Thus, the hash index does not replace the table; rather, it helps the DBMS find relevant tuples much more quickly for equality predicates.

8.1.1 Basic Idea

Conceptually, a hash index stores entries of the form

$$\text{key} \ \rightarrow \ \text{RID list},$$

where RID stands for a record identifier or tuple location. For example, if the indexed attribute is `student_id`, then a hash index might logically contain entries such as

$$100123 \ \rightarrow \ [(0, 14, 3)].$$

If the indexed attribute is `major`, then duplicate keys are possible, so an entry may look like

$$\texttt{"Computer Science"} \ \rightarrow \ [(0, 12, 1), (0, 15, 2), (1, 3, 5)].$$

The key observation is that the hash index is not storing the full tuples themselves. Instead, it stores the search key together with the locations of tuples having that key.

8.1.2 Structure of a Hash Index

A hash index is usually organized into a fixed number of buckets. A hash function $h(k)$ maps each key k to one of these buckets. If there are N buckets, a simple bucket selection rule is

$$\text{bucket\_id} = h(k) \bmod N.$$

Each bucket contains the entries whose keys hash to that bucket. In a simplified pedagogical implementation, a bucket may contain a list of pairs

$$(k, \text{RID}),$$

or it may group duplicate keys together as

$$k \ \rightarrow \ [\text{RID}_1, \text{RID}_2, \ldots].$$

For instance, suppose the system has 8 buckets. If

$$h(100123) \bmod 8 = 5,$$

then the key `100123` will be stored in bucket 5. A lookup for `100123` computes the same bucket number and searches only inside that bucket.

8.1.3 How Lookup Works

The most important operation supported by a hash index is equality lookup. Consider the query:

```
SELECT * FROM Student WHERE student_id = 100123;
```

If `student_id` has a hash index, the DBMS performs the following steps:

1. Apply the hash function to the search key `100123`.
2. Compute the bucket number.
3. Go directly to that bucket.
4. Search within the bucket for entries whose key equals `100123`.
5. Return the matching RID or RID list.
6. Fetch the actual tuple(s) from the heap table using those RID values.

This is much faster than a full table scan because the DBMS avoids examining unrelated tuples.

8.1.4 Insert, Delete and Update Operation

Whenever a new tuple is inserted into the table, the hash index must also be updated. Suppose a tuple is inserted into the heap table and receives an RID. Then the index maintenance process is:

1. Extract the value of the indexed attribute from the tuple.
2. Apply the hash function to that value.
3. Determine the correct bucket.
4. Insert a new entry into that bucket containing the key and the tuple RID.

For example, if a new student with `student_id = 100150` is inserted and its RID is $(0, 20, 4)$, then the hash index on `student_id` must add an entry mapping `100150` to $(0, 20, 4)$ in the corresponding bucket.

If a tuple is deleted from the heap table, the corresponding index entry must also be removed. Otherwise, the index would point to a tuple that no longer exists.

The delete procedure is:

1. Obtain the indexed key value of the tuple being deleted.
2. Hash that key to locate the correct bucket.
3. Search within the bucket for the matching key–RID pair.
4. Remove that pair from the bucket.

If duplicate keys are supported, the system must remove only the RID corresponding to the deleted tuple, not all entries for that key.

An update affects the hash index only if the indexed attribute changes.

Case 1: non-indexed attribute changes. If the tuple is updated but the indexed attribute remains the same, then no index change is needed.

Case 2: indexed attribute changes. If the indexed value changes from k_{old} to k_{new}, then the DBMS must:

1. remove the old entry $(k_{\text{old}}, \text{RID})$,
2. insert the new entry $(k_{\text{new}}, \text{RID})$.

So index maintenance for updates is often logically a delete followed by an insert. This illustrates an important principle: the index must remain consistent with the heap table at all times.

8.1.5 Duplicate Keys

In many real tables, an indexed attribute is not unique. For example, many students may share the same major, and many employees may belong to the same department. Therefore, a hash index must support duplicate keys.

This means the logical mapping is not always

$$\text{key} \rightarrow \text{one RID},$$

but often

$$\text{key} \to \text{a list of RIDs.}$$

For example:

$$\texttt{"Mathematics"} \to [(0,1,2),(0,8,4),(1,2,0)].$$

During lookup, the DBMS returns all RIDs associated with that key and then fetches all matching tuples.

8.1.6 What a Hash Index Does Not Do Well

A hash index is designed primarily for equality search, but its relationship to range queries is more subtle than simply saying that it "cannot" support them. In practice, whether a hash index can help with a range predicate depends on the structure of the domain, the size of the range, and whether the DBMS can efficiently enumerate all candidate key values inside that range.

Case 1: small, discrete, enumerable range. Suppose the indexed attribute is an integer-valued attribute such as `student_id`, and consider the query

```
SELECT * FROM Student
WHERE student_id >= 100010 AND student_id <= 100020;
```

Because the domain is discrete and the range is very small, the DBMS can decompose the range predicate into a sequence of equality lookups:

$$100010,\ 100011,\ 100012,\ \ldots,\ 100020.$$

In other words, the system may probe the hash index once for each value in the interval and combine all returned RID lists. In this situation, using the hash index may still be faster than a full table scan, especially if the table is large and the number of equality probes remains small. Therefore, for a small discrete range, a hash index can still be useful.

Case 2: large discrete range. Now consider a larger integer range such as

```
SELECT * FROM Student
WHERE student_id >= 100000 AND student_id <= 900000;
```

Although the same decomposition idea is still theoretically possible, it is no longer attractive in practice. The DBMS would need to perform an enormous number of independent hash probes, one for each possible key value in the interval. Even if each probe is individually efficient, the total work becomes very large. In this case, the hash index loses its advantage, because it does not provide any mechanism for scanning a contiguous interval of keys as one ordered region. Thus, for large discrete ranges, a hash index is usually inefficient.

Case 3: sparse discrete domain. Even if the attribute is discrete, the domain may be sparse. For example, employee IDs or account numbers may not appear consecutively. Consider:

```
SELECT * FROM Employee
WHERE emp_id >= 1000 AND emp_id <= 5000;
```

If only a small fraction of IDs in that interval actually exist, then the DBMS must still issue many separate equality probes that return no result. The cost of probing absent values can dominate the execution. So even though the domain is discrete, sparsity makes repeated hash probing inefficient. In such a case, the hash index does not exploit the logical continuity of the range; it merely tests values one by one.

Case 4: floating-point or continuous-looking values. Suppose the predicate is

```
SELECT * FROM Student
WHERE gpa >= 3.50 AND gpa <= 3.80;
```

Here the attribute is numeric, but the set of possible values is not naturally enumerable in the same way as a small integer interval. Even if internally the data is stored with finite precision, the DBMS generally cannot treat the predicate as a manageable set of individual equality lookups. The range is better understood as an interval rather than as a short list of exact keys. Therefore, for floating-point attributes or other effectively continuous domains, a hash index is typically not a practical structure for range evaluation.

Case 5: string ranges. Now consider a string predicate such as

```sql
SELECT * FROM Student
WHERE name >= 'Adam' AND name <= 'Brian';
```

A hash index offers little help here. Unlike integers in a short consecutive interval, strings in a lexical range are not naturally probed one by one. The DBMS cannot sensibly enumerate all possible strings between `'Adam'` and `'Brian'` and perform one hash lookup per candidate. Because hashing destroys key order, the index provides no direct way to find the first matching string and then scan forward through adjacent strings. For lexical range predicates, a hash index is therefore generally unsuitable.

Case 6: ordered access requirements. Sometimes the query does not merely ask for a range predicate, but also relies on order, as in:

```sql
SELECT * FROM Student
WHERE student_id >= 100010 AND student_id <= 100020
ORDER BY student_id;
```

or even

```sql
SELECT * FROM Student ORDER BY student_id;
```

A hash index does not preserve any ordering relationship among keys. Nearby values do not map to nearby buckets, and entries are not stored in sorted order. Therefore, even if repeated equality probes can identify the matching tuples, the index itself does not support efficient ordered traversal. Any required ordering must be imposed separately after retrieval. This is another reason why hash indexes are considered poor for general range and order-based access patterns.

8.2 External Hashing

In many database tasks, fully sorting the data is more work than we actually need. For example, in operations such as `GROUP BY`, duplicate elimination, or equality-based partitioning, our goal is simply to bring identical values together; we do not care about the global sorted order of all records. In such cases, hashing is often a more natural and more efficient technique than sorting.

The challenge in a database system is that the dataset is often much larger than available main memory. In an in-memory data structures course, one may build a standard hash table by inserting all records into RAM directly. However, this assumption no longer holds in a DBMS, where the input may span many disk pages. As a result, we need an external-memory or out-of-core hashing method that can process data incrementally using a limited number of buffer pages while still ensuring that identical values are grouped together correctly.

> **External hashing is necessary only when the DBMS cannot keep the required working state in memory.**

A natural first idea is to read part of the data into memory, build a hash table for that part, then repeat the same process for the remaining data and concatenate the results. Unfortunately, this does not solve the grouping problem. If a value appears in multiple independently processed chunks, then its occurrences may end up scattered across multiple hash tables. For instance, if the value `"Brian"` appears in two different chunks, then after concatenation the corresponding records may still be separated rather than grouped together.

To avoid this problem, we must guarantee the following property: whenever some occurrences of a value are assigned to a particular intermediate structure, all occurrences of that value must be assigned there. In other words, equal keys must always be routed to the same partition. Once this property is enforced, each distinct key appears in exactly one partition, and the partitions can later be processed independently and safely combined.

This leads to the standard external hashing approach:

1. **Partition the input** into multiple smaller groups using a hash function.
2. Ensure that all identical keys are sent to the same partition.
3. Recursively repartition any partition that is still too large to fit in memory.
4. Once a partition becomes small enough, build an in-memory hash table for it.

We use a divide-and-conquer strategy consisting of a **partition phase** and a **build phase**. Assume the DBMS has B buffer pages available in memory. As in external sorting, memory is limited, so we must carefully allocate these buffers during execution.

Buffer usage. During partitioning, we reserve:

- 1 page as the **input buffer**, and
- $B - 1$ pages as **output buffers**.

Each output buffer corresponds to one partition.

Partition phase. We scan the input relation page by page from disk into the input buffer. For each record, we apply a hash function h_1 to its partitioning attribute and use the result to decide which of the $B - 1$ output partitions it belongs to. The record is then appended to the corresponding output buffer. Whenever an output buffer becomes full, it is flushed to disk. All pages written for the same partition are stored contiguously or logically associated as one partition file.

After one pass over the input, we obtain at most $B - 1$ disk partitions. The crucial property is:

> **If two records have the same key value, then they hash to the same partition under the partitioning hash function.**

Therefore, every occurrence of a given key is guaranteed to lie in exactly one partition after the partition phase.

Why partitioning helps. Because identical values always go to the same partition, the global grouping problem is reduced to a collection of smaller local grouping problems. Instead of handling the full relation at once, we only need to process one partition at a time.

When a partition is small enough. Suppose one partition occupies at most B pages. Then it can fit entirely in memory, so we can load it into the buffer pool and build a normal in-memory hash table for that partition. At this point, the problem for that partition is solved.

Recursive repartitioning. Some partitions may still be too large to fit in memory after the first pass. In that case, we apply another partitioning pass to that partition only, using a different hash function, say h_2. Using a different hash function is important: if we reused the same one, each record would be sent to the same partition as before, and the large partition would not shrink. By changing the hash function, we redistribute records more finely into smaller subpartitions.

This repartitioning process may continue recursively:

$$\text{large partition} \;\rightarrow\; \text{subpartitions} \;\rightarrow\; \text{smaller subpartitions} \;\rightarrow\; \cdots$$

until every partition fits in memory.

Build phase. Once a partition is small enough, we switch to the build phase. We load the partition into memory and construct an in-memory hash table using a build hash function, which may again differ from the partitioning hash function. This final hash table groups identical keys together efficiently and supports the required operation, such as duplicate elimination, grouping, or hash-based join processing.

8.2.1 Why We Need Different Hash Functions

It is important to distinguish between the roles of the hash functions used in external hashing:

- A **partitioning hash function** decides which disk partition a record belongs to.
- A **build hash function** organizes records inside an in-memory hash table once the partition fits in RAM.

If recursive partitioning is needed, each new partitioning pass should use a new hash function. Otherwise, oversized partitions will not be broken down further in any meaningful way. Conceptually, each partitioning pass refines the data distribution, while the build phase performs the final in-memory organization.

8.2.2 Drawbacks of Hashing and Recursive External Hashing

Why hashing is not always ideal. Although hashing is very effective for equality-based operations, it also has several important limitations. The most significant issue is data skew. Hashing works best when the input keys are distributed relatively uniformly, so that the hash function spreads records evenly across partitions or buckets. In practice, however, real datasets are often far from uniform. Some values may occur much more frequently than others,

and some hash functions may map many values to the same partition. When this happens, the resulting partitions become unbalanced: a few partitions may be very large, while many others remain small. This is undesirable because the large partitions dominate the total cost and may not fit into memory, forcing additional partitioning passes.

Another limitation is that not all hash functions are equally good. In the ideal case, we would like a uniform hash function, meaning that records are distributed as evenly as possible across all output partitions. A non-uniform hash function, by contrast, creates uneven partitions and increases the chance that recursive repartitioning will be necessary. Thus, the practical performance of external hashing depends not only on the number of buffer pages available, but also on the quality of the hash function and the distribution of the underlying data.

8.2.3 Example: GROUP BY

Consider the SQL query

```
SELECT major, COUNT(*)
FROM Student
GROUP BY major;
```

Assume that the `Student` table is very large, the available main memory is limited, and the in-memory hash table needed for grouping does not fit in memory all at once. In this case, the DBMS cannot simply scan the entire relation and maintain a single global hash table of the form

$$\texttt{major} \rightarrow \texttt{count}$$

for the whole table. Instead, it uses external hash aggregation, and if necessary, it applies the partitioning step recursively.

Goal. The goal of the query is to compute the count for each distinct value of `major`. Conceptually, the final result has the form

$$\texttt{major} \rightarrow \texttt{count},$$

for example:

$$\begin{aligned} \texttt{CS} &\rightarrow 420 \\ \texttt{Math} &\rightarrow 310 \\ \texttt{English} &\rightarrow 205 \\ \texttt{Biology} &\rightarrow 280 \\ \texttt{History} &\rightarrow 150 \\ &\vdots \end{aligned}$$

The difficulty is that the input relation is too large to support this grouping in one pass using only RAM.

Memory setting. Assume the DBMS has

$$B = 4$$

buffer pages available in memory. A standard partitioning configuration is:

- 1 input buffer page, and
- $B - 1 = 3$ output buffer pages.

Therefore, in one partitioning pass, the DBMS can create at most 3 partitions simultaneously.

Step 1: Scan the Relation and Create the First-Level Partitions Suppose the table contains tuples of the form

$$(\texttt{StudentID}, \texttt{Name}, \texttt{Major}),$$

and suppose the major values appearing in the table include

$$\texttt{CS}, \texttt{Math}, \texttt{English}, \texttt{Biology}, \texttt{History}, \texttt{Physics}, \texttt{Chemistry}, \texttt{Music}.$$

During the first partitioning pass, the DBMS hashes each tuple on the grouping key `major` using a partitioning hash function h_1. The partition number is computed as

$$p = h_1(\texttt{major}) \bmod 3.$$

Assume that the hash function distributes the major values as follows:

$$\texttt{CS},\ \texttt{Math},\ \texttt{Physics} \to P_0$$
$$\texttt{English},\ \texttt{Biology} \to P_1$$
$$\texttt{History},\ \texttt{Chemistry},\ \texttt{Music} \to P_2$$

As the DBMS scans the `Student` table:

1. each tuple is read into the input buffer,
2. the system computes its partition ID using h_1,
3. the tuple is appended to the corresponding output buffer,
4. whenever an output buffer becomes full, that page is flushed to disk as part of the corresponding partition.

At the end of this pass, the DBMS has created three temporary partition files on disk:

$$P_0, \qquad P_1, \qquad P_2.$$

Step 2: Check the Sizes of the Partitions Suppose that after the first partitioning pass, the partition sizes are:

$$|P_0| = 7 \text{ pages}$$
$$|P_1| = 2 \text{ pages}$$
$$|P_2| = 3 \text{ pages}$$

Since the memory capacity is only

$$B = 4 \text{ pages},$$

we compare each partition size against the available memory:

- P_1 fits in memory because $2 \leq 4$,
- P_2 fits in memory because $3 \leq 4$,
- P_0 does not fit in memory because $7 > 4$.

This means:

- P_1 and P_2 can be processed directly by in-memory aggregation,
- P_0 is too large and must be partitioned again.

This is where the recursive part of recursive external hashing begins.

Step 3: Recursively Repartition the Oversized Partition Now consider only the oversized partition P_0. The DBMS scans P_0 and applies a new partitioning hash function h_2 to repartition it. The new partition number is

$$p' = h_2(\texttt{major}) \bmod 3.$$

Inside P_0, suppose the only major values present are:

$$\texttt{CS},\ \texttt{Math},\ \texttt{Physics}.$$

Assume that under the second hash function h_2, these values are distributed as:

$$\texttt{CS} \to P_{00}$$
$$\texttt{Math} \to P_{01}$$
$$\texttt{Physics} \to P_{02}$$

So the DBMS scans P_0 and creates three second-level partitions:

$$P_{00}, \qquad P_{01}, \qquad P_{02}.$$

Suppose their sizes are:

$$|P_{00}| = 3 \text{ pages}$$
$$|P_{01}| = 2 \text{ pages}$$
$$|P_{02}| = 2 \text{ pages}$$

Now all of them fit into memory because each satisfies

$$|P_{0i}| \leq B.$$

Therefore, the recursion stops at this point.

Step 4: Aggregate One Final Partition at a Time At this stage, every final partition fits in memory:

$$P_{00}, \quad P_{01}, \quad P_{02}, \quad P_1, \quad P_2.$$

The DBMS now processes these partitions one by one. For each final partition, it performs the following steps:

1. load the partition into memory,
2. build an in-memory hash table,
3. compute the local counts for the keys appearing in that partition,
4. output the grouped result for that partition.

For example, when processing P_{00}, the DBMS may build an in-memory table such as

$$\texttt{CS} \to \texttt{count}.$$

When processing P_{01}, it may build

$$\texttt{Math} \to \texttt{count}.$$

When processing P_1, it may build

$$\texttt{English} \to \texttt{count}, \qquad \texttt{Biology} \to \texttt{count}.$$

Because the external hashing process guarantees that all tuples with the same `major` are routed to the same final partition, the local counts computed in each final partition are already the final correct counts for those groups. No later merging of same-key counts across different partitions is needed.

This is a good example for illustrating how external hashing works, but it is not a very realistic real-world case. The number of majors is usually small, so the DBMS can simply build an in-memory hash table while scanning the table once. Using that hash table, it can easily count and output the number of students in each major.

8.2.4 Example 2: External Hashing for a Large Equality Join

Consider the following query:

```sql
SELECT O.order_id, C.customer_name
FROM Orders O
JOIN Customers C
  ON O.customer_id = C.customer_id;
```

This is a much better example for external hashing than a query such as `GROUP BY major`, because the difficulty here is not merely reading the input tables. The real difficulty is that the DBMS must organize tuples by the join key so that matching tuples can be found efficiently.

A full scan alone is not enough. A full table scan can read all pages of `Orders` and `Customers`, but scanning by itself does not solve the join problem. To perform the join efficiently, the DBMS would like to build a hash table on the join key `customer_id`. If one relation is small enough, this is easy: build an in-memory hash table on the smaller relation, then scan the larger relation and probe the hash table.

However, if the smaller relation is still too large to fit in memory, then a single in-memory hash join is no longer possible. In that case, the DBMS needs an external-memory version of hashing.

When external hashing becomes necessary. External hashing is necessary not because the table itself is large, but because the working state of the hash-based operation is too large to fit in memory. In this example, the working state is the hash table built on the join key. If the DBMS cannot hold that hash table in memory, then it must first partition the data into smaller pieces.

Main idea. The DBMS partitions both relations using the same hash function on the join key `customer_id`. This guarantees that if two tuples can join, they will be sent to the same partition pair.

Let the partitions be:

$$O_0, O_1, \ldots, O_{k-1} \qquad \text{and} \qquad C_0, C_1, \ldots, C_{k-1}$$

where each tuple is assigned by:

$$h(\texttt{customer\_id}) \bmod k$$

Then a tuple from `Orders` in partition O_i can only join with tuples from `Customers` in partition C_i. There is no need to compare it with tuples from any other partition.

Why partitioning helps. Partitioning turns one large join problem into many smaller join problems. The goal is to make each partition of the smaller relation small enough to fit in memory. Once that happens, each partition pair can be joined separately.

Important clarification: both partitions do not need to fit in memory at once. A common misunderstanding is to think that, for partition pair (O_i, C_i), the DBMS must load both partitions fully into memory at the same time. That is usually not necessary.

Instead, the DBMS normally chooses the smaller partition as the build side and the larger partition as the probe side:

1. Load the smaller partition (say C_i) into memory.
2. Build an in-memory hash table on `customer_id`.
3. Scan the matching partition O_i page by page.
4. Probe the in-memory hash table for each tuple in O_i.
5. Output joined tuples as matches are found.

So the requirement is not:

> both O_i and C_i must each fit in half of memory.

Instead, the real requirement is:

> the build-side partition must fit in memory, while the probe-side partition can be streamed page by page.

Streaming output. The DBMS also does not need to keep the entire join result in memory. When a match is found, the join operator can immediately construct the joined tuple and place it into an output buffer. Once that buffer page is full, it can be flushed to disk or passed to the next operator in the query plan.

Thus, memory is mainly used for:

- the build-side hash table,
- one input buffer for the probe relation,
- one output buffer for produced join tuples.

The full result table can be generated incrementally. This is a standard streaming behavior in DBMS query execution.

8.3 HashIndex Class

This class implements a **very small in-memory hash index** for EduDB. Its purpose is to support efficient **exact-match lookup** on one indexed column.

The core idea is simple:

$$\texttt{key} \longrightarrow \texttt{list of GlobalRID}$$

That means the index maps each search key to a list of record identifiers pointing to the matching rows in the table. This implementation is intentionally minimal.

This is exactly what a DBMS index usually does:

- the table stores the real tuples,
- the index stores keys and references to those tuples.

Why the Value Is a List of RIDs? A single key may correspond to multiple tuples. For example, if we index the column `major` in a student table, then many students may have the same major:

student_id	name	major
101	Alice	CS
102	Bob	Math
103	Carol	CS

Then the hash index might conceptually store:

$$\texttt{"CS"} \longrightarrow [\texttt{RID}_{Alice}, \texttt{RID}_{Carol}]$$
$$\texttt{"Math"} \longrightarrow [\texttt{RID}_{Bob}]$$

So the value must be a **list** of RIDs rather than just one RID.

The file defines one public class `HashIndex`. Its job is to:

- remember which table is indexed,
- remember which column is indexed,
- map each key to the list of matching RIDs,
- support equality lookup,
- support incremental updates when rows are inserted or deleted.

8.3.1 Method `__init__`

```python
def __init__(self, table_name: str, key_column: str):
    self.table_name = table_name
    self.key_column = key_column
    self.map: Dict[Any, List[GlobalRID]] = {}
```

Purpose This constructor creates a brand-new, empty hash index.

- `self.table_name`: the name of the indexed table,
- `self.key_column`: the column on which the index is built,
- `self.map`: the internal dictionary implementing the hash index.

8.3.2 Method `build`

```python
def build(self, table: HeapTable) -> None:
    self.map.clear()
    for rid, row in table.scan():
        key = row.get(self.key_column)
        self.map.setdefault(key, []).append(rid)
```

Purpose This method builds the entire hash index from a full table scan.

It is used when:

- a new index is first created,
- the index needs to be rebuilt from the table contents,
- or during initialization after loading existing data.

Step 1: clear old contents This removes all existing entries from the index.

Why is this important? Because `build()` is supposed to reconstruct the index from scratch. if we did not clear the old contents first, then repeated builds would duplicate entries.

Step 2: scan the whole table. This iterates through every row in the heap table.

Step 3: extract the indexed key. This retrieves the value of the indexed column from the row.

For example, if:

- `self.key_column = "major"`

and the row is:

```
{"student_id": 101, "name": "Alice", "major": "CS"}
```

then:

```
key = "CS"
```

Step 4: insert the RID into the hash map

```
self.map.setdefault(key, []).append(rid)
```

This is the most important line in the method.

What does `setdefault` do? `setdefault(key, default)` means:

- if `key` is already in the dictionary, return its existing value,
- otherwise insert `key` with the given default value and return that default.

So here:

```
self.map.setdefault(key, [])
```

means:

- if the key already exists, return its RID list,
- otherwise create a new empty RID list for that key.

Then:

```
.append(rid)
```

adds the current record identifier into that list.

8.3.3 Method `insert`

```
def insert(self, key: Any, rid: GlobalRID) -> None:
    self.map.setdefault(key, []).append(rid)
```

Purpose. This method incrementally adds one new index entry. It is typically called when a new row is inserted into the table.

8.3.4 Method `delete`

```
def delete(self, key: Any, rid: GlobalRID) -> None:
    lst = self.map.get(key)
    if not lst:
        return
    try:
        lst.remove(rid)
    except ValueError:
        pass
    if not lst:
        self.map.pop(key, None)
```

Purpose This method removes one specific (key, rid) pair from the index.

It is typically called when:

- a row is deleted from the table,
- or an indexed column is updated and the old index entry must be removed.

Step 1: get the RID list for the key

```
lst = self.map.get(key)
```

This looks up the RID list for the given key. If the key is absent, `get(key)` returns `None`.

Step 2: if the key does not exist, stop

```
if not lst:
    return
```

This means:

- if the key is not in the index,
- or if its RID list is empty,
- then there is nothing to delete.

So the method simply returns.

Step 3: remove the RID from the list

```
try:
    lst.remove(rid)
except ValueError:
    pass
```

The call:

```
lst.remove(rid)
```

removes the first occurrence of `rid` from the list.

Why is this inside `try/except`? Because if the key exists but the specific RID is not in its list, Python raises a `ValueError`.

Instead of crashing, the code ignores that situation:

```
except ValueError:
    pass
```

So deletion is **safe** even if the caller asks to remove a non-existent RID.

Step 4: remove the key entirely if its list becomes empty

```
if not lst:
    self.map.pop(key, None)
```

After removing the RID, the list may become empty.

For example:

$$\texttt{"Math"} \longrightarrow [rid2]$$

After deleting `rid2`, the list becomes empty:

"Math" $\longrightarrow$ []

At that point, the code removes the key from the dictionary completely.

Why is this a good idea? Because an empty RID list carries no useful information. It is cleaner and more efficient to remove the key entirely.

8.3.5 Method `probe`

```
def probe(self, key: Any) -> List[GlobalRID]:
    return list(self.map.get(key, []))
```

Purpose This method performs an **exact-match lookup**. It answers the question:

> Which records have indexed key equal to this key?

How it works First, the method does:

```
self.map.get(key, [])
```

This means:

- if the key exists, return its RID list,
- otherwise return an empty list.

Then it wraps the result in:

```
list(...)
```

So the returned list is a **copy** of the stored RID list.

Why return a copy? This is an important design choice. If the method returned the internal list directly, then outside code could accidentally modify the hash index.

8.3.6 Limitations of This Implementation

Although it is a useful teaching implementation, it is far simpler than a production hash index.

- It is fully in memory.
- It uses Python's built-in dictionary rather than implementing buckets explicitly.
- It has no overflow-page logic.
- It has no extendible or linear hashing mechanism.
- It does not store pages on disk.
- It does not support range search.
- It does not handle concurrency or recovery internally.

So this code should be viewed as a **logical model** of hashing, not a full storage-engine implementation.

8.4 B+ Tree

A B+ Tree is a balanced, disk-oriented index structure widely used in database systems to support efficient search, insertion, deletion, and range queries. It is designed to minimize disk I/O by storing many keys in each node, so the tree remains shallow even for very large tables. In a B+ Tree, internal nodes guide the search process by storing separator keys and child pointers, while all actual data entries appear at the leaf level. The leaf nodes are typically linked from left to right, which makes sequential access and range scanning especially efficient. Because the tree automatically remains balanced after updates, every search path from the root to a leaf has the same length, providing predictable performance and making the B+ Tree one of the most important indexing structures in modern database management systems.

8.4.1 Properties of B+ Trees

A B+ tree is parameterized by an integer d, called the **order** of the tree. For a B+ tree of order d:

- Each non-root node must have between d and $2d$ entries, assuming no deletes. In symbols,

$$d \leq x \leq 2d.$$

  This is the **occupancy invariant**. Entries inside a node are **sorted**.
- If an inner node has at most $2d$ entries, then it has at most $2d + 1$ child pointers. This is the tree's **fanout**.
- In an inner node, keys in the child to the left of an entry are less than that entry; keys in the child to the right are greater than or equal to that entry.
- All leaves are at the same depth, so the tree is always balanced. Only **leaf nodes** contain the actual records (or pointers to records). Inner nodes only guide search.

Consider the following inner node of an order $d = 2$ tree:

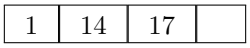
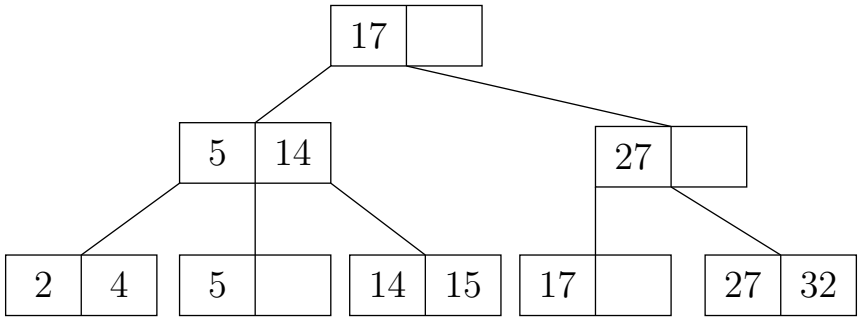


Figure 9: An inner node with entries 1, 14, and 17.

Its child pointers correspond to these key ranges:

$$(-\infty, 1), \quad [1, 14), \quad [14, 17), \quad [17, \infty).$$

Below is a B+ Tree with d = 1:



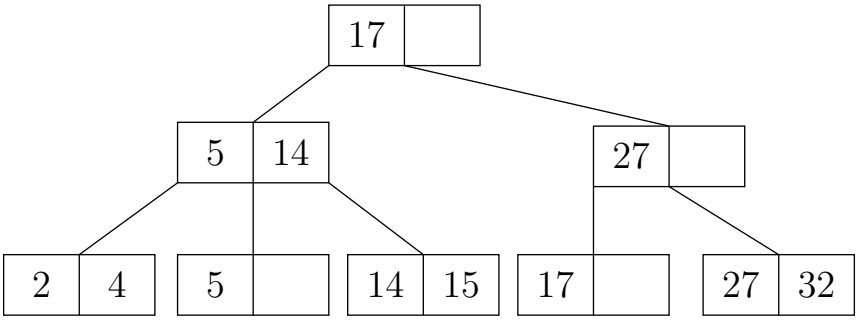
8.4.2 Insertion

To insert a key into a B+ tree:

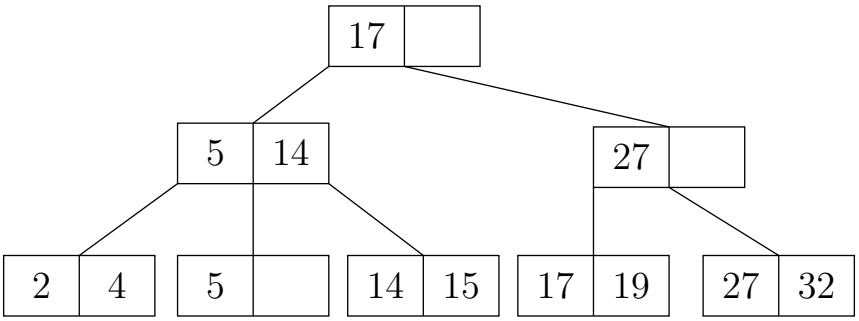
1. Find the target leaf node L by traversing from the root. Insert the new key into L in sorted order.
2. If L overflows (that is, it now has more than $2d$ entries):
   (a) Split it into L_1 and L_2. Keep d entries in L_1, and put the remaining $d + 1$ entries into L_2.
   (b) If the overflowed node was a **leaf**, **copy** the first entry of L_2 into the parent.
   (c) If the overflowed node was an **inner node**, **move** the first entry of L_2 into the parent.
   (d) Adjust pointers.
3. If the parent overflows, recurse upward. This is the only situation that can increase tree height.

Important distinction. When a leaf splits, we **copy** a key upward because all actual table keys must still remain in the leaves. When an inner node splits, we **move** a key upward because inner nodes do not store actual records; they only direct search.

We begin with the following order $d = 1$ tree.

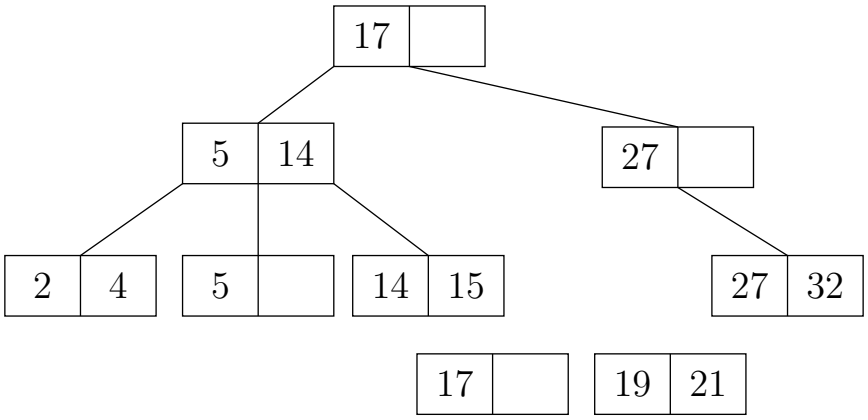


Insert 19 Key 19 belongs in the leaf containing 17. That leaf has room, so we simply insert 19:

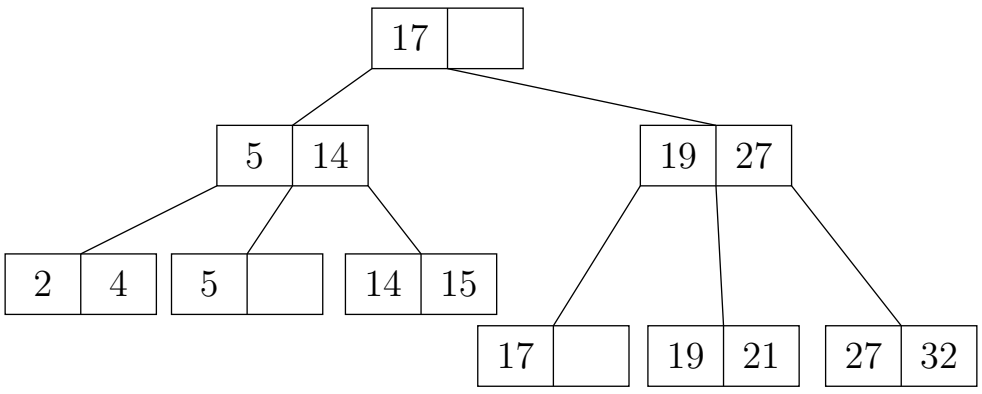


Insert 21 Now insert 21. The leaf $[17, 19]$ overflows because for $d = 1$, a leaf may hold at most 2 entries. We split it into:

$$L_1 = [17], \qquad L_2 = [19, 21].$$



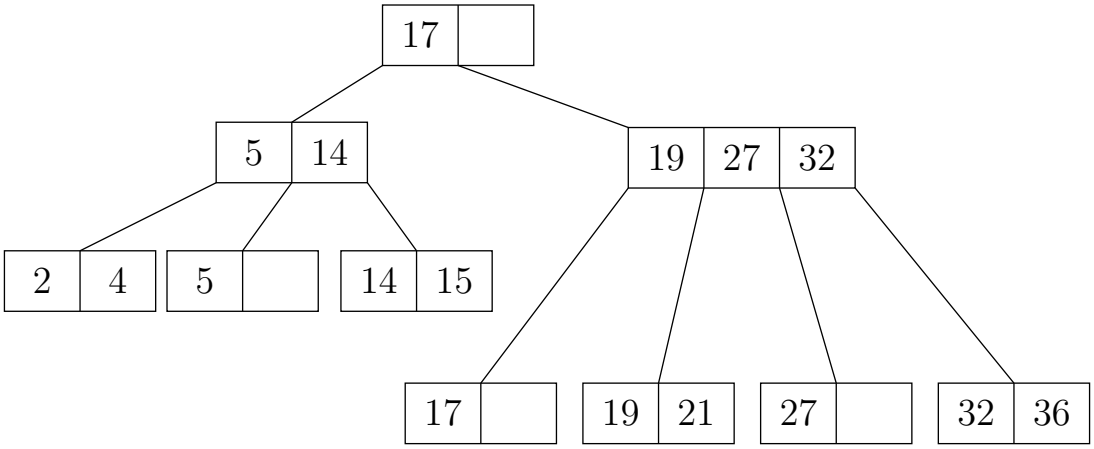
Because a **leaf** overflowed, we **copy** the first key of L_2, namely 19, into the parent.



Insert 36 Insert 36 into the leaf $[27, 32]$. This causes another leaf overflow. Split:

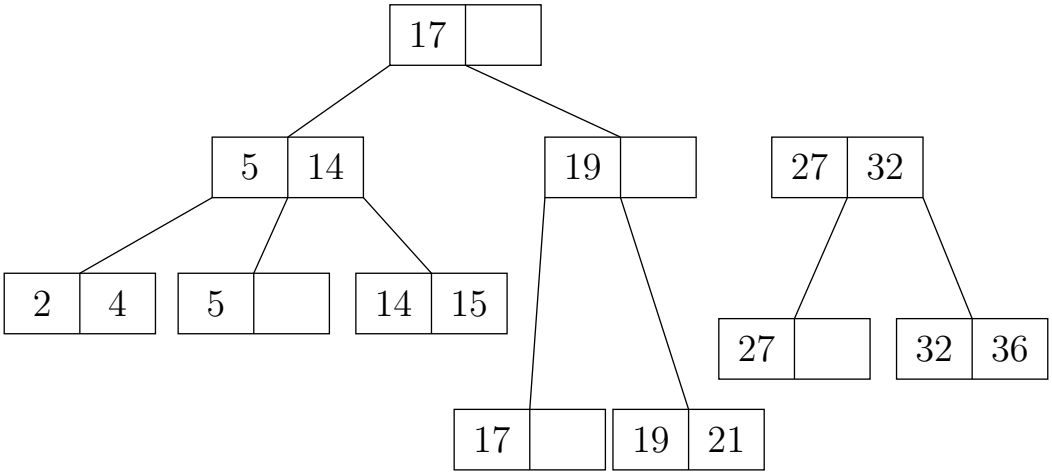
$$[27, 32, 36] \longrightarrow [27] \text{ and } [32, 36].$$

Since a leaf overflowed, we **copy** 32 into the parent.

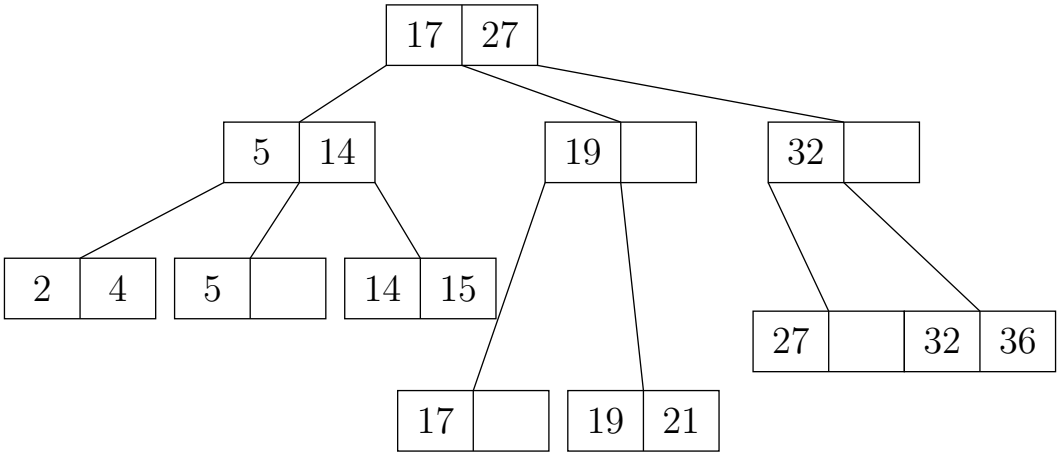


The parent becomes $[19, 27, 32]$, which overflows because for $d = 1$, inner nodes can have at most 2 entries. Now split the overflowing inner node. Let

$$L_1 = [19], \qquad L_2 = [27, 32].$$



Because an **inner node** overflowed, we **move** the first key of L_2, namely 27, into the parent.



8.4.3 Deletion

The notes use a simplified deletion rule:

1. Find the appropriate leaf.
2. Delete the unwanted value from that leaf.

In this simplified treatment, underflow handling is ignored. This is acceptable in many scenarios, because insertions are much more common than deletions, and future inserts may quickly restore occupancy.

Important reminder. We never delete inner-node keys directly, because inner-node keys are only for search guidance, not for storing actual data records.

8.4.4 How Records Are Stored in the Leaves

In EduDB, records are stored in heap-file pages, and each record can be identified by a `GlobalRID` of the form

$$(\texttt{seg\_id}, \texttt{pid}, \texttt{sid}),$$

where `seg_id` is the segment number, `pid` is the page id inside that segment, and `sid` is the slot id inside the slotted page. This matches the way EduDB scans a table: it yields a pair

$$(\texttt{GlobalRID}, \texttt{decoded\_row}).$$

Assume the table is:

```
Student(student_id, name, major, gpa)
```

and suppose several records are stored in the heap file as follows:

student_id	name	major	gpa	GlobalRID
100010	Alice	CS	3.80	(0,12,0)
100014	Bob	Math	3.40	(0,12,1)
100017	Carol	CS	3.95	(0,13,0)
100019	David	BIO	3.20	(0,13,1)
100027	Eva	CS	3.60	(0,14,0)
100032	Frank	Math	3.70	(0,14,1)

Alternative 1: By Value In Alternative 1, the leaf pages store the actual records themselves. In other words, the leaf pages are also the data pages.

For the `Student` table, a leaf might store entries such as

$$(100010, \text{Alice}, \text{CS}, 3.80), \quad (100014, \text{Bob}, \text{Math}, 3.40).$$

This is easy to understand, but it does not match the usual heap-file design, because the table records are already stored in slotted heap pages. If the B+ tree also stores full records in its leaves, then the data would effectively be duplicated.

Alternative 2: By Reference In Alternative 2, the leaf pages store pairs of the form

$$(\text{search\_key}, \texttt{GlobalRID}).$$

This is the most natural model for EduDB. The actual records remain in the heap file, and the B+ tree leaf stores only the indexed key and a pointer to the record location.

For a B+ tree on `student_id`, the leaves might store:

$$(100010, (0, 12, 0)),\ (100014, (0, 12, 1)),\ (100017, (0, 13, 0)),\ (100019, (0, 13, 1)),\ (100027, (0, 14, 0)),\ (100032, (0, 14, 1)).$$

This is the standard approach for a heap-file-based system. The table is stored once in the heap file, and the B+ tree acts as an access path. Searching the B+ tree gives a `GlobalRID`, and then EduDB fetches the corresponding page and slot to obtain the full row.

Alternative 3: By List of References Alternative 3 is useful when the indexed attribute is not unique. Instead of storing one entry per record, each leaf stores

$$(\text{search\_key}, [\texttt{GlobalRID}_1, \texttt{GlobalRID}_2, \dots]).$$

This is a good fit for a secondary index on `major`, because many students can share the same major.

For example:

$$(\texttt{BIO}, [(0, 13, 1)]), \quad (\texttt{CS}, [(0, 12, 0), (0, 13, 0), (0, 14, 0)]), \quad (\texttt{Math}, [(0, 12, 1), (0, 14, 1)]).$$

This representation can save space when many records share the same search key. It also makes the meaning of a non-unique secondary index very clear: one logical key value may point to many table records.

8.4.5 Clustering

In EduDB, the B+ tree index is built by reference: the index does not store full table records in its leaves. Instead, a leaf entry maps a search key to one or more `GlobalRID` values, where a `GlobalRID` has the form

$$(\texttt{seg\_id}, \texttt{pid}, \texttt{sid}).$$

The actual tuples remain in the heap table, stored in slotted pages. Therefore, when we discuss clustering, we are not talking about the logical ordering of keys in the B+ tree itself; the leaf keys are always sorted. Instead, we are talking about the physical layout of the heap-file pages pointed to by those leaf entries.

Suppose we build a B+ tree on the attribute `student_id` for the table

```
Student(student_id, name, major, gpa)
```

and suppose the leaf entries conceptually look like

$$(100010, [(0, 12, 0)]),\ (100014, [(0, 12, 1)]),\ (100017, [(0, 13, 0)]),\ (100019, [(0, 13, 1)]),\ (100027, [(0, 14, 0)]),\ (100032, [(0, 14, 1)]).$$

Even though these entries are sorted by `student_id` in the B+ tree, the corresponding heap-file pages may or may not be stored in a way that preserves that ordering physically.

Key idea. A clustered index means that records with nearby search-key values are likely to be stored in the same heap page or in nearby heap pages. An unclustered index means that the referenced records may be scattered across many unrelated pages.

In an **unclustered** index, the B+ tree may return a sequence of sorted keys, but the heap tuples pointed to by those keys can be located almost anywhere in the table file. For example, suppose the following records exist:

student_id	name	GlobalRID
100010	Alice	(0,25,0)
100014	Bob	(0,91,1)
100017	Carol	(0,12,0)
100019	David	(0,77,2)
100027	Eva	(0,40,1)
100032	Frank	(0,63,0)

Now consider the range query

```
SELECT * FROM Student
WHERE student_id BETWEEN 100014 AND 100019;
```

The B+ tree can find the correct leaf entries quickly, but after that EduDB must follow the returned `GlobalRID`s into the heap file. If these RIDs point to many different pages, then the DBMS may need one data-page read for each matching record. In other words, the index lookup is cheap, but the subsequent data-page fetches are expensive.

An unclustered index is still very useful for highly selective queries, such as exact-match lookups or queries that return only a few tuples. However, it becomes less efficient for large range scans, because many matching keys may require many scattered heap-page reads.

In a **clustered** index, the heap file is organized so that records with nearby `student_id` values are stored together, or at least roughly together, in the same region of the file. For example, the same logical records might instead be laid out like this:

student_id	name	GlobalRID
100010	Alice	(0,12,0)
100014	Bob	(0,12,1)
100017	Carol	(0,13,0)
100019	David	(0,13,1)
100027	Eva	(0,14,0)
100032	Frank	(0,14,1)

Now the same range query

```
SELECT * FROM Student
WHERE student_id BETWEEN 100014 AND 100019;
```

still uses the B+ tree to locate the matching key interval, but the data-page reads are much cheaper. Since the matching tuples are stored in pages 12 and 13, EduDB may need to read only those two pages to retrieve all qualifying tuples.

The advantage of clustering is not that the tree lookup becomes faster; the advantage is that the heap-page accesses become much more sequential and cache-friendly. This is why clustered indexes are especially beneficial for range predicates, ordered scans, and workloads that repeatedly read records with nearby key values.

the cost of an index is not only "how fast can I find the key?" but also "how expensive is it to fetch the actual tuples after I find them?"

8.4.6 Bulk Loading

The ordinary insertion procedure is appropriate when we already have a B+ tree and want to add new entries one by one. However, if we want to build an index from scratch in EduDB, repeated insertion is wasteful: each insertion would need its own search path, and the resulting leaf pages often start only partially full. A better strategy is bulk loading: first sort the data on the search key, then build the leaf level sequentially, and finally build upper levels on top of those leaves.

The actual table rows remain in the heap file, while the B+ tree stores entries of the form

$$(\texttt{student\_id}, [\texttt{GlobalRID}_1, \texttt{GlobalRID}_2, \ldots]).$$

For a unique attribute such as `student_id`, each key usually has exactly one `GlobalRID`, so conceptually each leaf entry behaves like

$$(\texttt{student\_id}, [\texttt{rid}]).$$

Suppose we want to build a B+ tree index on

```
Student(student_id, name, major, gpa)
```

and suppose the table scan yields the following sorted (`student_id`, `GlobalRID`) pairs:

student_id	GlobalRID
100002	(0,10,0)
100004	(0,10,1)
100005	(0,10,2)
100014	(0,11,0)
100015	(0,11,1)
100017	(0,11,2)
100019	(0,12,0)
100021	(0,12,1)
100027	(0,12,2)
100032	(0,13,0)
100036	(0,13,1)
100040	(0,13,2)
100044	(0,14,0)
100048	(0,14,1)
100050	(0,14,2)
100055	(0,15,0)
100060	(0,15,1)
100064	(0,15,2)

For readability, the figures below show only the last two digits of each `student_id`: for example, "02" means `100002`, and "64" means `100064`.

Bulk-Loading Procedure Assume an order $d = 2$ B+ tree. Then each non-root node can hold between d and $2d$ entries, so the maximum capacity of a node is 4 entries. We choose a leaf fill factor of

$$f = \frac{3}{4}.$$

Therefore, during bulk loading, each leaf is filled to 3 entries before we start a new leaf. The fill factor applies only to leaf nodes. Inner nodes are filled according to the usual B+ tree rule, up to their full capacity of $2d = 4$ entries.

The bulk-loading procedure is:

1. Sort the data on the search key.
2. Scan the sorted data from left to right and pack leaf nodes to the chosen fill factor.
3. Whenever a new leaf is started, insert a separator into the parent.
4. If an inner node overflows, split it:
   (a) keep d entries in the left inner node,
   (b) place the remaining $d + 1$ entries in the right inner node,
   (c) move the first entry of the new right inner node up to the parent,
   (d) adjust pointers.
5. Repeat until the root is stable.

Step 1: Pack the Leaf Level With fill factor 3/4, the sorted EduDB entries are packed into leaves as follows:

$$[02, 04, 05], \quad [14, 15, 17], \quad [19, 21, 27], \quad [32, 36, 40], \quad [44, 48, 50], \quad [55, 60, 64].$$

Each leaf entry really stores a key and a RID-list, such as

$$(100019, [(0, 12, 0)]), \qquad (100021, [(0, 12, 1)]), \qquad (100027, [(0, 12, 2)]).$$

But in the diagrams below, only the keys are shown so the structure is easier to read.

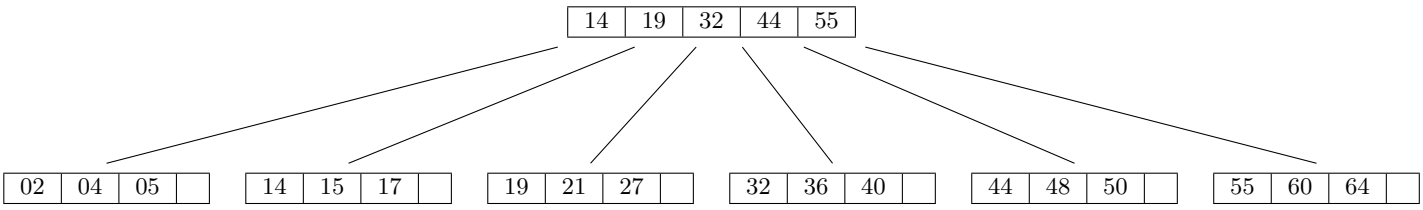


Figure 10: After packing the leaves, the parent has accumulated separators $14, 19, 32, 44, 55$, so it overflows.

At this point the leaf level is already complete and nicely packed. The problem is that the current parent has 5 entries, but an order-2 inner node can hold at most 4 entries.

Step 2: Split the Overflowing Inner Node We now split the overflowing parent just as we would split an overflowing inner node during normal insertion. Since $d = 2$, we keep 2 entries in the left node and place the remaining 3 entries in the right node:

$$L_1 = [14, 19], \qquad L_2 = [32, 44, 55].$$

Because this is an inner-node split, we do not copy a key upward. Instead, we move the first entry of the new right node upward. So the key 32 is moved to a new root.

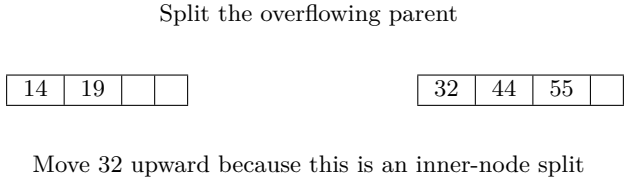


Figure 11: Temporary result after splitting the overflowing parent.

Step 3: Create the New Root After moving 32 upward, we create a new root containing that separator. The final B+ tree is:

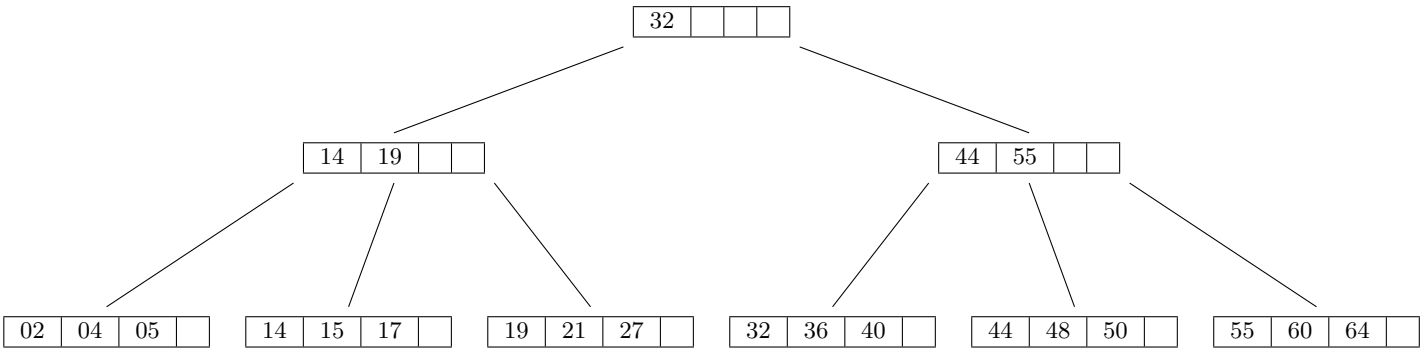


Figure 12: Final B+ tree after bulk loading the EduDB `student_id` index.

Why Bulk Loading Is Better Bulk loading improves two things at once.

Better construction cost. If we inserted the 18 keys one by one, each insertion would repeatedly traverse the tree and perform many local reorganizations. In contrast, bulk loading scans the sorted entries once, packs the leaves left-to-right, and then builds upper levels from those leaves. This makes construction more efficient.

Better leaf utilization. Because the leaves are packed deliberately to a chosen fill factor, we avoid the common situation where repeated insertions leave many half-empty leaf pages. In this example, each leaf begins with three entries out of a possible four, which is exactly the chosen fill factor.

8.5 B+ Tree Class

This file implements a **node-based, in-memory B+Tree index** for EduDB. This implementation is closes to a real B+Tree because it introduces:

- **internal nodes** for navigation,
- **leaf nodes** for actual indexed entries,
- **split propagation** when nodes overflow,
- **linked leaves** for efficient range scan,
- **duplicate-key buckets** storing multiple `GlobalRID`s for the same key,
- and a **delete** operation that removes a specific (key, rid) pair.

However, this is still a **teaching-grade** B+Tree rather than a fully industrial implementation. In particular:

- it is entirely in memory,
- it does not serialize nodes to disk pages,
- and deletion does not implement full redistribution or merge balancing.

That means this code is ideal for learning the **core logic** of B+Tree indexing without introducing too much engineering complexity.

The B+Tree index is designed to support queries such as:

- **exact lookup:** "find all tuples whose indexed key equals k,"
- **range lookup:** "find all tuples whose indexed key is between lo and hi,"
- **incremental maintenance:** insert or remove index entries as tuples are inserted, updated, or deleted.

Instead of storing full tuples in the tree, each leaf entry stores one key and a list of `GlobalRID`s:

$$\texttt{key} \longrightarrow [\texttt{GlobalRID}_1, \texttt{GlobalRID}_2, \dots]$$

Each `GlobalRID` is a record identifier that points to the tuple's physical location in the table storage.

The file defines four classes:

1. `_Node`: base class for all nodes,
2. `_LeafNode`: stores keys and RID buckets,
3. `_InternalNode`: stores separator keys and child pointers,
4. `BPlusTreeIndex`: the public wrapper class representing the whole tree.

8.5.1 Class `_Node`: Base Class

```python
class _Node:
    """Base class for internal and leaf nodes."""

    __slots__ = ("keys", "parent")

    def __init__(self):
        self.keys: List[Any] = []
        self.parent: Optional["_InternalNode"] = None

    @property
    def is_leaf(self) -> bool:
        return False
```

The `_Node` class is the shared base class of both leaf and internal nodes.

Stored fields

- `keys`: the sorted keys stored in this node,
- `parent`: a pointer to the parent internal node, or `None` if this node is the root.

Why is this useful? Both internal and leaf nodes need:

- a key list,
- a parent pointer.

So putting these fields into a shared base class reduces duplication.

8.5.2 Class `_LeafNode`: Leaf Storage

```python
class _LeafNode(_Node):
    """Leaf node: stores keys and RID buckets."""

    __slots__ = ("keys", "parent", "values", "next", "prev")

    def __init__(self):
        super().__init__()
        self.values: List[List[GlobalRID]] = []
        self.next: Optional["_LeafNode"] = None
        self.prev: Optional["_LeafNode"] = None

    @property
    def is_leaf(self) -> bool:
        return True
```

A leaf node stores the actual index entries.

Extra fields in a leaf

- `values`: a list of RID buckets,
- `next`: pointer to the next leaf,
- `prev`: pointer to the previous leaf.

Parallel structure A leaf maintains the invariant:

$$\texttt{keys[i]} \longleftrightarrow \texttt{values[i]}$$

That means:

- `keys[i]` is one indexed key,
- `values[i]` is the list of all `GlobalRID`s having that key.

Example Suppose a leaf contains:

```
keys   = [10, 20, 35]
values = [
    [(0,1,2)],
    [(0,3,4), (2,5,1)],
    [(1,2,9)]
]
```

Then:

- key 10 maps to one tuple,
- key 20 maps to two tuples,
- key 35 maps to one tuple.

Why are leaves linked? The `next` and `prev` pointers connect the leaves into a doubly-linked list. This is a standard B+Tree design because range scans often need to move from one leaf to the next.

8.5.3 Class `_InternalNode`: Navigation Nodes

```
class _InternalNode(_Node):
    """Internal node: stores separator keys and child pointers."""

    __slots__ = ("keys", "parent", "children")

    def __init__(self):
        super().__init__()
        self.children: List[_Node] = []
```

Internal nodes are used only for navigation.

Stored fields

- `keys`: separator keys,
- `children`: pointers to child nodes.

Important invariant If an internal node has m keys, it must have $m + 1$ children.

So if:

$$\texttt{keys} = [20, 50]$$

then the children conceptually represent:

$$(-\infty, 20], \quad (20, 50], \quad (50, +\infty)$$

The exact routing rule depends on how the implementation uses binary search. In this code, `bisect_right` is used when descending through internal nodes, so equal keys go to the child on the **right** side of the separator.

8.5.4 Class `BPlusTreeIndex`: Public Tree Wrapper

`Constructor`

```
class BPlusTreeIndex:

    def __init__(self, table_name: str, key_column: str, order: int = 16):
        if order < 1:
            raise ValueError("B+Tree order d must be at least 1")

        self.table_name = table_name
        self.key_column = key_column
        self.order = order

        self.min_leaf_keys = order
        self.max_leaf_keys = 2 * order

        self.min_internal_keys = order
        self.max_internal_keys = 2 * order

        self.root: _Node = _LeafNode()
```

This creates a new B+Tree index object. Initially, the tree consists of a single empty leaf node, which means a tree of height 0 is just one leaf.

8.5.5 Search Routing

```
    def _find_leaf(self, key: Any) -> _LeafNode:
        node = self.root
        while not node.is_leaf:
            node = node.children[bisect_right(node.keys, key)]
        return node
```

Purpose. This method finds the leaf where a given key belongs.

Code idea. Starting from the root:

- while the current node is not a leaf,
- choose the appropriate child using `bisect_right(node.keys, key)`,
- keep descending.

When the loop ends, the current node is the target leaf.

Why `bisect_right`? In this implementation, each separator key stored in an internal node represents the first key of the child subtree to its right. Therefore, if the search key is equal to a separator, the search must go to the right child.

For example, if an internal node has separator keys

$$[20, 50]$$

then its children are interpreted as:

$$\text{child 0: keys } < 20, \qquad \text{child 1: keys } \geq 20 \text{ and } < 50, \qquad \text{child 2: keys } \geq 50.$$

Thus, when searching for key 20, we must go to child 1 rather than child 0. This is exactly the behavior of `bisect_right`, which returns the insertion position to the right of equal keys.

This matches the leaf split rule used in the code: when a leaf splits, the separator copied up to the parent is the first key of the new right leaf.

```
    def _leftmost_leaf(self) -> _LeafNode:
        node = self.root
        while not node.is_leaf:
            node = node.children[0]
        return node
```

Purpose. Return the leftmost leaf in the tree.

How it works. Starting at the root, repeatedly take child 0 until reaching a leaf.

Why is this useful? It is used for:

- flattening all keys in order,
- finding the minimum key,
- scanning leaves from left to right.

```
    def _rightmost_leaf(self) -> _LeafNode:
        node = self.root
        while not node.is_leaf:
            node = node.children[-1]
        return node
```

Purpose. Return the rightmost leaf in the tree.

How it works. Starting at the root, repeatedly take the last child until reaching a leaf.

Why is this useful? It is used for:

- finding the maximum key,
- scanning from the far right if needed.

```
    def _leftmost_nonempty_leaf(self) -> Optional[_LeafNode]:
        leaf = self._leftmost_leaf()
        while leaf is not None and not leaf.keys:
            leaf = leaf.next
        return leaf

    def _rightmost_nonempty_leaf(self) -> Optional[_LeafNode]:
```

```
        leaf = self._rightmost_leaf()
        while leaf is not None and not leaf.keys:
            leaf = leaf.prev
        return leaf
```

Purpose. Return the first leaf from the left/right that actually contains at least one key.

Why is this needed? Because the simplified delete algorithm does **not** rebalance or remove empty leaves. So after deletions, it is possible that the leftmost/rightmost leaf exists but contains no keys. This helper avoids returning an empty leaf when the user asks for the minimum/maximum key.

How it works.

1. Find the leftmost leaf.
2. While the current leaf exists and is empty, move to `leaf.next`.
3. Return the first non-empty leaf or `None`.

8.5.6 Insert Path

```
    def insert(self, key: Any, rid: GlobalRID) -> None:
        leaf = self._find_leaf(key)
        i = bisect_left(leaf.keys, key)

        # Duplicate key -> append RID to bucket
        if i < len(leaf.keys) and leaf.keys[i] == key:
            leaf.values[i].append(rid)
            return

        leaf.keys.insert(i, key)
        leaf.values.insert(i, [rid])

        # CS186 overflow: more than 2d entries
        if len(leaf.keys) > self.max_leaf_keys:
            self._split_leaf(leaf)
```

Purpose. Insert one (key, rid) pair into the tree.

Step 1: find the target leaf. The method first calls:

`leaf = self._find_leaf(key)`

This locates the leaf where the key should go.

Step 2: find the position inside the leaf. The method then does:

`i = bisect_left(leaf.keys, key)`

This returns the sorted insertion position within the leaf.

Why `bisect_left`? Once we already know the correct leaf, that leaf still contains a **sorted list of keys**. For example, suppose the leaf stores:

$$[10, 20, 35, 50]$$

Now the problem is no longer "which leaf should contain the key?" Instead, the problem becomes:

At what position inside this leaf should the key be found or inserted?

The function `bisect_left` returns the **leftmost position** where `key` can be inserted while keeping the list sorted.

Example. If

$$\texttt{leaf.keys} = [10, 20, 35, 50]$$

then:

- `bisect_left(leaf.keys, 5)` returns 0,
- `bisect_left(leaf.keys, 20)` returns 1,
- `bisect_left(leaf.keys, 25)` returns 2,
- `bisect_left(leaf.keys, 60)` returns 4.

So this line answers the question:

> The key can be inserted while keeping the list sorted.

Why not `bisect_right`? `bisect_left` is the natural choice because it returns the position of the existing key directly if the key is already present.

For example, suppose a leaf contains:

$$[10, 20, 35, 50]$$

Then:

$$\texttt{bisect\_left(..., 20)} = 1$$

This works nicely for both cases:

- finding an existing key,
- inserting a new key in sorted order.

By contrast, if we use `bisect_right` on the same leaf, then:

$$\texttt{bisect\_right(..., 20)} = 2$$

because `bisect_right` returns the position immediately to the **right** of equal keys. It is not convenient for us to do update which we need the exact position of the key if exists.

Step 3: duplicate-key case. If the key is already present at position i, then we do **not** create a second leaf entry. Instead, we append the new RID to the existing bucket:

```
leaf.values[i].append(rid)
```

Why? Because the tree stores one leaf entry per **distinct key**, and duplicate rows are represented by multiple RIDs in the same bucket.

Step 4: new-key case. If the key is not already present, we insert:

- the key into `leaf.keys`,
- a one-element RID bucket `[rid]` into `leaf.values`.

Step 5: overflow check. A leaf may hold at most $2d$ entries. So overflow happens when:

$$\texttt{len(leaf.keys)} > 2d$$

If that happens, the method calls `_split_leaf(leaf)`.

8.5.7 Split Leaf

```
    def _split_leaf(self, leaf: _LeafNode) -> None:
        """
        - if a leaf has 2d + 1 entries, keep d in the left leaf
        - keep d + 1 in the new right leaf
        - COPY the first key of the new right leaf into the parent
        """
        right = _LeafNode()

        split = self.order  # keep d entries in the left leaf
        right.keys = leaf.keys[split:]
        right.values = leaf.values[split:]

        leaf.keys = leaf.keys[:split]
        leaf.values = leaf.values[:split]

        right.next = leaf.next
        if right.next is not None:
            right.next.prev = right
        leaf.next = right
        right.prev = leaf

        sep_key = right.keys[0]  # COPY right's first key up
        self._insert_into_parent(leaf, sep_key, right)
```

Purpose. Split a leaf that overflowed.

If a leaf temporarily has $2d + 1$ entries:

- keep d entries in the left leaf,
- put $d + 1$ entries into the new right leaf,
- copy the first key of the new right leaf into the parent.

How the code does this. It creates a new leaf called `right`, then sets:

```
split = self.order
```

So the left leaf keeps the first d entries, and the right leaf gets the rest.

Example. Suppose $d = 2$, so a leaf can hold at most 4 entries. If a leaf temporarily contains:

$$[10, 20, 25, 30, 40]$$

then after splitting:

$$\text{left leaf} = [10, 20]$$
$$\text{right leaf} = [25, 30, 40]$$

The separator copied upward is:

$$25$$

because it is the first key of the new right leaf.

Leaf-link repair. The method must also repair the linked list of leaves:

- `right.next = leaf.next`
- if that next leaf exists, set its `prev` to `right`
- `leaf.next = right`
- `right.prev = leaf`

This preserves the left-to-right leaf chain needed for range scans.

Promoting the separator. Finally, the method computes:

```
sep_key = right.keys[0]
```

and inserts that separator into the parent by calling:

```
self._insert_into_parent(leaf, sep_key, right)
```

8.5.8 Insert into Parent Node

```
    def _insert_into_parent(self, left: _Node, sep_key: Any, right: _Node) -> None:
        parent = left.parent

        # Split reached the old root -> create a new root
        if parent is None:
            new_root = _InternalNode()
            new_root.keys = [sep_key]
            new_root.children = [left, right]
            left.parent = new_root
            right.parent = new_root
            self.root = new_root
            return

        pos = parent.children.index(left)
        parent.keys.insert(pos, sep_key)
        parent.children.insert(pos + 1, right)
        right.parent = parent

        # overflow: more than 2d entries
        if len(parent.keys) > self.max_internal_keys:
            self._split_internal(parent)
```

Purpose. Handle the upward propagation after a split.

Meaning of the arguments.

- `left`: the old left node,
- `sep_key`: the separator key to insert upward,
- `right`: the new right node created by the split.

Case 1: the split reached the root. If `left.parent is None`, then the old root has split. We must create a brand-new root internal node.

The new root gets:

- `keys = [sep_key]`
- `children = [left, right]`

Then both child nodes get their `parent` pointer updated to the new root.

Why is this important? This is the only way the tree height increases.

Case 2: the node already had a parent. If the parent exists:

- find the position of `left` in the parent's child list,
- insert the separator into the parent's key list at that position,
- insert `right` immediately after `left` in the child list,
- set `right.parent = parent`.

Overflow propagation. After insertion into the parent, the parent itself may now overflow. If it has more than $2d$ keys, call `_split_internal(parent)`.

8.5.9 Split Internal Node

```python
    def _split_internal(self, node: _InternalNode) -> None:
        """
        - on overflow there are 2d + 1 keys
        - keep d keys in the left node
        - MOVE the middle separator key up to the parent
        - keep the remaining d keys in the new right node
        """
        right = _InternalNode()

        mid = self.order
        sep_key = node.keys[mid]  # MOVE this key upward

        right.keys = node.keys[mid + 1:]
        right.children = node.children[mid + 1:]
        for child in right.children:
            child.parent = right

        node.keys = node.keys[:mid]
        node.children = node.children[:mid + 1]

        self._insert_into_parent(node, sep_key, right)
```

Purpose. Split an internal node that overflowed.

If an internal node temporarily has $2d + 1$ keys:

- keep d keys in the left node,
- move the middle key up to the parent,
- keep the remaining d keys in the right node.

Important distinction from leaf split. For leaves, we **copy** a separator upward. For internal nodes, we **move** the separator upward.

This is because:

- leaf keys represent actual data entries and must remain in the leaves,
- internal keys are only routing information.

How the code does it. It chooses:

$$\texttt{mid = self.order}$$

Then:

- `sep_key = node.keys[mid]`
- left node keeps `node.keys[:mid]`
- right node gets `node.keys[mid+1:]`

Children are split consistently:

- left keeps the first $mid + 1$ children,
- right gets the remaining children.

Every child moved to the right node has its `parent` pointer updated.

Then the method inserts `sep_key` into the parent by calling `_insert_into_parent()`.

Example. Suppose $d = 2$, so a node may hold at most 4 keys. If an internal node temporarily contains:

$$[10, 20, 30, 40, 50]$$

then:

- left keeps $[10, 20]$,

- 30 is moved upward,
- right gets $[40, 50]$.

8.5.10 Probe

```python
    def probe(self, key: Any) -> List[GlobalRID]:
        leaf = self._find_leaf(key)
        i = bisect_left(leaf.keys, key)
        if i < len(leaf.keys) and leaf.keys[i] == key:
            return list(leaf.values[i])
        return []
```

Purpose. Perform exact-match lookup.

How it works.

1. Find the target leaf with `_find_leaf(key)`.
2. Use `bisect_left` to find the candidate position within that leaf.
3. If the key exists, return a **copy** of its RID bucket.
4. Otherwise return the empty list.

Why return a copy? The code does:

`return list(leaf.values[i])`

This prevents outside code from accidentally modifying the tree's internal bucket.

8.5.11 Range

```python
    def range(self, lo: Any, hi: Any) -> List[GlobalRID]:
        if hi < lo:
            return []

        out: List[GlobalRID] = []
        leaf = self._find_leaf(lo)
        first_leaf = True

        while leaf is not None:
            start = bisect_left(leaf.keys, lo) if first_leaf else 0
            first_leaf = False

            for i in range(start, len(leaf.keys)):
                key = leaf.keys[i]
                if key > hi:
                    return out
                out.extend(leaf.values[i])

            leaf = leaf.next

        return out
```

Purpose. Return all RIDs for keys in the inclusive interval:

$$lo \leq key \leq hi$$

Step 1: invalid interval check. If `hi < lo`, the method immediately returns `[]`.

Step 2: find the first relevant leaf. The method starts with:

```python
leaf = self._find_leaf(lo)
```

That finds the leaf where the lower bound would belong.

Step 3: scan through linked leaves. A flag called `first_leaf` is used so that:

- in the first leaf, scanning begins at `bisect_left(leaf.keys, lo)`,
- in later leaves, scanning starts at index 0.

Step 4: stop condition. For each key encountered:

- if the key is greater than `hi`, stop immediately and return the output,
- otherwise append all RIDs in that bucket to the result list.

Why is this efficient? Because it does not restart from the root for every key. It descends once to the first relevant leaf, then walks across the leaf chain.

Example. Suppose the leaves are:

$$[10, 20] \to [25, 30, 35] \to [40, 50]$$

Then:

`range(20, 35)`

returns all RIDs under keys:

$$20,\ 25,\ 30,\ 35$$

and stops as soon as 40 is seen.

8.5.12 Delete

```python
    def delete(self, key: Any, rid: GlobalRID) -> bool:
        """
        Remove one specific (key, rid) entry.

        - delete from the leaf only,
        - if a RID bucket becomes empty, remove that leaf entry,
        - do not rebalance or merge on delete,
        - do not remove/update internal separator keys just because a data key disappeared.

        This can leave sparse or even empty leaves, which is acceptable for a
        compact teaching implementation and still preserves correct lookup.
        """
        leaf = self._find_leaf(key)
        i = bisect_left(leaf.keys, key)

        if i >= len(leaf.keys) or leaf.keys[i] != key:
            return False

        bucket = leaf.values[i]
        try:
            bucket.remove(rid)
        except ValueError:
            return False

        # Duplicate-key bucket still has remaining RIDs
        if bucket:
            return True

        # Remove empty leaf entry; do not rebalance or touch internal separators
        leaf.keys.pop(i)
        leaf.values.pop(i)

        return True
```

Purpose. Remove one specific (key, rid) pair from the tree.

Very important design choice. This delete algorithm is deliberately simplified in teaching style:

- delete only from the leaf,
- remove the whole leaf entry only if its bucket becomes empty,
- do not rebalance,
- do not merge,
- do not update or remove separator keys in internal nodes.

Why is that acceptable? Because internal keys are only search guides. Even if some become "stale," the search path still reaches a correct leaf. The tree may become sparse, but lookup remains correct.

Detailed behavior.

1. Find the leaf where the key belongs.
2. Locate the key inside that leaf using `bisect_left`.
3. If the key is not present, return `False`.
4. Retrieve the key's RID bucket.
5. Attempt to remove the specified RID from that bucket.
6. If the RID is not in the bucket, return `False`.
7. If the bucket still contains other RIDs, return `True`.
8. If the bucket became empty, remove both the key and the bucket entry from the leaf.
9. Return `True`.

9 DB Design

9.1 Why the ER Model? (Conceptual Schema Design)

The **Entity–Relationship (ER) model** is a **conceptual data model**: it describes database structure and meaning independent of any specific DBMS. Its purpose is to provide a stable and communicable description of:

- the **database contents** (what exists in the mini-world),
- the **semantics** (what the data means),
- **interrelationships** among data,
- and key **constraints** (cardinality, participation, keys).

The ER model has three main concepts:

1. **Entities and entity types**
2. **Attributes (simple / composite / multivalued / derived)**
3. **Relationships and relationship types** (with constraints)

9.1.1 Entities, Entity Types, and Entity Sets

Entity vs. Entity Type:

- An **entity** is a specific object (an instance) in the mini-world (e.g., student #1001).
- An **entity type** groups entities with the same basic attributes (e.g., `STUDENT`).
- An **entity set** is the current collection of entity instances stored in the database for an entity type (e.g., student #1001, #1002, #1003...)..

9.1.2 Attributes and Value Sets

Attributes are properties used to describe an entity. Each attribute has a **value set** (data type / domain).

- Simple (atomic) attribute: `gpa`, `credits`.
- Composite attribute: `name` could be decomposed into (`first_name`, `last_name`). In conceptual design you may model the composite form even if you later store a single `name` field.
- Multivalued attribute: Example: a student may have multiple email addresses, one for communication, one for backup. In ER notation, multivalued attributes are drawn as **double ovals**.
- Derived attribute: `years_enrolled` can be derived from `enroll_date` and "today." In ER notation, derived attributes are often drawn as **dashed ovals**.

An attribute (or set of attributes) whose values uniquely identify each entity is a **key**.

- **Simple key**: one attribute (e.g., `student_id`).
- **Composite key**: multiple attributes together form a key.
- An entity type can have **more than one key**. The database designer chooses one as the **primary key**.

9.1.3 Relationships and Relationship Types

- A **relationship instance** relates specific entities (e.g., student 1001 enrolls in CSC211). A **relationship set** is the current set of relationship instances stored for a relationship type.
- A **relationship type** is the schema-level description: its name, participating entity types, and constraints.

Two major categories of constraints:

- **Cardinality ratio** (maximum participation).

  $$1{:}1, \quad 1{:}N \text{ (or } N{:}1), \quad M{:}N$$

  This answers what is the maximum number of relationship instances an entity can participate in?
- **Participation** (minimum participation).
  - Optional participation (minimum 0): entity can exist without participating.
  - Mandatory / total participation (minimum 1): existence-dependent (entity must participate at least once).

This answers, must every entity participate in this relationship at least once?

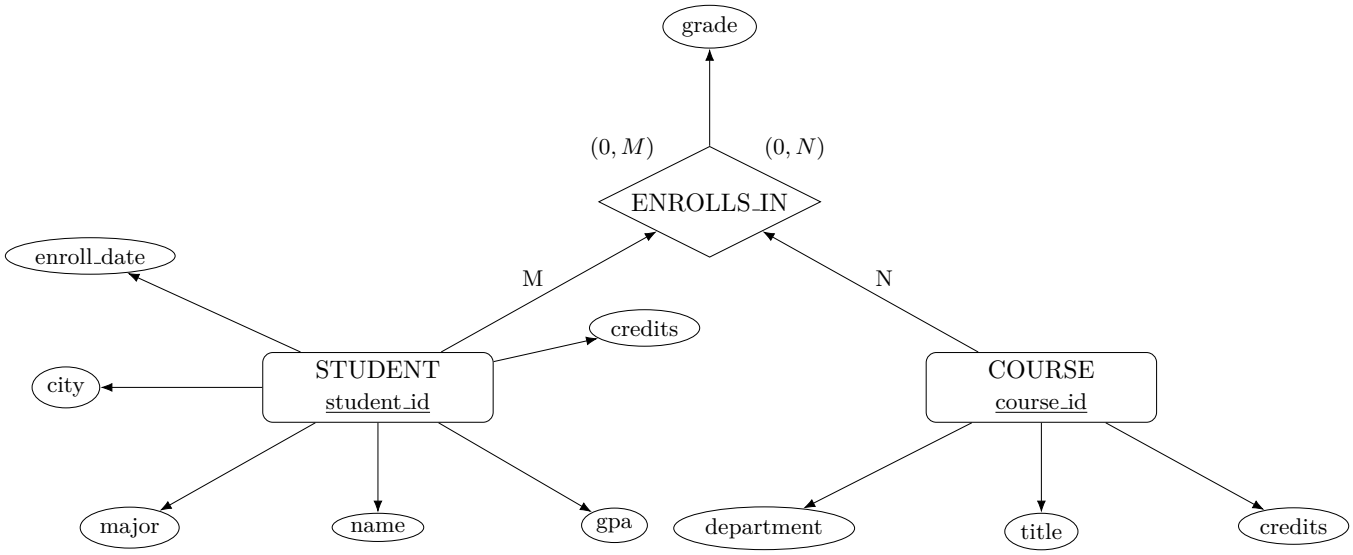
M:N Relationship Example: Student–Course Enrollment

Conceptually, we identify the entity types `STUDENT` and `COURSE`,

$$\texttt{STUDENT}(\underline{\texttt{student\_id}}, \texttt{name}, \texttt{major}, \texttt{gpa}, \texttt{enroll\_date}, \texttt{credits}, \texttt{city})$$

$$\texttt{COURSE}(\underline{\texttt{course\_id}}, \texttt{department}, \texttt{title}, \texttt{credits})$$

and the relationship type `ENROLLS_IN`, where `grade` is an attribute of the relationship.



Min–max constraints: STUDENT $(0, M) \longleftrightarrow$ ENROLLS_IN $\longleftrightarrow$ $(0, N)$ COURSE

Interpretation.

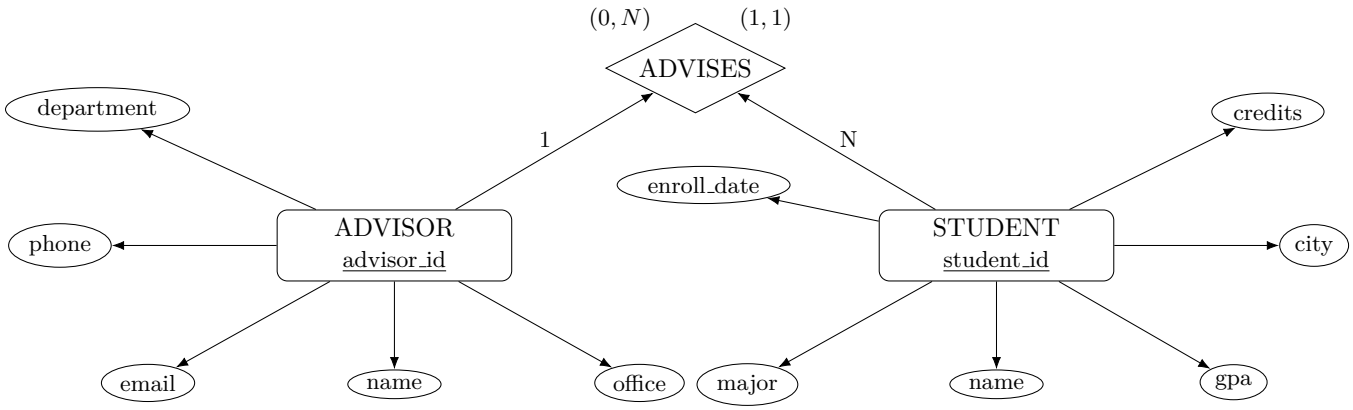
- A `STUDENT` may enroll in 0 to many courses.
- A `COURSE` may have 0 to many students.
- `grade` belongs to the relationship instance (student, course), not to Student alone and not to Course alone.
-

1:N Relationship Example: Student–Advisor

Conceptually, we identify a new entity type `ADVISOR`:

$$\texttt{ADVISOR}(\underline{\texttt{advisor\_id}}, \texttt{name}, \texttt{office}, \texttt{email}, \texttt{phone}, \texttt{department})$$

and the relationship type `ADVISES`,



Min–max constraints: `ADVISOR` $(0, N)$ — ADVISES — $(1, 1)$ `STUDENT`

Interpretation.

- An `ADVISOR` may advise 0 to many students.
- A `STUDENT` must be advised by exactly one advisor.
- The relationship `ADVISES` connects advisor instances and student instances.
- The existence of a `STUDENT` depends on participation in `ADVISES` if total participation is required. (A STUDENT must be related to an ADVISOR in order to exist in the database)

if you change the relationship type to:

Min–max constraints: `ADVISOR` $(0, N)$ — ADVISES — $(0, 1)$ `STUDENT`

A student can exist without an advisor. The existence of STUDENT does NOT depend on ADVISES. It is Partial participation.

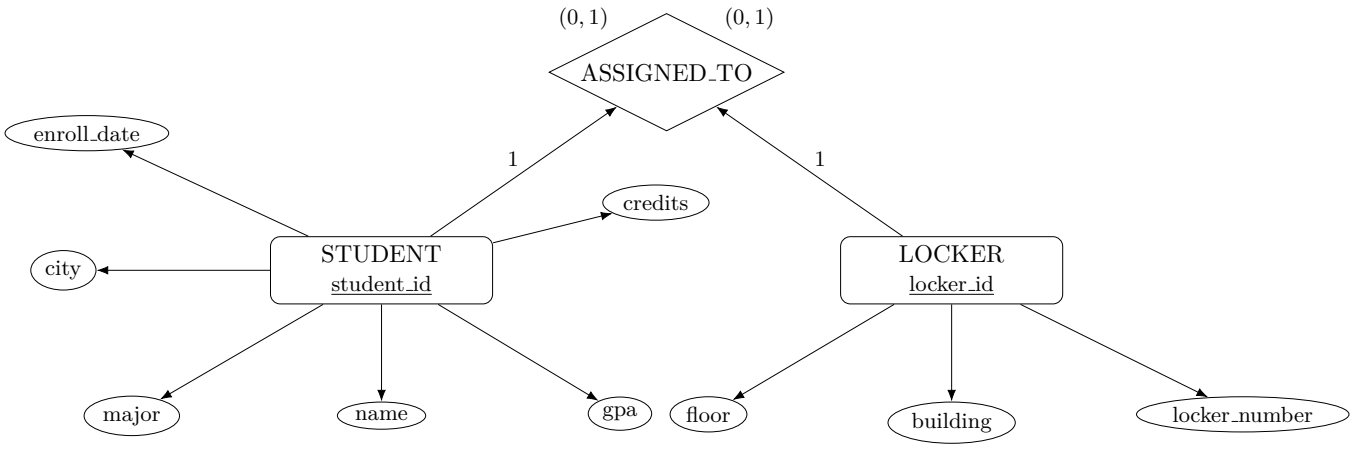
1:1 Relationship Example: Student–Locker

Each student is assigned at most one locker, and each locker is assigned to at most one student.

Add a new entity type:

`LOCKER`(`locker_id`, `building`, `floor`)

and the relationship type `ASSIGNED_TO` is:



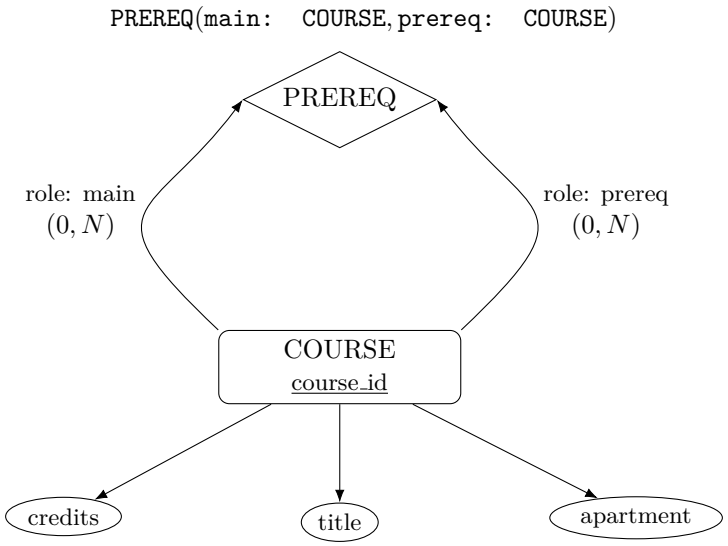
Min–max constraints: `STUDENT` $(0, 1)$ — — — `ASSIGNED_TO` — — — $(0, 1)$ `LOCKER`

Interpretation.

- A `STUDENT` may be assigned 0 or 1 locker.
- A `LOCKER` may be assigned 0 or 1 student.
- The relationship `ASSIGNED_TO` connects student instances and locker instances.

Recursive (Unary) Relationship Example:Course Prerequisite

A recursive relationship happens when the same entity type participates in distinct roles. Because `COURSE` participates twice in the same relationship type, we must use **role names** to distinguish the two participations:



Min–max constraints: `COURSE` $(0, N)$ — — — `PREREQ` — — — $(0, N)$ `COURSE`

Interpretation.

- A `COURSE` in the main role may require 0 to many prerequisite courses.
- A `COURSE` in the prereq role may be required by 0 to many other courses.
- The relationship type `PREREQ` connects instances of the same entity type (`COURSE`) in two distinct roles.
- Role names (`main` and `prereq`) are required to avoid ambiguity: they specify which course requires and which course is required.
- Since both sides have maximum participation N, `PREREQ` is a recursive M:N relationship type.

Weak Entity Type Example: Course Section

A weak entity type cannot be uniquely identified by its own attributes alone; it depends on an **owner** entity type via an **identifying relationship**.

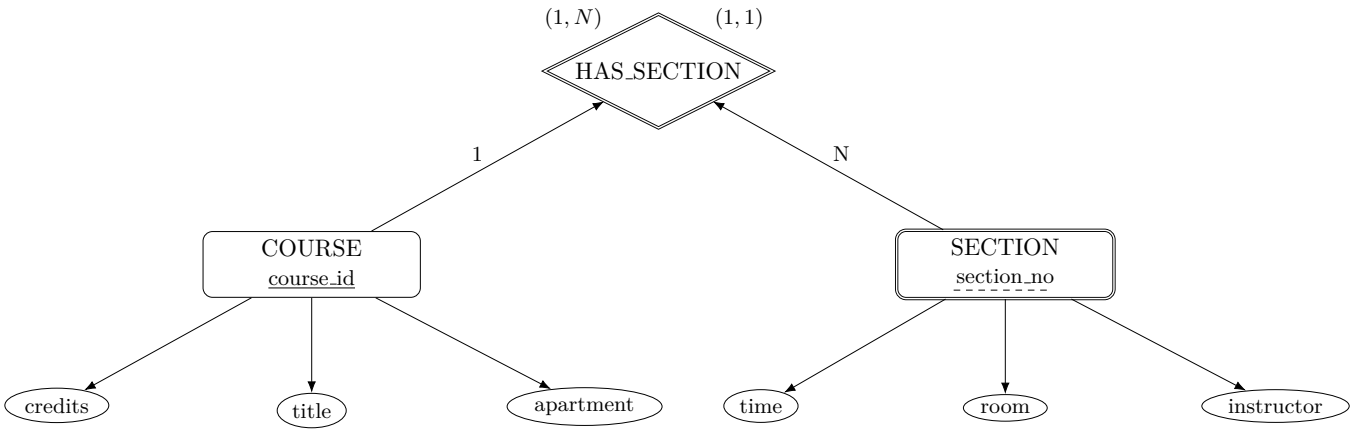
Owner entity:

`COURSE`(`course_id`, `title`, `credits`, `apartment`)

Weak entity:

`SECTION`(`section_no`, `room`, `time`, `instructor`)

Here `section_no` is only unique **within a course**. It is a **partial key**.



Min–max constraints: `COURSE` $(1, N)$ — — — `HAS_SECTION` — — — $(1, 1)$ `SECTION`

Interpretation.

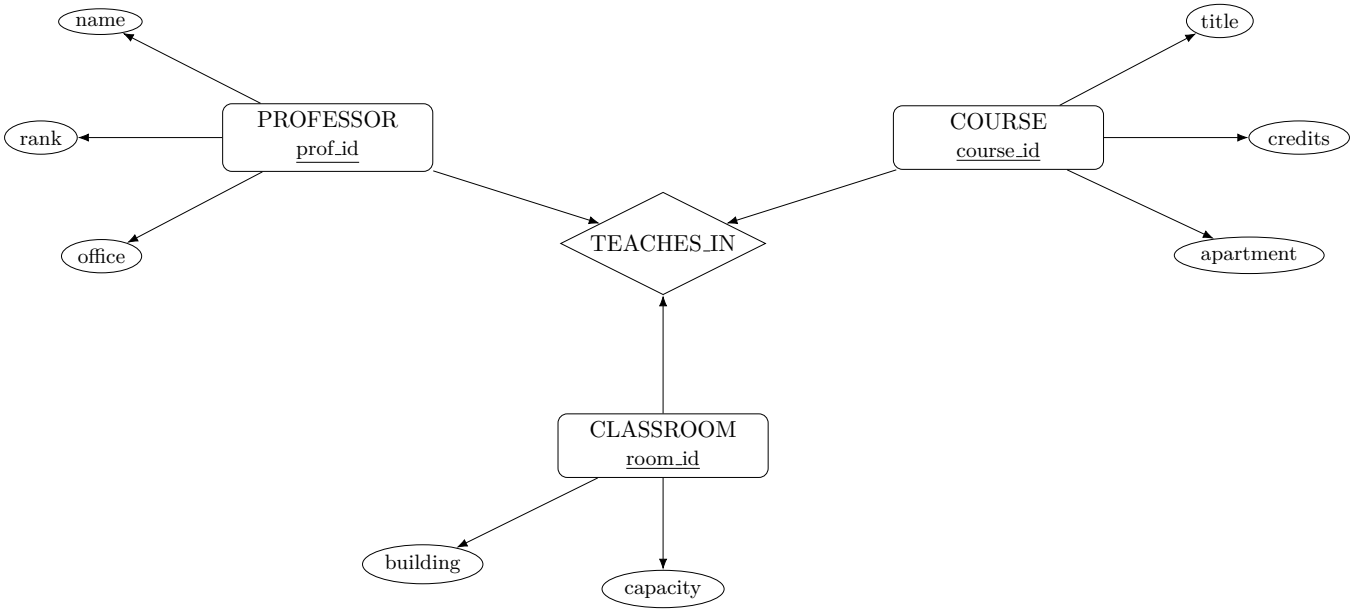
- A `COURSE` may have 1 to many sections.
- A `SECTION` must belong to exactly one course.
- The entity type `SECTION` is a weak entity because it does not have a full key of its own.
- The attribute `section_no` is a partial key and is only unique within a given course.
- The identifying relationship `HAS_SECTION` establishes the existence dependency of `SECTION` on `COURSE`.
- A `SECTION` cannot exist without a corresponding `COURSE`.

Ternary Relationship Example: PROFESSOR teaches COURSE in CLASSROOM

We introduce a new Entity:

`CLASSROOM`(`room_id`, `building`, `capacity`)

and the relationship type `TEACHES_IN`:



Min–max constraints: `PROFESSOR` $(0, N)$ — — — `COURSE` $(0, N)$ — — — `CLASSROOM` $(0, N)$

Interpretation.

- A `PROFESSOR` teaches a particular `COURSE` in a specific `CLASSROOM`.
- The meaning of the relationship depends on all three participating entity types.
- This is a ternary relationship because no single pair of entities fully determines the teaching assignment.

9.2 Mapping to Schema

Mapping Rules Used (Ordered):

- Step 1 (Strong entity): Create a relation with all simple attributes; choose a key as the primary key.
- Step 2 (Weak entity): Create a relation containing weak-entity attributes + owner key as FK; PK = (owner PK + partial key).
- Step 3 (Binary 1:1): Use FK approach (preferred), or merge, or create a separate relation.
- Step 4 (Binary 1:N): Add FK of the 1-side into the N-side relation; move any relationship attributes to the N-side.
- Step 5 (Binary M:N): Create a new relation; include both participating PKs as FKs; PK is their combination; include relationship attributes.

9.2.1 College Database Overview

Entities and relationships:

Strong Entities:

- STUDENT(student_id, name, major, gpa, enroll_date, credits, city)
- ADVISOR(advisor_id, name, office, email, phone, department)
- COURSE(course_id, title, credits, department)
- PROFESSOR(prof_id, name, rank, office)
- CLASSROOM(room_id, building, capacity)
- LOCKER(locker_id, building, floor)

Weak Entity:

- SECTION(section_no, time, room, instructor)

Relationships:

- Advisor advises Student (1:N)
- Student assigned Locker (1:1)
- Student enrolls in Course (M:N, attribute: grade)
- Course has Section (1:N identifying)
- Course prerequisite Course (recursive M:N)
- Professor teaches Course in Classroom (ternary)

ER min–max constraints are enforced in SQL using:

- **Minimum = 1** → use `NOT NULL`
- **Maximum = 1** → use `UNIQUE`
- **Existence dependency** → use `ON DELETE CASCADE`

9.2.2 Step 1: Strong Entities

For every regular (strong) entity in our ER diagram — such as Student, Advisor, or Course — we create one corresponding table in the relational schema.

```
CREATE TABLE Student (
  student_id   INT PRIMARY KEY,
  name         VARCHAR(100) NOT NULL,
  major        VARCHAR(100),
  gpa          DECIMAL(3,2),
  enroll_date  DATE,
```

```
  credits      INT DEFAULT 0,
  city         VARCHAR(100)
);

CREATE TABLE Advisor (
  advisor_id   INT PRIMARY KEY,
  name         VARCHAR(100),
  office       VARCHAR(50),
  email        VARCHAR(120),
  phone        VARCHAR(30),
  department   VARCHAR(100)
);

CREATE TABLE Course (
  course_id   VARCHAR(20) PRIMARY KEY,
  title       VARCHAR(200),
  credits     INT,
  department  VARCHAR(100)
);
```

9.2.3 Step 2: Weak Entity (Section)

A weak entity becomes its own table. It includes all its attributes. It also includes the owner's primary key as a foreign key. Its primary key is a combination of: the owner's primary key plus its own partial key.

```
CREATE TABLE Section (
  course_id   VARCHAR(20) NOT NULL,
  section_no  INT NOT NULL,
  time        VARCHAR(50),
  room        VARCHAR(50),
  instructor  VARCHAR(100),
  PRIMARY KEY (course_id, section_no),
  FOREIGN KEY (course_id)
    REFERENCES Course(course_id)
    ON DELETE CASCADE
);
```

9.2.4 Step 3: Binary 1:1 (Student–Locker)

Many ways to achieve:

- **Foreign Key approach:**Choose one of the two tables (either Student or Locker) and add a foreign key that references the primary key of the other table. It is usually best to place the foreign key in the table whose participation in the relationship is total (minimum = 1). For example: If every student must have a locker, add locker_id as a foreign key in Student. If every locker must be assigned to a student, add student_id as a foreign key in Locker.

```
-- Entity table for lockers
CREATE TABLE Locker (
  locker_id INT PRIMARY KEY,
  building  VARCHAR(100),
  floor     INT
);

-- Add locker_id with full 1:1 enforcement
ALTER TABLE Student
  ADD locker_id INT UNIQUE NOT NULL,
  ADD CONSTRAINT fk_student_locker
    FOREIGN KEY (locker_id)
    REFERENCES Locker(locker_id);

```

- **Merged relation option**: If both Student and Locker have total participation (every student has a locker and every locker belongs to exactly one student), the two entity types and their relationship can be merged into a single table. In this design, locker attributes become part of the Student table (or vice versa), and no separate relationship table is needed.

```
-- Add locker identity + locker attributes directly to Student
ALTER TABLE Student
  ADD locker_id INT NOT NULL UNIQUE,
  ADD locker_building VARCHAR(100),
```

```
  ADD locker_floor INT;
```

- **Cross-reference or relationship relation option**: Create a third table (for example, StudentLocker) that stores the primary keys of both Student and Locker. This table cross-references the two entities and can also include any attributes of the relationship (such as assignment date). This approach keeps the two entity tables separate and is especially useful if the relationship has its own attributes or may evolve later.

```
-- Locker entity table remains separate
CREATE TABLE Locker (
  locker_id INT PRIMARY KEY,
  building  VARCHAR(100),
  floor     INT
);

-- Relationship table enforces 1:1 using PK + UNIQUE
CREATE TABLE StudentLocker (
  student_id INT NOT NULL,
  locker_id  INT NOT NULL,

  PRIMARY KEY (student_id),   -- each student appears at most once
  UNIQUE (locker_id),         -- each locker appears at most once

  CONSTRAINT fk_sl_student
    FOREIGN KEY (student_id) REFERENCES Student(student_id)
    ON DELETE CASCADE,

  CONSTRAINT fk_sl_locker
    FOREIGN KEY (locker_id) REFERENCES Locker(locker_id)
    ON DELETE CASCADE
);
```

9.2.5 Step 4: Binary 1:N (Advisor–Student)

For a 1:N relationship, place the foreign key of the 1-side into the table representing the N-side, along with any relationship attributes.

```
ALTER TABLE Student
ADD advisor_id INT NOT NULL,
ADD FOREIGN KEY (advisor_id)
  REFERENCES Advisor(advisor_id);
```

9.2.6 Step 5: Binary M:N (Student–Enrollment)

For a many-to-many relationship, create a new table. Put the primary keys of both participating entities into that table as foreign keys. Their combination becomes the primary key. Add any attributes that belong to the relationship itself.

```
CREATE TABLE Enrollment (
  student_id INT NOT NULL,
  course_id  VARCHAR(20) NOT NULL,
  grade      VARCHAR(5),
  PRIMARY KEY (student_id, course_id),
  FOREIGN KEY (student_id)
    REFERENCES Student(student_id),
  FOREIGN KEY (course_id)
    REFERENCES Course(course_id)
);
```

9.2.7 Step 6: Ternary Relationship (Professor–Course–Classroom)

For a ternary relationship:Create a new table. Put the primary keys of all three participating entities into that table as foreign keys. Their combination usually becomes the primary key. Add any attributes that belong to the relationship itself.

```
CREATE TABLE TeachesIn (
  prof_id   INT NOT NULL,
  course_id VARCHAR(20) NOT NULL,
  room_id   INT NOT NULL,
  PRIMARY KEY (prof_id, course_id, room_id),
```

```
  FOREIGN KEY (prof_id) REFERENCES Professor(prof_id),
  FOREIGN KEY (course_id) REFERENCES Course(course_id),
  FOREIGN KEY (room_id) REFERENCES Classroom(room_id)
);
```

Composite key captures full relationship dependency.

In summary, practical Design Rules:

- Strong entity: own table
- Weak entity: composite key table
- 1:N: FK on N-side
- M:N: intersection table

Here we do not discuss theoretical details about function dependency analysis (FDs) and Boyce–Codd Normal Form (BCNF).

9.3 Schema Class

The file `schema.py` defines the schema layer of EduDB. Its job is to describe the structure of a row and perform all validation that can be done using only:

- the schema metadata, and
- the row currently being inserted, updated, encoded, or decoded.

It supports:

- `NOT NULL` checking,
- primary-key non-null checking,
- type checking,
- default values,
- allowed-value constraints,
- minimum and maximum value constraints,
- minimum and maximum length constraints,
- regular-expression pattern constraints,
- normalization of `date` and `datetime` values,
- foreign-key metadata,
- foreign-key referential action metadata such as `CASCADE` and `RESTRICT`.

However, it still intentionally does not perform database-state checks. In particular, this file does not check:

- primary-key uniqueness,
- whether a foreign-key value exists in the referenced table,
- whether cascading actions should actually be executed,
- any cross-row or cross-table constraint.

Those responsibilities belong to the database layer, not the schema layer.

```
_MISSING = object()
```

This object is used as a sentinel value to distinguish:

- a column that is entirely missing from the input row, and
- a column that is explicitly present with value `None`.

This distinction matters because those two cases should not always be treated the same way. Suppose a column has a default value. If the field is missing, the default should be applied. But if the user explicitly provides `None`, then the system should treat that as an explicit null attempt and enforce `nullable` rules. So the sentinel is necessary to represent:

"No value was supplied at all."

Foreign-Key Action Set:

```
_FK_ACTIONS = {"RESTRICT", "CASCADE", "SET_NULL", "NO_ACTION", "SETNULL"}
```

This constant defines the supported referential actions for foreign keys. These actions describe what the database layer should do if a referenced parent row is deleted or updated.

Helper Function `_normalize_fk_action()`:

```
def _normalize_fk_action(action: str) -> str:
    s = str(action).strip().upper().replace(" ", "_")
    if s == "SETNULL":
        s = "SET_NULL"
    if s not in _FK_ACTIONS:
        raise ValueError(...)
    return s
```

This helper converts a user-provided foreign-key action into a canonical normalized form.

1. Convert the action to a string.
2. Remove leading and trailing spaces.
3. Convert to uppercase.
4. Replace spaces with underscores.
5. Convert `SETNULL` into `SET_NULL`.
6. Reject unsupported actions.

It allows the schema to accept slightly different user spellings while still storing one consistent representation. For example:

- `"cascade"` becomes `"CASCADE"`
- `" set null "` becomes `"SET_NULL"`
- `"SETNULL"` becomes `"SET_NULL"`

This makes the schema metadata more robust and predictable.

9.3.1 `Column` Class

```
@dataclass(frozen=True)
class Column:
    name: str
    col_type: str
    nullable: bool = True
    default: Any = _MISSING
    allowed_values: Optional[List[Any]] = None
    min_value: Optional[Any] = None
    max_value: Optional[Any] = None
    min_length: Optional[int] = None
    max_length: Optional[int] = None
    pattern: Optional[str] = None
```

A `Column` object stores the metadata and row-local constraints for one column in a table.

- `name`

  The column name.

- `col_type`

  The logical type name. Supported built-in types are:

  - `int`
  - `float`
  - `str`
  - `bool`

  - `date`
  - `datetime`
  - `json`
  - `any`
- `nullable`

  Whether `None` is allowed.
- `default`

  The default value to use when the field is missing entirely.
- `allowed_values`

  An optional finite domain restriction.
- `min_value` and `max_value`

  Optional lower and upper bounds.
- `min_length` and `max_length`

  Optional length restrictions.
- `pattern`

  An optional regular expression that string values must satisfy.

Why `frozen=True`? The class is immutable after creation. This is a good design decision because column metadata should generally not change after the schema has been defined.

```
def __post_init__(self) -> None:
    if not self.name:
        raise ValueError(...)
    if not self.col_type:
        raise ValueError(...)
    if self.min_length is not None and self.min_length < 0:
        raise ValueError(...)
    if self.max_length is not None and self.max_length < 0:
        raise ValueError(...)
    if self.min_length is not None and self.max_length is not None
       and self.min_length > self.max_length:
        raise ValueError(...)
    if self.pattern is not None:
        re.compile(self.pattern)
```

This checks whether the `Column` object itself is valid at schema-construction time.

What it enforces.

- column name cannot be empty,
- type name cannot be empty,
- length bounds cannot be negative,
- minimum length cannot exceed maximum length,
- regex pattern must be syntactically valid.

```
@property
def has_default(self) -> bool:
    return self.default is not _MISSING
```

This checks whether the column truly has a default value.

Why not check `default is not None`? Because `None` itself may be a legitimate default. So the sentinel `_MISSING` is necessary to distinguish:

- no default provided,
- default provided and its value happens to be `None`.

```
@property
def compiled_pattern(self) -> Optional[Pattern[str]]:
    if self.pattern is None:
        return None
    return re.compile(self.pattern)
```

This compiles the regex string into a pattern object when needed. This recompiles the regex on each access. That is acceptable in a teaching implementation, though a production implementation might cache it.

9.3.2 `ForeignKey` Class

```
@dataclass(frozen=True)
class ForeignKey:
    column: str
    ref_table: str
    ref_column: str
    on_delete: str = "RESTRICT"
    on_update: str = "RESTRICT"
```

This class stores foreign-key metadata plus referential-action metadata.

It describes:

- which child-table column is the foreign key,
- which parent table is referenced,
- which parent-table column is referenced,
- what action should happen if the parent row is deleted,
- what action should happen if the parent key is updated.

The supported action values are:

- `RESTRICT`
- `CASCADE`
- `SET_NULL`
- `NO_ACTION`

These actions are metadata only at the schema layer.

For example, this object:

```
ForeignKey(
    column="student_id",
    ref_table="Student",
    ref_column="student_id",
    on_delete="CASCADE",
    on_update="RESTRICT"
)
```

means:

> If the referenced `Student` row is deleted, the database layer should perform a cascade. If the referenced key is updated, the database layer should restrict that change.

```
def __post_init__(self) -> None:
    if not self.column:
        raise ValueError(...)
    if not self.ref_table:
        raise ValueError(...)
    if not self.ref_column:
        raise ValueError(...)
    object.__setattr__(self, "on_delete",
                       _normalize_fk_action(self.on_delete))
    object.__setattr__(self, "on_update",
                       _normalize_fk_action(self.on_update))
```

This method validates the foreign-key metadata and normalizes the action names.

- local foreign-key column cannot be empty,
- referenced table name cannot be empty,
- referenced column name cannot be empty,
- `on_delete` must be a supported action,
- `on_update` must be a supported action.

Why use `object.__setattr__`? Because the dataclass is frozen. Normally its fields cannot be changed. However, during `__post_init__`, we still want to normalize the action names into canonical form. So `object.__setattr__()` is used to bypass immutability just for initialization.

9.3.3 `Schema` Class

```
@dataclass
class Schema:
    columns: List[Column]
    primary_key: Optional[str] = None
    foreign_keys: List[ForeignKey] = field(default_factory=list)
    allow_extra_keys: bool = True
```

This is the main schema object. It groups columns, primary-key metadata, foreign-key metadata, and row-validation logic into one class.

Why `field(default_factory=list)` is correct

```
foreign_keys: List[ForeignKey] = field(default_factory=list)
```

creates a fresh list for each `Schema` object. This avoids the classic mutable-default pitfall. If the code had used:

```
foreign_keys = []
```

then all schema objects would share the same list, which would be incorrect.

```
def __post_init__(self) -> None:
    if not self.columns:
        raise ValueError("Schema must contain at least one column")

    self._columns_by_name: Dict[str, Column] = {}
    for c in self.columns:
        if c.name in self._columns_by_name:
            raise ValueError(...)
        self._columns_by_name[c.name] = c

    if self.primary_key is not None:
        if self.primary_key not in self._columns_by_name:
            raise ValueError(...)

    for fk in self.foreign_keys:
        if fk.column not in self._columns_by_name:
            raise ValueError(...)
```

This validates the schema definition itself and prepares a fast lookup dictionary for columns.

What it enforces

- a schema must have at least one column,
- column names must be unique,
- the primary key must refer to a defined column,
- each foreign-key child column must refer to a defined local column.

Why build `_columns_by_name`? The schema keeps:

- `self.columns` for ordered iteration,
- `self._columns_by_name` for fast lookup by name.

This is a common and useful design pattern.

```
@property
def column_names(self) -> List[str]:
    return [c.name for c in self.columns]
```

This returns the column names in schema order. Schema order matters for:

- consistent presentation,
- predictable iteration,
- future physical layout decisions.

```
def get_column(self, name: str) -> Column:
    try:
        return self._columns_by_name[name]
    except KeyError as e:
        raise KeyError(f"Unknown column: {name}") from e
```

This returns a column definition by name.

```
def validate_row(self, row: Dict[str, Any]) -> Dict[str, Any]:
    if not isinstance(row, dict):
        raise TypeError(...)

    if not self.allow_extra_keys:
        extras = set(row.keys()) - set(self._columns_by_name.keys())
        if extras:
            raise ValueError(...)

    out: Dict[str, Any] = {}

    for c in self.columns:
        raw_val = row[c.name] if c.name in row else _MISSING
        out[c.name] = self._validate_and_normalize_value(c, raw_val)

    if self.primary_key is not None and out[self.primary_key] is None:
        raise ValueError(...)

    return out
```

This is the main row-validation method. It validates one row using only schema-local information.

What it enforces

- row must be a dictionary,
- extra keys may be rejected in strict mode,
- missing columns are handled with defaults,
- `NOT NULL` is enforced,
- type checks are enforced,
- allowed-value, length, pattern, and range checks are enforced,
- the primary key cannot be `NULL`.

```
def _validate_and_normalize_value(self, c: Column, raw_val: Any) -> Any:
    if raw_val is _MISSING:
        if c.has_default:
            raw_val = c.default
        else:
            raw_val = None

    if raw_val is None:
        if not c.nullable:
```

```
            raise ValueError(...)
        return None

    val = self._coerce_type(c, raw_val)

    if c.allowed_values is not None and val not in c.allowed_values:
        raise ValueError(...)

    if c.min_length is not None or c.max_length is not None:
        ...

    if c.pattern is not None:
        ...

    if c.min_value is not None:
        ...

    if c.max_value is not None:
        ...

    return val
```

This method validates and normalizes one value for one column.

1. Phase 1: missing-field handling. If the field is missing entirely, use the default if one exists; otherwise treat it as `None`.
2. Phase 2: null handling. If the result is `None`, reject it if the column is not nullable.
3. Phase 3: type normalization. Convert or validate the value according to the declared logical type.
4. Phase 4: additional constraints. Apply finite-domain checks, length checks, regex checks, and range checks.

Allowed-value checking

```
if c.allowed_values is not None and val not in c.allowed_values:
    raise ValueError(...)
```

This implements a finite-domain constraint. A column declared as:

```
Column("grade", "str", allowed_values=["A", "B", "C", "D", "F"])
```

rejects any value not in that list.

```
def _coerce_type(self, c: Column, val: Any) -> Any:
    t = c.col_type.lower()

    if t in ("any", "object"):
        return val
    if t == "int":
        ...
    if t == "float":
        ...
    if t == "str":
        ...
    if t == "bool":
        ...
    if t == "date":
        ...
    if t == "datetime":
        ...
    if t == "json":
        ...
    raise ValueError(...)
```

This interprets the logical type name and validates the value accordingly.

```
@staticmethod
def _normalize_date(col_name: str, val: Any) -> str:
    if isinstance(val, datetime):
        return val.date().isoformat()
    if isinstance(val, date):
        return val.isoformat()
```

```
    if isinstance(val, str):
        return date.fromisoformat(val).isoformat()
    raise TypeError(...)
```

This normalizes date input into canonical ISO string form.

Accepted inputs

- `datetime` object,
- `date` object,
- ISO date string.

Why normalize to string? Because rows are stored as JSON bytes, and strings are JSON-friendly. Native Python date objects are not directly JSON-serializable.

```
@staticmethod
def _normalize_datetime(col_name: str, val: Any) -> str:
    if isinstance(val, datetime):
        return val.isoformat(timespec="seconds")
    if isinstance(val, str):
        return datetime.fromisoformat(val).isoformat(timespec="seconds")
    raise TypeError(...)
```

This normalizes datetime values into ISO string form with second precision.

Why use second precision? It removes unnecessary microsecond detail and makes stored representations more consistent and readable.

```
def _normalize_bound_for_compare(self, c: Column, bound: Any) -> Any:
    return self._coerce_type(c, bound)
```

This ensures that range bounds are normalized in the same way as actual values before comparison.

```
def encode(self, row: Dict[str, Any]) -> bytes:
    row2 = self.validate_row(row)
    return json.dumps(
        row2,
        separators=(",", ":"),
        ensure_ascii=False,
    ).encode("utf-8")
```

This converts a validated row into UTF-8 encoded JSON bytes suitable for storage.

Why JSON is used JSON is used here because it is:

- easy to read,
- easy to debug,
- appropriate for a teaching DBMS.

It is not the most space-efficient format, but it is very clear for instructional purposes.

```
def decode(self, payload: bytes) -> Dict[str, Any]:
    obj = json.loads(payload.decode("utf-8"))
    if not isinstance(obj, dict):
        raise ValueError("Decoded payload must be a JSON object")
    return self.validate_row(obj)
```

This converts stored bytes back into a row and revalidates it.

Why revalidate on decode? Because the stored payload may be malformed or inconsistent. Revalidating ensures the reconstructed row still satisfies schema-local rules.

What the updated schema layer now enforces

- valid column metadata,
- valid foreign-key metadata,
- valid foreign-key action metadata,
- duplicate-column rejection,
- primary-key column existence,
- local foreign-key column existence,
- default handling,
- `NOT NULL`,
- primary-key non-null,
- type checking,
- finite-domain checking,
- length checking,
- regex checking,
- lower and upper bound checking,
- normalization of date and datetime values,
- JSON-serializable row storage.

Now, we can clearly know that:

- **Schema** knows what one valid row looks like
- **HeapTable** knows how one row is physically stored
- **Database** knows whether the operation is valid given the whole database state

9.4 Database Class

The file `db.py` defines the core engine layer of EduDB. This file is the main coordinator of the system. It connects together:

- the catalog,
- the heap-table layer,
- the schema layer,
- the index layer,
- the transaction and lock manager,
- the SQL parser and optimizer.

This revised version makes the boundary between layers very clear:

- `schema.py` performs only schema-local validation,
- `table.py` performs only physical storage operations,
- `db.py` performs all state-dependent relational checks.

So this file is responsible for:

- primary-key uniqueness,
- foreign-key existence,
- referential actions on delete and update,
- index-backed probes for constraint checking,
- transaction-aware locking,
- table creation and catalog loading,
- SQL execution.

```
"""
edudb.db
Core engine layer of EduDB.

This revision keeps schema-local validation in schema.py and moves all
state-dependent relational checks into the database layer:

- primary-key uniqueness
- foreign-key existence
- referential actions on delete/update (RESTRICT by default, CASCADE supported)
- index-backed probes for PK/FK checks
- table-level transaction-aware lock acquisition for related tables
"""
```

This docstring already gives the core design idea of the file.

Very important meaning. This file is where EduDB decides whether an operation is legal with respect to the whole database state.

For example:

- `schema.py` can check whether a primary-key value is `NULL`,
- but only `db.py` can check whether that primary-key value already exists in the table.

That is why this file is the correct place for relational integrity enforcement.

9.4.1 Imports

```
from __future__ import annotations
import os
import json
import threading
from dataclasses import dataclass
from typing import Any, Dict, List, Optional, Tuple, Iterable, Set

from .disk_file import PageFile
from .buffer_manager import BufferPool
from .schema import Schema, Column, ForeignKey
from .table import HeapTable, GlobalRID
from .index_hash import HashIndex
from .index_bplustree import BPlusTreeIndex
from .transaction import LockManager, Transaction
from edudb import parser
```

These imports show the central role of this file.

- `PageFile` and `BufferPool` are needed for physical table access.
- `Schema`, `Column`, and `ForeignKey` are needed for catalog reconstruction and metadata handling.
- `HeapTable` and `GlobalRID` are needed for physical row operations.
- `HashIndex` and `BPlusTreeIndex` are needed for fast probes and query processing.
- `LockManager` and `Transaction` are needed for transaction-aware checks.
- `parser` is needed for SQL execution.

So `db.py` is the layer where all major engine components come together.

9.4.2 `TableInfo` Dataclass

```
@dataclass
class TableInfo:
    table: HeapTable
    hash_indexes: Dict[str, HashIndex]
    btree_indexes: Dict[str, BPlusTreeIndex]
```

This small dataclass groups together all runtime information for one loaded table.

For each table, EduDB keeps:

- the `HeapTable` object,
- all hash indexes on that table,
- all B+ tree indexes on that table.

This is convenient because the system often needs to access the heap table and its indexes together.

9.4.3 `Database.__init__`

```
class Database:
    def __init__(
        self,
        dir_path: str,
        page_size: int = 4096,
        buffer_pages: int = 256,
        default_index: str = "btree",
    ):
        self.dir_path = dir_path
        os.makedirs(self.dir_path, exist_ok=True)

        self.page_size = page_size
        self.bp = BufferPool(capacity_pages=buffer_pages)

        idx = default_index.strip().lower()
        if idx not in {"btree", "hash"}:
            raise ValueError('default_index must be "btree" or "hash"')
        self.default_index = idx

        self._catalog_path = os.path.join(self.dir_path, "catalog.json")
        self._catalog: Dict[str, Tuple[Schema, str]] = {}
        self._tables: Dict[str, TableInfo] = {}
        self._index_meta: Dict[str, List[Tuple[str, str]]] = {}

        self._load_catalog()

        self._optimizer = None
        self.lockmgr = LockManager()
        self._lock_mu = threading.RLock()
```

This constructor initializes the entire database engine.

Important fields.

- `dir_path`: the database directory on disk
- `page_size`: page size used by page files
- `bp`: the global buffer pool
- `default_index`: default index type for automatic index creation
- `_catalog_path`: path to `catalog.json`
- `_catalog`: metadata loaded from catalog
- `_tables`: loaded table objects
- `_index_meta`: stored index metadata
- `lockmgr`: global lock manager
- `_lock_mu`: mutex protecting lock acquisition

Why call `_load_catalog()` here? Because when the database starts, it must reconstruct table schemas and index metadata from disk before normal operations can continue.

9.4.4 Catalog Section

The catalog is the metadata layer of the database. It records:

- which tables exist,
- each table's schema,
- each table's data file name,

- which indexes exist.

Column serialization helper

```
@staticmethod
def _column_to_catalog_dict(c: Column) -> Dict[str, Any]:
    return {
        "name": c.name,
        "col_type": c.col_type,
        "nullable": c.nullable,
        "default": None if not c.has_default else c.default,
        "has_default": c.has_default,
        "allowed_values": c.allowed_values,
        "min_value": c.min_value,
        "max_value": c.max_value,
        "min_length": c.min_length,
        "max_length": c.max_length,
        "pattern": c.pattern,
    }
```

This helper converts one `Column` object into a JSON-friendly dictionary for catalog storage.

This is necessary because the schema is a Python object in memory, but the catalog on disk must be plain JSON data.

Catalog loading

```
def _load_catalog(self):
    if not os.path.exists(self._catalog_path):
        return

    with open(self._catalog_path, "r") as f:
        obj = json.load(f)
```

This begins loading the catalog from `catalog.json`. If the file does not exist yet, the method simply returns, meaning the database starts with no stored tables.

Schema reconstruction

```
for name, info in obj.get("tables", {}).items():
    sch_obj = info.get("schema", {})
    columns: List[Column] = []
    for c in sch_obj.get("columns", []):
        kwargs = {
            "name": c["name"],
            "col_type": c["col_type"],
            "nullable": c.get("nullable", True),
            "allowed_values": c.get("allowed_values"),
            "min_value": c.get("min_value"),
            "max_value": c.get("max_value"),
            "min_length": c.get("min_length"),
            "max_length": c.get("max_length"),
            "pattern": c.get("pattern"),
        }
        if c.get("has_default", False):
            kwargs["default"] = c.get("default")
        columns.append(Column(**kwargs))
```

This part reconstructs each `Column` object from JSON metadata. The database is effectively rebuilding the schema objects that originally existed in memory.

Foreign-key reconstruction

```
fks: List[ForeignKey] = []
for fk in sch_obj.get("foreign_keys", []) or []:
    try:
        fks.append(ForeignKey(
            column=fk["column"],
            ref_table=fk["ref_table"],
            ref_column=fk["ref_column"],
```

```
            on_delete=fk.get("on_delete", "RESTRICT"),
            on_update=fk.get("on_update", "RESTRICT"),
        ))
    except Exception:
        continue
```

This part reconstructs foreign-key metadata. Notice that it preserves:

- referenced table,
- referenced column,
- `on_delete`,
- `on_update`.

So foreign-key actions survive database restart.

Schema object creation

```
schema = Schema(
    columns=columns,
    primary_key=sch_obj.get("primary_key"),
    foreign_keys=fks,
    allow_extra_keys=sch_obj.get("allow_extra_keys", True),
)
self._catalog[name] = (schema, info["file_name"])
```

This finally reconstructs the `Schema` object and stores it in the in-memory catalog.

Index metadata loading

```
for table_name, metas in obj.get("indexes", {}).items():
    clean: List[Tuple[str, str]] = []
    for meta in metas or []:
        col = meta.get("column")
        method = (meta.get("method") or self.default_index).lower()
        if col and method in {"btree", "hash"}:
            clean.append((col, method))
    if clean:
        self._index_meta[table_name] = clean
```

This loads index metadata so that indexes can later be recreated when the table is opened.

Catalog saving

```
def _save_catalog(self):
    obj = {"tables": {}, "indexes": {}}
    for name, (schema, file_name) in self._catalog.items():
        obj["tables"][name] = {
            "schema": {
                "primary_key": schema.primary_key,
                "allow_extra_keys": schema.allow_extra_keys,
                "columns": [self._column_to_catalog_dict(c) for c in schema.columns],
                "foreign_keys": [
                    {
                        "column": fk.column,
                        "ref_table": fk.ref_table,
                        "ref_column": fk.ref_column,
                        "on_delete": getattr(fk, "on_delete", "RESTRICT"),
                        "on_update": getattr(fk, "on_update", "RESTRICT"),
                    }
                    for fk in schema.foreign_keys
                ],
            },
            "file_name": file_name,
        }
```

This writes back all table metadata, including foreign-key action metadata.

Very important. Because `on_delete` and `on_update` are saved here, cascade or restrict behavior is not lost after restart.

9.4.5 Transactions and Locking

This section handles transaction-aware lock acquisition.

```
def begin(self) -> Transaction:
    return Transaction(self.lockmgr)
```

This starts a new transaction object.

```
def _acquire_lock(self, tx: Transaction, resource: str, mode: str) -> None:
    with self._lock_mu:
        self.lockmgr.acquire(tx.txid, resource, mode)
```

This acquires one lock safely under a mutex. The mutex ensures lock-manager access is synchronized.

```
def _acquire_table_locks(self, tx: Transaction, lock_map: Dict[str, str]) -> None:
    rank = {"S": 0, "X": 1}
    merged: Dict[str, str] = {}
    for table_name, mode in lock_map.items():
        mode = mode.upper()
        prev = merged.get(table_name)
        if prev is None or rank[mode] > rank[prev]:
            merged[table_name] = mode
    for table_name in sorted(merged):
        self._acquire_lock(tx, f"table:{table_name}", merged[table_name])
```

This method takes a set of requested table locks, merges them, and acquires them in sorted order.

Why is this important?

- If a table needs both `S` and `X`, the method keeps the stronger one.
- Sorted acquisition order helps reduce deadlock risk.

Meaning of lock modes.

- `S`: shared lock
- `X`: exclusive lock

9.4.6 Table Handling

This section manages loading and creation of tables.

```
def _ensure_table_info(self, name: str) -> TableInfo:
    ti = self._tables.get(name)
    if ti is not None:
        return ti

    if name not in self._catalog:
        raise KeyError(f"Unknown table {name}")

    schema, file_name = self._catalog[name]
    pf = PageFile(os.path.join(self.dir_path, file_name), page_size=self.page_size)
    ht = HeapTable(name, schema, [pf], self.bp)
    ti = TableInfo(ht, {}, {})
    self._tables[name] = ti

    for col, method in self._index_meta.get(name, []):
        self.create_index(name, col, method)
    return ti
```

This method ensures that a table is loaded into memory.

If the table is already loaded, it returns the existing `TableInfo`. Otherwise it:

1. looks up the schema and file name in the catalog,
2. opens the table's `PageFile`,
3. builds a `HeapTable`,

4. creates a fresh `TableInfo`,
5. recreates the table's indexes.

So this method lazily loads tables only when needed.

```
def create_table(self, name_or_def, schema: Optional[Schema] = None):
    if schema is None and hasattr(name_or_def, "name") and hasattr(name_or_def, "schema"):
        name = getattr(name_or_def, "name")
        schema = getattr(name_or_def, "schema")
    else:
        name = name_or_def
```

This method supports two styles of table creation:

- passing the table name and schema directly,
- passing an object that already has `name` and `schema`.

```
if name in self._catalog:
    return

file_name = f"{name}.pages"
self._catalog[name] = (schema, file_name)
self._save_catalog()
self._ensure_table_info(name)
```

This stores the table in the catalog, saves the catalog, and loads the table immediately.

Automatic PK/FK index creation

```
if schema.primary_key:
    self.create_index(name, schema.primary_key, self.default_index)
for fk in schema.foreign_keys:
    self.create_index(name, fk.column, self.default_index)
```

This is a very important design choice.

If a table has:

- a primary key,
- foreign-key columns,

then the database automatically creates indexes on those columns.

Why? Because primary-key uniqueness and foreign-key checks need fast lookup. Without indexes, those checks would require full table scans.

9.4.7 Index Handling

This section manages index creation and index lookup.

```
def create_index(self, table_name: str, col: str, method: str = "btree"):
    method = method.strip().lower()
    if method not in {"btree", "hash"}:
        raise ValueError("method must be 'btree' or 'hash'")
```

The database supports two index types:

- hash index
- B+ tree index

```
if method == "btree":
    if col in ti.btree_indexes:
        return ti.btree_indexes[col]
    idx = BPlusTreeIndex(table_name, col)
    ti.btree_indexes[col] = idx
else:
    if col in ti.hash_indexes:
```

```
        return ti.hash_indexes[col]
    idx = HashIndex(table_name, col)
    ti.hash_indexes[col] = idx
```

This creates the requested index object only if it does not already exist.

Index rebuild from heap scan

```
for rid, row in ti.table.scan():
    key = row.get(col)
    if key is not None:
        idx.insert(key, rid)
```

This rebuilds the index from the heap table by scanning all rows and inserting key-RID pairs into the new index.

Meaning. Indexes are derived structures. They can always be rebuilt from the base table.

```
def _ensure_index(self, table_name: str, col: str):
    ti = self._ensure_table_info(table_name)
    if col in ti.hash_indexes:
        return ti.hash_indexes[col]
    if col in ti.btree_indexes:
        return ti.btree_indexes[col]
    return self.create_index(table_name, col, self.default_index)
```

This guarantees that an index exists on a column before the database tries to probe that column.

```
def _probe_rids(self, table_name: str, col: str, key: Any) -> List[GlobalRID]:
    if key is None:
        return []
    idx = self._ensure_index(table_name, col)
    rids = idx.probe(key)
    if not rids:
        return []
    return list(rids)
```

This is the main index-backed lookup helper. Given a table, column, and key, it returns all matching RIDs.

Very important. This method is the foundation for:

- primary-key uniqueness checking,
- foreign-key existence checking,
- child-row collection for cascade or restrict behavior.

9.4.8 Constraint Helpers

These methods implement the state-dependent integrity logic of the database.

Primary-key uniqueness

```
def _check_pk_unique(self, ti: TableInfo, row: Dict[str, Any], *, ignore_rids: Optional[Set[GlobalRID]] = None) -> None:
    pk = ti.table.schema.primary_key
    if not pk:
        return
    pk_val = row.get(pk)
    if pk_val is None:
        return
    hits = set(self._probe_rids(ti.table.name, pk, pk_val))
    if ignore_rids:
        hits.difference_update(ignore_rids)
    if hits:
        raise ValueError(
            f"Duplicate PRIMARY KEY {pk}={pk_val!r} for table {ti.table.name}"
        )
```

This method checks whether a primary-key value already exists in the table.

Why is `ignore_rids` needed? During update, the row being updated may already have the same primary-key value as before. In that case, the old RID should not count as a duplicate.

Foreign-key existence

```python
def _check_foreign_keys(self, row: Dict[str, Any], schema: Schema) -> None:
    for fk in schema.foreign_keys:
        val = row.get(fk.column)
        if val is None:
            continue
        if not self._probe_rids(fk.ref_table, fk.ref_column, val):
            raise ValueError(
                f"FOREIGN KEY violation: {schema.get_column(fk.column).name}={val!r} "
                f"references {fk.ref_table}.{fk.ref_column}"
            )
```

This checks that every non-null foreign-key value actually exists in the referenced parent table.

Meaning. This is exactly the kind of rule that cannot be checked inside `schema.py`, because it requires access to another table.

Find all referencing foreign keys

```python
def _referencing_fks(self, parent_table: str, parent_column: Optional[str] = None):
    out = []
    for child_table, (schema, _) in self._catalog.items():
        for fk in schema.foreign_keys:
            if fk.ref_table != parent_table:
                continue
            if parent_column is not None and fk.ref_column != parent_column:
                continue
            out.append((child_table, fk))
    return out
```

This method finds all child tables whose foreign keys point to a given parent table.

Why is this important? Because when a parent row is deleted or its key is updated, the database must know which child tables may be affected.

Foreign-key action helpers

```python
def _fk_on_delete(self, fk: ForeignKey) -> str:
    return str(getattr(fk, "on_delete", "RESTRICT")).upper().replace(" ", "_")

def _fk_on_update(self, fk: ForeignKey) -> str:
    return str(getattr(fk, "on_update", "RESTRICT")).upper().replace(" ", "_")
```

These normalize the stored FK action metadata.

Collect child RIDs

```python
def _collect_child_rids(self, child_table: str, fk_column: str, parent_value: Any) -> Set[GlobalRID]:
    return set(self._probe_rids(child_table, fk_column, parent_value))
```

This returns all child rows whose foreign-key column matches a specific parent value.

This is the basis for:

- delete restriction,
- cascade delete,
- cascade update,
- set-null behavior.

9.4.9 Transaction-Aware Table Touch Analysis

These helpers determine which tables must be locked for an operation.

Insert

```
def _tables_touched_by_insert(self, table_name: str, schema: Schema) -> Dict[str, str]:
    locks = {table_name: "X"}
    for fk in schema.foreign_keys:
        locks[fk.ref_table] = "S"
        self._ensure_index(fk.ref_table, fk.ref_column)
    return locks
```

For insert:

- the target table needs an exclusive lock,
- referenced parent tables need shared locks because FK existence must be checked.

Delete

```
def _tables_touched_by_delete(self, table_name: str) -> Dict[str, str]:
    locks = {table_name: "X"}
    for child_table, fk in self._referencing_fks(table_name):
        locks[child_table] = "X"
        self._ensure_index(child_table, fk.column)
    return locks
```

For delete:

- the parent table needs an exclusive lock,
- child tables may also need exclusive locks because rows may be deleted or updated due to referential actions.

Update

```
def _tables_touched_by_update(self, table_name: str, schema: Schema) -> Dict[str, str]:
    locks = {table_name: "X"}
    for fk in schema.foreign_keys:
        locks[fk.ref_table] = "S"
        self._ensure_index(fk.ref_table, fk.ref_column)
    for child_table, fk in self._referencing_fks(table_name):
        locks[child_table] = "X"
        self._ensure_index(child_table, fk.column)
    return locks
```

For update:

- the updated table needs an exclusive lock,
- parent tables may need shared locks for FK checking,
- child tables may need exclusive locks for cascade or restrict handling.

Meaning. This is why the file claims to support transaction-aware checks. It does not just validate constraints; it also acquires appropriate locks on the related tables.

9.4.10 Physical Index Maintenance

Indexes must stay consistent with the heap table.

```
def _index_insert_row(self, ti: TableInfo, rid: GlobalRID, row: Dict[str, Any]) -> None:
    for col, idx in ti.hash_indexes.items():
        key = row.get(col)
        if key is not None:
            idx.insert(key, rid)
    for col, idx in ti.btree_indexes.items():
        key = row.get(col)
        if key is not None:
            idx.insert(key, rid)
```

This adds a newly inserted row into all relevant indexes.

```python
def _index_delete_row(self, ti: TableInfo, rid: GlobalRID, row: Dict[str, Any]) -> None:
    for col, idx in ti.hash_indexes.items():
        key = row.get(col)
        if key is not None:
            idx.delete(key, rid)
    for col, idx in ti.btree_indexes.items():
        key = row.get(col)
        if key is not None:
            idx.delete(key, rid)
```

This removes a deleted row from all relevant indexes.

Meaning. The heap table stores the real tuples, but indexes must be updated whenever tuples are inserted, deleted, or physically replaced.

9.4.11 Internal Physical DML Helpers

These methods call the heap-table layer and then keep indexes in sync.

Physical insert

```python
def _insert_physical(self, ti: TableInfo, row: Dict[str, Any]) -> GlobalRID:
    rid = ti.table.insert(row)
    self._index_insert_row(ti, rid, row)
    return rid
```

This inserts a row into the heap table and then inserts the corresponding index entries.

Physical delete

```python
def _delete_physical(self, ti: TableInfo, rid_set: Set[GlobalRID]) -> List[Tuple[GlobalRID, Dict[str, Any]]]:
    deleted_rows = ti.table.delete_where(rid_set=rid_set)
    for rid, old_row in deleted_rows:
        self._index_delete_row(ti, rid, old_row)
    return deleted_rows
```

This deletes rows physically from the heap table and removes their index entries.

Physical replacement

```python
def _replace_physical(
    self,
    ti: TableInfo,
    replacements: Iterable[Tuple[GlobalRID, Dict[str, Any], Dict[str, Any]]],
) -> List[Tuple[GlobalRID, GlobalRID, Dict[str, Any], Dict[str, Any]]]:
    out = []
    for old_rid, old_row, new_row in replacements:
        new_rid = ti.table.insert(new_row)
        deleted = ti.table.delete_where({old_rid})
        if not deleted:
            ti.table.delete_where({new_rid})
            raise RuntimeError(f"Failed to delete old RID during update: {old_rid}")
        self._index_delete_row(ti, old_rid, old_row)
        self._index_insert_row(ti, new_rid, new_row)
        out.append((old_rid, new_rid, old_row, new_row))
    return out
```

This implements physical update as:

$$\texttt{insert new row} + \texttt{delete old row} + \texttt{update indexes}$$

Why is this useful? Because the updated row may not fit into the old physical location. So it is safer to treat update as a replacement operation.

9.4.12 DML Public API

Now we reach the most important part of the file: public insert, delete, and update.

`insert()`

```python
def insert(self, table_name: str, row: Dict[str, Any], tx: Transaction) -> GlobalRID:
    if tx is None:
        raise ValueError("Transaction is required for insert().")

    ti = self._ensure_table_info(table_name)
    schema = ti.table.schema
    row2 = self._normalize_input_row(schema, row)

    self._acquire_table_locks(tx, self._tables_touched_by_insert(table_name, schema))

    if schema.primary_key:
        self._ensure_index(table_name, schema.primary_key)
    self._check_pk_unique(ti, row2)
    self._check_foreign_keys(row2, schema)

    return self._insert_physical(ti, row2)
```

This is the full insert pipeline.

Step-by-step meaning.

1. A transaction must exist.
2. Load the table information.
3. Perform schema-local validation through `schema.validate_row()`.
4. Acquire all required table locks.
5. Ensure PK index exists if needed.
6. Check primary-key uniqueness.
7. Check foreign-key existence.
8. Only then perform the physical heap insert and index maintenance.

This is an excellent example of clean layering.

`delete_where()` and internal delete logic

```python
def _delete_where_internal(self, table_name: str, rid_set: Set[GlobalRID], tx: Transaction) -> int:
    ti = self._ensure_table_info(table_name)
    rows = ti.table.read_where(rid_set)
    if not rows:
        return 0

    schema = ti.table.schema
    self._acquire_table_locks(tx, self._tables_touched_by_delete(table_name))
```

This begins delete processing by reading the target rows and acquiring the necessary locks.

Referential-action enforcement on delete

```python
for rid, row in rows:
    for child_table, fk in self._referencing_fks(table_name):
        parent_val = row.get(fk.ref_column)
        if parent_val is None:
            continue
        child_rids = self._collect_child_rids(child_table, fk.column, parent_val)
        if not child_rids:
            continue

        action = self._fk_on_delete(fk)
        if action == "CASCADE":
            self._delete_where_internal(child_table, child_rids, tx)
        elif action in {"SET_NULL", "SETNULL"}:
            self._update_where_internal(child_table, child_rids, {fk.column: None}, tx)
```

```
        else:
            raise ValueError(
                f"Delete restricted: {table_name}.{fk.ref_column}={parent_val!r} "
                f"is referenced by {child_table}.{fk.column}"
            )
```

This is one of the most important blocks in the whole file.

When a parent row is being deleted, the database checks every referencing child table.

Then it applies the foreign-key action:

- `CASCADE`: delete child rows too
- `SET_NULL`: set child FK values to `None`
- otherwise: treat as `RESTRICT` and reject the delete

What does RESTRICT mean here? It means:

> Do not allow the parent row to be deleted if child rows still reference it.

Final physical delete

```
deleted_rows = self._delete_physical(ti, {rid for rid, _ in rows})
return len(deleted_rows)
```

Only after referential checks succeed does the database actually delete the parent rows physically.

`update_where()` and internal update logic

```
def _update_where_internal(self, table_name: str, rid_set: Set[GlobalRID], updates: dict, tx: Transaction) -> int:
    ti = self._ensure_table_info(table_name)
    schema = ti.table.schema
    old_rows = ti.table.read_where(rid_set)
    if not old_rows:
        return 0
```

This begins update processing by loading the affected rows.

Acquire locks and prepare validation

```
self._acquire_table_locks(tx, self._tables_touched_by_update(table_name, schema))
if schema.primary_key:
    self._ensure_index(table_name, schema.primary_key)
```

This ensures the database holds all relevant locks before changing any data.

Validate updated rows before touching storage

```
prepared: List[Tuple[GlobalRID, Dict[str, Any], Dict[str, Any]]] = []
old_rids = {rid for rid, _ in old_rows}

for old_rid, old_row in old_rows:
    merged = dict(old_row)
    merged.update(updates)
    new_row = self._normalize_input_row(schema, merged)
    ignore = {old_rid}
    self._check_pk_unique(ti, new_row, ignore_rids=ignore)
    self._check_foreign_keys(new_row, schema)
    prepared.append((old_rid, old_row, new_row))
```

This is another very important design choice.

Before the system touches storage, it validates all updated rows:

- schema-local validation,
- primary-key uniqueness,

- foreign-key existence.

So storage changes happen only after all updated rows pass validation.

Duplicate new primary keys within the same batch

```
pk = schema.primary_key
if pk:
    seen: Dict[Any, GlobalRID] = {}
    for old_rid, _, new_row in prepared:
        key = new_row.get(pk)
        if key is None:
            continue
        prev = seen.get(key)
        if prev is not None and prev != old_rid:
            raise ValueError(f"Duplicate PRIMARY KEY {pk}={key!r} within update batch")
        seen[key] = old_rid
```

This catches a subtle case: even if every updated row is individually valid, two different rows in the same update batch must not end up with the same new primary-key value.

Apply physical parent-row updates first

```
applied = self._replace_physical(ti, prepared)
```

This performs the actual physical replacement of the updated parent rows.

Then enforce ON UPDATE actions

```
for _, _, old_row, new_row in applied:
    for child_table, fk in self._referencing_fks(table_name):
        old_parent_val = old_row.get(fk.ref_column)
        new_parent_val = new_row.get(fk.ref_column)
        if old_parent_val == new_parent_val:
            continue
        child_rids = self._collect_child_rids(child_table, fk.column, old_parent_val)
        if not child_rids:
            continue

        action = self._fk_on_update(fk)
        if action == "CASCADE":
            self._update_where_internal(child_table, child_rids, {fk.column: new_parent_val}, tx)
        elif action in {"SET_NULL", "SETNULL"}:
            self._update_where_internal(child_table, child_rids, {fk.column: None}, tx)
        else:
            raise ValueError(
                f"Update restricted: changing {table_name}.{fk.ref_column} from {old_parent_val!r} "
                f"to {new_parent_val!r} would break {child_table}.{fk.column}"
            )
```

This enforces foreign-key actions for parent-key update.

Meaning. If a referenced parent key changes, child rows may need to be:

- updated too (`CASCADE`),
- set to `NULL` (`SET_NULL`),
- or blocked (`RESTRICT`).

So this is the update-time counterpart of delete-time referential action handling.

9.4.13 SQL Execution

This section turns SQL strings into actual engine operations.

```
def execute_sql(self, sql: str):
    print("[SQL]", sql)

    with self.begin() as tx:
```

```
        parsed = parser.parse(sql)
```

Every SQL statement runs inside a transaction.

Handling INSERT

```
if op == "INSERT":
    _, table, row = parsed
    return self.insert(table, row, tx)
```

If the parser returns an INSERT form, the database calls the public insert API described earlier.

Handling DELETE

```
if op == "DELETE_PLAN":
    _, table_name, logical_plan = parsed
    ti = self._ensure_table_info(table_name)
    optimizer = self._get_optimizer()
    plan = optimizer.compile(logical_plan)
    plan.open()
    target_rids = set()
    while True:
        row = plan.next()
        if row is None:
            break
        target_rids.add(row["_rid"])
    plan.close()
    return self.delete_where(table_name, target_rids, tx)
```

DELETE works by:

1. compiling the logical plan,
2. executing it to find target RIDs,
3. passing those RIDs into `delete_where()`.

This is a very clean design: query planning identifies which rows to affect, then the DML layer performs the actual relationally safe delete.

Handling UPDATE_EQ

```
if op == "UPDATE_EQ":
    _, table_name, sets, col, val = parsed
    ti = self._ensure_table_info(table_name)
    rid_set = set(self._probe_rids(table_name, col, val))
    if not rid_set:
        return 0
    return self.update_where(table_name, rid_set, sets, tx)
```

This update form uses an index-backed equality lookup to find the target RIDs, then calls `update_where()`.

SELECT and other query plans

```
optimizer = self._get_optimizer()
plan = optimizer.compile(parsed)
plan.open()
out = []
while True:
    row = plan.next()
    if row is None:
        break
    out.append(row)
plan.close()
return out
```

If the SQL is not a DML special case, the database compiles it into a physical plan and executes it as a query.

9.4.14 Miscellaneous Methods

Close database

```
def close(self):
    self.bp.flush_all()
    for ti in self._tables.values():
        ti.table.close()
```

This flushes all dirty pages from the buffer pool and closes all loaded tables.

Scan helper

```
def scan(self, table_name: str, tx: Optional[Transaction] = None):
    ti = self._ensure_table_info(table_name)
    if tx is not None:
        self._acquire_lock(tx, f"table:{table_name}", "S")
    yield from ti.table.scan()
```

This provides a direct scan interface over a table. If a transaction is provided, it acquires a shared table lock first.

9.4.15 What this file now teaches clearly

Now, we can clearly know that:

- **Schema** knows what one valid row looks like
- **HeapTable** knows how one row is physically stored
- **Database** knows whether the operation is valid given the whole database state

This file is the best example of that third sentence.

9.4.16 What this database layer enforces

- table metadata and catalog loading,
- automatic table opening,
- automatic index reconstruction,
- primary-key uniqueness,
- foreign-key existence,
- delete-time referential actions,
- update-time referential actions,
- index-backed constraint checking,
- transaction-aware lock acquisition,
- SQL execution and DML routing.

9.4.17 What it intentionally leaves to other layers

- row-local type and null checks belong to `schema.py`,
- slotted-page storage belongs to `table.py`,
- low-level page and buffer management belong to `disk_file.py` and `buffer_manager.py`,
- plan generation belongs to the optimizer.

So `db.py` is not doing everything. It is doing the things that require knowledge of the whole database state.

9.4.18 Core meaning of `db.py`

The core meaning of this file is:

> The database layer is the relational integrity manager of EduDB. It is the place where local row validation becomes full database-aware correctness.

9.4.19 Summary

The file `db.py` is the engine coordinator of EduDB.

It manages:

- the catalog,
- loaded tables,
- indexes,
- transactions,
- locking,
- PK and FK enforcement,
- referential actions,
- SQL execution.

Most importantly, it shows the correct DBMS architecture:

- `schema.py` handles local row rules,
- `table.py` handles physical storage,
- `db.py` handles global relational rules.

That is exactly the right separation for a teaching DBMS.

10 Sql Parsing

SQL processing in the DB design follows a staged pipeline:

$$\text{SQL} \xrightarrow{\text{parser}} \text{AST} \xrightarrow{\text{lowering}} \text{logical plan} \xrightarrow{\text{optimizer}} \text{physical operators} \xrightarrow{\text{execution}} \text{results / database changes.}$$

First, the parser reads the SQL string, checks its syntax, and converts it into an abstract syntax tree (AST), which captures the structure of the statement in a clean and uniform form. Next, the lowering phase translates the AST into an internal execution-oriented representation: a relational algebra logical plan for queries such as `SELECT`, or a DML plan for statements such as `INSERT`, `DELETE`, and `UPDATE`. The optimizer then examines the logical plan and chooses an efficient physical strategy, such as whether to use a full table scan, a hash index, or a B+Tree range scan, and compiles the plan into physical operators. Finally, the execution engine runs these physical operators, typically through an `open()`–`next()`–`close()` interface, to produce query results or apply modifications to the database.

10.1 AST

AST stands for **Abstract Syntax Tree**. An AST is a tree-structured internal representation of a statement. It keeps the logical structure of the SQL query, but removes unnecessary surface details such as extra spaces, punctuation style, or exact formatting.

For example, the SQL:

```
SELECT name FROM Student WHERE gpa >= 3.5;
```

can be represented by an AST that says:

- this is a `SelectStmt`,
- the `FROM` table is `Student`,
- the select list contains one column item `name`,
- the `WHERE` clause is a comparison `gpa >= 3.5`.

So the AST preserves the meaningful structure of the query.

It is called abstract because it ignores low-level syntax details that do not matter semantically. For example, these SQL statements mean the same thing:

```
SELECT name FROM Student WHERE gpa >= 3.5;
SELECT   name   FROM   Student   WHERE   gpa>=3.5
```

A parser may read different text forms, but the AST should be the same. An AST creates a clean separation between two stages:

$$\text{SQL text} \longrightarrow \text{AST} \longrightarrow \text{logical plan / execution representation}$$

This separation is valuable because:

- parsing becomes easier to reason about,
- translation to relational algebra becomes a separate step,
- optimization can inspect a structured query representation,
- future features such as joins, nested queries, and `GROUP BY` can be added more naturally.

10.1.1 Our AST design

`SQLNode` `SQLNode` is the root base class for the entire SQL abstract syntax tree in this module. It does not store any fields or define any behavior by itself; instead, it provides a common parent type for all AST nodes, including statements, expressions, and select-list items. This design gives the parser and later phases a shared conceptual foundation, making it clear that all of these objects are pieces of the same syntax tree even though they play different roles.

```
class SQLNode:
    # Root base class for every SQL AST node in this file.
    #
    # This class is intentionally empty.
    # Its purpose is organizational rather than operational:
```

```
    # it tells us that all subclasses belong to the SQL AST hierarchy.
    #
    # Examples of subclasses:
    #   - Statement
    #   - Expr
    #   - SelectItem
    #
    # Having a single top-level base class makes the AST design cleaner,
    # because the rest of the system can reason about these objects as
    # "SQL nodes" in a unified way.
    pass
```

`Statement` `Statement` is the base class for top-level SQL statement nodes such as `SELECT`, `INSERT`, `DELETE`, and `UPDATE`. Like `SQLNode`, it is intentionally empty, but it plays an important structural role: it distinguishes complete executable SQL statements from smaller components such as expressions or select-list items. In other words, `Statement` marks the nodes that can stand alone as full SQL commands.

```
class Statement(SQLNode):
    # Base class for complete SQL statements.
    #
    # A Statement is something that can stand on its own as a full command,
    # such as:
    #   SELECT ...
    #   INSERT ...
    #   DELETE ...
    #   UPDATE ...
    #
    # This class is empty because its main job is to categorize AST nodes.
    # Later code can test:
    #   isinstance(node, Statement)
    # to determine whether a parsed node is a full SQL statement.
    pass
```

`Expr` `Expr` is the base class for expression nodes inside SQL statements. In this file, it is primarily used for predicates appearing in `WHERE` clauses, such as column references, literal values, comparisons, and conjunctions. By separating expressions from complete statements, the AST can represent both the whole SQL command and the smaller logical pieces that appear inside it.

```
class Expr(SQLNode):
    # Base class for SQL expression nodes.
    #
    # Expressions are smaller building blocks that appear inside statements,
    # especially inside WHERE clauses.
    #
    # Examples:
    #   ColumnRef("gpa")
    #   Literal(3.5)
    #   Compare(">=", ColumnRef("gpa"), Literal(3.5))
    #   And(expr1, expr2)
    #
    # Like the other base classes, Expr is intentionally empty.
    # It provides a common type for all expression AST nodes.
    pass
```

`ColumnRef` `ColumnRef` represents a reference to a column name inside an expression. Its single field, `name`, stores the name of the referenced attribute. This node is especially important in predicates, since conditions such as `gpa >= 3.5` need a way to represent the column part separately from the literal value and the comparison operator.

```
@dataclass
class ColumnRef(Expr):
    # Name of the referenced column.
    #
    # Example:
    #   In the predicate:
    #       gpa >= 3.5
    #   the "gpa" part is represented as:
    #       ColumnRef("gpa")
    #
    # This class inherits from Expr because a column reference is a kind
    # of expression node.
```

```
    name: str
```

Literal `Literal` represents a constant value appearing in an expression, such as a number, string, or `NULL`. Its field `value` stores the Python value corresponding to the parsed literal. For example, the SQL literal `3.5` may become the Python float `3.5`, and `NULL` may become `None`. This node complements `ColumnRef` in predicates by representing the constant side of a comparison.

```
@dataclass
class Literal(Expr):
    # Constant value used inside an expression.
    #
    # Examples:
    #   Literal(1001)
    #   Literal(3.5)
    #   Literal("Alice")
    #   Literal(None)    # for SQL NULL
    #
    # This class inherits from Expr because a literal value is also
    # an expression node.
    value: Any
```

Compare `Compare` represents a comparison expression between two subexpressions. It stores the comparison operator in `op`, the left operand in `left`, and the right operand in `right`. In the current design, this node is used for simple predicates such as equality and range conditions. It is a central part of the `WHERE` clause representation because it captures the actual logical test being performed.

```
@dataclass
class Compare(Expr):
    # Comparison operator, such as:
    #   "="
    #   ">="
    #   "<="
    #
    # Example:
    #   Compare(
    #       op=">=",
    #       left=ColumnRef("gpa"),
    #       right=Literal(3.5)
    #   )
    #
    # This represents the condition:
    #   gpa >= 3.5
    op: str          # '=', '>=', '<='

    # Left-hand side of the comparison.
    # Usually this is a ColumnRef in the current parser design.
    left: Expr

    # Right-hand side of the comparison.
    # Usually this is a Literal in the current parser design.
    right: Expr
```

And `And` represents a logical conjunction of two expressions. It stores the left and right subexpressions and models conditions connected by the SQL keyword `AND`. This node allows the AST to represent compound predicates such as `gpa >= 3.5 AND gpa <= 3.8`, making it possible for later phases to analyze or lower multi-condition filters systematically.

```
@dataclass
class And(Expr):
    # Left side of the conjunction.
    #
    # Example:
    #   gpa >= 3.5 AND gpa <= 3.8
    #
    # The left side might be:
    #   Compare(">=", ColumnRef("gpa"), Literal(3.5))
    left: Expr

    # Right side of the conjunction.
    #
```

```
    # In the same example, the right side might be:
    #   Compare("<=", ColumnRef("gpa"), Literal(3.8))
    #
    # So the full AST node represents:
    #   left AND right
    right: Expr
```

SelectItem SelectItem is the base class for items that appear in the SELECT list. This includes selecting all columns with *, selecting specific columns, and selecting aggregate expressions. The class itself has no fields, but it provides a shared type so that the select_items field of a SelectStmt can hold a list of different kinds of select-list elements in a uniform way. :contentReference[oaicite:8]index=8

```
@dataclass
class SelectItem(SQLNode):
    # Base class for things that can appear in the SELECT list.
    #
    # Examples of subclasses:
    #   Star()                  -> SELECT *
    #   ColumnItem("name")      -> SELECT name
    #   AggregateItem(...)      -> SELECT COUNT(*)
    #
    # This class is empty because it is mainly used for typing
    # and AST organization.
    pass
```

Star Star represents the special * symbol in a SELECT list. It indicates that all columns from the source relation should be returned. Since * carries no additional information beyond its presence, this class has no fields. Its purpose is simply to distinguish the SELECT * case from ordinary column selection or aggregation. :contentReference[oaicite:9]index=9

```
@dataclass
class Star(SelectItem):
    # Represents the '*' symbol in a SELECT list.
    #
    # Example:
    #   SELECT * FROM Student
    #
    # Since '*' does not need extra data, this class has no fields.
    # Its mere presence in the AST is enough to tell later phases
    # that the query wants all columns.
    pass
```

ColumnItem ColumnItem represents a plain column name in the SELECT list. Its field name stores the selected attribute. This node is used for ordinary projections such as SELECT name, gpa FROM Student, where each chosen column is represented separately in the AST before later phases turn the selection into a projection operator. :contentReference[oaicite:10]index=10

```
@dataclass
class ColumnItem(SelectItem):
    # Name of the selected column in the SELECT list.
    #
    # Example:
    #   SELECT name FROM Student
    #
    # The "name" part becomes:
    #   ColumnItem("name")
    #
    # If the query selects multiple columns, the parser will create
    # multiple ColumnItem objects inside the select_items list.
    name: str
```

AggregateItem AggregateItem represents an aggregate expression in the SELECT list, such as COUNT(*), MIN(gpa), MAX(student_id), or AVG(gpa). The field func stores the aggregate function name, while arg stores the argument, which may be either a column name or *. This node allows the parser to distinguish aggregate queries from ordinary projections at the AST level. :contentReference[oaicite:11]index=11

```
@dataclass
class AggregateItem(SelectItem):
```

```
    # Name of the aggregate function.
    #
    # Supported examples in this parser:
    #   MIN
    #   MAX
    #   COUNT
    #   AVG
    func: str        # MIN, MAX, COUNT, AVG

    # Argument of the aggregate.
    #
    # Examples:
    #   "*"     for COUNT(*)
    #   "gpa"   for AVG(gpa)
    #   "id"    for MAX(id)
    arg: str         # '*' or column name
```

SelectStmt SelectStmt represents a full SELECT statement in the AST. It stores the parsed select list in select_items, the source table name in from_table, and an optional predicate expression in where. This class is the main container for query statements, bringing together the projection, source relation, and filtering condition in a single structured object that later phases can lower into relational algebra. :contentReference[oaicite:12]index=12

```
@dataclass
class SelectStmt(Statement):
    # List of items that appear after SELECT.
    #
    # Examples:
    #   [Star()]
    #   [ColumnItem("name"), ColumnItem("gpa")]
    #   [AggregateItem("COUNT", "*")]
    #
    # This field captures what the query wants to return.
    select_items: List[SelectItem]

    # Name of the table in the FROM clause.
    #
    # Example:
    #   SELECT * FROM Student
    # gives:
    #   from_table = "Student"
    from_table: str

    # Optional WHERE predicate represented as an expression AST.
    #
    # Example:
    #   WHERE gpa >= 3.5
    # may become:
    #   Compare(">=", ColumnRef("gpa"), Literal(3.5))
    #
    # If there is no WHERE clause, this field is None.
    where: Optional[Expr] = None
```

InsertStmt InsertStmt represents a full INSERT statement in the AST. It stores the target table name, the list of column names being assigned, and the list of values to be inserted. This class gives a direct structured representation of an insertion command, making it easy for later lowering logic to pair each column with its corresponding value and build an insertion plan. :contentReference[oaicite:13]index=13

```
@dataclass
class InsertStmt(Statement):
    # Name of the table receiving the new tuple.
    #
    # Example:
    #   INSERT INTO Student (...)
    # gives:
    #   table = "Student"
    table: str

    # Ordered list of column names that appear in the INSERT statement.
    #
    # Example:
    #   INSERT INTO Student (student_id, name, gpa) ...
    # gives:
    #   ["student_id", "name", "gpa"]
```

```
    columns: List[str]

    # Ordered list of inserted values corresponding to the columns list.
    #
    # Example:
    #   VALUES (1001, 'Alice', 3.8)
    # becomes something like:
    #   [1001, "Alice", 3.8]
    #
    # Later lowering logic typically combines columns and values
    # into a dictionary representing one row.
    values: List[Any]
```

DeleteStmt DeleteStmt represents a full DELETE statement in the AST. It stores the target table and an optional WHERE expression identifying which rows should be removed. If the where field is None, the statement logically refers to deleting all rows from the table. This class provides a compact representation of row-deletion commands before they are lowered into a delete plan. :contentReference[oaicite:14]index=14

```
@dataclass
class DeleteStmt(Statement):
    # Name of the table from which rows will be deleted.
    #
    # Example:
    #   DELETE FROM Student ...
    # gives:
    #   table = "Student"
    table: str

    # Optional WHERE condition describing which rows should be removed.
    #
    # Example:
    #   DELETE FROM Student WHERE gpa < 2.0
    #
    # In the current parser, supported conditions are stored as Expr nodes
    # such as Compare or And.
    #
    # If this field is None, the statement means delete all rows.
    where: Optional[Expr] = None
```

Assignment Assignment represents one column-value pair inside the SET clause of an UPDATE statement. Its field column stores the name of the attribute being modified, and value stores the new assigned value. By representing each assignment separately, the AST can naturally support updates that modify multiple columns at once. :contentReference[oaicite:15]index=15

```
@dataclass
class Assignment(SQLNode):
    # Name of the column being assigned a new value.
    #
    # Example:
    #   SET gpa = 4.0
    # gives:
    #   column = "gpa"
    column: str

    # New value assigned to that column.
    #
    # Example:
    #   SET gpa = 4.0
    # gives:
    #   value = 4.0
    #
    # A full UPDATE statement may contain multiple Assignment objects.
    value: Any
```

UpdateStmt UpdateStmt represents a full UPDATE statement in the AST. It stores the target table name, a list of Assignment objects representing the SET clause, and an optional predicate expression in where. This class combines both modification information and conditional logic, since an update must specify not only what values to change but also which rows should receive those changes. It therefore serves as the AST representation of one of the most structurally rich DML statements in the file. :contentReference[oaicite:16]index=16

```python
@dataclass
class UpdateStmt(Statement):
    # Name of the table whose rows will be updated.
    #
    # Example:
    #   UPDATE Student SET ...
    # gives:
    #   table = "Student"
    table: str

    # List of Assignment objects representing the SET clause.
    #
    # Example:
    #   SET gpa = 4.0, major = 'CS'
    # becomes something like:
    #   [
    #       Assignment("gpa", 4.0),
    #       Assignment("major", "CS")
    #   ]
    assignments: List[Assignment]

    # Optional WHERE predicate limiting which rows are updated.
    #
    # Example:
    #   WHERE student_id = 1001
    # may become:
    #   Compare("=", ColumnRef("student_id"), Literal(1001))
    #
    # If this field is None, the update logically applies to all rows.
    where: Optional[Expr] = None
```

10.2 Parser

The parser has one main job:

> Read a SQL string and build a structured abstract syntax tree (AST) that represents the meaning of the statement at the syntax level.

In other words, the parser should answer the question:

> What does this SQL statement say?

It should not yet answer:

> How should the database execute it?

That second question belongs to lowering, optimization, and execution.

10.2.1 Why Use an AST?

The main advantage of the AST is that it gives a uniform parser output. The AST design provides three major benefits:

1. It separates **syntax** from **execution**.
2. It makes the system easier to extend later.
3. It makes the code easier to reason about.

10.2.2 How the Parser Works

Step 1: normalize the SQL string At the top level, the parser first normalizes the input SQL string. It does three useful cleanup steps:

1. collapse repeated whitespace,
2. strip leading/trailing spaces,
3. remove a trailing semicolon.

This is a simple but effective preprocessing step because it makes the later regular-expression matches more reliable.

Step 2: dispatch by statement type After normalization, the parser converts a copy of the SQL string to uppercase and checks its prefix:

- `SELECT` → `_parse_select`
- `INSERT` → `_parse_insert`
- `DELETE` → `_parse_delete`
- `UPDATE` → `_parse_update`

So the parser behaves like a simple dispatcher:

$$\text{SQL string} \longrightarrow \text{statement-specific parser.}$$

If no supported statement prefix matches, the parser raises an error.

10.2.3 Parsing the WHERE Clause

The `WHERE` clause is parsed by `_parse_where(where_clause)`. This function returns an AST expression of type `Expr`, not an executable predicate.

The parser supports predicate pieces of the form:

$$col = literal, \qquad col \geq literal, \qquad col \leq literal$$

and combines multiple parts using `AND`.

For example:

```
WHERE age >= 12 AND age <= 20
```

is parsed into two `Compare` nodes, then combined using `And`.

Suppose the parser reads three conditions:

$$c_1,\ c_2,\ c_3.$$

It first builds a list of individual comparison expressions, then folds them into a left-associated tree:

$$And(And(c_1, c_2), c_3).$$

This means the parser represents conjunction structurally rather than flattening it into a string.

10.2.4 Parsing the SELECT List

The helper `_parse_select_items(select_expr)` determines what kind of select list is present.

There are three cases.

1. star: If the expression is exactly:

   ```
   *
   ```

   the parser returns:

   ```
   [Star()]
   ```

   This represents `SELECT *`.

2. aggregate: If the expression matches:

   ```
   MIN(...)
   MAX(...)
   COUNT(...)
   AVG(...)
   ```

then the parser returns:

```
[AggregateItem(func, arg)]
```

For example:

```
SELECT COUNT(*) FROM Student
```

becomes a `SelectStmt` whose `select_items` field contains one `AggregateItem("COUNT", "*")`.

3. ordinary column list: Otherwise, the parser splits the string on commas and builds:

```
[ColumnItem(...), ColumnItem(...), ...]
```

For example:

```
SELECT name, major, gpa FROM Student
```

produces three `ColumnItem` nodes.

10.2.5 Parsing Each Statement Type

SELECT A `SELECT` statement is matched using a regular expression of the form:

```
SELECT <select_expr> FROM <table> [WHERE <where_clause>]
```

The result is:

```
SelectStmt(
    select_items=...,
    from_table=...,
    where=...
)
```

So the parser output for `SELECT` is now a clean AST object.

Example SQL:

```
SELECT name, age FROM dogs WHERE age >= 12 AND age <= 20;
```

AST idea:

```
SelectStmt(
    select_items=[ColumnItem("name"), ColumnItem("age")],
    from_table="dogs",
    where=And(
        Compare(">=", ColumnRef("age"), Literal(12)),
        Compare("<=", ColumnRef("age"), Literal(20))
    )
)
```

INSERT An `INSERT` statement is matched in the form:

```
INSERT INTO table(col1, col2, ...) VALUES(v1, v2, ...)
```

The parser extracts:

- the table name,
- the column list,
- the value list.

Then it returns:

```
InsertStmt(table=..., columns=[...], values=[...])
```

Example SQL:

```
INSERT INTO Student(student_id, name, major)
VALUES (1001, 'Alice', 'CS');
```

AST idea:

```
InsertStmt(
    table="Student",
    columns=["student_id", "name", "major"],
    values=[1001, "Alice", "CS"]
)
```

DELETE A `DELETE` statement is matched in the form:

```
DELETE FROM table [WHERE ...]
```

and returns:

```
DeleteStmt(table=..., where=...)
```

Example SQL:

```
DELETE FROM Student WHERE student_id = 1001;
```

AST idea:

```
DeleteStmt(
    table="Student",
    where=Compare("=", ColumnRef("student_id"), Literal(1001))
)
```

UPDATE An `UPDATE` statement is matched in the form:

```
UPDATE table SET col1=v1, col2=v2 [WHERE ...]
```

The parser builds a list of assignments:

```
Assignment(column, value)
```

and returns:

```
UpdateStmt(table=..., assignments=[...], where=...)
```

Example SQL:

```
UPDATE Student SET major='Math', gpa=3.9 WHERE student_id = 1001;
```

AST idea:

```
UpdateStmt(
    table="Student",
    assignments=[
        Assignment("major", "Math"),
        Assignment("gpa", 3.9)
    ],
    where=Compare("=", ColumnRef("student_id"), Literal(1001))
```

```
)
```

10.2.6 Our Implementation

`parse(sql: str) -> Statement` The function `parse` is the main entry point of the SQL parser. Its job is to receive a raw SQL string, normalize its formatting, and then determine which specific parsing routine should be used based on the statement type. It first removes extra whitespace, strips any trailing semicolon, and creates an uppercase version of the SQL string for keyword checking. Then it dispatches the input to one of the specialized helper functions such as `_parse_select`, `_parse_insert`, `_parse_delete`, or `_parse_update`. In this sense, `parse` acts as the top-level router of the parsing phase: it does not parse every statement itself, but it decides which lower-level parser should process the statement and return the corresponding AST node.

```python
def parse(sql: str) -> Statement:
    # Normalize whitespace so that inputs like:
    #   "SELECT   *   FROM   Student ;"
    # become easier to parse consistently.
    sql = " ".join(sql.split())

    # Remove leading/trailing spaces and any final semicolon.
    # This lets the parser accept both:
    #   SELECT * FROM Student
    # and
    #   SELECT * FROM Student;
    sql = sql.strip().rstrip(";")

    # Create an uppercase copy only for keyword testing.
    # We keep the original 'sql' because table names and literal values
    # should not necessarily be forced to uppercase.
    up = sql.upper()

    # Route the SQL statement to the appropriate parser
    # according to its first keyword.
    if up.startswith("SELECT"):
        return _parse_select(sql)
    if up.startswith("INSERT"):
        return _parse_insert(sql)
    if up.startswith("DELETE"):
        return _parse_delete(sql)
    if up.startswith("UPDATE"):
        return _parse_update(sql)

    # If the statement type is unknown, reject it explicitly.
    raise ValueError("Unsupported SQL")
```

`_parse_literal(text: str)` The function `_parse_literal` converts a raw textual token from SQL into a Python value that can be stored inside the AST. It first trims surrounding whitespace, then checks whether the token is `NULL`, in which case it returns Python's `None`. Otherwise, it tries to use `ast.literal_eval` to interpret the token safely as a Python literal, which allows numbers and quoted strings to be converted into their corresponding Python types. If that conversion fails, the function falls back to returning the raw text itself. This design makes the parser simple and flexible, since it can correctly interpret common literal values without building a complete SQL type system.

```python
def _parse_literal(text: str):
    # Remove surrounding whitespace from the raw token.
    s = text.strip()

    # SQL NULL is represented internally as Python None.
    if s.upper() == "NULL":
        return None

    try:
        # Try to interpret the token as a Python literal.
        #
        # Examples:
        #   "123"      -> 123
        #   "3.14"     -> 3.14
        #   "'Alice'"  -> "Alice"
        #
        # Using ast.literal_eval is safer than eval because it only
        # handles Python literals rather than arbitrary code.
        return pyast.literal_eval(s)

    except Exception:
```

```
        # If literal_eval fails, treat it as a bare token.
        #
        # Example:
        #   CS
        # would remain "CS" if it was not quoted and could not be
        # interpreted as a Python literal.
        return s
```

_parse_where(where_clause: str) -> Expr The function _parse_where parses the text of a WHERE clause and converts it into an expression AST. In the current implementation, it supports simple comparison predicates using =, >=, and <=, as well as conjunctions connected by AND. The function first splits the clause into separate conditions, then parses each condition into a Compare node whose left side is a ColumnRef and whose right side is a Literal. If multiple conditions are present, it combines them into a left-associated tree of And nodes. As a result, this function transforms textual filtering conditions into structured AST expressions that later phases, such as lowering and optimization, can analyze more systematically.

```
def _parse_where(where_clause: str) -> Expr:
    # Split the WHERE clause on the keyword AND, ignoring case.
    #
    # Example:
    #   "gpa >= 3.5 AND gpa <= 3.8"
    # becomes:
    #   ["gpa >= 3.5", "gpa <= 3.8"]
    parts = [p.strip() for p in re.split(r"\bAND\b", where_clause, flags=re.IGNORECASE)]

    exprs = []

    # Parse each individual condition.
    for part in parts:
        # Supported shape:
        #   column op value
        # where op is one of =, >=, <=
        m = re.match(r"(\w+)\s*(=|>=|<=)\s*(.+)$", part, re.IGNORECASE)
        if not m:
            raise ValueError(f"Unsupported WHERE condition: {part}")

        # Extract the column name, operator, and raw value text.
        col = m.group(1)
        op = m.group(2)
        raw_val = m.group(3)

        # Build a Compare AST node:
        #   Compare(
        #       op=">=",
        #       left=ColumnRef("gpa"),
        #       right=Literal(3.5)
        #   )
        exprs.append(
            Compare(
                op=op,
                left=ColumnRef(col),
                right=Literal(_parse_literal(raw_val)),
            )
        )

    # Reject an empty WHERE clause explicitly.
    if not exprs:
        raise ValueError("Empty WHERE clause")

    # If there is more than one predicate, combine them using And nodes.
    #
    # Example:
    #   expr1, expr2, expr3
    # becomes:
    #   And(And(expr1, expr2), expr3)
    node = exprs[0]
    for e in exprs[1:]:
        node = And(node, e)

    return node
```

_parse_select_items(select_expr: str) The function _parse_select_items parses the list of expressions that appear between SELECT and FROM. Its job is to determine whether the query is selecting all columns, selecting an

aggregate, or selecting one or more plain column names. If the text is exactly `*`, it returns a `Star` node. If the text matches an aggregate form such as `COUNT(*)`, `MIN(col)`, `MAX(col)`, or `AVG(col)`, it returns an `AggregateItem`. Otherwise, it treats the input as a comma-separated list of simple column names and returns a list of `ColumnItem` objects. This function is important because it converts the syntactic appearance of the select list into a structured representation that later stages can distinguish easily.

```
def _parse_select_items(select_expr: str):
    # Remove surrounding spaces from the SELECT-list text.
    select_expr = select_expr.strip()

    # Special case: SELECT *
    if select_expr == "*":
        return [Star()]

    # Try to match a supported aggregate:
    #   MIN(col)
    #   MAX(col)
    #   COUNT(*)
    #   AVG(col)
    #
    # The regex captures:
    #   group(1) -> function name
    #   group(2) -> argument column or *
    agg_match = re.fullmatch(r"(MIN|MAX|COUNT|AVG)\(((\*|\w+)\)", select_expr, re.IGNORECASE)
    if agg_match:
        return [AggregateItem(
            # Store the aggregate function in uppercase
            # for consistency.
            func=agg_match.group(1).upper(),

            # Store the argument, for example "*" or "gpa".
            arg=agg_match.group(2),
        )]

    # Otherwise, treat the SELECT list as a comma-separated list of columns.
    #
    # Example:
    #   "student_id, name, gpa"
    # becomes:
    #   [ColumnItem("student_id"), ColumnItem("name"), ColumnItem("gpa")]
    return [ColumnItem(c.strip()) for c in select_expr.split(",")]
```

`_parse_select(sql: str) -> SelectStmt` The function `_parse_select` parses an entire `SELECT` statement into a `SelectStmt` AST node. It first normalizes spacing, then uses a regular expression to separate the statement into three conceptual parts: the select list, the table name in the `FROM` clause, and the optional `WHERE` clause. After extracting these parts, it delegates the parsing of the select list to `_parse_select_items` and the parsing of the condition to `_parse_where`. Finally, it packages the results into a `SelectStmt` object. This function is therefore the main constructor for parsed `SELECT` queries, assembling multiple smaller parsing results into one coherent AST node.

```
def _parse_select(sql: str) -> SelectStmt:
    # Normalize spacing and remove a trailing semicolon if present.
    s = " ".join(sql.strip().rstrip(";").split())

    # Match the overall SELECT structure:
    #   SELECT <select_expr> FROM <table> [WHERE <where_clause>]
    #
    # group(1) -> select list
    # group(2) -> table name
    # group(3) -> optional WHERE clause
    m = re.match(
        r"SELECT\s+(.+?)\s+FROM\s+(\w+)(?:\s+WHERE\s+(.+))?$",
        s,
        re.IGNORECASE,
    )
    if not m:
        raise ValueError(f"Malformed SELECT: {sql}")

    # Extract the main parts of the SELECT statement.
    select_expr = m.group(1)
    table = m.group(2)
    where_clause = m.group(3)

    # Build and return the SelectStmt AST node.
    return SelectStmt(
```

```
        # Parse the SELECT list into Star / AggregateItem / ColumnItem nodes.
        select_items=_parse_select_items(select_expr),

        # Store the table from the FROM clause.
        from_table=table,

        # Parse the WHERE clause into an expression AST if it exists;
        # otherwise store None.
        where=_parse_where(where_clause) if where_clause else None,
    )
```

`_parse_insert(sql: str) -> InsertStmt` The function `_parse_insert` parses an `INSERT` statement into an `InsertStmt` AST node. It assumes the supported form `INSERT INTO table (cols) VALUES (vals)` and uses a regular expression to extract the target table, the list of column names, and the list of raw values. It then splits the column and value lists by commas, converts each value with `_parse_literal`, and returns an `InsertStmt` containing the final structured result. In effect, this function translates a textual insertion command into a form where the target relation, inserted columns, and inserted values are all explicitly represented in the AST.

```
def _parse_insert(sql: str) -> InsertStmt:
    # Normalize spacing and remove any trailing semicolon.
    s = " ".join(sql.strip().rstrip(";").split())

    # Match the supported INSERT form:
    #   INSERT INTO table_name (col1, col2, ...)
    #   VALUES (val1, val2, ...)
    #
    # group(1) -> table name
    # group(2) -> comma-separated column list
    # group(3) -> comma-separated value list
    m = re.match(
        r"INSERT\s+INTO\s+(\w+)\s*\((.*?)\)\s*VALUES\s*\((.*?)\)$",
        s,
        re.IGNORECASE,
    )
    if not m:
        raise ValueError(f"Malformed INSERT: {sql}")

    # Extract the target table.
    table = m.group(1)

    # Split the column list and strip spaces around each name.
    cols = [c.strip() for c in m.group(2).split(",")]

    # Split the raw value list and strip spaces around each value text.
    raw_vals = [v.strip() for v in m.group(3).split(",")]

    # Convert each raw textual value into a Python literal / None / token.
    vals = [_parse_literal(v) for v in raw_vals]

    # Return the structured InsertStmt AST node.
    return InsertStmt(table=table, columns=cols, values=vals)
```

`_parse_delete(sql: str) -> DeleteStmt` The function `_parse_delete` parses a `DELETE` statement into a `DeleteStmt` AST node. It supports the general form `DELETE FROM table` with an optional `WHERE` clause. After normalizing the input, it uses a regular expression to extract the target table name and any condition that follows `WHERE`. If a condition is present, it is parsed into an expression AST by `_parse_where`; otherwise the `where` field is set to `None`. This function therefore converts a textual delete command into a structured object that explicitly describes both the relation being modified and the condition identifying which rows should be deleted.

```
def _parse_delete(sql: str) -> DeleteStmt:
    # Normalize spacing and remove any trailing semicolon.
    s = " ".join(sql.strip().rstrip(";").split())

    # Match the supported DELETE form:
    #   DELETE FROM table_name
    #   DELETE FROM table_name WHERE ...
    #
    # group(1) -> table name
    # group(2) -> optional WHERE clause
    m = re.match(
        r"DELETE\s+FROM\s+(\w+)(?:\s+WHERE\s+(.+))?$",
        s,
```

```
        re.IGNORECASE,
    )
    if not m:
        raise ValueError(f"Malformed DELETE: {sql}")

    # Extract the target table and optional condition text.
    table = m.group(1)
    where_clause = m.group(2)

    # Return the structured DeleteStmt AST node.
    return DeleteStmt(
        table=table,
        where=_parse_where(where_clause) if where_clause else None,
    )
```

`_parse_update(sql: str) -> UpdateStmt` The function `_parse_update` parses an `UPDATE` statement into an `UpdateStmt` AST node. It supports the form `UPDATE table SET assignments [WHERE condition]`. The function first uses a regular expression to separate the table name, the assignment list, and the optional `WHERE` clause. Then it splits the assignment list on commas, parses each assignment into an `Assignment` object, and converts the right-hand-side value using `_parse_literal`. If a `WHERE` clause exists, it is processed by `_parse_where`. The final result is an `UpdateStmt` whose fields clearly describe which table is being updated, which columns receive new values, and which rows are affected by the update.

```
def _parse_update(sql: str) -> UpdateStmt:
    # Match the supported UPDATE form:
    #   UPDATE table_name SET col1 = val1, col2 = val2 [WHERE ...]
    #
    # group(1) -> table name
    # group(2) -> assignment list
    # group(3) -> optional WHERE clause
    m = re.match(r"UPDATE\s+(\w+)\s+SET\s+(.+?)(?:\s+WHERE\s+(.+))?$", sql, re.IGNORECASE)
    if not m:
        raise ValueError("Malformed UPDATE")

    # Extract the main parts of the UPDATE statement.
    table = m.group(1)
    set_part = m.group(2)
    where_clause = m.group(3)

    assignments = []

    # Split the SET clause by commas, so each item looks like:
    #   column = value
    for assign in set_part.split(","):
        # Split only on the first '=' in case the value contains '=' later.
        c, v = assign.split("=", 1)

        # Build an Assignment AST node with:
        #   left side  -> column name
        #   right side -> parsed literal value
        assignments.append(Assignment(c.strip(), _parse_literal(v.strip())))

    # Return the structured UpdateStmt AST node.
    return UpdateStmt(
        table=table,
        assignments=assignments,
        where=_parse_where(where_clause) if where_clause else None,
    )
```

10.3 Relational Algebra

In SQL, we usually write declarative queries: we specify what result we want, but we do not explicitly describe how the database system should compute it. This is convenient for users because SQL is relatively easy to write and read.

Relational algebra plays a different role. It is a more procedural way to describe a query: **an expression explicitly states which operators are applied and in what order.** Because of this, relational algebra is extremely useful for:

- understanding query meaning precisely,
- reasoning about equivalent query plans,
- studying query optimization,

- connecting SQL to the internal execution strategies used by a DBMS.

A relational algebra expression is built by combining operators. Each operator takes one or more relations as input and produces another relation as output.

10.3.1 Relations are sets of tuples

In classical relational algebra, a relation is a set of tuples, therefore:

- duplicate tuples are not allowed in a relation,
- operators that would otherwise create duplicates automatically eliminate them.

This is an important difference from SQL. In SQL, duplicate rows are generally preserved unless we explicitly write `DISTINCT`. In relational algebra, duplicates are removed automatically because the output must still be a relation.

Relations are sets, so there is no concept of tuple order. If two query results contain exactly the same tuples, they are considered the same relation even if displayed in a different order.

Suppose we have a relation `dogs` with schema:

$$dogs(name, age)$$

and contents:

name	age
Scooby	10
Buster	15
Buster	20

The projection expression

$$\pi_{name}(dogs)$$

keeps only the `name` column. The result is:

name
Scooby
Buster

Notice that although the two tuples $(Buster, 15)$ and $(Buster, 20)$ are distinct in the original relation, after projecting only the `name` attribute they both become $(Buster)$, so only one copy remains in the result.

10.3.2 Projection Operator π

The projection operator selects only certain attributes from a relation.

$$\pi_{A_1, A_2, \ldots, A_k}(R)$$

Here, R is a relation, and the output contains only attributes $A_1, A_2, \ldots, A_k$. The output schema is exactly the list of projected attributes, in the order written.

Projection corresponds roughly to the `SELECT` list in SQL.

For example, the SQL query

```sql
SELECT name
FROM dogs;
```

can be written in relational algebra as

$$\pi_{name}(dogs)$$

Because relational algebra uses set semantics, projection removes duplicates automatically. In SQL, the corresponding behavior would be

```sql
SELECT DISTINCT name
FROM dogs;
```

rather than plain `SELECT name FROM dogs;`.

10.3.3 Selection Operator σ

The selection operator filters tuples according to a predicate.

$$\sigma_{\theta}(R)$$

where θ is a condition such as $age = 12$, $name =' Timmy'$, or a compound condition.

Selection does not change the attributes. The output schema is the same as the input schema. Selection corresponds roughly to the `WHERE` clause in SQL.

For example, the SQL query

```sql
SELECT name, age
FROM dogs
WHERE age = 12;
```

can be written as

$$\pi_{name,age}(\sigma_{age=12}(dogs))$$

Another equivalent expression is

$$\sigma_{age=12}(\pi_{name,age}(dogs))$$

provided that the projection still contains all attributes needed later. But Selection is often pushed as low as possible in a query tree, because filtering early reduces the size of intermediate results. So the first expression is more efficient.

Compound predicates Relational algebra supports compound conditions:

- $\wedge$ means logical AND,
- $\vee$ means logical OR,
- $\neg$ means logical NOT.

For example,

```sql
SELECT name, age
FROM dogs
WHERE age = 12 AND name = 'Timmy';
```

becomes

$$\pi_{name,age}(\sigma_{age=12 \wedge name='Timmy'}(dogs))$$

10.3.4 Union $\cup$

Two relations are union-compatible if:

- they have the same number of attributes,
- corresponding attributes have compatible domains/types,
- in many presentations, the corresponding attributes are also expected to represent the same meaning.

$$R \cup S$$

returns all tuples that are in R, or in S, or in both.

Suppose:

$$dogs(name, age)$$

contains

name	age
Scooby	10
Buster	15
Garfield	20

and

$$cats(name, age)$$

contains

name	age
Tom	8
Garfield	10

Then

$$\pi_{name}(dogs) \cup \pi_{name}(cats)$$

returns

name
Scooby
Buster
Garfield
Tom

This corresponds to SQL `UNION`, not `UNION ALL`.

10.3.5 Set Difference −

$$R - S$$

returns the tuples that are in R but not in S.

$$\pi_{name}(dogs) - \pi_{name}(cats)$$

returns

name
Scooby
Buster

because `Garfield` also appears in `cats`.

This corresponds to SQL `EXCEPT` (or `MINUS` in some systems).

10.3.6 Intersection ∩

$$R \cap S$$

returns only the tuples that appear in both R and S.

10.3.7 Example

$$\pi_{name}(dogs) \cap \pi_{name}(cats)$$

returns

name
Garfield

Intersection can be expressed using difference:

$$R \cap S = R - (R - S)$$

This is sometimes useful because some minimal relational algebra systems define fewer primitive operators.

10.3.8 Cross Product $\times$

The cross product combines every tuple of one relation with every tuple of another relation.

$$R \times S$$

If R has m tuples and S has n tuples, then $R \times S$ has $m \cdot n$ tuples.

Let

$$dogs(name, age)$$

and

$$parks(park, city)$$

with contents:

park	city
Golden Gate Park	San Francisco
Central Park	New York City

Then

$$dogs \times parks$$

produces one tuple for each dog–park pair. This corresponds to

```sql
SELECT *
FROM dogs, parks;
```

when no join condition is specified. Cross product by itself is usually too large to be useful directly, but it is conceptually important because joins can be viewed as:

$$\text{join} = \text{cross product} + \text{filtering}$$

10.3.9 Join Operator $\bowtie$

Theta Join A theta join joins two relations using a general predicate θ:

$$R \bowtie_\theta S$$

For example, to join `cats` and `dogs` on matching names:

$$cats \bowtie_{cats.name=dogs.name} dogs$$

A theta join can be expressed as:

$$R \bowtie_\theta S = \sigma_\theta(R \times S)$$

This identity is important because it shows that join is not fundamentally a mysterious new operation. It is a convenient shorthand for combining tuples and then keeping only those that satisfy the join condition.

Natural Join If we simply write

$$R \bowtie S$$

this usually means a natural join. A natural join matches all attributes with the same name and keeps only one copy of each matching attribute in the result.

For example:

$$cats \bowtie dogs$$

would join on all common attribute names between the two relations. If R and S share common attributes $A_1, A_2, \ldots, A_n$, then the natural join is equivalent to selecting tuples from $R \times S$ where all shared attributes are equal, and then removing duplicate copies of those shared attributes.

Conceptually:

$$R \bowtie S = \pi_{\text{desired attrs}}\Big(\sigma_{R.A_1=S.A_1 \wedge \cdots \wedge R.A_n=S.A_n}(R \times S)\Big)$$

Outer joins Standard relational algebra usually focuses on inner join. Left outer join, right outer join, and full outer join are commonly treated as extended operators rather than core primitive operators.

10.3.10 Rename Operator ρ

Rename changes the name of a relation, one or more attributes, or both.

For example, to rename attribute `name` in `dogs` to `dname`:

$$\rho_{name \to dname}(dogs)$$

Then a join can be written as

$$cats \bowtie_{name=dname} \rho_{name \to dname}(dogs)$$

Rename is important when:

- two relations contain attributes with the same name but different meanings,
- we want to avoid ambiguity,
- we need self-joins.

10.3.11 Aggregation and Grouping γ

Aggregation and grouping are often represented by the operator γ. Different textbooks use slightly different notations, but the general idea is the same.

A grouping by age with an aggregate can be written as:

$$\gamma_{age,\ \text{COUNT}(*)}(dogs)$$

This means: group tuples by `age` and compute COUNT$(*)$ for each group.

The SQL query

```sql
SELECT age
FROM dogs
GROUP BY age
HAVING COUNT(*) > 5;
```

may be written in one common notation as

$$\pi_{\text{age, sumWeight}}\left(\sigma_{\text{cnt}>5}\left(\gamma_{\text{age; SUM(weight)}\to\text{sumWeight, COUNT}(*)\to\text{cnt}}(\text{dogs})\right)\right)$$

10.3.12 Mapping SQL to Relational Algebra

A rough correspondence is:

SQL construct	Relational algebra idea
`FROM R`	relation R
`SELECT cols`	projection π
`WHERE cond`	selection σ
`UNION`	union $\cup$
`EXCEPT`	difference $-$
`INTERSECT`	intersection $\cap$
`JOIN ... ON ...`	theta join $\bowtie_\theta$
`GROUP BY`	grouping γ
`AS` alias	rename ρ

These equivalences are important because a DBMS optimizer may transform one relational algebra expression into another equivalent one.

- Selection decomposition:
$$\sigma_{\theta_1 \wedge \theta_2}(R) = \sigma_{\theta_1}(\sigma_{\theta_2}(R))$$
- Selection commutativity:
$$\sigma_{\theta_1}(\sigma_{\theta_2}(R)) = \sigma_{\theta_2}(\sigma_{\theta_1}(R))$$
- Projection idempotence:
$$\pi_{L_1}(\pi_{L_2}(R)) = \pi_{L_1}(R) \qquad \text{if } L_1 \subseteq L_2$$
- Selection pushdown through join:

  If θ refers only to attributes of R, then

  $$\sigma_\theta(R \bowtie S) = \sigma_\theta(R) \bowtie S$$

  This is very useful because it reduces the size of the join input.
- Join as derived operator:

$$R \bowtie_\theta S = \sigma_\theta(R \times S)$$

Why multiple equivalent expressions matter?

Consider:

$$\pi_{name,age}(\sigma_{age=12}(dogs))$$

and

$$\sigma_{age=12}(\pi_{name,age}(dogs))$$

Both may produce the same final result in a suitable setting, but the first is usually better because selection reduces the number of tuples earlier. This illustrates the foundation of query optimization: two expressions can be logically equivalent but have very different execution costs.

10.3.13 Our Implementation

`RelNode` `RelNode` is the abstract base class for all logical query-plan nodes in this module. It does not store any fields or implement behavior by itself; instead, it serves as a common parent type so that the rest of the system can treat all logical operators uniformly. In other words, `Scan`, `Select`, `Project`, `Join`, and `Aggregate` are all specific kinds of `RelNode`. This design is useful because the optimizer and other planning components can work with a general relational-algebra node interface without needing separate handling logic for every class at the type level.

```
@dataclass
class RelNode: ...
```

`Scan` `Scan` represents the logical operation of reading tuples from a table. It is usually the starting point of a logical query plan, because most queries begin with some base relation from the `FROM` clause. The class stores only one field, `table`, which identifies the name of the relation being accessed. At the logical level, `Scan` simply means "read this table," without yet deciding how the table will actually be accessed. Later, the optimizer may keep it as a regular table scan or replace it with a more efficient physical access path such as an index-based scan.

```
@dataclass
class Scan(RelNode):
    table: str
```

`Select` `Select` represents logical filtering, corresponding to the `WHERE` clause in SQL. It takes an input relational node, stored in `child`, and a predicate function, stored in `pred`, and conceptually returns only those rows from the child node that satisfy the predicate. This means `Select` does not create new rows or change the table structure; instead, it reduces the set of rows that continue through the plan. In your implementation, the predicate is a callable that accepts a row and returns `True` or `False`, and it may also carry extra metadata that helps the optimizer detect equality predicates or range predicates for possible index usage.

```
@dataclass
class Select(RelNode):
    child: RelNode
    pred: Callable[[Row], bool]
```

`Project` `Project` represents logical projection, which corresponds to choosing specific columns in the `SELECT` list of a query. It stores a child relational node and a list of column names in `cols`. Conceptually, this operator takes each row produced by the child node and removes any attributes that are not listed in the projection. Unlike `Select`, which filters rows, `Project` keeps the same number of qualifying rows but changes the shape of each row by limiting which columns are returned. This makes `Project` the logical representation of column selection in relational algebra.

```
@dataclass
class Project(RelNode):
    child: RelNode
    cols: List[str]
```

`Join` `Join` represents the logical operation of combining rows from two input relations based on a join condition. It stores the left and right input nodes, the join key pair `on`, and a `kind` field that indicates a preferred implementation style such as `"hash"` or `"nested"`. The `on` field records which column from the left side should match which column from the right side, such as joining `Student.student_id` with `Enrollment.student_id`. Logically, `Join` expresses the idea that rows from two relations should be matched and merged when their join attributes agree. Although it already stores a simple implementation hint, it still belongs to the logical plan layer, since the final choice of concrete physical operator is made later by the optimizer or executor.

```
@dataclass
class Join(RelNode):
    left: RelNode
    right: RelNode
    on: Tuple[str, str]  # (left_col, right_col)
    kind: str = "hash"   # 'hash' or 'nested'
```

`Aggregate` `Aggregate` represents logical aggregation, used for queries involving functions such as `COUNT`, `SUM`, `AVG`, `MIN`, or `MAX`. It stores the input relation in `child`, the aggregate function name in `func`, and the target column in `col`.

Conceptually, this operator consumes a set of rows and produces a summarized result rather than returning each row individually. For example, `COUNT(*)` computes the total number of input rows, while `AVG(gpa)` computes the average value of a specific column. Unlike the other classes in the file, `Aggregate` is written as a regular class with an explicit constructor instead of as a dataclass, but its role in the logical query plan is the same: it captures the meaning of an aggregate query before physical execution decisions are made.

```
class Aggregate:
    def __init__(self, child, func, col):
        self.child = child
        self.func = func
        self.col = col
```

`DMLPlan` `DMLPlan` is the base class for all data-modification plans in this module. Unlike relational-algebra nodes, which describe the logical structure of queries such as `SELECT`, the classes derived from `DMLPlan` represent statements that modify the database state, such as `INSERT`, `DELETE`, and `UPDATE`. The class itself is intentionally minimal and contains no fields or behavior; its main purpose is to provide a common parent type so that the rest of the system can recognize that `InsertPlan`, `DeletePlan`, and `UpdatePlan` all belong to the same category of executable DML plan objects.

```
class DMLPlan:
    # Base class for all Data Manipulation Language (DML) plans.
    #
    # This class does not define fields or methods.
    # Its role is mainly organizational: it tells the system that
    # InsertPlan, DeletePlan, and UpdatePlan are all kinds of DML plans.
    #
    # That is useful because other parts of the code can test:
    #   isinstance(plan, DMLPlan)
    # instead of testing each subclass separately.
    pass
```

`InsertPlan` `InsertPlan` represents an executable plan for an `INSERT` statement. It stores the name of the target table in `table` and the new row to be inserted in `row`, where the row is represented as a Python dictionary mapping column names to values. In other words, this class provides a direct and concrete description of what data should be inserted and where it should go. Since insertion does not require scanning existing tuples to locate qualifying rows, the plan is much simpler than a query plan or delete plan: it only needs the destination relation and the tuple contents.

```
@dataclass
class InsertPlan(DMLPlan):
    # Name of the table that will receive the new row.
    #
    # Example:
    #   INSERT INTO Student (...)
    # would produce:
    #   table = "Student"
    table: str

    # The row to be inserted, represented as:
    #   column_name -> value
    #
    # Example:
    #   {
    #       "student_id": 1001,
    #       "name": "Alice",
    #       "gpa": 3.8
    #   }
    #
    # This dictionary is usually built during lowering by pairing
    # the column list and the VALUES list from the SQL statement.
    row: Dict[str, Any]
```

`DeletePlan` `DeletePlan` represents an executable plan for a `DELETE` statement. It stores the target table in `table` and a logical relational-algebra plan in `logical_plan`. The logical plan identifies which rows should be deleted, typically by starting with a scan of the table and optionally applying a selection predicate if there is a `WHERE` clause. This design is important because deleting rows is not just about naming a table; the system must also know which tuples qualify

for deletion. By embedding a relational-algebra plan inside the delete plan, the implementation can reuse the same logical filtering machinery that is already used for query processing.

```
@dataclass
class DeletePlan(DMLPlan):
    # Name of the table from which rows will be deleted.
    #
    # Example:
    #   DELETE FROM Student WHERE gpa < 2.0
    # would produce:
    #   table = "Student"
    table: str

    # A logical relational-algebra plan describing which rows
    # should be removed.
    #
    # Typical example:
    #   Select(
    #       child=Scan("Student"),
    #       pred=<predicate function>
    #   )
    #
    # So DeletePlan does not directly list row IDs here.
    # Instead, it stores a logical plan that can be executed to
    # find the matching rows first.
    logical_plan: relalg.RelNode
```

UpdatePlan `UpdatePlan` represents an executable plan for an `UPDATE` statement. It stores the target table name, a dictionary of assignments describing which columns should receive new values, and an optional `where_expr` field containing the original AST expression for the `WHERE` clause. The `assignments` dictionary summarizes the `SET` clause in a compact form, while `where_expr` preserves the condition that identifies which rows should be modified. This makes `UpdatePlan` a hybrid structure: it is direct like `InsertPlan` in its handling of new values, but it also needs conditional logic like `DeletePlan` because updates typically affect only rows that satisfy a predicate.

```
@dataclass
class UpdatePlan(DMLPlan):
    # Name of the table whose rows will be updated.
    #
    # Example:
    #   UPDATE Student SET gpa = 4.0 WHERE student_id = 1001
    # would produce:
    #   table = "Student"
    table: str

    # Dictionary describing the new values to assign.
    #
    # Example:
    #   SET gpa = 4.0, major = 'CS'
    # becomes:
    #   {
    #       "gpa": 4.0,
    #       "major": "CS"
    #   }
    #
    # This is a compact representation of the SET clause.
    assignments: Dict[str, Any]

    # The WHERE condition kept as an AST expression object.
    #
    # Why keep the AST here?
    # Because later code may still need to interpret or lower it
    # to determine which rows should be updated.
    #
    # If there is no WHERE clause, this field is None,
    # which usually means all rows are eligible for update.
    where_expr: Optional[object] = None   # keep AST Expr here
```

10.4 Lowering Phase

The lowering phase is the bridge between the **AST** and the **logical execution representation** used by the DBMS. In our system, parsing and AST construction answer the question:

> "What does the SQL statement mean structurally?"

The lowering phase answers the next question:

> "How should that AST be converted into the internal plan objects that the optimizer and executor understand?"

Therefore, the lowering phase is the stage:

$$\text{AST} \longrightarrow \text{Relational-algebra-style plan / DML plan}$$

The AST is a clean, structured representation of SQL syntax. However, the executor does not usually run the AST directly. Instead, we translate the AST into a lower-level internal form, such as:

- `relalg.Scan`
- `relalg.Select`
- `relalg.Project`
- `relalg.Aggregate`
- `InsertPlan`
- `DeletePlan`
- `UpdatePlan`

This translation is called **lowering**. The lowering phase is useful because it separates the system into clean stages:

$$\text{SQL text} \rightarrow \text{Parser} \rightarrow \text{AST} \rightarrow \text{Lowering} \rightarrow \text{Logical plan / DML plan} \rightarrow \text{Optimizer / Executor}$$

This separation is important for several reasons:

- the parser only focuses on syntax and AST structure,
- the lowering phase only focuses on internal plan construction,
- the optimizer can inspect plan objects instead of raw SQL,
- the executor can run a standard plan representation.

10.4.1 `lower_statement(stmt: Statement)`

The function `lower_statement` is the main entry point of the lowering phase. Its job is to inspect the type of the AST node stored in `stmt` and then dispatch the work to the correct lowering logic. This makes the function a dispatcher: it does not perform all of the lowering itself, but instead decides which specialized path should be used for each kind of SQL statement. The input is a general `Statement` object, and the output is either a relational algebra plan or a DML plan object, depending on the statement type.

```
def lower_statement(stmt: Statement):
    """
    Convert a parsed SQL statement (AST) into an internal plan.

    This function is the top-level dispatcher for the lowering phase.
    Depending on the concrete statement type, it will:

    - lower a SELECT into relational algebra
    - convert an INSERT into an InsertPlan
    - convert a DELETE into a DeletePlan
    - convert an UPDATE into an UpdatePlan
    """
    if isinstance(stmt, SelectStmt):
        # SELECT statements are lowered into relational algebra trees.
        return lower_select(stmt)

    if isinstance(stmt, InsertStmt):
        # Pair each listed column with its corresponding value.
        # Example:
        #   INSERT INTO Student(student_id, name) VALUES (1001, 'Alice')
        # becomes:
        #   {'student_id': 1001, 'name': 'Alice'}
        row = dict(zip(stmt.columns, stmt.values))
        return InsertPlan(stmt.table, row)

    if isinstance(stmt, DeleteStmt):
        # DELETE starts as a scan of the target table.
```

```
        node = relalg.Scan(stmt.table)

        # If there is a WHERE clause, lower it into an executable predicate
        # and wrap the scan in a Select node.
        if stmt.where is not None:
            pred = lower_predicate(stmt.where)
            node = relalg.Select(node, pred)

        # The DeletePlan carries the table name plus the logical node
        # that identifies which rows should be deleted.
        return DeletePlan(stmt.table, node)

    if isinstance(stmt, UpdateStmt):
        # Convert assignment AST nodes into a dictionary:
        #   SET col1 = v1, col2 = v2
        # becomes:
        #   {'col1': v1, 'col2': v2}
        assigns = {a.column: a.value for a in stmt.assignments}

        # The WHERE clause is passed through as-is here.
        # Presumably a later stage or the executor will interpret it.
        return UpdatePlan(stmt.table, assigns, stmt.where)

    # Any unsupported statement type is rejected explicitly.
    raise TypeError(f"Unsupported statement type: {type(stmt)}")
```

10.4.2 `lower_select(stmt: SelectStmt)`

The function `lower_select` handles the lowering of `SELECT` statements into relational algebra nodes. Its job is to take a parsed `SelectStmt` AST and build a relational algebra tree that represents how the query should be logically evaluated. The function begins with the most basic operation in relational query processing: scanning the source table. It creates a scan node using `relalg.Scan(stmt.from_table)` and stores it in `node`. This means that every `SELECT` query starts from the idea of reading tuples from the table named in the `FROM` clause.

```
def lower_select(stmt: SelectStmt):
    """
    Lower a parsed SELECT statement into a relational algebra tree.

    Typical flow:
    1. Start with a Scan(from_table)
    2. Add a Select(...) if there is a WHERE clause
    3. Add either:
       - nothing      for SELECT *
       - Project(...) for column selection
       - Aggregate(...) for aggregate queries
    """
    # Every SELECT begins by scanning the source table.
    node = relalg.Scan(stmt.from_table)

    # Apply WHERE filtering if present.
    if stmt.where is not None:
        pred = lower_predicate(stmt.where)
        node = relalg.Select(node, pred)

    # Handle the case where the SELECT list has multiple items.
    # In this implementation, multiple items must all be simple columns.
    if len(stmt.select_items) != 1:
        cols = []
        for item in stmt.select_items:
            if not isinstance(item, ColumnItem):
                # Mixed SELECT lists such as:
                #   SELECT name, COUNT(*)
                # are not supported here.
                raise ValueError("Mixed or unsupported SELECT list")
            cols.append(item.name)

        # Build a projection over the selected columns.
        return relalg.Project(node, cols)

    # From here on, there is exactly one SELECT item.
    item = stmt.select_items[0]

    if isinstance(item, Star):
        # SELECT * means return all columns,
        # so no projection node is needed.
        return node
```

```
    if isinstance(item, AggregateItem):
        # Aggregate query such as:
        #   SELECT COUNT(*)
        #   SELECT AVG(gpa)
        return relalg.Aggregate(node, item.func, item.arg)

    if isinstance(item, ColumnItem):
        # Single-column projection such as:
        #   SELECT name FROM Student
        return relalg.Project(node, [item.name])

    # Any other SELECT item type is unsupported.
    raise TypeError(f"Unsupported SELECT item: {type(item)}")
```

10.4.3 `lower_predicate(expr: Expr)`

The function `lower_predicate` is responsible for converting a predicate AST into an executable Python function that can be applied to a row. At the same time, it attaches extra metadata to that function so that the optimizer can inspect the predicate and potentially choose an index-based access path. This dual purpose is very important. The executor needs a real function that returns `True` or `False` for each row, but the optimizer also needs structured information such as "this is an equality predicate on column `student_id`" or "this is a range predicate on column `gpa` between two bounds." The docstring of the function explicitly states this design goal: convert an AST expression to an executable predicate while still attaching optimizer metadata.

```
def lower_predicate(expr: Expr):
    """
    Convert an expression AST into an executable predicate(row) function.

    This function also attaches metadata attributes to the resulting
    predicate function so that optimizer.py can inspect the predicate and
    decide whether an index can be used.

    Supported patterns:
    - column = literal
    - column >= literal
    - column <= literal
    - conjunctions (AND) of the above

    Notes:
    - Only ColumnRef op Literal is supported
    - Only '=', '>=', '<=' are supported
    - The result is a Python function that accepts a row dictionary
    """

    # These variables capture the recognized predicate structure.
    # Equality case:
    #   eq_col = column name
    #   eq_val = literal value
    eq_col = None
    eq_val = None

    # Range case:
    #   range_col = column name
    #   lo        = lower bound from >=
    #   hi        = upper bound from <=
    range_col = None
    lo = None
    hi = None

    def collect(e: Expr):
        ......

    # First, analyze the AST and extract comparison metadata.
    collect(expr)

    def predicate(row):
    ... ...

    # Attach optimizer metadata for equality lookup.
    # Example use: optimizer can inspect predicate._eq_col
    # and predicate._eq_val to decide whether a hash index applies.
    if eq_col is not None:
        predicate._eq_col = eq_col
        predicate._eq_val = eq_val

```

```
    # Attach optimizer metadata for range lookup.
    # Example use: optimizer can inspect predicate._range_col,
    # _range_lo, _range_hi to decide whether a B+ tree range scan applies.
    if range_col is not None:
        predicate._range_col = range_col
        predicate._range_lo = lo
        predicate._range_hi = hi

    return predicate
```

10.4.4 `collect(e: Expr)`

The nested function `collect` is defined inside `lower_predicate`. Its purpose is to recursively inspect the AST of a predicate expression and extract the parts of the condition that this educational database currently supports. The function begins with a `nonlocal` declaration for `eq_col`, `eq_val`, `range_col`, `lo`, and `hi`. This means `collect` is allowed to modify the variables that were created in the surrounding `lower_predicate` function. Without `nonlocal`, assignments inside `collect` would create new local variables instead of updating the shared state needed by the later `predicate` function.

```
    def collect(e: Expr):
        """
        Recursively walk the AST predicate expression and extract
        supported comparison information into the outer variables above.

        This helper does not itself return a predicate function.
        Instead, it collects metadata that will later be used both by:
        - the executable predicate(row)
        - the optimizer hints attached to that predicate
        """
        nonlocal eq_col, eq_val, range_col, lo, hi

        # AND combines two predicates, so recursively collect both sides.
        if isinstance(e, And):
            collect(e.left)
            collect(e.right)
            return

        # Every non-AND predicate must be a Compare node.
        if not isinstance(e, Compare):
            raise ValueError(f"Unsupported predicate node: {type(e)}")

        # Only support predicates of the form:
        #   column op literal
        # Example:
        #   gpa >= 3.5
        if not isinstance(e.left, ColumnRef) or not isinstance(e.right, Literal):
            raise ValueError("Only ColumnRef op Literal is supported")

        col = e.left.name
        val = e.right.value

        # Equality predicate: col = val
        if e.op == "=":
            eq_col = col
            eq_val = val

        # Lower bound: col >= val
        elif e.op == ">=":
            range_col = col
            lo = val

        # Upper bound: col <= val
        elif e.op == "<=":
            range_col = col
            hi = val

        else:
            # Other operators such as >, <, != are not yet supported.
            raise ValueError(f"Unsupported comparison operator: {e.op}")
```

10.4.5 `predicate(row)`

The nested function `predicate` is the executable result produced by `lower_predicate`. It closes over the values extracted by `collect` and uses them to evaluate whether a given row satisfies the predicate. The row is expected to

behave like a dictionary, since the function calls `row.get(...)` to retrieve column values. This design matches the common representation of decoded tuples as Python dictionaries in educational database implementations.

```
    def predicate(row):
        """
        Return True if the given row satisfies the lowered predicate.

        The row is expected to behave like a dictionary:
            row.get(column_name)
        """

        # Equality predicate takes priority if it was collected.
        if eq_col is not None:
            return row.get(eq_col) == eq_val

        # Otherwise evaluate the range predicate if present.
        if range_col is not None:
            val = row.get(range_col)

            # Reject rows below the lower bound.
            if lo is not None and val < lo:
                return False

            # Reject rows above the upper bound.
            if hi is not None and val > hi:
                return False

            # The row is within the allowed range.
            return True

        # If nothing was collected, treat it as always true.
        return True
```

10.5 Database Operators

In a database system, an operator is a small execution unit that performs one specific part of a query. Instead of executing a SQL query as one giant monolithic procedure, a DBMS breaks the query into a sequence or tree of operators. Each operator is responsible for one relational task, such as scanning a table, filtering rows, projecting selected columns, joining two inputs, or computing an aggregate. This design is extremely important because it makes query execution modular, composable, and easier to optimize.

For example, the SQL query

```
SELECT name FROM Student WHERE major = 'CS';
```

can be viewed as a pipeline of operators:

$$\texttt{TableScan(Student)} \rightarrow \texttt{Selection(major = 'CS')} \rightarrow \texttt{Projection(name)}.$$

Each operator consumes tuples from its child operator and produces new tuples for its parent. In this sense, operators are the fundamental building blocks of query execution.

10.5.1 The Volcano Iterator Model

Our implementation follows the Volcano-style execution model. In this model, every operator supports three basic methods:

- `open()`: initialize the operator,
- `next()`: produce the next output row,
- `close()`: release resources.

This is sometimes called the iterator interface, because each operator behaves like an iterator over tuples. A parent operator repeatedly calls `next()` on its child until the child returns `None`, which means there are no more rows. This execution model is elegant because every operator follows the same interface, regardless of whether it performs scanning, filtering, joining, or aggregation.

Conceptually, the abstract base class can be described as:

```
class Op:
    def open(self) -> None: ...
    def next(self) -> Optional[Row]: ...
    def close(self) -> None: ...
```

The importance of this design is that it decouples what an operator does from how the entire query is executed. The optimizer can rearrange operators, swap one implementation for another, or build different execution trees, as long as each operator obeys the same interface.

Operators are important for several reasons. First, they provide a clean bridge between logical query plans and physical execution. Logical query processing describes what needs to happen, such as selection, projection, and join. Operators provide the physical machinery that actually performs these tasks. Second, they allow the optimizer to choose among multiple implementations. For example, a join can be carried out with a nested-loop join or a hash join. Third, they make the engine extensible: once a new operator class is added, it can be combined with the rest of the system using the same `open/next/close` protocol. In other words, operators are where abstract relational algebra becomes executable code.

The operators in this file can be grouped into several major categories:

1. **Access operators**: operators that retrieve tuples from storage, such as `TableScan`, `IndexScan`, and `RangeIndexScan`.
2. **Unary relational operators**: operators that transform one input stream, such as `Selection`, `Projection`, and `Aggregate`.
3. **Binary relational operators**: operators that combine two input streams, such as `NestedLoopJoin` and `HashJoin`.
4. **Special-purpose operators**: operators like `ConstantAggregate`, which return a fixed precomputed result.

This classification is helpful because it mirrors how DBMS textbooks present physical query processing.

10.5.2 Access Operators

TableScan A `TableScan` is the most basic access operator. Its job is to read every tuple in a table one by one. This corresponds to a full relation scan in database theory. It is simple and general: no matter what query you issue, the system can always fall back to scanning the entire table.

In this implementation, `TableScan.open()` initializes an iterator over the table by calling `self.table.scan()`. Then `next()` repeatedly pulls the next pair `(rid, row)` from that iterator. The operator packages the result into a dictionary that includes both the RID and the row contents:

```
return {"_rid": rid, **row}
```

This tells us an important design choice: the table scan does not merely return logical tuples; it returns tuples enriched with physical location information. That makes the scan operator useful not only for read-only queries, but also for operations that may later need to update or delete tuples.

The main advantage of `TableScan` is universality. It works even when no index exists. The main disadvantage is cost: it may read many irrelevant tuples. For a selective query such as

```
SELECT * FROM Student WHERE student_id = 1001
```

a full table scan is usually much more expensive than an index-based lookup.

IndexScan An `IndexScan` is an access operator for equality predicates. Instead of reading the whole table, it asks an index for the RIDs of tuples matching a given key. In this implementation, `open()` performs:

```
self._rids = list(self.index.probe(self.key))
```

So the index is probed once, and the matching RIDs are stored in memory. Then each call to `next()` fetches one RID from that list and resolves it into a full row using `_fetch()`.

The `_fetch()` method is particularly important because it shows how logical query execution meets physical storage management. Given a RID `(seg_id, pid, sid)`, the operator:

1. locates the correct segment file,
2. fetches the page through the buffer pool,
3. constructs a `SlottedPage`,
4. reads the slot payload,
5. decodes the payload with the schema,

6. unpins the page from the buffer pool.

This is a very realistic design. An index typically stores search keys and tuple references, not the full tuple itself. Therefore, after the index search finds candidate RIDs, the engine must still fetch the actual data records from storage. That is exactly what this operator models.

The benefit of `IndexScan` is efficiency for equality predicates. If the index is selective, the system can jump directly to the relevant tuples instead of scanning the full table. The cost, however, may still include many random page fetches if the matching tuples are scattered across the heap file.

RangeIndexScan A `RangeIndexScan` is similar to `IndexScan`, but it is designed for range predicates rather than exact equality. This type of operator is naturally associated with ordered indexes such as B^+-trees. Queries like

```
student_id >= 100010 AND student_id <= 100020
```

are classic examples where a range scan is appropriate.

In this implementation, `RangeIndexScan.open()` first normalizes the lower and upper bounds. If the query leaves one side open, the operator tries to infer the missing bound using `min_key()` or `max_key()` from the index. If those are unavailable, it falls back to a legacy `_keys` structure. If no valid bounds can be established, it returns an empty result. Otherwise, it materializes:

```
self._rids = list(self.index.range(lo, hi))
```

This design reveals two important ideas. First, a range scan depends on the index preserving order. That is why B^+-trees are suitable while hash indexes are not. Second, the operator still returns full tuples by reusing the tuple-fetching logic from `IndexScan`. In other words, the difference between equality scan and range scan is mainly in how the RIDs are found, not in how the underlying tuples are fetched.

10.5.3 Unary Operators

Selection A `Selection` operator filters tuples according to a predicate. In relational algebra, this corresponds to the selection operator

$$\sigma_{\text{predicate}}(R).$$

It takes one child operator as input and returns only those tuples that satisfy the predicate.

In this implementation, the constructor receives a child operator and a Python callable:

```
def __init__(self, child: Op, pred: Callable[[Row], bool]):
```

The `next()` method repeatedly asks the child for a tuple. If the child returns `None`, the selection is done. Otherwise, the predicate is applied. If the predicate is true, the tuple is returned; if not, the operator keeps looping until it finds a tuple that passes.

This is a classic example of lazy tuple-at-a-time execution. The selection operator does not preload all rows and then filter them in bulk. Instead, it processes one tuple at a time as requested by its parent. This matches the Volcano model very well.

For example, if the child is `TableScan(Student)` and the predicate is `major == "CS"`, then the operator emits only the student rows whose major is CS.

Projection A `Projection` operator keeps only selected columns from each input tuple. In relational algebra, this corresponds to

$$\pi_{\text{cols}}(R).$$

For example, if a child produces tuples with columns `student_id`, `name`, `major`, and `gpa`, then a projection on `[name, gpa]` will discard the other fields.

In this implementation, the operator constructs a new dictionary:

```
projected = {c: row.get(c) for c in self.cols}
```

Then it preserves `_rid` if present. That detail is important. Even though projection is logically about column reduction, the implementation chooses to keep the RID metadata alive. This is very practical in an educational DBMS because later stages may still need tuple identity.

Projection is often cheap in terms of logic, but it can still matter a lot for performance. Reducing the number of columns early can lower memory usage, reduce copying cost, and simplify later processing.

Aggregate The `Aggregate` operator computes a summary value from all tuples in its child stream. In SQL, this corresponds to functions such as `MIN`, `MAX`, `COUNT`, and `AVG`. Unlike selection and projection, which may produce many output rows, this operator usually produces a single output row.

In this implementation, `open()` fully consumes the child operator and stores the values needed for aggregation in a list. If the requested column is `"*"`, the operator appends `1` for each tuple, which supports `COUNT(*)`. Otherwise, it extracts the designated column value.

A particularly important feature is that the implementation follows SQL null-handling semantics:

> ignore `NULL` for aggregates except `COUNT(*)`.

This is a subtle but correct database behavior. For example, `AVG(gpa)` should ignore missing GPA values rather than treating them as zero.

After reading all tuples, the operator computes the requested function. Then `next()` returns the result exactly once in dictionary form, such as:

```
{"COUNT": 2000}
```

or

```
{"AVG": 3.41}
```

Notice that this operator is blocking: it cannot produce its result until it has consumed all of its child input. This is an important contrast with pipelined operators such as selection and projection.

ConstantAggregate The `ConstantAggregate` operator is a simplified special case that returns a precomputed aggregate value. It does not read from a child operator at all. Instead, it stores a function name and a value, and returns that result once.

This kind of operator can be useful when the optimizer already knows the answer, or when an aggregate has been computed through some shortcut. For example, if the system can answer `MIN` or `MAX` directly from an index root or boundary leaf, it may wrap the answer as a constant aggregate rather than scanning tuples.

Although simple, this operator highlights an important optimizer principle: sometimes the best execution plan is not to evaluate a normal operator tree at all, but to replace it with a cheaper equivalent result.

10.5.4 Binary Operators: Join Processing

Joins are among the most important and most expensive operations in a DBMS. A join combines rows from two input relations based on a matching condition. For example:

```sql
SELECT * FROM Student JOIN Enrollment ON Student.student_id = Enrollment.student_id;
```

Conceptually, a join operator has two children: a left input and a right input. Its job is to find pairs of tuples that satisfy the join condition and emit merged output rows.

The implementation includes two join algorithms:

- `NestedLoopJoin`
- `HashJoin`

These two operators compute the same relational result but use different physical strategies.

NestedLoopJoin A `NestedLoopJoin` is one of the most straightforward join algorithms. The basic idea is:

1. take one tuple from the left input,
2. compare it with tuples from the right input,
3. output each matching pair,
4. then move to the next left tuple.

In this implementation, the right input is fully materialized into a list during `open()`. Then the operator scans the left input one tuple at a time. For each left tuple, it iterates through the stored right tuples and checks whether the join attributes are equal. If so, it merges the two rows into one output dictionary.

An interesting detail is how duplicate column names are handled. If a right-side column name already exists in the output dictionary, it is renamed with the prefix `right.`. This avoids overwriting the left tuple's values. For example, if both relations contain `student_id`, the output may contain:

```
student_id
right.student_id
```

The nested-loop join is easy to understand and works for any join condition, but it can be expensive. If there are m tuples on the left and n tuples on the right, the naive comparison cost is roughly $m \times n$. It is therefore usually best for small relations or when no better join structure is available.

HashJoin A `HashJoin` is a much more efficient algorithm for equality joins when memory is sufficient. Its idea is to divide the work into two phases:

1. **Build phase**: read one input and store its tuples in a hash table keyed by the join attribute.
2. **Probe phase**: read the other input and use the hash table to find matching tuples quickly.

In this implementation, the left child is the build side. During `open()`, the operator reads all left tuples, extracts the join key, and stores them in:

```
1 self._hash.setdefault(key, []).append(lr)
```

Then it opens the right child and starts probing. For each right tuple, it computes the right join key, looks up matching left tuples in the hash table, and emits joined rows one by one.

This is usually much faster than nested loops for equality joins because each right tuple can jump directly to matching left tuples rather than comparing against every left tuple. Instead of repeated full comparisons, the algorithm relies on efficient hash lookup.

However, hash join also has limitations. It is naturally suited to equality predicates, not general range joins. It also requires enough memory to build the in-memory hash table, unless the system implements external or partitioned hashing.

Both join operators return the same logical result, but their physical behavior differs:

- **Nested-loop join** is simple and general, but may be slow.
- **Hash join** is usually faster for equality joins, but depends on hashing and memory.

This comparison is a classic example of why physical operators matter. SQL alone only says join these relations; it does not say how. The optimizer chooses the operator implementation that seems cheapest under the circumstances.

10.5.5 Pipelining, Blocking, and Execution Behavior

One of the most important lessons from these operators is that not all operators behave the same way with respect to tuple flow.

A pipelined operator can produce output tuples incrementally as it reads its input. In this file, `TableScan`, `Selection`, `Projection`, `IndexScan`, and `RangeIndexScan` are mostly pipelined. Their parent can repeatedly call `next()` and get results one row at a time. This is efficient because it avoids materializing the entire result before continuing. It also reduces memory usage and enables operator composition.

A blocking operator must consume all or much of its input before producing output. In this file:

- `Aggregate` is blocking because it must read all child rows before computing the final result.
- `NestedLoopJoin` materializes the entire right input before starting comparisons.
- `HashJoin` materializes the build side into a hash table before probing.

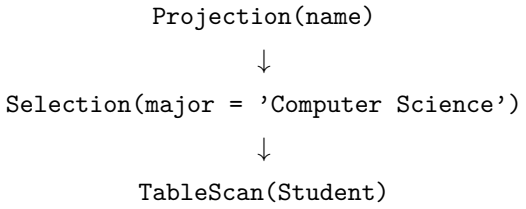
Blocking behavior is common in real DBMSs. Operators such as sorting, grouping, and some joins cannot always produce output immediately. Understanding which operators are blocking is important for reasoning about memory use and latency.

10.5.6 A Complete Example of Operator Composition

Suppose we want to execute:

```
SELECT name FROM Student WHERE major = 'Computer Science';
```

A possible operator tree is:



Execution proceeds as follows:

1. The top-level projection is opened.
2. Projection opens its child selection.
3. Selection opens its child table scan.
4. The parent repeatedly calls `next()` on projection.
5. Projection calls `next()` on selection.
6. Selection calls `next()` on table scan.
7. Table scan returns one tuple at a time.
8. Selection filters out non-matching tuples.
9. Projection keeps only the `name` column and returns the result upward.

This example illustrates the core beauty of the Volcano model: each operator only needs to know how to interact with its immediate child, yet the whole query plan works as one coordinated pipeline.

10.6 Join Algorithms

A join combines two relations by matching tuples that satisfy a join condition. If we join relation R and relation S on condition θ, then for each tuple $r_i \in R$, we look for tuples $s_j \in S$ such that $\theta(r_i, s_j)$ is true, and we output the concatenated tuple $\langle r_i, s_j \rangle$.

In our course database, a common example is:

```
SELECT *
FROM Student
INNER JOIN Enrollment
ON Student.student_id = Enrollment.student_id;
```

Here, `Student` and `Enrollment` are joined on `student_id`. Every join algorithm below computes the same logical result, but they do so with very different I/O behavior.

Throughout this section, we use the following standard notation:

- R: the outer relation
- S: the inner relation
- $[R]$: the number of pages in relation R
- $[S]$: the number of pages in relation S
- $|R|$: the number of records in relation R
- B: the number of buffer pages available

When interpreting the examples, you can think of:

$$R = \texttt{Student}, \qquad S = \texttt{Enrollment}$$

unless otherwise stated.

The join algorithms can be grouped as follows:

- **Loop-based joins:** SNLJ, PNLJ, BNLJ, INLJ
- **Hash-based joins:** Naive Hash Join, Grace Hash Join
- **Order-based joins:** Sort-Merge Join

Loop joins repeatedly compare one side against the other. Hash joins use hashing to group possible matches together. Sort-merge join uses sorted order and synchronized scanning.

10.6.1 Simple Nested Loop Join (SNLJ)

Simple Nested Loop Join is the most direct join algorithm. For each record in R, we scan every record in S looking for matches. Conceptually, it is just two nested loops over tuples.

In the `Student`–`Enrollment` example, this means:

> For each student tuple, scan all enrollment tuples and output the ones with the same `student_id`.

```
for each record r_i in R :
  for each record s_j in S :
if θ(r_i, s_j) is true, output ⟨r_i, s_j⟩
```

The I/O cost is:

$$[R] + |R|[S]$$

The intuition is:

- we read all pages of R once;
- for each record in R, we scan all pages of S.

SNLJ is easy to understand, but it is usually a poor choice because it rescans S once for every tuple in R. It is mainly useful as a conceptual baseline.

10.6.2 Page Nested Loop Join (PNLJ)

Page Nested Loop Join improves on SNLJ by using pages rather than individual tuples as the outer loop. Instead of rescanning all of S once for every tuple of R, it rescans all of S once for every page of R.

So in the college database setting:

> For each page of `Student`, scan all pages of `Enrollment`, and compare tuples inside those two pages.

```
for each page p_r in R :
  for each page p_s in S :
   for each record r_i in p_r :
     for each record s_j in p_s :
   if θ(r_i, s_j) is true, output ⟨r_i, s_j⟩
```

The I/O cost is:

$$[R] + [R][S]$$

This is better than SNLJ because $[R]$ is usually much smaller than $|R|$.

PNLJ is better than SNLJ, but it still rescans the full inner relation many times. It is a useful stepping stone toward more efficient loop joins.

10.6.3 Block Nested Loop Join (BNLJ)

Block Nested Loop Join uses the buffer pool more effectively. Instead of keeping only one page of R in memory, it keeps a whole block of $B-2$ pages from R. One page is reserved for scanning S, and one page is reserved for output.

The key insight is:

> The larger the outer block from R, the fewer times we need to rescan S.

```
for each block of (B − 2) pages B_r in R :
    for each page p_s in S :
   for each record r_i in B_r :
     for each record s_j in p_s :
   if θ(r_i, s_j) is true, output ⟨r_i, s_j⟩
```

The I/O cost is:

$$[R] + \left\lceil \frac{[R]}{B-2} \right\rceil [S]$$

This is often much better than PNLJ because the number of times we scan S is reduced from $[R]$ times to

$$\left\lceil \frac{[R]}{B-2} \right\rceil$$

times.

10.6.4 Index Nested Loop Join (INLJ)

Index Nested Loop Join is useful when the inner relation S has an index on the join attribute. Then, instead of scanning all of S for each tuple in R, we use the index to directly look up matching tuples in S.

For example, if `Enrollment.student_id` has an index, then for each student tuple we can probe the index to find enrollments with the same `student_id`.

```
for each record r_i in R:
    probe the index on S using the join key of r_i
    for each matching record s_j in S:
        output ⟨r_i, s_j⟩
```

The generic cost formula is:

$$[R] + |R| \cdot (\text{cost to look up matching records in } S)$$

The lookup cost depends on:

- the index type;
- whether the index is clustered or unclustered;
- how many matching tuples exist.

INLJ can be excellent when:

- the outer relation is not too large, and
- the inner index makes lookups cheap.

It is especially attractive when the join attribute is indexed and the join is selective.

10.6.5 Naive Hash Join

If the smaller relation R fits in memory, we can build an in-memory hash table on R using the join key, then scan S once and probe the hash table for matches. This is called Naive Hash Join.

This works only when the build relation fits in memory, specifically when:

$$[R] \leq B - 2$$

1. Read all pages of R into memory.
2. Build a hash table on the join key.
3. Scan all pages of S once.
4. For each tuple in S, probe the hash table and output any matches.

The I/O cost is:

$$[R] + [S]$$

This is extremely attractive, but only feasible if the build side fits in memory. Naive Hash Join is very efficient for equijoins when the smaller relation fits in memory. Its main limitation is the memory requirement.

10.6.6 Grace Hash Join (GHJ)

Naive Hash Join fails when the build relation does not fit in memory. Grace Hash Join fixes this by partitioning both relations into smaller buckets so that corresponding partitions can be joined one pair at a time.

The basic idea is:

- partition R and S using the same hash function;
- matching tuples must end up in the same partition pair;
- for each partition pair, build a smaller in-memory hash table and perform a local hash join.

1. Partition R into $R_1, R_2, \ldots$.
2. Partition S into $S_1, S_2, \ldots$ using the same partitioning rule.
3. For each pair (R_i, S_i):
    - if one side now fits in memory, perform a Naive Hash Join;
    - if neither side fits, partition again recursively.

The cost is:

$$\text{cost of partitioning} + \text{cost of Naive Hash Join on the partition pairs}$$

So the exact cost depends on:

- how many partitioning passes are required;
- whether the hash distribution is uniform.

Grace Hash Join is sensitive to key skew. If too many tuples hash to the same partition, then that partition may still be too large to fit in memory, forcing more partitioning or causing poor performance.

GHJ is a strong algorithm for large equijoins when neither relation fits in memory. But it works best when hash partitioning is reasonably balanced.

10.6.7 Sort-Merge Join (SMJ)

Sort-Merge Join is useful when the relations are already sorted on the join key, or when sorting is acceptable and we want the result in sorted order. The algorithm sorts both inputs, then scans them together with pointers.

It is especially attractive when a later operator benefits from sorted output.

Suppose R and S are sorted on the join key.

1. Start at the beginning of both relations.
2. If the current key in R is smaller, advance R.
3. If the current key in S is smaller, advance S.
4. When keys match, mark the matching position in S, output matching pairs, and continue.
5. When the block of matches in S ends, reset S to the mark and advance R.

The average I/O cost is:

$$\text{sort cost of } R + \text{sort cost of } S + ([R] + [S])$$

The worst-case cost is:

$$\text{sort cost of } R + \text{sort cost of } S + ([R] + |R|[S])$$

The worst case happens when many tuples match many tuples, causing repeated rescans of matching ranges.

SMJ has two major advantages:

- it works well for equijoins when sorting is acceptable;
- it can produce output sorted on the join key, which may be useful later.

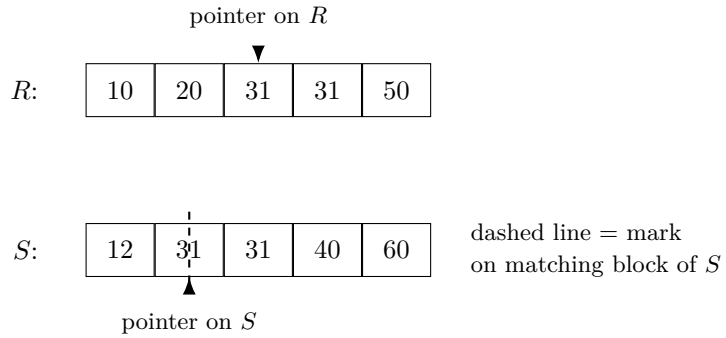


Figure 13: Sort-Merge Join: both relations are sorted, pointers advance until a match is found, and a mark is used on the matching block of S.

10.6.8 How to Choose Among Them

A useful practical summary is:

- Use **BNLJ** when no index exists and a loop join is required.
- Use **INLJ** when the inner side has a useful index on the join key.
- Use **Naive Hash Join** when one side fits in memory.
- Use **Grace Hash Join** for large equijoins when inputs do not fit in memory.
- Use **SMJ** when sorted order is useful later or sorting is already available.

10.7 The Query Optimizer

A query optimizer is the component of a database system that decides how a SQL query should be executed. SQL is a declarative language: the user specifies the result they want, but not the exact algorithm that should be used to produce it. The optimizer fills in this gap. It examines the query, considers different execution strategies, and chooses a physical plan that is expected to be efficient.

For example, suppose we ask the database to find all students whose GPA is greater than 3.5, or to join `Student` with `Enrollment`. There are many possible ways to answer such a query. The system could scan the entire table, use an index, perform a nested-loop join, use a hash join, or sort both relations and perform a sort-merge join. All of these strategies may be logically correct, but they can differ enormously in cost. The optimizer exists to choose among these alternatives.

A typical database pipeline can be described as:

$$\text{SQL} \to \text{Parser} \to \text{AST} \to \text{Lowering} \to \text{Logical Plan} \to \text{Optimizer} \to \text{Physical Plan} \to \text{Execution}$$

The parser reads the SQL text and builds an abstract syntax tree (AST). The lowering phase converts the AST into a logical plan, often using relational algebra operators such as selection, projection, join, and aggregation. The optimizer then takes this logical plan and chooses concrete physical operators, such as table scan, index scan, hash join, nested-loop join, or aggregate operator. The execution engine finally runs that physical plan. Thus, the optimizer is the bridge between the meaning of a query and the mechanism used to execute it.

10.7.1 The Optimizer's Goal: Minimize Cost

Query optimization is primarily about minimizing the number of disk I/Os. This is a classical database viewpoint: disk access is much slower than CPU operations, so I/O cost is often the dominant performance factor. Therefore, when we say one plan is better than another, we often mean that it is expected to perform fewer I/Os.

Of course, in a real system we do not know the exact cost of a plan until we run it. That means the optimizer must work with estimates. It estimates how many tuples will survive filters, how many pages will be read, how large join outputs will be, and which execution algorithm is likely to be cheapest. This is why query optimization is not about finding an absolutely guaranteed best plan, but about using statistics, formulas, and heuristics to choose a good plan.

10.7.2 Logical Optimization

Equivalent Rewriting Logical optimization uses the fact that many relational algebra expressions are equivalent. Two logical plans are equivalent if they always produce the same result. The optimizer can rewrite one plan into another if the rewritten version is expected to be cheaper.

For example, a selection can often be pushed below a join:

$$\sigma_{A.x=10}(A \bowtie B) \quad \Longrightarrow \quad (\sigma_{A.x=10}(A)) \bowtie B$$

This is beneficial because it reduces the number of tuples entering the join.

Similarly, projections can often be pushed downward:

$$\pi_{a,b}(A \bowtie B) \;\equiv\; \pi_{a,b}\Big(\pi_{a,j}(A) \bowtie \pi_{b,j}(B)\Big)$$

can often be rewritten so that only the needed columns are carried into the join. This reduces tuple width, memory usage, and sometimes page count.

10.7.3 Selectivity Estimation

A key part of optimization is estimating how many tuples survive an operator. This is called selectivity estimation. **The selectivity of an operator is an approximation for what percentage of pages will make it through the operator onto the operator above it.** If an operator is highly selective, then it filters out many tuples, and we usually want to apply it early so that later operators process fewer pages.

For example, if only 1% of the rows satisfy a condition, then pushing that condition down before a join can greatly reduce the join cost.

A common set of simple formulas is:

$$\text{Sel}(X = a) = \frac{1}{V(X)}$$

where $V(X)$ is the number of distinct values of X.

For an equality join:

$$\text{Sel}(X = Y) = \frac{1}{\max(V(X), V(Y))}$$

For a range predicate on integers:

$$\text{Sel}(X > a) = \frac{\max(X) - a}{\max(X) - \min(X) + 1}$$

For conjunction:

$$\text{Sel}(c_1 \wedge c_2) = \text{Sel}(c_1) \cdot \text{Sel}(c_2)$$

These formulas are simple approximations, often assuming uniformity and independence. They are not perfect, but they give the optimizer a practical way to estimate output size.

10.7.4 Estimating Join Output Size

Suppose we join relations A and B on $A.id = B.id$. Without any condition, the Cartesian product has $|A| \cdot |B|$ tuples. With the equality join condition, we estimate:

$$|A \bowtie B| \approx \frac{|A| \cdot |B|}{\max(V(A.id), V(B.id))}$$

This gives an estimated number of output tuples. To estimate the number of output pages, we must also estimate how many joined tuples fit on one page. The important point is that page count cannot be derived by simply multiplying input page counts by selectivity; tuple widths and output layout matter.

10.7.5 Common Heuristics

A practical optimizer cannot consider every possible plan, because the plan space grows too quickly. Therefore, optimizers often use heuristics to reduce the search space. A classic set of heuristics includes:

1. Push down selections as far as possible.
2. Push down projections as far as possible.
3. Only consider left-deep join plans.
4. Avoid cross joins unless there is no alternative.

These heuristics are important because they dramatically reduce the number of candidate plans and usually improve performance. Selection pushdown reduces the number of tuples. Projection pushdown reduces tuple width. Left-deep plans are easier to pipeline and cheaper to enumerate than arbitrary bushy plans. Avoiding cross joins prevents huge intermediate results.

Why Left-Deep Join Plans Are Often Preferred? A left-deep join plan is a join tree in which the right child of every join is always a base relation, not the result of another join. For example, with relations R, S, and T, the following is left-deep:

$$((R \bowtie S) \bowtie T)$$

but the following is not left-deep:

$$(R \bowtie S) \bowtie (T \bowtie U)$$

It is important to note that left-deep plans are not always universally better than every other possible plan. Instead, they are often used as an optimizer heuristic because they make optimization much simpler and they work very well in many practical cases.

Reason 1: The Search Space Becomes Much Smaller The first advantage is that only considering left-deep plans greatly reduces the number of join trees the optimizer must explore. If we allow every possible bushy plan, then even for a moderate number of relations the number of possible plans becomes enormous. By restricting attention to left-deep plans, the optimizer only needs to decide an order in which relations are added one at a time.

For example, suppose we want to join four relations:

$$R,\ S,\ T,\ U$$

A left-deep optimizer may only consider plans such as:

$$(((R \bowtie S) \bowtie T) \bowtie U)$$

$$(((R \bowtie T) \bowtie S) \bowtie U)$$

$$(((S \bowtie U) \bowtie R) \bowtie T)$$

and so on. In each case, we start with one relation and keep attaching one more base table to the right.

If bushy plans are allowed, the optimizer must also consider shapes such as:

$$(R \bowtie S) \bowtie (T \bowtie U)$$

or

$$(R \bowtie (S \bowtie T)) \bowtie U$$

or many other possibilities. This increases the plan space dramatically. Since optimization itself takes time, reducing the search space is very important. A smaller search space means the optimizer can make decisions faster and with less implementation complexity.

Reason 2: Left-Deep Plans Can Be Pipelined More Easily The second major advantage is that left-deep plans are usually much easier to pipeline. Pipelining means that the system does not need to fully write the result of one join to disk before starting the next join. Instead, tuples can flow upward one at a time.

Consider the left-deep plan:

$$((R \bowtie S) \bowtie T)$$

Imagine that the DBMS is using a join algorithm such as index nested-loop join or hash join in a pipelined execution engine. It can do the following:

1. produce one tuple from $R \bowtie S$,
2. immediately feed that tuple into the join with T,
3. produce an output tuple,
4. then continue with the next tuple.

So the intermediate result of $R \bowtie S$ does not necessarily have to be fully materialized on disk. This saves I/O and often improves performance.

Why Bushy Plans Often Need More Materialization? Suppose we have:

$$(R \bowtie S) \bowtie (T \bowtie U)$$

At the top join, both the left child and the right child are already join results. That means both sides may need substantial work before the top operator can proceed. In many implementations, this leads to one or both subresults being materialized.

For example:

- compute $R \bowtie S$ first and store it,
- compute $T \bowtie U$ first and store it,
- then join the two stored intermediate relations.

This can be much more expensive in terms of temporary space and disk I/O, especially when the intermediate join results are large.

10.7.6 Pass 1 of System R

System R is one of the classic foundations of relational query optimization. More precisely, it is a cost-based optimizer framework that takes a logical query plan, applies a small set of important heuristics, estimates the cost of alternative physical plans, and keeps only the most promising candidates for later passes. In our course, System R is useful not only because of its historical importance, but also because it provides a clear and practical model for how a DBMS can choose access paths and join orders without exhaustively considering every possible execution strategy.

The first pass of System R focuses on single-table access. At this stage, the optimizer looks at each base table independently and asks: What is the best way to read this table? The answer is not always unique. For each table, Pass 1 considers both the cheapest access method and any access method that produces an interesting order, which may help later operators such as joins, `ORDER BY`, or `GROUP BY`.

For a table P, the two basic categories of access paths considered in Pass 1 are:

- **Full scan**
- **Index scan** (for each index built on the table)

At this point, the optimizer applies all single-table predicates as early as possible, following the selection-pushdown heuristic. For example, in a query involving `Student` and `Enrollment`, a condition such as `Student.gpa >= 3.0` can be applied during the access of `Student`, because it refers only to one table. In contrast, a condition such as

```
Student.student_id = Enrollment.student_id
```

is a join predicate, so it cannot yet be applied in Pass 1 because the optimizer is still evaluating each table separately.

Because all candidate access paths for the same table apply the same single-table predicates, they produce the same logical output and therefore have the same selectivity. However, they may have very different I/O costs.

Full scan cost. If table P occupies $[P]$ data pages, then a full scan always costs

$$[P]$$

I/Os, because every page of the table must be read.

Alternative 1 index cost. For an Alternative 1 index, the leaves themselves contain the data entries in sorted order. The cost of an index scan is:

$$\text{cost to reach the level above the leaf} + \text{number of leaf pages read}$$

The key idea is that we do not necessarily need to read every leaf page. If the query has a predicate on the indexed column, then we can use that condition to jump directly to the first relevant leaf and scan forward only as long as the condition remains true. However, predicates on non-indexed columns cannot help narrow the index range; they are applied only after the candidate entries have been found.

Example using the `Student` table. Suppose the `Student` table occupies $[S]$ pages, and there is an Alternative 1 index on `gpa` with height 2. Consider the following single-table conditions:

`gpa >= 3.0` and `credits >= 60`

Assume GPA values are uniformly distributed over a range for which `gpa >= 3.0` has selectivity 0.5, and credits are uniformly distributed so that `credits >= 60` also has selectivity 0.5. Since the conditions are connected by `AND`, the combined selectivity is

$$0.5 \times 0.5 = 0.25$$

Even though the final result contains only 25% of the tuples, only the predicate on `gpa` can help reduce the index scan range, because the index is built on `gpa`, not on `credits`. Therefore, the scan only needs to examine approximately $0.5[S]$ leaf pages, not all of them. If it takes 2 I/Os to reach the level above the leaf level, then the total cost is:

$$2 + 0.5[S]$$

Alternative 2/3 index cost. For Alternative 2 or Alternative 3 indexes, the leaves do not store the full data records directly. Instead, they store references to records, so the DBMS must perform additional I/Os to retrieve the actual table pages. The cost formula becomes:

$$\text{cost to reach the level above the leaf} + \text{number of leaf pages read} + \text{number of data pages read}$$

Again, predicates on the indexed column can reduce the number of leaf pages read, and they also reduce the number of qualifying data records. The difference between clustered and unclustered indexes becomes very important here.

If the index is clustered, then qualifying records tend to be stored close together in the data file, so the number of data pages read is approximately:

$$\text{selectivity} \times [P]$$

If the index is unclustered, qualifying records may be scattered across the table, so the DBMS may need roughly one I/O per qualifying record. In that case, the number of data-page I/Os is closer to:

$$\text{selectivity} \times |P|$$

where $|P|$ is the number of records in the table.

Example using the `Enrollment` table. Suppose the `Enrollment` table has $[E]$ data pages and $|E|$ records. Assume there is an Alternative 2 index on `student_id`, the index height is 2, and the leaf level contains $[L]$ pages. Consider the predicates

`student_id <= 100999` and `semester = 'Fall 2025'`

Assume, for simplicity, that each condition has selectivity 0.5. Then the combined selectivity is again

$$0.5 \times 0.5 = 0.25$$

but only the predicate on `student_id` can reduce the index range, because the index is built on `student_id`. Therefore, the scan reads approximately $0.5[L]$ leaf pages.

If the index is clustered, then the number of data pages read is approximately $0.5[E]$, so the total cost is:

$$2 + 0.5[L] + 0.5[E]$$

If the index is unclustered, then the data records may be scattered, so the DBMS may need about $0.5|E|$ data-page fetches. In that case, the total cost is:

$$2 + 0.5[L] + 0.5|E|$$

This example shows why clustering matters so much: even when two access paths have the same logical selectivity, their physical I/O behavior can be very different.

Which access paths survive Pass 1? After estimating the cost of all candidate scans for a table, System R does not keep every possibility. Instead, for each table it keeps:

- the **optimal access path**, meaning the one with the fewest estimated I/Os; and
- any access path that produces an **interesting order**.

An interesting order is an output ordering that may reduce the cost of a later operator. In practice, an order is interesting if it is on a column that will later be:

- used in an `ORDER BY`,
- used in a `GROUP BY`, or
- used in a downstream join.

The first two cases are straightforward: if a table is already read in an order needed later by `ORDER BY` or `GROUP BY`, the DBMS may avoid an extra sort. The third case is also important: if tuples are read in join-key order, the optimizer may later take advantage of that order in a sort-merge join or another order-sensitive operation.

A full scan generally does not produce an interesting order, because the table is not assumed to be stored in the required sorted order. By contrast, an index scan does produce output in the order of the indexed column. Of course, that order is only worth keeping if it is actually useful later in the query.

An Example Consider the query:

```
SELECT *
FROM Student INNER JOIN Enrollment
ON Student.student_id = Enrollment.student_id
WHERE Student.gpa >= 3.5
ORDER BY Student.name;
```

For this query, the attributes that may produce an interesting order are:

- `Student.name`, because it is used in the final `ORDER BY`;
- `Student.student_id` and `Enrollment.student_id`, because they are used in the downstream join.

Suppose the optimizer finds the following candidate access paths:

- Full Scan `Student` 100 I/Os
- Index Scan `Student.gpa` 60 I/Os
- Index Scan `Student.student_id` 110 I/Os
- Index Scan `Student.name` 140 I/Os
- Full Scan `Enrollment` 220 I/Os
- Index Scan `Enrollment.student_id` 250 I/Os
- Index Scan `Enrollment.course_id` 260 I/Os

Then Pass 1 would keep the following access paths:

- `Index Scan Student.gpa`, because it is the cheapest access path for `Student`;
- `Index Scan Student.student_id`, because it produces an interesting order for the downstream join on `student_id`;
- `Index Scan Student.name`, because it produces an interesting order for the final `ORDER BY Student.name`;
- `Full Scan Enrollment`, because it is the cheapest access path for `Enrollment`;
- `Index Scan Enrollment.student_id`, because it produces an interesting order for the downstream join on `student_id`.

The access path `Index Scan Enrollment.course_id` does not survive, even though it is an index scan, because `course_id` is not used anywhere later in this query. It does not appear in the `WHERE` clause, the join condition, or the `ORDER BY`, so its output order is not interesting.

This example illustrates an important point: Pass 1 does not keep only the cheapest access path for each table. It also keeps any access path whose output order may help later operators. In this query, both sides of the join may contribute interesting orders on `student_id`, and `Student.name` contributes an interesting order for the final sort requirement.

Summary. Pass 1 of System R is the stage where the optimizer decides how to access each base table individually. It applies single-table predicates as early as possible, estimates the I/O cost of full scans and index scans, and then keeps only the cheapest access path plus any access paths that produce useful interesting orders. These surviving candidates form the input to later passes, where the optimizer begins to build multi-table join plans.

10.7.7 Passes $2 \dots n$ of System R

After Pass 1 has determined the promising access paths for each individual table, the remaining passes of System R focus on joining tables together. The basic idea is very systematic: Pass i constructs plans that join exactly i tables by combining:

- a plan from Pass $i-1$, which already joins $i-1$ tables, and
- one base-table access path from Pass 1.

For example, in Pass 2 the optimizer joins two single-table plans from Pass 1. In Pass 3, it takes a two-table plan from Pass 2 and joins it with one additional base table from Pass 1. In general, Pass i extends a previously computed plan on $i-1$ tables by attaching one more base table on the right. This structure naturally enforces the left-deep plan restriction: at every stage, the left side is an existing join result, while the right side is always a base relation.

This restriction is important because it keeps the search space manageable and fits well with pipelined execution. Instead of considering every possible bushy join tree, System R only explores plans that can be built incrementally from left to right.

What Pass i produces. For each set of i tables that can be connected without requiring a cross join, Pass i produces at least one candidate plan, assuming such a plan exists. Just as in Pass 1, the optimizer does not keep every candidate. For each join set, it keeps:

- the cheapest overall plan, and
- the cheapest plan for each useful interesting order, if such an order exists.

Thus, the same principle continues across all passes: keep the best plan overall, but also preserve slightly more expensive plans if their output ordering may save work later.

Only non-cross-join sets are considered. A very important heuristic is that System R does not consider cross joins unless they are unavoidable. So if two subsets of relations do not have a join condition connecting them, the optimizer simply does not generate that join candidate.

This dramatically reduces the number of plans under consideration and also avoids obviously bad intermediate results. In most realistic queries, cross joins are not intended, so ignoring them is both safe and efficient.

Join algorithms in later passes. When the optimizer tries to join a Pass $i-1$ plan with a base table from Pass 1, it considers each join algorithm implemented by the DBMS. Different algorithms may have different estimated I/O costs, and they may also differ in whether they preserve or create an interesting output order.

Among the common join algorithms discussed in System R style optimization, **sort-merge join (SMJ)** is especially important because it produces an output sorted on the join attributes. Therefore, SMJ is the main way for a join result to acquire a new interesting order.

Some other join algorithms may preserve the ordering of the left input rather than creating a new one. For example, **simple nested loop join (SNLJ)** and **index nested loop join (INLJ)** preserve the order of the left relation because they process one left tuple at a time and generate all matching outputs for that tuple before moving on to the next one. As a result, if the left input is already sorted on an interesting attribute, that order is preserved in the output.

By contrast, **grace hash join (GHJ)**, **page nested loop join (PNLJ)**, and **block nested loop join (BNLJ)** generally do not produce a useful interesting order. Grace hash join partitions relations by hash value, so the overall output order is determined by hash partitions rather than by a meaningful logical attribute order. Page nested loop join and block nested loop join process tuples in page-sized or block-sized chunks rather than strictly tuple-by-tuple, so they do not preserve the exact tuple order of the left relation either.

Suppose we are optimizing the following query:

```
SELECT *
FROM Student
INNER JOIN Enrollment
    ON Student.student_id = Enrollment.student_id
INNER JOIN Course
```

```
    ON Enrollment.course_id = Course.course_id
ORDER BY Course.course_id;
```

This query involves three tables: `Student`, `Enrollment`, and `Course`. The join graph tells us that:

- `Student` can join with `Enrollment` using `student_id`,
- `Enrollment` can join with `Course` using `course_id`,
- but `Student` and `Course` do not have a direct join predicate.

Pass 2. In Pass 2, the optimizer only considers two-table sets that can be joined directly without a cross join. Therefore, it considers:

$$\{\texttt{Student}, \texttt{Enrollment}\} \quad \text{and} \quad \{\texttt{Enrollment}, \texttt{Course}\}$$

It does not consider:

$$\{\texttt{Student}, \texttt{Course}\}$$

because there is no join predicate connecting these two tables directly.

To keep the example simple, suppose our DBMS only implements **sort-merge join (SMJ)** and **block nested loop join (BNLJ)**, and suppose Pass 1 returned only one access plan for each base table, namely a full scan. Assume the optimizer estimates the following costs:

`Student BNLJ Enrollment`	cost 1200
`Enrollment BNLJ Student`	cost 1600
`Student SMJ Enrollment`	cost 2100
`Enrollment BNLJ Course`	cost 900
`Course BNLJ Enrollment`	cost 700
`Enrollment SMJ Course`	cost 1100

Now we decide which plans survive Pass 2.

For the set $\{\texttt{Student}, \texttt{Enrollment}\}$, the cheapest plan is:

```
Student BNLJ Enrollment
```

with estimated cost 1200.

For the set $\{\texttt{Enrollment}, \texttt{Course}\}$, the cheapest plan is:

```
Course BNLJ Enrollment
```

with estimated cost 700.

However, we also need to consider interesting orders. Our query has:

```
ORDER BY Course.course_id
```

so any join plan whose output is sorted on `Course.course_id` may be useful later. The plan

```
Enrollment SMJ Course
```

produces output sorted on the join attributes, namely `Enrollment.course_id` and `Course.course_id`. Because `Course.course_id` is needed later by the final `ORDER BY`, this order is interesting. Therefore, even though it is not the cheapest plan for $\{\texttt{Enrollment}, \texttt{Course}\}$, it also survives Pass 2.

So the plans that advance from Pass 2 are:

- `Student BNLJ Enrollment` (cheapest for $\{\texttt{Student}, \texttt{Enrollment}\}$)
- `Course BNLJ Enrollment` (cheapest for $\{\texttt{Enrollment}, \texttt{Course}\}$)
- `Enrollment SMJ Course` (interesting order on `Course.course_id`)

Why $\{\texttt{Student}, \texttt{Course}\}$ is excluded. It is worth emphasizing again that the pair $\{\texttt{Student}, \texttt{Course}\}$ is not considered in Pass 2. Even though these tables both appear in the query, they cannot be joined directly without going through `Enrollment`. Considering them would require a cross join, which System R avoids.

Pass 3. Now the optimizer constructs plans for all three tables. Since we are restricted to left-deep plans, each candidate must take a two-table plan from Pass 2 and attach the remaining base table on the right. Therefore, the optimizer considers:

`(Student BNLJ Enrollment) BNLJ Course`	cost 10000
`(Student BNLJ Enrollment) SMJ Course`	cost 12000
`(Course BNLJ Enrollment) BNLJ Student`	cost 8500
`(Course BNLJ Enrollment) SMJ Student`	cost 19000
`(Enrollment SMJ Course) BNLJ Student`	cost 22000
`(Enrollment SMJ Course) SMJ Student`	cost 17000

Notice something very important: at this point, the optimizer cannot arbitrarily rearrange the tree shape. It can only attach one base relation to the right of an existing two-table result. That is exactly the left-deep restriction in action.

Which Pass 3 plans survive? All of these plans join the full set

$$\{\texttt{Student}, \texttt{Enrollment}, \texttt{Course}\}$$

so now we are selecting survivors for the complete query.

The cheapest overall plan is:

```
(Course BNLJ Enrollment) BNLJ Student
```

with cost 8500.

But we also need to keep the cheapest plan that preserves an interesting order on `Course.course_id`, because the `ORDER BY` has not yet been evaluated. Among the candidates, the plan

```
(Student BNLJ Enrollment) SMJ Course
```

is the best such plan, with cost 12000.

The reason this plan produces an output sorted on `Course.course_id` is that its final join is a sort-merge join on:

```
Enrollment.course_id = Course.course_id
```

and sort-merge join produces output sorted on the join attributes. Since `Enrollment.course_id` and `Course.course_id` are equal in the join output, this gives us an ordering that is useful for the final `ORDER BY Course.course_id`.

By contrast, plans whose final join is on

```
Student.student_id = Enrollment.student_id
```

may produce output ordered on `student_id`, but that order is not interesting here because `student_id` is not needed later by either `ORDER BY` or another unevaluated join.

Therefore, the surviving Pass 3 plans are:

- the cheapest overall plan for all three tables;
- the cheapest plan that preserves an interesting order on `Course.course_id`.

What this example shows. This example illustrates several core ideas of System R.

First, later passes build larger join plans incrementally from smaller ones, rather than starting from scratch each time.

Second, the left-deep restriction means that the optimizer only extends previous plans by joining them with one additional base table on the right.

Third, cross joins are excluded unless unavoidable, so only connected relation sets are considered.

Finally, the optimizer continues to keep both the cheapest plan and any plan with a useful interesting order. This is important because the cheapest plan locally is not always the best choice globally if another plan can avoid a later sort.

Summary. Passes $2 \dots n$ of System R are the stages where the optimizer builds join plans of increasing size. Each pass extends previously computed plans by attaching one base table, which naturally enforces left-deep plans. For each joinable set of tables, the optimizer keeps the cheapest plan overall and also the cheapest plan for each useful interesting order. In this way, System R balances two goals: minimizing immediate cost and preserving structural properties, such as ordering, that may reduce the cost of later operations.

10.8 Calculating I/O Costs for Join Operations in Query Optimization

When we first learn join algorithms, we usually memorize their baseline I/O formulas. For example, Block Nested Loop Join has cost

$$[R] + \left\lceil \frac{[R]}{B-2} \right\rceil [S]$$

and Sort-Merge Join has average-case cost

$$\text{Sort}(R) + \text{Sort}(S) + ([R] + [S]).$$

These formulas are very useful, but in query optimization they are not always enough by themselves.

The reason is that the optimizer does not cost a join in isolation. Instead, it costs the join inside a query plan. That means the optimizer must also consider:

1. whether previous operators produce intermediate results that are materialized to disk or streamed directly into the join,
2. whether selections and projections reduce the effective size of a join input,
3. whether previous operators produce an interesting order that can reduce the cost of a later join, especially Sort-Merge Join.

So the central lesson of this section is:

> In query optimization, the I/O cost of a join is not always obtained by plugging the original table sizes directly into the basic join formula. We must first understand what data actually reaches the join operator and how it reaches it.

We will use the following notation throughout:

- R, S: the two inputs of a join
- $[R]$, $[S]$: number of pages in relations R and S
- $|R|$, $|S|$: number of tuples in relations R and S
- B: number of buffer pages available
- $V(A)$: number of distinct values of attribute A

In our examples, we will often use:

$$R = \texttt{Student}, \qquad S = \texttt{Enrollment}$$

10.8.1 Part I: Estimating the Size of Join Inputs and Join Outputs

If a selection is pushed below a join, then the join may not see all pages of the original table. For example, if only 30 pages of `Student` survive a predicate, then a later BNLJ should use 30 pages as its effective left input, not the original $[Student]$.

So before costing a join, we often estimate:

1. how many pages remain after each pushed-down predicate,
2. how many tuples the join will produce,
3. how many output pages the join result occupies.

Join Selectivity For an equality join

$$R.X = S.Y$$

a common estimate for join selectivity is:

$$\text{Sel}(R.X = S.Y) = \frac{1}{\max(V(X), V(Y))}$$

Then the estimated number of output tuples is:

$$|R \bowtie S| \;\approx\; |R| \cdot |S| \cdot \mathrm{Sel}(R.X = S.Y) = \frac{|R| \cdot |S|}{\max(V(X), V(Y))}$$

To estimate join output pages, the procedure is:

1. Compute join selectivity.
2. Compute the estimated number of output tuples.
3. Divide by the estimated number of joined tuples that fit on one page.

So if t_{join} is the number of joined tuples per page, then:

$$[\text{output}] \;\approx\; \left\lceil \frac{|R| \cdot |S| \cdot \mathrm{Sel}(R.X = S.Y)}{t_{\text{join}}} \right\rceil$$

We cannot usually estimate output pages by simply multiplying page counts by selectivity. Page counts do not behave like tuple counts, because joined tuples are wider and may fit fewer per page.

Suppose:

- `Student` has $|Student| = 2000$ tuples,
- `Enrollment` has $|Enrollment| = 4000$ tuples,
- we join on `Student.student_id = Enrollment.student_id`,
- $V($`Student.student_id`$) = 2000$,
- $V($`Enrollment.student_id`$) = 1800$.

Then:

$$\mathrm{Sel} = \frac{1}{\max(2000, 1800)} = \frac{1}{2000}$$

So the estimated number of output tuples is:

$$|Student \bowtie Enrollment| \approx \frac{2000 \cdot 4000}{2000} = 4000$$

If the joined tuples are large and only 20 joined tuples fit on one page, then the estimated output size is:

$$\left\lceil \frac{4000}{20} \right\rceil = 200 \text{ pages}$$

10.8.2 Part II: Why Basic Join Formulas Must Be Adjusted in Query Optimization

Two Key Corrections When calculating join I/O cost inside an optimizer, two issues matter especially:

Correction 1: Materialization versus streaming. If an intermediate result is materialized, then we pay to write it to disk and later read it back. If it is streamed, then the next operator consumes it directly without a write-read cycle.

Correction 2: Interesting orders. If an earlier access path produces tuples already sorted on a useful attribute, then a later Sort-Merge Join may be cheaper because one input no longer needs to be sorted.

10.8.3 Part III: Worked Examples with Diagrams

Common Setup We will use the following query throughout:

```
SELECT *
FROM Student S INNER JOIN Enrollment E
ON S.student_id = E.student_id
WHERE S.gpa >= 3.5 AND E.semester = 'Fall 2025';
```

Assume:

$$B = 5, \qquad [S] = 50, \qquad [E] = 100$$

and suppose:

- the predicate `S.gpa >= 3.5` leaves 30 pages of `Student`,
- the predicate `E.semester = 'Fall 2025'` leaves 40 pages of `Enrollment`.

Since $B = 5$, a BNLJ left block has size:

$$B - 2 = 3$$

So the reduced `Student` input has:

$$\left\lceil \frac{30}{3} \right\rceil = 10$$

left blocks.

Example 1: Materialize Both Selections Before the Join In this first plan, both selections are materialized before the BNLJ begins.

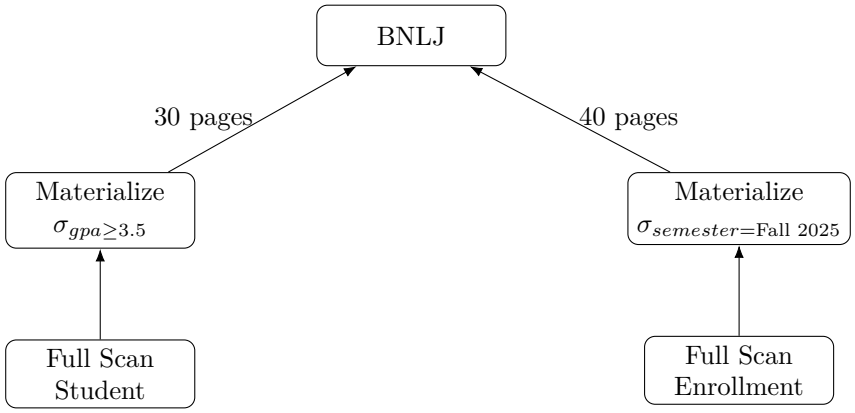


Figure 14: Example 1: both filtered inputs are written to disk before the BNLJ.

Cost calculation. The total cost is:

$$(50 + 30) + (100 + 40) + \left(30 + \left\lceil \frac{30}{5-2} \right\rceil \cdot 40\right)$$

Why?

- Scan `Student`: 50 I/Os.
- Write the 30-page filtered `Student` result: 30 I/Os.
- Scan `Enrollment`: 100 I/Os.
- Write the 40-page filtered `Enrollment` result: 40 I/Os.
- Now BNLJ runs on the materialized 30-page and 40-page relations:

$$30 + \left\lceil \frac{30}{3} \right\rceil \cdot 40 = 30 + 10 \cdot 40 = 430$$

So the total is:

$$80 + 140 + 430 = 650 \text{ I/Os}$$

Example 2: Materialize Only the Right Selection Now suppose we materialize only the filtered `Enrollment` relation, while the selection on `Student` is streamed into the left side of BNLJ.

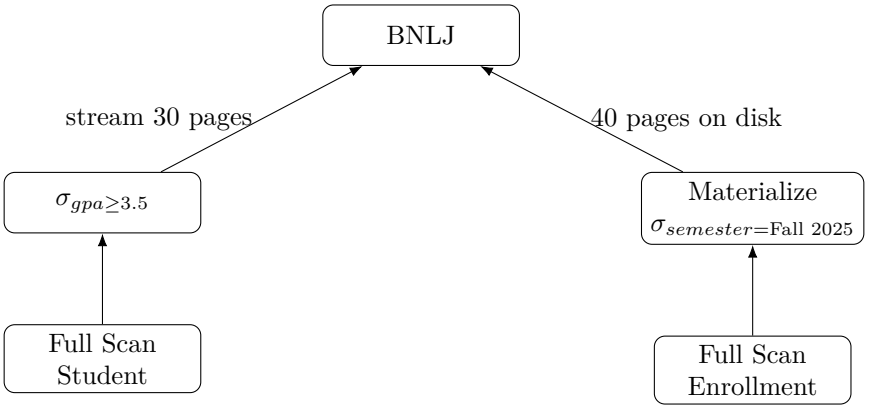


Figure 15: Example 2: the right filtered input is materialized, but the left filtered input is streamed.

Cost calculation. The total cost is:

$$(100 + 40) + \left(50 + \left\lceil \frac{30}{5-2} \right\rceil \cdot 40\right)$$

Why?

- Materialize filtered `Enrollment`: $100 + 40$.
- Full scan of `Student`: 50.
- Only 30 pages survive the left selection, so BNLJ uses

$$\left\lceil \frac{30}{3} \right\rceil = 10$$

  left blocks.
- For each left block, BNLJ reads the 40-page filtered `Enrollment` relation from disk.

So:

$$(100 + 40) + (50 + 10 \cdot 40) = 140 + 450 = 590 \text{ I/Os}$$

Important lesson. Here the left-side selection helps because the left side of BNLJ is only passed once, block by block. Reducing the left input from 50 pages to 30 pages reduces the number of left blocks, which reduces how many times the right side is rescanned.

The execution proceeds as follows:

1. Scan the original `Student` table once, page by page. This costs 50 I/Os.
2. Apply the predicate $\sigma_{\texttt{gpa} \geq 3.5}$ while scanning.
3. Keep only the qualifying tuples, and use them to fill the left-side BNLJ buffer up to 3 pages.
4. Once the left buffer is full, scan the materialized 40-page filtered `Enrollment` relation once and perform the join.
5. After that right-side scan finishes, reuse the same 3 left-buffer pages for the next batch of qualifying `Student` tuples.
6. Continue this process until the full `Student` scan is complete.

Example 3: Materialize Only the Left Selection Now suppose we materialize only the filtered `Student` relation, while the selection on `Enrollment` is applied on the fly during every scan of the right side.

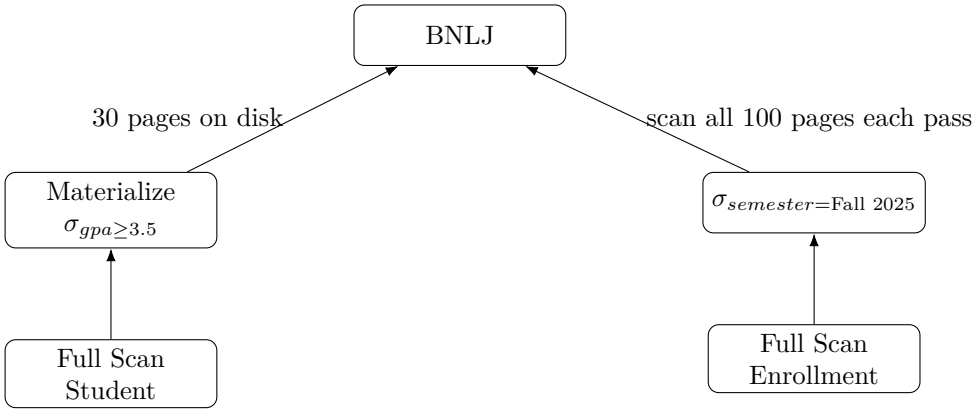


Figure 16: Example 3: the left filtered input is materialized, but the right filtered input is not.

Cost calculation. The total cost is:

$$(50 + 30) + \left(30 + \left\lceil \frac{30}{5-2} \right\rceil \cdot 100\right)$$

Why?

- Scan `Student`: 50.
- Write the 30-page filtered `Student` relation: 30.
- BNLJ then reads those 30 pages once as its left input.
- But the right side is not materialized after filtering, so for each left block we must scan all 100 pages of `Enrollment` and apply the predicate on the fly.

Thus:

$$(50 + 30) + (30 + 10 \cdot 100) = 80 + 1030 = 1110 \text{ I/Os}$$

Important lesson. Even though the right selection is pushed down logically, it does not reduce I/O here, because the filtered right relation was never saved to disk. Every time BNLJ needs the right input, it still scans all 100 original pages again.

Example 4: System R Style — No Intermediate Materialization In the simplified System R setting used in this tutorial, intermediate outputs are assumed to be streamed rather than materialized. So the optimizer would consider a plan like this:

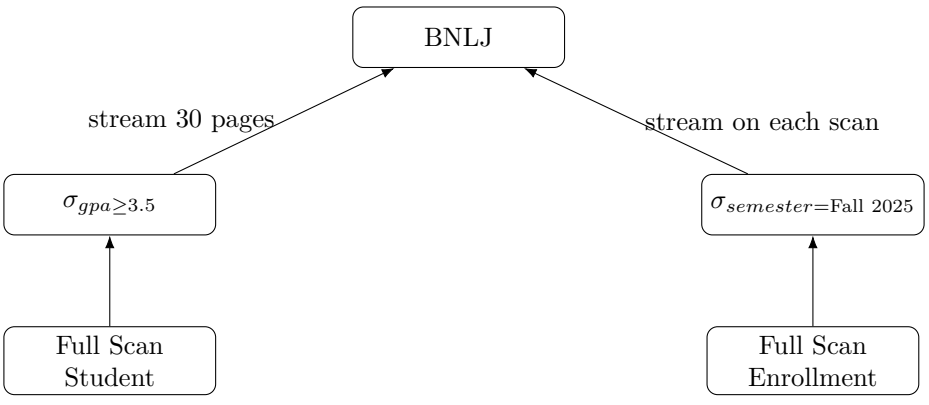


Figure 17: Example 4: System R style plan, where intermediate results are streamed rather than materialized.

Cost calculation. The total cost is:

$$50 + \left\lceil \frac{30}{5-2} \right\rceil \cdot 100$$

Why?

- We scan `Student` once: 50 I/Os.
- The selection leaves 30 pages, so the left side of BNLJ has
$$\left\lceil \frac{30}{3} \right\rceil = 10$$
blocks.
- The right side is not materialized, so each left block causes a full scan of all 100 pages of `Enrollment`.

Therefore:
$$50 + 10 \cdot 100 = 1050 \text{ I/Os}$$

10.8.4 Part IV: Interesting Orders and Their Effect on Join Cost

Now let us look at the second big optimizer correction: an earlier operator may produce tuples already sorted on a useful attribute.

Suppose we want to optimize:

```
SELECT *
FROM Student S INNER JOIN Enrollment E
ON S.student_id = E.student_id
WHERE S.student_id > 100200;
```

Assume:

- $[S] = 50$,
- $[E] = 100$,
- the optimizer chooses an index scan on `Student.student_id` costing 60 I/Os,
- the optimizer chooses a full scan on `Enrollment` costing 100 I/Os.

Because the index scan on `Student.student_id` returns tuples in `student_id` order, it produces an interesting order for the downstream join on `student_id`.

Baseline SMJ Cost Without Interesting Orders. If both inputs were unsorted, the average-case SMJ cost would be:
$$\text{Sort}(S) + \text{Sort}(E) + (50 + 100)$$

But now `Student` already arrives sorted from the index scan, so we no longer need to sort it. Instead, we can stream the index-scan output directly into the merge phase. Thus the estimated cost becomes:
$$60 + \text{Sort}(E) + 100$$

Interpretation. The 60 replaces the need to sort the left input and also serves as the way we access the left input for the join. The right input still needs to be sorted and then read during the merge process.

Important lesson. An interesting order is not necessarily the cheapest single-table access path. It is kept because it may reduce the cost of a later operator. Here, the index scan is valuable not only for filtering, but also because it produces join-key order for Sort-Merge Join.

10.8.5 Part V: A General Procedure for Costing Join Plans in an Optimizer

When you cost a join in a query-optimization problem, a good step-by-step process is:

1. Identify the join algorithm being used.
2. Start from the baseline formula for that join.
3. Determine what selections and projections have been pushed below the join.
4. Estimate how many pages actually reach the join on each side.
5. Ask whether those intermediate results are materialized or streamed.
6. If streamed, ask whether the join reads that side once or many times.
7. If the join is SMJ, ask whether either side already has a useful interesting order.
8. Only after these adjustments, plug the effective page counts into the cost expression.

11 Transactions & Concurrency

In most situations, we usually do not have just one person accessing a database at a time. Many users can make requests to a database at the same time which can cause concurrency issues. What happens when one user writes and then another user reads from the same resource? What if both users try to write to the same resource? There are several problems we can run into when several users are using the database at the same time if we are not careful:

- **Inconsistent Reads (Write-Read Conflict):** A user reads only part of what was updated.
  - User 1 updates Table 1 and then updates Table 2.
  - User 2 reads Table 2 (which User 1 has not updated yet) and then Table 1 (which User 1 already updated) so it reads the database in an intermediate state.
- **Lost Update (Write-Write Conflict):** Two users try to update the same record at the same time so one of the updates gets lost. For example:
  - User 1 updates a toy's price to be `price * 2`.
  - User 2 updates a toy's price to be `price + 5`, removing User 1's update.
- **Dirty Reads (Write-Read Conflict):** One user reads an update that was never committed.
  - User 1 updates a toy's price but this gets aborted.
  - User 2 reads the update before it was rolled back.
- **Unrepeatable Reads (Read-Write Conflict):** A user reads two different values for the same record because another user updated the record in between the two reads.
  - User 1 reads in toy price.
  - User 2 updates toy price to be `toy price * 2` and commits.
  - User 1 reads in toy price again. User 1 reads in two different values for toy price because User 2's write operation was interleaved. User 1 should be reading in the same value within one transaction. As such, User 1 will have to abort.

11.1 Transactions

Our solution to those problems is to define a set of rules and guarantees about operations. We will do this by using transactions. A transaction is a sequence of multiple actions that should be executed as a single, logical, atomic unit. Transactions guarantee the ACID properties to avoid the problems discussed above:

- **Atomicity:** A transaction ends in two ways: it either commits or aborts. Atomicity means that either all actions in the transaction happen, or none happen.
- **Consistency:** If the DB starts out consistent, it ends up consistent at the end of the transaction.
- **Isolation:** Execution of each transaction is isolated from that of others. In reality, the DBMS will interleave actions of many transactions and not execute each in order of one after the other. The DBMS will ensure that each transaction executes as if it ran by itself.
- **Durability:** If a transaction commits, its effects persist. The effects of a committed transaction must survive failures.

11.1.1 Motivation for Concurrent Execution

It seems as though concurrency causes a lot of trouble, so why bother with it? There are two main advantages that come with implementing concurrency in our DBMS:

- **Increase in Throughput (transactions per second):** If the DBMS is running on a single core, one transaction can use the CPU while another transaction is reading to or writing from the disk. If the DBMS is running on a multicore processor, we can ideally scale throughput with respect to the number of processors.
- **Decrease in Latency (response time per transaction):** Multiple transactions can run at the same time, so one transaction's latency need not be dependent on another unrelated transaction.

11.1.2 Concurrency Control

In this note we will discuss how to enforce the isolation property of transactions (we will learn how the other properties are enforced in the note about recovery). To do this, we will analyze transaction schedules which show the order that operations are executed in. These operations include: Begin, Read, Write, Commit and Abort. The easiest way to

ensure isolation is to run all the operations of one transaction to completion before beginning the operations of the next transaction. This is called a serial schedule. For example, the following schedule is a serial schedule because T_1's operations run completely before T_2 runs.

T1: Transfer $100 from A to B	T2: Add 10% interest to A & B
begin	
read(A)	
A = A - 100	
write(A)	
read(B)	
B = B + 100	
write(B)	
commit	
	begin
	read(A)
	A = A * 1.1
	write(A)
	read(B)
	B = B * 1.1
	write(B)
	commit

Figure 18: A serial schedule.

The problem with these schedules, however, is that it is not efficient to wait for an entire transaction to finish before starting another one. Ideally, we want to get the same results as a serial schedule (because we know serial schedules are correct) while also getting the performance benefits of running schedules simultaneously. Basically, we are looking for a schedule that is equivalent to a serial schedule. For schedules to be equivalent they must satisfy the following three rules:

1. They involve the same transactions.
2. Operations are ordered the same way within the individual transactions.
3. They each leave the database in the same state.

If we find a schedule whose results are equivalent to a serial schedule, we call the schedule serializable. For example, the following schedule is serializable because it is equivalent to the schedule above. You can work through the following schedule and see that resources A and B end up with the same value as the serial schedule above.

T1: Transfer $100 from A to B	T2: Add 10% interest to A & B
begin	
read(A)	
A = A - 100	
write(A)	
	begin
	read(A)
	A = A * 1.1
	write(A)
read(B)	
B = B + 100	
write(B)	
commit	
	read(B)
	B = B * 1.1
	write(B)
	commit

Figure 19: A serializable interleaving equivalent to the serial schedule.

11.1.3 Conflict Serializability

Now the question is: how do we ensure that two schedules leave the database in the same final state without running through the entire schedule to see what the result is? We can do this by looking for conflicting operations. For two operations to conflict they must satisfy the following three rules:

1. The operations are from different transactions.
2. Both operations operate on the same resource.
3. At least one operation is a write.

We then check if the two schedules order every pair of conflicting operations in the same way. If they do, we know for sure that the database will end up in the same final state. When two schedules order their conflicting operations in the same way the schedules are said to be conflict equivalent, which is a stronger condition than being equivalent. Now that we have a way of ensuring that two schedules leave the database in the same final state, we can check if a schedule is conflict equivalent to a serial schedule without running through the entire schedule. We call a schedule that is conflict equivalent to some serial schedule conflict serializable. Note: if a schedule is conflict serializable then it implies that it is serializable.[1]

11.2 Conflict Dependency Graph

Now we have a way of checking if a schedule is serializable. We can check if the schedule is conflict equivalent to some serial schedule because conflict serializable implies serializable. We can check conflict serializability by building a dependency graph. Dependency graphs have the following structure:

- One node per transaction.
- Edge from T_i to T_j if:
  - an operation O_i of T_i conflicts with an operation O_j of T_j, and
  - O_i appears earlier in the schedule than O_j.

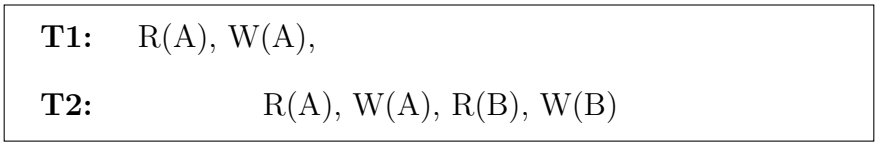
A schedule is conflict serializable if and only if its dependency graph is acyclic. So all we have to do is check if the graph is acyclic to know for sure that it is serializable.

Let's take a look at two examples:

- The following schedule is conflict serializable and the conflict graph is acyclic. There are two conflicting operations:

[1] Not all serializable schedules are conflict serializable.

- T_1 reads A and then T_2 writes to A. Because of this, there will be an edge from T_1 to T_2.
- T_1 writes to A and then T_2 reads from A. Since there already is an edge from T_1 to T_2, we do not have to add the edge again.



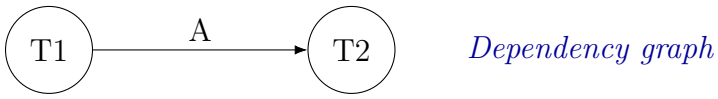
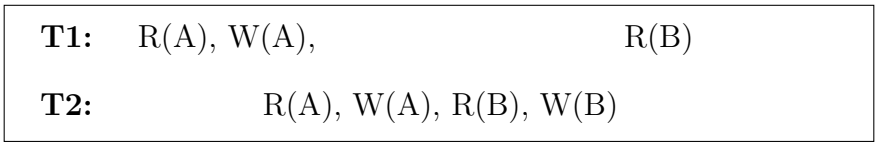


Figure 20: Conflict-serializable example.

- The following schedule is not conflict serializable and the conflict graph is not acyclic. Some conflicting operations:
  - T_1 reads A and then T_2 writes to A. Because of this, there will be an edge from T_1 to T_2.
  - T_2 writes to B and then T_1 reads B. Because of this, there will be an edge from T_2 to T_1.



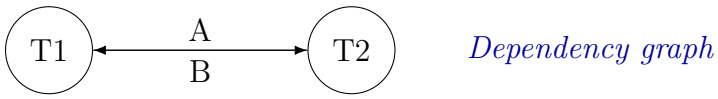


Figure 21: Non-conflict-serializable example.

11.3 View Serializability

View serializability is an alternate way to determine overall serializability. View serializability has fewer false negatives with respect to conflict serializability, meaning that it will identify more serializable schedules than conflict serializability. Schedules S_1 and S_2 are view equivalent if they have:

1. Same initial reads.
2. Same dependent reads.
3. Same winning writes.

View serializability allows a few more schedules than conflict serializability. View serializability finds all schedules that are conflict serializable, plus a couple more that have blind writes. Blind writes are sequential writes to a data page with no interleaving reads to the page. Conflict serializability is generally preferred because it is easier and more efficient to enforce, while only missing out on a few serializable schedules with blind writes.

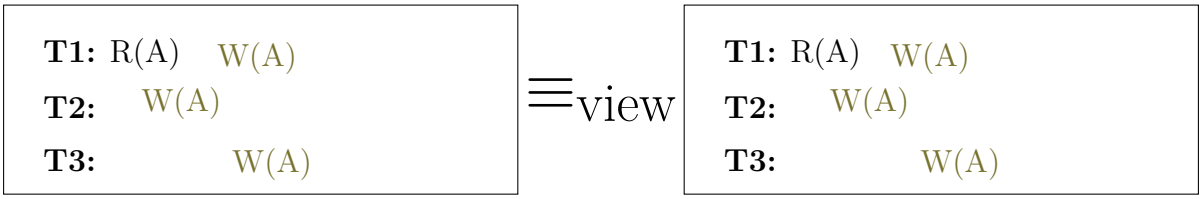


Figure 22: A stylized illustration of view equivalence with blind writes.

A serial schedule means that one transaction runs to completion before the next begins.

A conflict serializable schedule may still be interleaved, but it behaves like some serial schedule with respect to all conflicting operations.

If two schedules order every conflicting pair the same way, they are conflict equivalent, and if a schedule is conflict equivalent to some serial schedule, then it is called conflict serializable.

11.4 Two Phase Locking

In the last note, we introduced the concept of isolation as one of the ACID properties. Let us revisit our definition here:

Isolation: Execution of each transaction is isolated from that of others. In reality, the DBMS will interleave actions of many transactions and not execute each in order of one after the other. The DBMS will ensure that each transaction executes as if it ran by itself.

What are locks, and why are they useful? Locks are basically what allows a transaction to read and write data. For example, if transaction T_1 is reading data from resource A, then it needs to make sure no other transaction is modifying resource A at the same time. So a transaction that wants to read data will ask for a Shared (S) lock on the appropriate resource, and a transaction that wants to write data will ask for an Exclusive (X) lock on the appropriate resource. Only one transaction may hold an exclusive lock on a resource, but many transactions can hold a shared lock on data. Two phase locking (2PL) is a scheme that ensures the database uses conflict serializable schedules. The two rules for 2PL are:

- **Transactions must acquire a S (shared) lock before reading, and an X (exclusive) lock before writing.**
- **Transactions cannot acquire new locks after releasing any locks – this is the key to enforcing serializability through locking.**

	S	**X**
S	✓	–
X	–	–

Figure 23: Lock compatibility matrix for S and X locks.

Two Phase Locking guarantees conflict serializability. When a committing transaction has reached the end of its acquisition phase, let us call this the "lock point." At this point, it has everything that it needs locked. Any conflicting transactions either started the release phase before this point or are blocked waiting for this transaction. So the visibility of actions of two conflicting transactions can be ordered by their lock points. The order of these lock points gives us an equivalent serial schedule.

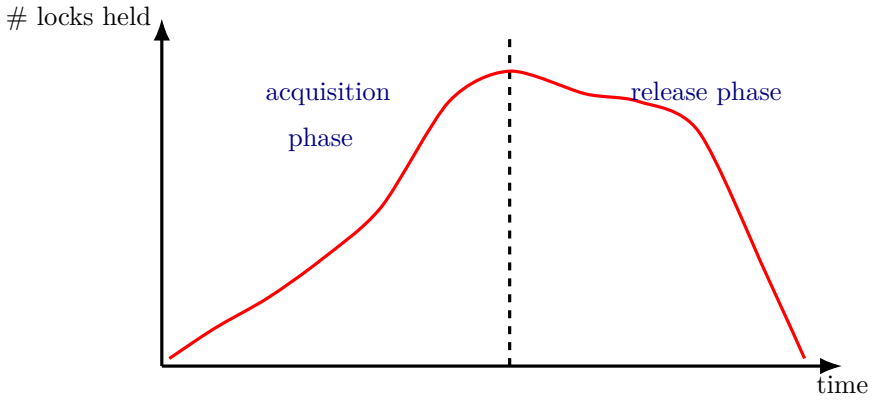


Figure 24: Two-phase locking: acquisition phase followed by release phase.

Example of legal 2PL. The following execution is allowed under 2PL:

```
T1:
lock-X(A)
write(A)
lock-S(B)
read(B)
unlock(A)
```

```
unlock(B)
commit
```

This execution is valid because the transaction first acquires all the locks it needs, and only afterward begins releasing them. Once it unlocks `A`, it never requests another new lock.

Example of illegal 2PL. The following execution is not allowed under 2PL:

```
T1:
lock-X(A)
write(A)
unlock(A)
lock-S(B)   % not allowed
read(B)
commit
```

This violates 2PL because after beginning the shrinking phase by releasing `A`, the transaction tries to acquire a new lock on `B`. That is exactly what 2PL forbids.

The importance of 2PL is that it guarantees conflict serializability. The note explains this using the idea of a transaction's lock point, which is the moment when the transaction has acquired all the locks it will ever need. Since conflicting transactions must be ordered according to their lock points, the resulting schedule is equivalent to some serial schedule. Thus, 2PL gives us a practical way to enforce correct concurrent execution.

The problem with this is that it does not prevent cascading aborts. For example,

- T_1 updates resource A and then releases the lock on A.
- T_2 reads from A.
- T_1 aborts.

In this case, T_2 must also abort because it read an uncommitted value of A.

To solve this, we will use Strict Two Phase Locking. Strict 2PL is the same as 2PL, except all locks get released together when the transaction completes.

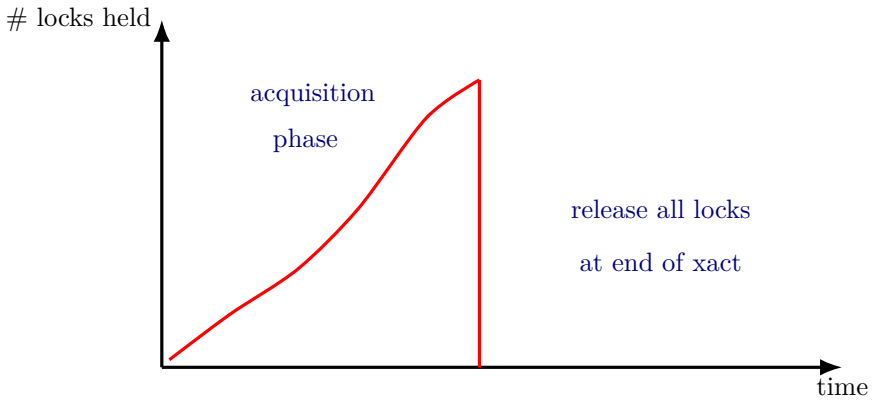


Figure 25: Strict 2PL releases all locks only at the end of the transaction.

A useful way to compare the two protocols is:

```
2PL:
acquire acquire acquire ... release release ...

Strict 2PL:
acquire acquire acquire ... commit/abort ... release everything
```

So the difference is the following:

- Under **2PL**, once a transaction releases its first lock, it cannot acquire any new locks.
- Under **Strict 2PL**, the transaction does not release any lock until the very end.

Because of this stronger rule, Strict 2PL not only guarantees conflict serializability, but also prevents other transactions from reading or overwriting uncommitted data. Therefore, Strict 2PL avoids cascading aborts and is the version more commonly used in practice.

11.5 Lock Management

Now we know what locks are used for and the types of locks. We will take a look at how the Lock Manager[2] manages these lock and unlock (or acquire and release) requests and how it decides when to grant the lock. The LM maintains a hash table, keyed on names of the resources being locked. Each entry contains a granted set (a set of granted locks/the transactions holding the locks for each resource), lock type (S or X or types we have not yet introduced), and a wait queue (queue of lock requests that cannot yet be satisfied because they conflict with the locks that have already been granted). See the following graphic:

	Granted Set	Mode	Wait Queue
A	{T1, T2}	S	T3(X) → T4(X)
B	{T6}	X	T5(X) → T7(S)

Figure 26: An example lock table entry layout.

When a lock request arrives, the Lock Manager checks if any transaction in the Granted Set or in the Wait Queue want a conflicting lock. If so, the requester gets put into the Wait Queue. If not, then the requester is granted the lock and put into the Granted Set. In addition, transactions can request a lock upgrade: this is when a transaction with shared lock can request to upgrade to exclusive. The Lock Manager will add this upgrade request at the front of the queue.

Here is some pseudocode for how to process the queue; note that it does not explicitly go over what to do in cases of promotion etc, but it is a good overview nevertheless.

```
# If queue skipping is not allowed, here is how to process the queue
H = set of held locks on A
Q = queue of lock requests for A

def request(lock_request):
    if Q is empty and lock_request is compatible with all locks in H:
        grant(lock_request)
    else:
        addToQueue(lock_request)

def release_procedure(lock_to_release):
    release(lock_to_release)
    for lock_request in Q: # iterate through the lock requests in order
        if lock_request is compatible with all locks in H:
            grant(lock_request) # grant the lock, updating the held set
        else:
            return
```

Note that this implementation does not allow queue skipping. When a request arrives under a queue skipping implementation, we first check if you can grant the lock based on what locks are held on the resource; if the lock cannot be granted, then put it at the back of the queue. When a lock is released and the queue is processed, grant any locks that are compatible with what is currently held.

11.6 Deadlock

We now have a lock manager that will put requesters into the Wait Queue if there are conflicting locks. But what happens if T_1 and T_2 both hold S locks on a resource and they both try to upgrade to X? T_1 will wait for T_2 to release the lock so that it can get an X lock while T_2 will wait for T_1 to release the lock so it can get an X lock. At this point, neither transaction will be able to get the lock because they are waiting on each other. This is called a deadlock, a cycle of transactions waiting for locks to be released by each other.

11.6.1 Avoidance

One way we can get around deadlocks is by trying to avoid getting into a deadlock. We will assign the transaction's priority by its age: `now - start time`. If T_i wants a lock that T_j holds, we have two options:

- **Wait-Die:** If T_i has higher priority, T_i waits for T_j; else T_i aborts.
- **Wound-Wait:** If T_i has higher priority, T_j aborts; else T_i waits.

[2]We will refer to the Lock Manager as LM sometimes.

Why do these prevent deadlock? A deadlock requires a cycle of waiting. For example, we could have

$$T_1 \text{ waits for } T_2, \qquad T_2 \text{ waits for } T_3, \qquad T_3 \text{ waits for } T_1.$$

This cycle is exactly what causes the deadlock.

Both **Wait-Die** and **Wound-Wait** prevent such cycles by forcing all waiting edges to go in only one direction with respect to transaction age.

In Wait-Die. A transaction is allowed to wait only if it is older than the lock holder. Thus, if T_i waits for T_j, then

$$T_i > T_j,$$

where ">" means "has higher priority" or equivalently "is older than."

Therefore, along any chain of waiting transactions, ages must strictly decrease:

$$T_1 > T_2 > T_3 > \cdots$$

Now suppose a cycle existed. Then eventually we would have to return to T_1, which would require some transaction T_k in the chain to satisfy

$$T_k > T_1.$$

But this contradicts the strictly decreasing chain that started from T_1. Hence, no cycle can exist, and therefore no deadlock is possible.

In Wound-Wait. A transaction is allowed to wait only if it is younger than the lock holder. Thus, if T_i waits for T_j, then

$$T_i < T_j.$$

Therefore, along any chain of waiting transactions, ages must strictly increase:

$$T_1 < T_2 < T_3 < \cdots$$

Again, if a cycle existed, we would eventually have to return to the starting transaction T_1. But that would require the ordering to loop back, contradicting the fact that the ages are strictly increasing along the chain. Hence, no cycle can exist, and therefore no deadlock is possible.

11.6.2 Detection

Although we avoid deadlocks in the method above, we end up aborting many transactions. We can instead try detecting deadlocks and then if we find a deadlock, we abort one of the transactions in the deadlock so the other transactions can continue. We will detect deadlocks by creating and maintaining a "waits-for" graph. This graph will have one node per transaction and an edge from T_i to T_j if:

- T_j holds a lock on resource X,
- T_i tries to acquire a lock on resource X, but T_j must release its lock on resource X before T_i can acquire its desired lock.

For example, the following graph has an edge from T_1 to T_2 because after T_2 acquires a lock on B, T_1 tries to acquire a conflicting lock on it. Thus, T_1 waits for T_2.

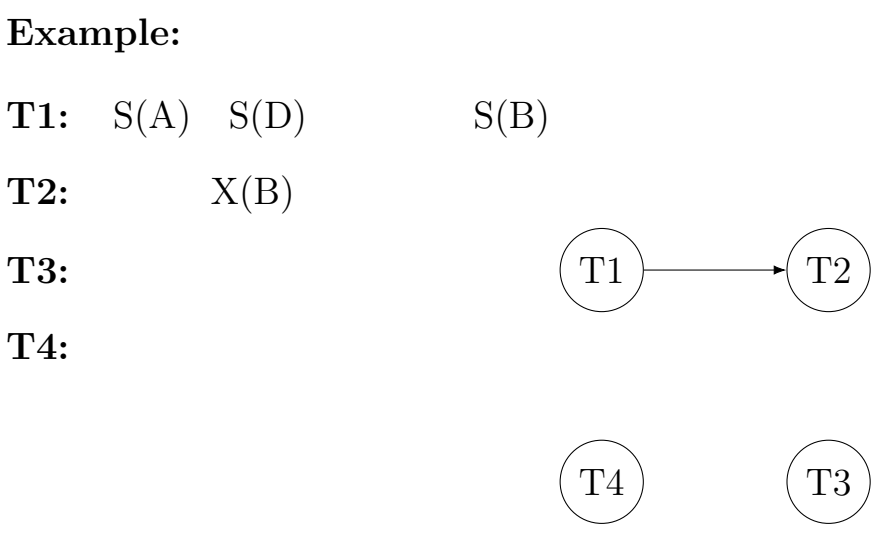


Figure 27: A waits-for graph with a single edge from T_1 to T_2.

If a transaction is waiting on another transaction (i.e. there is an edge from T_i to T_j), then T_i cannot acquire any new locks. Therefore, a transaction T_k will not wait for T_i on a resource X unless T_i had acquired a conflicting lock on X before it began waiting for T_j.

Consider the example below, while keeping in mind that only lock acquisitions are shown in schedule, not lock releases.

Example:

T1: X(A)

T2: S(A) X(B)

T3: S(B) X(A)

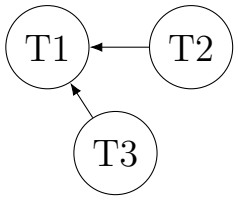


Figure 28: A waits-for graph where both T_2 and T_3 wait for T_1.

There is an edge from T_2 to T_1 because T_1 holds an X lock, when T_2 requests a conflicting S lock on resource A. Once T_2 waits for T_1 to finish with resource A, none of T_2's operations can proceed until it is removed from the wait queue. This is why T_3 does not wait for T_2 when acquiring an S lock on B, since T_2 was never actually able to acquire an X lock on B, as it was still waiting on T_1. Similarly, when T_3 goes to acquire an X lock on A, it need only wait for T_1 since at that point in time the only transaction with a conflicting lock on A is T_1. Note that at that point both T_2 and T_3 will be in the wait queue for resource A. We will periodically check for cycles in a graph, which indicate a deadlock. If a cycle is found, we will "shoot" a transaction in the cycle and abort it to break the cycle. An interesting empirical fact is that most deadlock cycles are small (2–3 transactions).

Concrete waits-for graph example. Suppose:

- T_1 holds lock A,
- T_2 holds lock B,
- T_3 holds lock C.

Now assume:

- T_1 requests B, which is held by T_2,
- T_2 requests C, which is held by T_3,
- T_3 requests A, which is held by T_1.

Then the waits-for graph contains the edges

$$T_1 \to T_2, \qquad T_2 \to T_3, \qquad T_3 \to T_1.$$

This forms the cycle

$$T_1 \to T_2 \to T_3 \to T_1,$$

which is a deadlock.

Breaking the cycle with Wait-Die. Assume the transactions are ordered by age as

$$T_1 > T_2 > T_3,$$

where ">" means "is older than."

Under Wait-Die:

- T_1 requests a lock held by T_2: since T_1 is older, T_1 waits;
- T_2 requests a lock held by T_3: since T_2 is older, T_2 waits;

- T_3 requests a lock held by T_1: since T_3 is younger, T_3 aborts.

Thus the edge $T_3 \to T_1$ is never added permanently, so the cycle is broken.

Breaking the cycle with Wound-Wait. Using the same age order $T_1 > T_2 > T_3$, under Wound-Wait:

- if an older transaction requests a lock held by a younger one, the younger one aborts;
- if a younger transaction requests a lock held by an older one, the younger one waits.

For example, when T_1 requests a lock held by T_2, T_2 is wounded and aborts immediately. Therefore, the cycle cannot form.

11.7 Lock Granularity

So now that we understand the concept of locking, we want to figure out what to actually lock. Do we want to lock the tuple containing the data we wish to write? Or the page? Or the table? Or maybe even the entire database, so that no transaction can write to this database while we are working on it? As you can guess, the decision we make will differ greatly based upon the situation we find ourselves in. Let us think of the database system as the tree below:

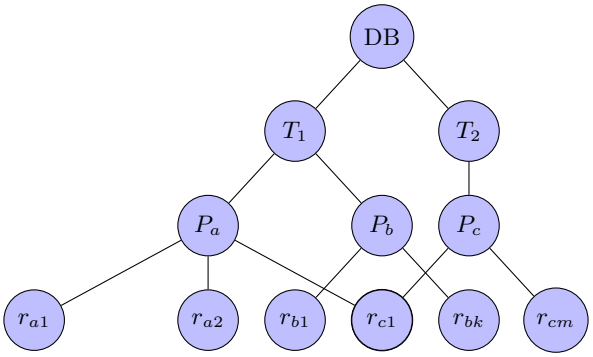


Figure 29: Hierarchy used for multigranularity locking.

The top level is the database. The next level is the table, which is followed by the pages of the table. Finally, the records of the table themselves are the lowest level in the tree. Remember that when we place a lock on a node, we implicitly lock all of its children as well (intuitively, think of it like this: if you place a lock on a page, then you are implicitly placing a lock on all the records and preventing anyone else from modifying it). So you can see how we would like to be able to specify to the database system exactly which level we would really like to place the lock on. That is why multigranularity locking is important; it allows us to place locks at different levels of the tree. We will have the following new lock modes:

- **IS:** Intent to get S lock(s) at finer granularity.
- **IX:** Intent to get X lock(s) at finer granularity. Note that two transactions can place an IX lock on the same resource – they do not directly conflict at that point because they could place the X lock on two different children. So we leave it up to the database manager to ensure that they do not place X locks on the same node later on while allowing two IX locks on the same resource.
- **SIX:** Like S and IX at the same time. This is useful if we want to prevent any other transaction from modifying a lower resource but want to allow them to read a lower level. Here, we say that at this level, I claim a shared lock; now, no other transaction can claim an exclusive lock on anything in this sub-tree (however, it can possibly claim a shared lock on something that is not being modified by this transaction – i.e. something we will not place the X lock on. That is left for the database system to handle).

Interestingly, note that no other transaction can claim an S lock on the node that has a SIX lock, because that would place a shared lock on the entire tree by two transactions, and that would prevent us from modifying anything in this sub-tree. The only lock compatible with SIX is IS. Here is the compatibility matrix below; interpret the axes as being transaction T_1 and transaction T_2. As an example, consider the entry X, S – this means that it is not possible for T_1 to hold an X lock on a resource while T_2 holds an S lock on the same resource. NL stands for no lock.

The matrix answers this question:

If one transaction already holds the row lock mode on a node, can another transaction get the column lock mode on the same node?

Mode	NL	IS	IX	S	SIX	X
NL	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
IS	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**No**
IX	**Yes**	**Yes**	**Yes**	**No**	**No**	**No**
S	**Yes**	**Yes**	**No**	**Yes**	**No**	**No**
SIX	**Yes**	**Yes**	**No**	**No**	**No**	**No**
X	**Yes**	**No**	**No**	**No**	**No**	**No**

Figure 30: Multigranularity lock compatibility matrix.

- *IS* **with** *IX*: **Yes.**

  One transaction may intend to read some descendants of the current node, while another may intend to write some descendants of that same node. This is still compatible at the current node, because the actual conflict may or may not occur lower in the hierarchy. In other words, intention locks are only announcements of future locking behavior on descendants; they do not themselves access the current node in a conflicting way.

- *IX* **with** *IX*: **Yes.**

  This often seems surprising at first. If two transactions both hold *IX* on the same node, it means that both intend to acquire exclusive locks somewhere below that node. However, they may be planning to update different descendants, so there is not necessarily a conflict at the current level. Therefore, we do not block them yet at this higher node. For example, two transactions may both hold *IX* on the same table because they intend to update different records in that table.

- *S* **with** *IX*: **No.**

  This case is very important. If one transaction holds an *S* lock on a node, then it is reading the entire node. If another transaction holds an *IX* lock on that same node, then it is announcing an intention to write some descendant of that node. This is not safe, because writing part of the node conflicts with another transaction reading the whole node. Therefore, *S* and *IX* are incompatible.

- *SIX* **with** *S*: **No.**

  Recall that *SIX* means two things at once:

  - shared access to the current node, and
  - intention to acquire exclusive locks on some descendants.

  Thus, if one transaction holds *SIX* on a node, it is reading the whole node while also planning to update parts below it. If another transaction were also allowed to hold *S* on the same node, then it would be reading the whole node at the same time that the first transaction may update some descendants. That is not allowed, so *SIX* and *S* are incompatible.

Multiple Granularity Locking Protocol

1. Each transaction starts from the root of the hierarchy.

   If you want to lock something deep in the hierarchy, you do not jump directly there. You begin at the root and move downward. So if you want a record lock, you first lock: database, then table, then page, then record with appropriate modes.

2. To get S or IS lock on a node, must hold IS or IX on parent node.

   If you are going to read a node or read something below it, your parent must already say: "I intend shared access below," or "I intend exclusive access below." So before taking S on a record, you need intention lock(s) on its ancestors.

3. To get X or IX on a node, must hold IX or SIX on parent node.

   If you are going to write a node or write below it, your parent must already announce writing intention. Writing is stronger, so the parent must carry stronger intent.

4. Must release locks in bottom-up order.

   You should unlock descendants before ancestors. That preserves the meaning of ancestor intent locks. You should not release a table-level intent lock while still holding a record lock under that table.

5. 2-phase and lock compatibility matrix rules enforced as well.
6. Protocol is correct in that it is equivalent to directly setting locks at leaf levels of the hierarchy.

   This means the protocol gives the same logical protection as if we only reasoned about the actual leaf objects being accessed. The ancestor intent locks are just a safe and efficient way to coordinate that behavior.

11.7.1 Concrete Examples of Multigranularity Locking

Example 1: A table-level read conflicts with a record-level update. Suppose the lock hierarchy is

$$\text{DB} \rightarrow \text{Table Students} \rightarrow \text{Page } P_5 \rightarrow \text{Record } r_{12}.$$

Now consider the following two transactions:

- T_1 updates record r_{12},
- T_2 reads the entire table Students.

Transaction T_1: update one record. To update r_{12}, transaction T_1 needs an X lock on that record. Following the multigranularity locking protocol, T_1 must first acquire the proper intention locks on all ancestors of r_{12}:

- acquire IX on DB,
- acquire IX on Table Students,
- acquire IX on Page P_5,
- acquire X on Record r_{12}.

Thus, T_1 holds

$$IX(\text{DB}), \qquad IX(\text{Students}), \qquad IX(P_5), \qquad X(r_{12}).$$

Transaction T_2: read the whole table. To read the entire table Students, transaction T_2 needs:

- IS on DB,
- S on Table Students.

Now check compatibility at the table node:

- T_1 already holds IX on Students,
- T_2 requests S on Students.

From the compatibility matrix,

$$IX \text{ with } S = \text{No}.$$

Therefore, T_2 must wait.

Why does T_2 have to wait? The reason is that T_1 intends to write somewhere inside the table, while T_2 wants to read the entire table. These two actions conflict. This is exactly the kind of conflict that intention locks are designed to detect efficiently at a higher level of the hierarchy.

Example 2: Two record-level updates can coexist. Now suppose:

- T_1 updates record r_{12} on page P_5,
- T_2 updates a different record r_{88} on page P_9.

In this case, both transactions may hold:

- IX on DB,
- IX on Table Students.

This is allowed because

$$IX \text{ with } IX = \text{Yes}.$$

The two transactions then continue down the hierarchy to different pages and different records. At the leaf level:

- T_1 acquires $X(r_{12})$,
- T_2 acquires $X(r_{88})$.

These locks do not conflict, because they are on different records. Hence, both transactions can proceed concurrently.

This illustrates an important benefit of multigranularity locking: we do not unnecessarily lock the entire table when only a few records are being modified.

Example 3: Using SIX. Suppose transaction T_1 wants to:

- scan the entire Students table, and
- update a few records while scanning.

Then T_1 can acquire:

- IS or IX on DB, depending on the access path,
- SIX on Table Students,
- X on each individual record that it updates.

Why use SIX? This is because T_1 needs:

- an S lock on the whole table in order to scan it, and
- an IX lock below the table in order to update some records.

That combination is exactly what SIX represents: shared access to the current node together with intention exclusive access on descendants.

Now suppose another transaction T_2 also wants an S lock on Students. The compatibility matrix says

$$SIX \text{ with } S = \text{No.}$$

Thus, T_2 cannot obtain the shared lock while T_1 holds SIX, because T_1 may modify some records while T_2 is trying to read the whole table.

A Simple Way to Remember the Lock Modes

A useful memory aid is:

- S: I read this.
- X: I write this.
- IS: I will read below.
- IX: I will write below.
- SIX: I read this and may write below.

This viewpoint often makes the multigranularity lock compatibility matrix much easier to understand.

12 Recovery

Database recovery is the component of a DBMS responsible for restoring the database to a correct state after failures. Its purpose is to preserve the **atomicity** and **durability** properties of transactions.

Informally:

- **Atomicity** means that a transaction happens all at once or not at all.
- **Durability** means that once a transaction commits, its effects persist even if the system crashes.

Recovery is necessary because a DBMS allows many transactions to execute while data pages are buffered in memory and written to disk asynchronously. If a failure occurs at the wrong time, the on-disk database may contain:

- some updates from transactions that never committed, and
- be missing some updates from transactions that did commit.

A correct recovery algorithm must fix both issues. We usually distinguish among the following failures.

Transaction Failure A single transaction may fail because of:

- deadlock resolution,
- constraint violations,
- explicit abort by the user,
- program errors.

In this case, the DBMS should undo the effects of only that transaction.

System Crash A crash may stop the DBMS or the entire machine:

- power outage,
- operating system crash,
- DBMS process failure.

In this case, main memory is lost but disk contents survive. Recovery after restart must restore correctness.

Media Failure A disk may be lost or corrupted. Recovery from media failure usually requires backups plus archived logs. In these notes, we focus primarily on **system crash recovery**.

12.1 Buffer Management Policies

A DBMS caches pages in a **buffer pool**. Transactions typically modify pages in memory first, and the buffer manager writes pages back to disk later. Two buffer management policies are especially important for recovery.

12.1.1 STEAL

Under **STEAL**, the buffer manager is allowed to write a dirty page to disk even if the transaction that modified it has not yet committed.

Consequence. The disk may contain effects of uncommitted transactions. Therefore recovery must be able to **undo** such effects.

12.1.2 FORCE

Under **FORCE**, when a transaction commits, all pages modified by that transaction must be written to disk before commit completes.

Consequence. If FORCE is used, then committed updates are guaranteed to be on disk at commit time, so crash recovery does not need to redo them.

12.1.3 The Common Choice: STEAL + NO-FORCE

Most systems use:

- **STEAL**, and
- **NO-FORCE**.

This is efficient, but it creates two recovery requirements:

- Since STEAL allows uncommitted updates on disk, we need **UNDO**.
- Since NO-FORCE allows committed updates to remain only in memory, we need **REDO**.

STEAL + NO-FORCE implies the recovery system must support both UNDO and REDO.

If we used FORCE then on every commit the DBMS would need to write all pages touched by that transaction to disk. That is expensive because:

- a transaction may touch many pages
- disk I/O is slow
- multiple transactions may repeatedly update the same page
- forcing the page now may cause unnecessary writes

If we used NO-STEAL then pages updated by uncommitted transactions could never be flushed to disk. That sounds nice for recovery, because then uncommitted changes never reach disk, so undo becomes simpler or unnecessary. But it is bad for memory management:

- dirty pages modified by active transactions must stay in memory
- long-running transactions can pin many pages
- the buffer pool can fill up
- the system may be unable to evict pages when it needs space

This can severely reduce throughput.

Suppose a long transaction dirties 10,000 pages. With NO-STEAL:

- those 10,000 pages cannot be written out
- they must remain in memory until commit or abort

That is often impractical.

12.2 Write Ahead Logging

To solve these complications we will use logging. A log is a sequence of log records that describe the operations that the database has done.

12.2.1 Update Log Record

Each write operation (SQL insert/delete/update) will get its own log `UPDATE` record.

An `UPDATE` log record looks like this:

$$\langle XID, pageID, offset, length, old\_data, new\_data \rangle$$

The fields are:

- **XID:** transaction ID — tells us which transaction did this operation
- **pageID:** what page has been modified
- **offset:** where on the page the data started changing (typically in bytes)
- **length:** how much data was changed (typically in bytes)
- **old_data:** what the data was originally (used for undo operations)
- **new_data:** what the data has been updated to (used for redo operations)

There are a few other record types we will use in our log. We will add fields to these log records throughout the note as the need for these fields becomes apparent.

- `COMMIT`: signifies that a transaction is starting the commit process
- `ABORT`: signifies that a transaction is starting the aborting process
- `END`: signifies that a transaction is finished (usually means that it finished committing or aborting)

12.2.2 WAL Requirements

Just like regular data pages, log pages need to be operated on in memory, but need to be written to disk to be stored permanently. **Write Ahead Logging (WAL)** imposes requirements for when we must write the logs to disk. In short, write ahead logging is a policy where log records describing an action such as modifying a data page or committing a transaction are flushed to disk before the actual action is flushed to disk or occurs, respectively.

The two rules are as follows:

1. Log records must be written to disk **before** the corresponding data page gets written to disk. This is how we will achieve atomicity. The intuition for this is that if a data page is written first and then the database crashes we have no way of **undoing** the operation because we do not know what operation happened!
2. All log records must be written to disk when a transaction commits. This is how we will achieve durability. The intuition is that we need to persistently track what operations a committed transaction has performed. Otherwise, we would have no idea what operations we need to redo. By writing all the logs to disk, we know exactly which operations we need to **redo** in the event that the database crashes before the modified data pages are written to disk!

12.2.3 WAL Implementation

To implement write ahead logging we are going to add a field to our log records called the **LSN**, which stands for Log Sequence Number. The LSN is a unique increasing number that helps signify the order of the operations (if you see a log record with `LSN = 20` then that operation happened after a record with `LSN = 10`). In this class the LSNs will increase by 10 each time, but this is just a convention.

We will also add a **prevLSN** field to each log record which stores the last operation from the same transaction (this will be useful for undoing a transaction).

The database will also keep track of the **flushedLSN** which is stored in RAM. The `flushedLSN` keeps track of the LSN of last log record that has been flushed to disk. When a page is **flushed**, it means that the page has been written to disk; it usually also implies that we evict the page from memory because we do not need it there anymore. The `flushedLSN` tells us that any log records before it should not be written to disk because they are already there. Log pages are usually appended to the previous log page on disk, so writing the same logs multiple times would mean we are storing duplicate data which would also mess up the continuity of the log.

flushedLSN is the largest LSN such that all log records up to that point have been written to disk. So if:

$$flushedLSN = 20$$

that means log records with LSN 10 and 20 are definitely on disk, but maybe not 30 yet. flushedLSN is about the log, not data pages. It answers: "How far into the log have we safely persisted?"

We will also add a piece of metadata to each data page called the **pageLSN**. The `pageLSN` stores the LSN of the operation that last modified the page. We will use this to help tell us what operations actually made it to disk and what operations must be redone.

Each data page stores its own `pageLSN`. For a page P, `pageLSN`(P) means:

> the LSN of the most recent update whose effects are reflected on this page

So if page $P5$ has

$$\texttt{pageLSN}(P5) = 40$$

that means the contents of page $P5$ already include the effects of log record 40, and therefore also earlier relevant updates.

Important. `pageLSN` is about a data page, not the log as a whole. It answers:

> "What is the latest logged update already applied to this page?"

Inequality Exercise Before page i is allowed to be flushed to disk, what inequality must hold?

$$pageLSN_i \_\_ flushedLSN$$

Answer: $\leq$

This comes from our first rule for WAL — **we must flush the corresponding log records before we can flush the data page to disk.** A data page is only flushed to disk if the LSN of the last operation to modify it is less than or equal to the `flushedLSN`. In other words, before page i can be flushed to disk, the log records for all operations that have modified page i must have been flushed to disk.

12.3 Aborting a Transaction

Before getting into recovering from a crash, let's figure out how a database can abort a transaction that is in progress. We may want to abort a transaction because of deadlock, or a user may decide to abort because the transaction is taking too long. Transactions can also be aborted to guarantee the C for consistency in ACID if an operation violates some integrity constraint. Finally, a transaction may need to be aborted due to a system crash! We need to ensure that none of the operations are still persisted to disk once the abort process finishes.

12.3.1 Abort and CLR Log Records

The first thing we will do is write an `ABORT` record to the log to signify that we are starting the process. Then we will start at the last operation in the log for that transaction. We will undo each operation in the transaction and write a `CLR` record to the log for each undone operation.

A **CLR** (Compensation Log Record) is a new type of record signifying that we are undoing a specific operation. It is essentially the same thing as an `UPDATE` record (it stores the previous state and the new state), but it tells us that this write operation happened due to an abort.

12.4 Recovery Data Structures

We will keep two tables of state to make the recovery process a little bit easier.

The first table is called the **transaction table** and it stores information on the active transactions. The transaction table has three fields:

- **XID:** transaction ID
- **status:** either running, committing, or aborting
- **lastLSN:** the LSN of the most recent operation for this transaction, so lastLSN helps us track the progress of a transaction.

An example of the transaction table is here:

Table 5: Example Transaction Table

XID	Status	lastLSN
1	R	33
2	C	42

If transaction $T1$ aborts, where do we begin undoing? We begin from its most recent log record:

$$lastLSN(T1)$$

Then we follow prevLSN backward.

The other table we maintain is called the **Dirty Page Table (DPT)**. The DPT keeps track of what pages are dirty. This information will be useful because it will tell us what pages have operations that have not yet made it to disk. The DPT only has two columns:

- **Page ID**
- **recLSN:** the first operation to dirty the page. so recLSN helps us track when a page first became dirty.

An example of the DPT is here:

Table 6: Example Dirty Page Table

PageID	recLSN
46	11
63	24

$$recLSN(P)$$

means:

the LSN of the first update that made page P dirty since the page was last flushed to disk.

This is very important in recovery. A page becomes dirty when it is modified in memory and that version has not yet been written back to disk. **So recLSN remembers the earliest possibly-missing update for that page.**

One thing to note is that both of these tables are stored in memory; so when recovering from a crash, you will have to use the log to reconstruct the tables. We will talk about a way to make this easier (checkpointing) later in the note.

More Inequality Questions

1. Fill in the equality below to enforce the WAL rule that all the logs must be flushed to disk before a transaction T can commit:

$$flushedLSN \_\_ lastLSN_T$$

Answer: $\geq$

If the `flushedLSN` is greater than the last operation of the transaction then we know all of the logs for that transaction are on disk.

2. For a page P that is in the DPT, fill in the following inequality for what must always be true:

$$recLSN_P \_\_ \text{in-memory } pageLSN_P$$

Answer: $\leq$

If a page is in the dirty page table then it must be dirty, so the last update must not have made it to disk. The `recLSN` is the first operation to dirty the page, so it must be smaller than the last operation to modify that page.

1. For a page P that is in the DPT, fill in the following inequality for what must always be true:

$$recLSN_P \_\_ \text{on-disk } pageLSN_P$$

Answer: $>$

If the page is dirty then the operation that dirtied the page (`recLSN`) must not have made it to disk, so it must have came after the operation that did make it to disk for that page.

12.5 Undo Logging

We've covered a lot of background information on how the database writes to its log and how it aborts transactions when it is running normally. Now, let's finally get into the reason for all this logging — recovering from failures. One possible recovery mechanism is **Undo Logging**. Note that Undo Logging does not, in fact, use write ahead logging (WAL) that we previously discussed (we will get back to that in a bit). Further, it uses the force and steal mechanisms in terms of buffer pool management.

The high level idea behind Undo Logging is that we want to undo the effects of all the transactions that have not yet been committed, while not doing so for those that have.

In order to do this, we establish 4 types of records: `Start`, `Commit`, `Abort` and `Update` (which contains old value). We also need to establish 2 rules regarding how we do logging and when to flush dirty data pages to disk:

1. If a transaction modifies a data element, then the corresponding update log record must be written to the disk before the dirty page containing it is written. We want this because we would like to ensure the old value is recorded on the disk before the new value replaces it permanently.
2. If a transaction commits, then the pages that are modified must be written to disk before the commit record itself is written to disk. That rule ensures that all changes that are made by the transaction have been written to the disk before the transaction itself actually commits. This is important because if we see the commit log record in the log, then we will consider the transaction as committed and will not undo its changes during recovery. Notice that this is different from write ahead logging; here, the dirty pages are written to the disk before writing the commit record itself to the disk.

Notice that the first rule implements the steal policy, since the dirty pages are written to the disk before a transaction commits, and the second rule implements the force policy.

Now that we have established these rules, we can discuss recovery with undo logging. When the system crashes, we first run the recovery manager. We scan the log from the end to determine whether each of the transactions is completed or not. The action that we take based on the log record we encounter is as follows:

1. `COMMIT/ABORT T`: mark T as completed
2. `UPDATE T, X, v`: if T is not completed, write $X = v$ to disk, else ignore
3. `START T`: ignore

And how far do we need to go? All the way to the start, unless we have checkpointing (discussed later).

Let us do a concrete example. Suppose the log is:

```
1. START T1
2. UPDATE T1, A, 100
3. START T2
4. UPDATE T2, B, 50
5. COMMIT T1
```

Then the system crashes.

Assume:

- `UPDATE T1, A, 100` means $T1$ changed A, and the old value was 100
- `UPDATE T2, B, 50` means $T2$ changed B, and the old value was 50

Now recovery scans the log from the end.

- **Step 1: Read `COMMIT T1`**
  Mark $T1$ as completed.
  $$completed = \{T1\}$$
- **Step 2: Read `UPDATE T2, B, 50`**
  Is $T2$ completed? No. So undo it:
  $$B := 50$$
- **Step 3: Read `START T2`**
  Ignore.
- **Step 4: Read `UPDATE T1, A, 100`**
  Is $T1$ completed? Yes. So ignore it.
- **Step 5: Read `START T1`**
  Ignore.

12.6 Redo Logging

Now, let's talk about another form of logging based recovery called **Redo Logging**. Here, Redo Logging implements the no-force no-steal strategy for buffer management. In Redo Logging, we have the same type of log records as Undo Logging, though only difference is for the `Update` log record, where instead of storing the previous value for particular data elements, we store the new value it is going to write.

The idea behind Redo Logging is similar to Undo Logging, except that at recovery time instead of undoing all the transactions that are incomplete, we redo the actions of all the transactions that were committed instead. Meanwhile, we leave all the uncommitted transactions alone.

Like Undo Logging, we also have also have a rule to abide by:

1. If a transaction modifies a data element X, then both the update record and commit record must be written to disk before dirty data page itself — this is the no-steal policy. Hence, the dirty data page is written later than the transaction commit record, and is essentially write ahead logging.

Recovery for Redo Logging is rather straightforward: we just read the log from the beginning, and redo all updates of committed transactions. While this may seem like a lot of operations, it can be optimized, like Undo Logging, through checkpointing.

12.7 ARIES Recovery Algorithm

When a database crashes, the only things it has access to are the logs that made it to disk and the data pages on disk. From this information, it should restore itself so that all committed transactions' operations have persisted (durability) and all transactions that didn't finish before the crash are properly undone (atomicity).

The recovery algorithm consists of 3 phases that execute in the following order:

1. **Analysis Phase:** reconstructs the Xact Table and the DPT
2. **Redo Phase:** repeats operations to ensure durability
3. **Undo Phase:** undoes operations from transactions that were running during the crash to ensure atomicity

Let's go through each phase in detail.

12.7.1 Analysis Phase

The entire purpose of the analysis phase is to rebuild what the Xact Table and the DPT looked like at the time of the crash. To do this, we scan through all of the records in the log beginning from the start. We modify our tables according to the following rules:

- On any record that is not an `END` record: add the transaction to the the Xact Table (if necessary). Set the `lastLSN` of the transaction to the LSN of the record you are on

- If the record is a `COMMIT` or an `ABORT` record, change the status of the transaction in the Xact Table accordingly
- If the record is an `UPDATE` record, if the page is not in the DPT add the page to the DPT and set `recLSN` equal to the LSN
- If the record is an `END` record, remove the transaction from the Xact Table.

At the end of the analysis phase, for any transactions that were committing we will also write the `END` record to the log and remove the transaction from the Xact Table. Additionally, any transactions that were running at the time of the crash need to be aborted and the abort record should be logged.

One problem with the analysis phase so far is that it requires the database to scan through the entire log. In production databases this is not realistic as there could be thousands or millions of records. To speed up the analysis phase, we will use **checkpointing**. Checkpointing writes the contents of the Xact Table and the DPT to the log. This way, instead of starting from the beginning of the log, we can start from the last checkpoint.

For the rest of this note, we consider a variant of checkpointing called **fuzzy checkpointing** that actually writes two records to the log, a `<BEGIN_CHECKPOINT>` record that says when checkpointing started and an `<END_CHECKPOINT>` record that says when we finished writing the tables to the log. The tables written to the log can be the state of tables at any point between the `<BEGIN_CHECKPOINT>` and `<END_CHECKPOINT>`. This means we need to start at the `<BEGIN_CHECKPOINT>` because we're not sure if the records after it are actually reflected in the tables that were written to the log.

12.7.2 Analysis Phase Example

This figure contains:

- a log with records:
  - 10: T1 updates P3
  - 20: T1 updates P1
  - 30: T2 updates P2
  - 40: T3 updates P1
  - 50: Begin Checkpoint
  - 60: T3 updates P3
  - 70: T3 Aborts
  - 80: End Checkpoint
  - 90: CLR undo T3 LSN 60
  - 100: T1 updates P4
  - 110: T1 commits
  - 120: T1 Ends
- a Transaction Table at checkpoint:

Transaction	Status	lastLSN
$T1$	*running*	20
$T2$	*running*	30
$T3$	*running*	40

- a Dirty Page Table at checkpoint:

Page ID	recLSN
$P1$	40
$P3$	10

Figure 31: Analysis Phase Example

So the fact that $P2$ is absent from the DPT in the checkpoint is itself evidence that it had been dirty earlier but was already flushed by then.

The database crashed and we are given the log above. The Xact Table and the DPT on the right are the tables found in the `<END_CHECKPOINT>` record. Let's go through the process.

First, we start at the record at LSN 60 because it is the record immediately after the begin checkpoint record. This is an `UPDATE` record and T3 is already in the Xact Table, so we will update the `lastLSN` to 60. The page it updates is already in the DPT, so we do not have to do anything with the DPT.

Now we go to the record at LSN 70. It is an `ABORT` record, so we need to change the status in our Xact Table to Aborting and update the `lastLSN`.

There is nothing to do for the end checkpoint record, so we move onto the `CLR` (UNDO) at LSN 90. T3 is in the Xact Table so we update the `lastLSN` and the page it is modifying (P3) is already in the DPT so we again do not have to modify the DPT.

At LSN 100 we have another update operation and T1 is already in the Xact Table so we will update its `lastLSN`. The page this record is updating is not in the DPT, however, so we will add it with a `recLSN` of 100 because this is the first operation to dirty the page.

Next is LSN 110 which is a `COMMIT` record. We need to change T1's status to committing and update the `lastLSN`.

Finally, LSN 120 is an `END` record meaning that we need to remove T1 from our Xact Table. This leaves us with a final answer of:

Table 7: Final Tables After Analysis Example

Transaction	Status	lastLSN
T2	Running	30
T3	Aborting	90

PageID	recLSN
P1	40
P3	10
P4	100

So even though $T1$ has ended, page $P4$ can still remain in the DPT if its updated contents have not yet been flushed to disk.

12.7.3 Redo Phase

The next phase in recovery is the Redo Phase which ensures durability. We will repeat history in order to reconstruct the state at the crash. We start at the smallest `recLSN` in the DPT because that is the first operation that may not have made it to disk.

We will redo all `UPDATE` and `CLR` operations unless one of the following conditions is met:

- The page is not in the DPT. If the page is not in the DPT it implies that all changes (and thus this one!) have already been flushed to disk.
- `recLSN` > `LSN`. This is because the first update that dirtied the page occurred after this operation. This implies that the operation we are currently at has already made it to disk, otherwise it would be the `recLSN`.
- `pageLSN(disk)` $\geq$ `LSN`. If the most recent update to the page that made it to disk occurred after the current operation, then we know the current operation must have made it to disk.

12.7.4 Redo Example

The log and final tables from the analysis example have been reproduced for your convenience. Now let's answer the following two questions:

1. Where should we start the recovery process?

   Answer: At LSN 10 because that is the smallest `recLSN` in the DPT.

2. What are the LSNs of the operations that get redone?

   Answer:

   - 10 — UPDATE that does not meet any of the conditions
   - Not 20 — `recLSN` > LSN
   - Not 30 — page not in DPT
   - 40 — UPDATE that does not meet any of the conditions
   - Not 50 — only redo UPDATEs and CLRs
   - 60 — UPDATE that does not meet any of the conditions
   - Not 70 — only redo UPDATEs and CLRs
   - Not 80 — only redo UPDATEs and CLRs
   - 90 — CLR that does not meet any of the conditions
   - 100 — UPDATE that does not meet any of the conditions

- Not 110 — only redo UPDATEs and CLRs
- Not 120 — only redo UPDATEs and CLRs

For a final answer of **10, 40, 60, 90, 100**.

12.7.5 Undo Phase

The final phase in the recovery process is the undo phase which ensures atomicity. The undo phase will start at the end of the log and works its way towards the start of the log. It undoes every `UPDATE` (only `UPDATE`s!) for each transaction that was active (either running or aborting) at the time of the crash so that we do not leave the database in an intermediate state. It will not undo an `UPDATE` if it has already been undone (and thus a `CLR` record is already in the log for that `UPDATE`).

For every `UPDATE` the undo phase undoes, it will write a corresponding `CLR` record to the log. `CLR` records have one additional field that we have not yet introduced called the **undoNextLSN**. The `undoNextLSN` stores the LSN of the next operation to be undone for that transaction (it comes from the `prevLSN` of the operation that you are undoing). Once you have undone all the operations for a transaction, write the `END` record for that transaction to the log.

12.7.6 Undo Example

Write down all of the log records that will be written during the undo phase.

Answer: First recognize that the log provided is missing one record from the analysis phase. Remember that at the very end of the analysis phase we need to write log entries for any aborting transactions. Therefore, there should be an `ABORT` record at LSN 130 that aborts T2. It will have a `prevLSN` of 30 because that is the last operation that T2 did before this `ABORT` operation. We include this record in the final answer for completeness, but note that it is not technically written during the Undo Phase, it is written at the end of the analysis phase.

We now move on to undoing the operations for T2 and T3. The most recent update for T3 occurs at LSN 60, but notice that there is already a CLR for that operation in the log (LSN 90). Because that operation is undone, we do not need undo it again. The next operation is the `UPDATE` at LSN 40. This update is not undone anywhere else in the log so we do need to undo it and write the corresponding CLR record. The `prevLSN` will be 90 because that CLR log record is the last operation for T3. The `undoNextLSN` will be null because there are no other operations in T3 to undo. Because there are no more actions to undo for T3, we must also write the `END` record for that transaction.

The next operation we need to undo is the update at LSN 30 for T2. The `prevLSN` for this record will be 130 because of the `ABORT` record we wrote before. The `undoNextLSN` will be null again because there are no other operations for T2 in the log. We will also need to write the `END` record for T2 because that was last operation we needed to undo. Here is the final answer:

Table 8: Log Records Written During Undo Example

LSN	Record	prevLSN	undoNextLSN
130	ABORT T2	30	
140	CLR Undo T3: LSN 40	90	null
150	END T3	140	
160	CLR Undo T2: LSN 30	130	null
170	END T2	160	