SQL Basics: Structured Notes

Generated on May 8, 2025

Contents

1 Introduction to Data Management

1.1 Overview

The field of data management encompasses the methods and systems for storing, organizing, and retrieving data in a database system.

Definition 1.1 (Data Model)**.** An abstraction for describing and representing data. A data model comprises three components:

- **Structure**: how data is organized (e.g., tables, documents).
- **Constraints**: rules that restrict the data (e.g., domains, keys).
- **Manipulation**: operations supported to query or update the data.

2 Important Data Models

2.1 Relational Data Model

Definition 2.1 (Relational Data Model)**.** Data organized into tables (relations) with rows (tuples) and columns (attributes). Tables are linked via foreign keys, suitable for structured data with predefined schemas.

- **Examples**: MySQL, PostgreSQL, Microsoft SQL Server, Oracle Database.
- **Use Cases**: transaction processing, BI/data warehousing.

2.2 Semistructured Data Model

Definition 2.2 (Semistructured Data Model)**.** Data that does not adhere to a fixed schema, represented in formats like XML or JSON.

2.3 Key-Value Data Model

Definition 2.3 (Key-Value Data Model)**.** Data stored as unique keys mapping to values.

2.4 Graph Data Model

Definition 2.4 (Graph Data Model)**.** Data represented as nodes and edges, ideal for complex relationships.

2.5 Array/Matrix Data Model

Definition 2.5 (Array/Matrix Data Model)**.** Data stored as multi-dimensional arrays or matrices.

2.6 DataFrame Data Model

Definition 2.6 (DataFrame Data Model)**.** Two-dimensional table with heterogeneous types, supporting relational and linear-algebra operations.

3 The Relational Data Model

Definition 3.1 (Relation (Table))**.** A relation is a set of tuples sharing the same attributes.

Definition 3.2 (Attribute)**.** An atomic, typed column in a relation, with a name and domain.

Example 3.1 (Student Relation)**.**

sid	**name**	**surname**	**age**	**gpa**
1	Alicia	Shan	20	3.5
2	Andre	Lorde	21	3.0
3	Yan	Ke	19	4.0
4	Sudip	Roy	22	4.0

4 Integrity Constraints

Definition 4.1 (Integrity Constraint)**.** Conditions specified on a schema that restrict allowable data in instances.

Definition 4.2 (Domain Constraint)**.** Limits attribute values to a defined domain (e.g., INT, VARCHAR).

Definition 4.3 (Key Constraint)**.** A minimal set of attributes that uniquely identifies each tuple.

Definition 4.4 (Primary Key)**.** The selected candidate key designated to uniquely identify tuples.

Definition 4.5 (Foreign Key Constraint)**.** An attribute or set whose values must match primary key values in another relation.

Example 4.1 (Enrolled Relation)**.**

course_id	**sid**	**grade**
DSC100	1	92
DSC080	2	90
DSC100	5	95 (* violates FK)

5 Normalization: First Normal Form

A relation is in First Normal Form (1NF) if all attributes are atomic.

Example 5.1 (Violation of 1NF)**.**
```
sid | name | surname | age | Enrolled
--- | ---- | ------- | --- | ----------------
1   | Alicia | Shan  | 20  | [(DSC100,97),(DSC080,90)]
```

Splitting into Student(sid,name,surname,age) and Enrolled(course_id,sid,grade) satisfies 1NF.

6 DataFrame Data Model Details

DataFrames support both relational operators and linear algebra, allow heterogeneous types, and explicit labels.

7 Overview of SQL

Definition 7.1 (SQL)**.** Structured Query Language, a declarative language for querying and managing relational databases.

8 SQL History

Developed at IBM in 1974 by Donald D. Chamberlin (originally SEQUEL), standardized in the 1980s.

9 Declarative vs. Imperative

Definition 9.1 (Declarative Language)**.** Specifies what data is desired without how to compute it.

Definition 9.2 (Imperative Language)**.** Specifies a sequence of steps to achieve a result.

10 Basic SQL Queries

The general form:

```
SELECT [DISTINCT] <column_list>
FROM <table_list>
WHERE <predicate>;
```

10.1 Projection

Example 10.1 (Projection).

```
SELECT name, genre
FROM Movies;
```

10.2 Selection

Example 10.2 (Selection).

```
SELECT *
FROM Movies
WHERE year > 2000;
```

10.3 Selection with Projection

Example 10.3 (Selection and Projection).

```
SELECT name
FROM Movies
WHERE year > 2000;
```

10.4 Joins via WHERE

Example 10.4 (Inner Join via WHERE).

```
SELECT DISTINCT m.genre
FROM Movie m, ActedIN a
WHERE m.name = a.moviename;
```

11 Joins in SQL

11.1 Overview

Joins combine rows from two or more tables based on a related column between them.

11.2 Inner Join

Definition 11.1 (Inner Join)**.** A join that returns rows when there is at least one match in both tables.

Example 11.1 (Inner Join Example).

```
SELECT DISTINCT genre
FROM   Movie
JOIN   ActedIN ON Movie.name = ActedIN.moviename
WHERE  ActedIN.actorname = 'Marlon Brando';
```

11.3 Self Join

Definition 11.2 (Self Join)**.** A join of a table to itself using table aliases.

Example 11.2 (Employee-Manager Self Join)**.**

```
SELECT e1.Name AS Employee,
       e2.Name AS Manager
FROM   Employees AS e1
JOIN   Employees AS e2 ON e1.ManagerID = e2.ID;
```

Example 11.3 (Crime and War Genres Self Join)**.**

```
SELECT DISTINCT z.actorname
FROM   Movie AS x
JOIN   ActedIN AS z ON x.name = z.moviename
JOIN   ActedIN AS w ON z.actorname = w.actorname
JOIN   Movie AS y ON w.moviename = y.name
WHERE  x.genre = 'Crime'
  AND  y.genre = 'War';
```

11.4 Outer Joins

Definition 11.3 (Left Outer Join)**.** Returns all rows from the left table, matched rows from the right; NULLs if no match.

Example 11.4 (Left Outer Join)**.**

```
SELECT m.name,
       m.genre,
       a.actorname
FROM   Movie AS m
LEFT   JOIN ActedIN AS a ON m.name = a.moviename
WHERE  m.year > 1975;
```

Definition 11.4 (Right Outer Join)**.** Returns all rows from the right table, matched rows from the left; NULLs if no match.

Example 11.5 (Right Outer Join)**.**

```
SELECT a.actorname,
       m.name
FROM   ActedIN AS a
RIGHT  JOIN Movie AS m ON a.moviename = m.name;
```

Definition 11.5 (Full Outer Join)**.** Returns rows when there is a match in one of the tables; NULLs for non-matching side.

Example 11.6 (Full Outer Join)**.**

```
SELECT m.name,
       a.actorname
FROM   Movie AS m
FULL  JOIN ActedIN AS a ON m.name = a.moviename;
```

12 Aggregation and Grouping

12.1 Aggregate Functions

SQL provides five fundamental aggregate functions, each operating on a set of values:

- **COUNT**: Counts the number of input rows. Example:
  Example 12.1 (Count Total Rows).
  ```
  SELECT COUNT(*)
  FROM Movie;
  ```
- **SUM**: Computes the sum of a numeric column. Example:
  Example 12.2 (Sum of Revenues).
  ```
  SELECT SUM(revenue)
  FROM Movie;
  ```
- **MIN**: Finds the minimum value in a column. Example:
  Example 12.3 (Minimum Budget).
  ```
  SELECT MIN(budget)
  FROM Movie;
  ```
- **MAX**: Finds the maximum value in a column. Example:
  Example 12.4 (Maximum Rate).
  ```
  SELECT MAX(rate)
  FROM Movie;
  ```
- **AVG**: Calculates the average of a numeric column. Example:
  Example 12.5 (Average Rating).
  ```
  SELECT AVG(rate)
  FROM Movie;
  ```

12.2 Aggregates and NULL Values

By default, aggregate functions ignore `NULL` values. They neither contribute to sums/averages nor are counted by `COUNT(column)`. Example:

Example 12.6 (Handling NULLs).
```
SELECT COUNT(*), COUNT(revenue), SUM(revenue)
FROM Movie;
```

Here, `COUNT(*)` includes all rows, but `COUNT(revenue)` and `SUM(revenue)` ignore rows where `revenue` is NULL.

12.3 Grouping and Aggregation

The `GROUP BY` clause partitions rows into groups, allowing aggregates per group. Example: total revenue by genre for movies after 2008.

Example 12.7 (Revenue by Genre).

```
SELECT genre, SUM(revenue) AS TotalRevenue
FROM Movie
WHERE year > 2008
GROUP BY genre;
```

Multiple aggregates and grouping by multiple attributes are also supported:

Example 12.8 (Multiple Aggregates).

```
SELECT year,
       SUM(budget)   AS SumBudget,
       MAX(revenue)  AS MaxRevenue
FROM Movie
GROUP BY year;

SELECT genre, SUM(revenue - budget) AS TotalProfit
FROM Movie
GROUP BY genre, year;
```

12.4 ORDER BY Clause

The `ORDER BY` clause sorts the result set. By default, it sorts ascending; use `DESC` for descending. Example:

Example 12.9 (Sort by Profit).

```
SELECT genre, SUM(revenue - budget) AS TotalProfit
FROM Movie
GROUP BY genre
ORDER BY TotalProfit DESC;
```

12.5 HAVING Clause

`HAVING` filters groups based on aggregate conditions. Unlike `WHERE`, it operates after grouping. Example: genres with total budget over 1B.

Example 12.10 (Filter Groups).

```
SELECT genre, SUM(revenue - budget) AS TotalProfit
FROM Movie
WHERE year > 2008
GROUP BY genre
HAVING SUM(budget) > 1000000000;
```

12.6 WHERE versus HAVING

- `WHERE` filters individual rows before grouping; cannot use aggregates.
- `HAVING` filters groups after aggregation; can use aggregates.

12.7 SQL Group-By Query Syntax and Semantics

General structure:

```
SELECT <attributes>, <aggregates>
FROM <tables>
WHERE <row_conditions>
GROUP BY <grouping_attributes>
HAVING <group_conditions>;
```

Evaluation order: FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY.

12.8 Grouping, Aggregation, and Join

Aggregates can be combined with joins to compute metrics across related tables. Example: count of high-profit movies per actor per year.

Example 12.11 (Actor High-Profit Counts)**.**

```
SELECT m.year,
       a.actorname,
       COUNT(*) AS HighProfitCount
FROM Movie AS m
JOIN ActedIN AS a ON m.name = a.moviename
WHERE (m.revenue - m.budget) > 100000000
GROUP BY m.year, a.actorname;
```

13 Nested Queries

13.1 Subqueries Overview

Definition 13.1 (Subquery)**.** A query that is part of another query, used to compute intermediate results.

Definition 13.2 (Nested Query)**.** A query that contains one or more embedded subqueries, possibly in SELECT, FROM, or WHERE clauses.

Subqueries may appear in:

- `SELECT` clause (to compute a value per row)
- `FROM` clause (to treat a subquery result as a table)
- `WHERE` or `HAVING` clauses (to filter based on computed sets)

13.2 Subqueries in `SELECT`

Example 13.1 (Correlated Subquery for Genre)**.** For each actor, return the genre of the movie they acted in:

```
SELECT a.actorname,
       (SELECT m.genre
        FROM   Movie m
        WHERE  m.name = a.moviename) AS genre
FROM   ActedIN a;
```

This is a *correlated* subquery, evaluated once per row of `ActedIN`.

This form may be inefficient for large tables. It can be *unnested* into a join:

Example 13.2 (Unnested Equivalent).

```
SELECT a.actorname,
       m.genre
FROM   ActedIN a
JOIN   Movie  m ON m.name = a.moviename;
```

13.3 Subqueries in `FROM`

Example 13.3 (Subquery in FROM). Find movies with rating between 8 and 9:

```
SELECT x.name, x.rating
FROM   (SELECT *
        FROM   Movie
        WHERE  rating > 8) AS x
WHERE  x.rating < 9;
```

This non-correlated subquery can also be expressed without nesting:

```
SELECT name, rating
FROM   Movie
WHERE  rating > 8 AND rating < 9;
```

13.4 Subqueries in `WHERE`

Subqueries in `WHERE` often express existential or universal conditions.

13.4.1 Existential: `EXISTS` / `IN`

Example 13.4 (Existential with EXISTS). Find actors in any Sci-Fi movie:

```
SELECT DISTINCT a.actorname
FROM   ActedIN a
WHERE  EXISTS (SELECT 1
               FROM   Movie m
               WHERE  m.name = a.moviename
                 AND  m.genre = 'Sci-Fi');
```

Example 13.5 (Existential with IN). Equivalent using `IN`:

```
SELECT DISTINCT a.actorname
FROM   ActedIN a
WHERE  a.moviename IN (SELECT m.name
                       FROM   Movie m
                       WHERE  m.genre = 'Sci-Fi');
```

13.4.2 Universal: `ALL`, `NOT EXISTS`, `NOT IN`

Example 13.6 (Universal with ALL)**.** Movies where all actors earn >$100k:

```
SELECT m.name
FROM   Movie m
WHERE  100000 < ALL (SELECT a.salary
                     FROM   ActedIN a
                     WHERE  m.name = a.moviename);
```

Example 13.7 (Universal with NOT EXISTS)**.** Equivalent using `NOT EXISTS`:

```
SELECT m.name
FROM   Movie m
WHERE  NOT EXISTS (SELECT 1
                   FROM   ActedIN a
                   WHERE  m.name = a.moviename
                     AND  a.salary <= 100000);
```

13.5 Unnesting and Monotonicity

Nested queries with only existential conditions are *monotone* and can be unnested into joins and group-by queries. Universal quantifiers or negation yield *non-monotone* queries, which generally cannot be unnested.

Definition 13.3 (Monotone Query)**.** A query whose result set does not lose tuples when input tables gain additional rows.

Example 13.8 (Non-monotone Universal Query)**.** Retrieve actors who acted only in Action movies:

```
SELECT a.actorname
FROM   ActedIN a
WHERE  a.actorname NOT IN (SELECT a2.actorname
                           FROM   ActedIN a2
                           JOIN   Movie m2 ON a2.moviename = m2.name
                           WHERE  m2.genre <> 'Action');
```

This uses negation and is non-monotone.

14 Set Operations

14.1 Overview

Definition 14.1 (Compound Query)**.** A query that combines the results of two or more SELECT statements using set operators.

Set operators allow merging, intersecting, and differentiating result sets:

- **UNION**: combines rows from multiple queries and removes duplicates.
- **UNION ALL**: combines rows and retains duplicates.
- **INTERSECT**: returns only rows common to both queries.
- **MINUS/EXCEPT**: returns rows from the first query not present in the second.

14.2 UNION and UNION ALL

Definition 14.2 (UNION)**.** Combines results of two SELECT statements and eliminates duplicate rows. Requires same number and compatible datatypes of columns.

Example 14.1 (UNION)**.**

```
SELECT id, name
FROM   Table1
UNION
SELECT id, name
FROM   Table2;
```

Result: All distinct (id,name) pairs from both tables.

Definition 14.3 (UNION ALL)**.** Combines results of two SELECT statements and retains all duplicates.

Example 14.2 (UNION ALL)**.**

```
SELECT id, name
FROM   Table1
UNION ALL
SELECT id, name
FROM   Table2;
```

Result: All rows from both tables, including duplicates.

14.3 INTERSECT

Definition 14.4 (INTERSECT)**.** Returns rows common to both SELECT statements, eliminating duplicates.

Example 14.3 (INTERSECT)**.**

```
SELECT id, name
FROM   Table1
INTERSECT
SELECT id, name
FROM   Table2;
```

Result: (id,name) pairs present in both tables.

14.4 MINUS and EXCEPT

Definition 14.5 (MINUS)**.** In Oracle and some other systems, returns rows in the first SELECT that are not returned by the second.

Definition 14.6 (EXCEPT)**.** ANSI/SQL standard operator (supported in SQL Server, PostgreSQL, SQLite) for difference; same semantics as MINUS.

Example 14.4 (MINUS/EXCEPT)**.**

```
-- Oracle
SELECT id, name
FROM   Table1
MINUS
SELECT id, name
FROM   Table2;
```

```
-- SQL Server / PostgreSQL / SQLite
SELECT id, name
FROM   Table1
EXCEPT
SELECT id, name
FROM   Table2;
```

Result: (id,name) pairs in Table1 not in Table2.

14.5 Practical Examples

Example 14.5 (Actors in Crime OR Horror)**.** Retrieve actors who acted in at least one 'Crime' or 'Horror' movie:

```
SELECT DISTINCT actorname
FROM   Movie m JOIN ActedIN a ON m.name = a.moviename
WHERE  m.genre = 'Crime'
UNION
SELECT DISTINCT actorname
FROM   Movie m JOIN ActedIN a ON m.name = a.moviename
WHERE  m.genre = 'Horror';
```

Example 14.6 (Actors in Both Crime AND Horror)**.** Retrieve actors who acted in both 'Crime' and 'Horror' movies using INTERSECT:

```
SELECT actorname
FROM   ActedIN
WHERE  moviename IN (
  SELECT name FROM Movie WHERE genre = 'Crime'
)
INTERSECT
SELECT actorname
FROM   ActedIN
WHERE  moviename IN (
  SELECT name FROM Movie WHERE genre = 'Horror'
);
```

Example 14.7 (Actors in Action but Not Drama)**.** Retrieve actors in 'Action' movies but not in 'Drama':

```
SELECT DISTINCT actorname
FROM   Movie m JOIN ActedIN a ON m.name = a.moviename
WHERE  m.genre = 'Action'
EXCEPT
SELECT DISTINCT actorname
FROM   Movie m JOIN ActedIN a ON m.name = a.moviename
WHERE  m.genre = 'Drama';
```

15 Formal Query Languages

15.1 Relational Algebra

Definition 15.1 (Relational Algebra)**.** A procedural query language for relations, used to describe how to compute queries via algebraic operators on relations.

Relational algebra expressions are used by DBMSs to represent query evaluation plans, often depicted as trees with operators.

15.1.1 Operands

The inputs to relational algebra operators, which are relations (tables) or variables representing relations.

15.1.2 Operators

Common core operators include:

- $\sigma_{\text{condition}}(R)$: Selection — picks rows satisfying a predicate.
- $\pi_{A_1,\dots,A_n}(R)$: Projection — picks specified columns.
- $R_1 \cup R_2$: Union — set union of two relations.
- $R_1 \cap R_2$: Intersection — common tuples in both relations.
- $R_1 - R_2$: Set difference — tuples in R_1 not in R_2.
- $R_1 \times R_2$: Cartesian product — pairs each tuple of R_1 with each tuple of R_2.
- $R_1 \bowtie_C R_2$: Theta-join — selection $\sigma_C(R_1 \times R_2)$ with condition C.
- $\rho_{B_1,\dots,B_n}(R)$: Renaming — assigns new names to relations or attributes.

Example 15.1 (Selection Example)**.** Using selection to filter movies released after 2010:

$$\sigma_{\text{year}>2010}(\text{Movie})$$

Example 15.2 (Projection Example)**.** Extracting only names and genres of movies:

$$\pi_{\text{name, genre}}(\text{Movie})$$

Example 15.3 (Union Example)**.** Combining action and drama movie titles:

$$\pi_{\text{name}}(\sigma_{\text{genre}='Action'}(\text{Movie})) \cup \pi_{\text{name}}(\sigma_{\text{genre}='Drama'}(\text{Movie}))$$

Example 15.4 (Intersection Example)**.** Actors in both 'Crime' and 'Horror' movies:

$$\pi_{\text{actorname}}(\sigma_{\text{genre}='Crime'}(\text{Movie}) \bowtie \text{ActedIN}) \cap \pi_{\text{actorname}}(\sigma_{\text{genre}='Horror'}(\text{Movie}) \bowtie \text{ActedIN})$$

Example 15.5 (Set Difference Example)**.** Actors in 'Action' but not in 'Drama':

$$\pi_{\text{actorname}}(\sigma_{\text{genre}='Action'}(\text{Movie}) \bowtie \text{ActedIN}) - \pi_{\text{actorname}}(\sigma_{\text{genre}='Drama'}(\text{Movie}) \bowtie \text{ActedIN})$$

Example 15.6 (Cartesian Product Example)**.** All combinations of movie titles and actor names:

$$\text{Movie} \times \text{Actor}$$

Example 15.7 (Renaming Example)**.** Renaming Movie to M and its attributes (name, year) to (title, yr):

$$\rho_{M(title,name),\, M(yr,year)}(\text{Movie})$$

Example 15.8 (Theta-Join Example)**.** Joining movies with their actors where the movie genre is 'Sci-Fi':

$$\sigma_{\text{Movie.name}=\text{ActedIN.moviename} \wedge \text{Movie.genre}='\text{Sci}-\text{Fi}'}(\text{Movie} \times \text{ActedIN})$$

15.2 Relational Calculus

Definition 15.2 (Relational Calculus)**.** A non-procedural (declarative) query language that specifies what data to retrieve, not how to compute it. Variants include tuple relational calculus and domain relational calculus.

Relational calculus underlies the declarative nature of SQL, allowing users to focus on desired results rather than data retrieval procedures.

Example 15.9 (Tuple Relational Calculus)**.** Retrieve names of movies rated above 8:

$$\{m.name \mid m \in \text{Movie} \wedge m.rating > 8\}$$

Example 15.10 (Domain Relational Calculus)**.** Retrieve actor names in Sci-Fi genre:

$$\{a \mid \exists m(m \in \text{Movie} \wedge m.genre =' Sci - Fi' \wedge (m.name, a) \in \text{ActedIN})\}$$

15.3 Benefits of Learning Relational Algebra

Understanding relational algebra offers several advantages:

- **Foundational Knowledge**: Forms the theoretical basis for SQL and DBMS internals.
- **Query Optimization**: Insights into how queries are executed and optimized by the database engine.
- **Better Data Manipulation**: Provides a framework for reasoning about data transformations.
- **Versatility**: Concepts translate to other data manipulation tools like R's data frames or Python pandas.

16 Midterm Review

16.1 Joins

- Default join is `INNER JOIN`; always explicitly specify `LEFT/RIGHT/FULL OUTER JOIN` when needed.
- To "show all movies and their actors": use `LEFT OUTER JOIN` so movies without actors still appear.
- To "show movies which have actors": use `INNER JOIN`.
- Example schema: `Movie(name,year,genre)`, `ActedIN(actorname,moviename)`.

16.2 Aggregations

- Five basic SQL aggregate functions: `COUNT`, `SUM`, `MIN`, `MAX`, `AVG`.
- By default, duplicates are included unless `DISTINCT` is used.
- You may combine multiple aggregates in one query (e.g. `SUM(revenue) - SUM(budget)`).
- SQL does *not* permit nested aggregates (e.g. `AVG(SUM(...))` is invalid).

16.3 Grouping and Filtering

- **GROUP BY rule**: every selected column must either appear in `GROUP BY` or be wrapped in an aggregate.
- **ORDER BY**: defaults to ascending; add `DESC` for descending.
- `WHERE` filters rows *before* aggregation; `HAVING` filters groups *after* aggregation.
- **Execution order**: `FROM` → `WHERE` → `GROUP BY` → `HAVING` → `SELECT` → `DISTINCT` → `ORDER BY` → `LIMIT`.

16.4 Subqueries

Definition 16.1 (Subquery)**.** A query embedded within another query. A nested query contains one or more subqueries.

- Can appear in `SELECT`, `FROM`, or `WHERE`/`HAVING`.
- Often used for existential checks (`EXISTS`, `IN`) or universal conditions (`ALL`, `NOT EXISTS`).

16.5 Sample Questions

Example 16.1 (Q1: Max-Budget Projects)**. Problem:** Find the projects with the highest allocated budget and the managers responsible.
Tables: `Projects(pid,pname,budget,managerid)`, `Managers(mid,mname,department)`
Solution:

```
SELECT m.mname, p.pname, p.budget
FROM Managers m
JOIN Projects p
  ON m.mid = p.managerid
WHERE p.budget = (SELECT MAX(budget) FROM Projects);
```

Example 16.2 (Q2: Students in Math & Physics)**. Problem:** Print names and ages of students enrolled in both "Math" and "Physics."
Tables: `Student(sid,sname,age)`, `Enrollment(sid,cid)`, `Course(cid,cname,instructor)`
Solution (join version):

```
SELECT s.sname, s.age
FROM Student s
JOIN Enrollment e1 ON s.sid = e1.sid
JOIN Course c1     ON e1.cid = c1.cid
JOIN Enrollment e2 ON s.sid = e2.sid
JOIN Course c2     ON e2.cid = c2.cid
WHERE c1.cname='Math'
  AND c2.cname='Physics';
```

Solution (INTERSECT version):

```
SELECT DISTINCT s1.sname, s1.age
FROM Student s1
JOIN Enrollment e1 ON s1.sid = e1.sid
JOIN Course c1     ON e1.cid = c1.cid
```

```
WHERE c1.cname='Math'
INTERSECT
SELECT DISTINCT s2.sname, s2.age
FROM Student s2
JOIN Enrollment e2 ON s2.sid = e2.sid
JOIN Course c2     ON e2.cid = c2.cid
WHERE c2.cname='Physics';
```

Example 16.3 (Q3: Courses with 5+ Long-Term Enrollees)**. Problem:** Which courses have at least 5 students enrolled for >2 years, and what is the average age of those students?
Solution:

```
SELECT c.cid, c.cname, AVG(s.age) AS avg_age
FROM Course c
JOIN Enrollment e ON c.cid = e.cid
JOIN Student   s ON e.sid = s.sid
WHERE e.years_enrolled > 2
GROUP BY c.cid, c.cname
HAVING COUNT(*) >= 5;
```

Example 16.4 (Q4: Courses with Young & Old Students)**. Problem:** Which courses have at least one student aged >40 and at least one aged <18?
Solution:

```
SELECT c.cid, c.cname
FROM Course c
WHERE EXISTS (
    SELECT 1
    FROM Enrollment e
    JOIN Student s ON e.sid = s.sid
    WHERE e.cid = c.cid AND s.age > 40
)
AND EXISTS (
    SELECT 1
    FROM Enrollment e
    JOIN Student s ON e.sid = s.sid
    WHERE e.cid = c.cid AND s.age < 18
);
```

Example 16.5 (Q5: Developer ↔ Project Analysis)**. Schema:** `Developer(did,name,hireYear)`, `WorksOn(pid,did,year)`, `Project(pid,name,startYear)`.
RA expression:

$$\pi_{did,name,\,\text{count}(pid)}\Big(\gamma_{did,\,\text{count}(pid)\to\text{projCount}}\big(\\ \sigma_{year-startYear\le 2\,\wedge\,hireYear+10<\max(startYear)}\big(\text{Developer}\ \bowtie\ \text{WorksOn}\ \bowtie\ \text{Project}\big)\big)\Big)$$

Interpretation: Retrieve developers who worked on projects within 2 years of their start and whose hireYear + 10 is less than the most recent project startYear they worked on; output developer ID, name, and count of projects.

16.6 Relational Algebra & SQL Summary

- **SQL**: declarative—specify *what* you want.
- **Relational Algebra**: procedural—specify *how* to compute it.
- Both languages can express the same class of queries.