UNIVERSITY OF CALGARY

ENSF 608
Dr. Ronnie de Souza Santos, PhD.

AI in SQL Databases: Making Databases Smarter
Artificial Intelligence (AI) and SQL databases are starting to work together in exciting ways. Traditionally, SQL databases follow strict rules for storing and retrieving data. You write a query (like *SELECT * FROM STUDENT*), and the system runs it based on fixed plans designed by human experts. But as data grows larger and more complex, those old methods start to slow down. AI helps SQL systems *learn* and *adapt*, making them faster, smarter, and more secure.

How AI Improves SQL Databases

1. Smarter Query Optimization. Normally, the database guesses the "best route" to get your data, similar to how Google Maps finds the quickest path. AI can learn from past queries and automatically adjust. Example: If a query with many joins (JOINs) is running slowly, an AI model can predict a better execution plan the next time. Some systems use *reinforcement learning*, where the database "learns by trial and error" to find faster ways to execute queries.
2. Automated Indexing and Tuning. Indexes make searches faster, but deciding which columns to index is hard. AI systems can now watch query patterns and automatically create or remove indexes. Example: If students keep searching by StudentName, the AI will index that column to speed it up.
3. Predictive and Real-Time Analytics. AI models can live *inside* SQL databases. This means predictions happen as part of your SQL queries. Example: *SELECT StudentID, Predict_Grade (GPA, Attendance) AS PredictedGrade FROM Student*. Here, *Predict_Grade()* could be an AI function trained on past data, the database runs it just like any other SQL command.
4. Security and Anomaly Detection. AI can spot unusual database activity, for example, someone trying to access too many records at once (which could signal an attack). These systems learn what "normal" looks like and alert admins when something's off.
5. Hybrid SQL + Big Data Systems. Modern databases are combining the structure of SQL with the flexibility of big data platforms. AI helps manage when to use which system for speed and cost efficiency.

Example: AI-Optimized Query in Action
In one study, researchers tested queries on a database with one million records:

- Traditional SQL: took 85 seconds.
- AI-optimized SQL: took only 55 seconds, i.e., 35% faster! The AI version also used less CPU and memory, showing both performance and energy savings.

What's Next

- Self-tuning databases: systems that manage themselves with minimal human input.
- AISQL: new versions of SQL that directly include AI operators (like TRAIN MODEL, PREDICT, or EXPLAIN).
- Smarter cloud databases: that adjust resources automatically and learn from global workloads.
- **Why It Matters for You?** Learning SQL today means understanding data logic nut also think about future AI application. Tomorrow's data engineers won't just *query* databases, they'll *train* them to improve themselves.

Back to the Basics: Everyday SQL Tips
Here is a compiled list of a few simple but powerful SQL tips to help you write cleaner, clearer, and smarter queries, whether you're just starting or already feel comfortable with SQL.

1. Understand the True Order of SQL. SQL doesn't always run top to bottom the way it looks. Here's the real logical order behind a SELECT query:

1. FROM – choose the tables
2. WHERE – filter the rows
3. GROUP BY – group similar rows
4. HAVING – filter groups (use for things like HAVING COUNT(*) > 1)
5. SELECT – pick the columns or calculations
6. ORDER BY – sort your results
7. LIMIT / FETCH – show just a portion of the result

Tip: If you ever get an error saying something like "not a GROUP BY expression," remember that grouping happens before selecting — so only grouped or aggregated values can appear in your result.

2. Write Readable Code. SQL is much easier to debug when it's tidy. Example**:**

```
SELECT
   employee_id,
   employee_name,
   salary
FROM employees
WHERE department = 'IT';
```

Tips:

- Put commas before new columns (makes them easier to see).
- Use indentation for JOINs, CASE statements, and subqueries.
- Comment your code to explain why, not just what, you did:

3. Simplify Joins. If the column name is the same in both tables, use **USING** instead of a long ON clause. Example:

```
SELECT *
FROM album
JOIN artist USING (artistid);
```

Tip: It makes it shorter and avoids duplicate column names.

4. Start Practicing GROUP BY and HAVING. GROUP BY combines rows with the same values, and HAVING filters after grouping.

```
SELECT department, COUNT(*) AS total
FROM employees
GROUP BY department
HAVING COUNT(*) > 5;
```

Tip: *Think of HAVING as "WHERE for groups."*

5. Avoid Common Mistakes

- NOT IN and NULL don't mix! Use NOT EXISTS instead.
- Always match data types, don't compare text to numbers.
- Give every calculated field a unique alias: *SELECT LEFT (product, 1)* ***AS*** *product_letter*

6. Performance Tips

- NOT EXISTS is usually faster than NOT IN.
- Don't rely on implicit conversions (e.g., comparing text with numbers).
- Read the documentation, it often saves hours of debugging!

7. Keep Queries Organized Tips

- Use clear file or query names (e.g., Monthly_Sales_Report_2025-10).
- Add comments for future you (or your teammate).
- Always specify which table a column belongs to in joins.

Continue Practice

We'll continue using the **University** database. Here are the four tables we've been working with (now updated based on our last lecture):

```
CREATE TABLE STUDENT (
    StudentID INT,
    Name VARCHAR(50),
    Address VARCHAR(100),
    Age INT CHECK (Age BETWEEN 18 AND 65)
   PRIMARY KEY (StudentID),
);


CREATE TABLE PROFESSOR (
    ProfessorID INT,
    Name VARCHAR(50),
    Contract VARCHAR(30),
    Background VARCHAR(50),
    YearsExperience INT
    PRIMARY KEY (ProfessorID),
);

CREATE TABLE COURSE (
    CourseID INT PRIMARY KEY,
    Name VARCHAR(50),
    Code VARCHAR(10),
    Schedule VARCHAR(50),
    Credits INT,
    Hours INT
   PRIMARY KEY CourseID),
);

CREATE TABLE ENROLL (
    StudentID INT,
    CourseID INT,
    PRIMARY KEY (StudentID, CourseID),
    FOREIGN KEY (StudentID) REFERENCES STUDENT(StudentID),
    FOREIGN KEY (CourseID) REFERENCES COURSE(CourseID)
);


CREATE TABLE TEACHES (
    ProfessorID INT,
    CourseID INT,
    PRIMARY KEY (ProfessorID, CourseID),
    FOREIGN KEY (ProfessorID) REFERENCES PROFESSOR(ProfessorID),
    FOREIGN KEY (CourseID) REFERENCES COURSE(CourseID)
);
```

1. Add More Data

- Insert 20 more students (with realistic ages between 18–65).
- Add 10 more professors (give them different years of experience).
- Add 5 more courses (with credits and hours).
- Enroll students in courses (populate the ENROLL table).
- Assign professors to teach courses (populate the TEACHES table).

Tip: Try using both simple INSERT INTO ... VALUES (...) and multiple-row inserts.

2. **Practice Searches (SELECTS)**

1. Find the names and ages of all students.
2. Show professors and their years of experience.
3. Find all students who are 25 years old.
4. List courses with more than 3 credits.
5. Display courses with exactly 4 hours.
6. Find students who are 20 years old or younger.
7. Show professors with at least 10 years of experience.
8. Count how many students exist in the database.
9. Calculate the average age of students.
10. Show the highest and lowest credit values among courses.
11. Show all unique schedules of courses (no duplicates).
12. Find students who live in a specific city (e.g., Calgary).
13. List professors sorted by experience, from most to least.
14. Show only a limited number of rows (e.g., first 5 students).
15. Find how many professors exist in the database.
16. Find the average age of all students.
17. Show the oldest and youngest student.
18. Show the maximum and minimum number of credits among courses.
19. Find the average number of hours across all courses.
20. Count how many students are enrolled in total.
21. Count how many students are enrolled per course.
22. Find the average experience (in years) among all professors.
23. Find the maximum years of experience and the name of that professor.
24. Show the average course credits per schedule.
25. Find all students whose names start with the letter 'A'.
26. Find all students whose names contain “an” anywhere in them.
27. Find all professors whose background is NOT “Mathematics.”
28. Find all students who are older than 25 AND live in Calgary.
29. Find all students who are younger than 21 OR older than 30.
30. Find all students who do not have an address.
31. Find all courses that have more than 3 credits AND fewer than 5 hours.
32. Display the maximum credits value and rename the column as Max_Credit.
33. Show the names of all courses and rename the column to Course_Name.
34. Show all professors who teach more than one course.