Query Planning and Optimization

Dr. Qichen Wang
EPFL
2025.5

Self-introduction

- Dr. Qichen Wang
  - PhD from Hong Kong University of Science and Technology, 2022
  - Research Assistant Professor, Hong Kong Baptist University 2022-2024
  - Postdoc, EPFL, 2024-now

- Teaching experiences:
  - Lecturer: Cloud Computing, Hong Kong Baptist University
  - TA: Big Data Technology, Combinatorial Optimization, HKUST

- Teaching interests:
  - Databases, Cloud Computing, Big Data Technology, Algorithms, Data Structures
  - Other BS/MS level CS courses

Prerequisite

- Fundamental relational concepts: tables, tuples, columns, primary and foreign keys
- Relational algebra
- Basic concepts of writing SQL queries, SELECT, FROM, WHERE, different types of joins, and subqueries
- Big-O analysis for algorithmic cost

Demo Database

- Student(sid, name, state), Course(cid, title), Enrolled(sid, cid, grade)

sid	name	state
1	Alice	CA
2	Bob	NY
3	Charlie	CA
4	Diana	TX
5	Eve	CA
6	Frank	TX
7	Grace	NY

cid	title
101	Database Systems
102	Operating Systems
103	Algorithms
104	Computer Networks

sid	cid	grade
1	101	A
1	103	B
2	101	B
2	102	A
3	101	A
3	102	B
3	103	A
3	104	A
4	103	C
5	101	B
5	102	A
6	101	A
7	104	A
8	101	A

Download the demo database

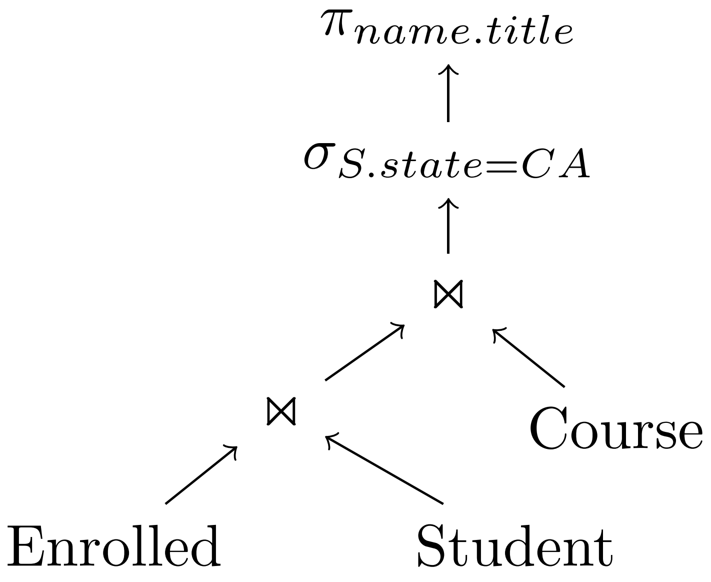
https://qichen-wang.github.io/files/demo.sql

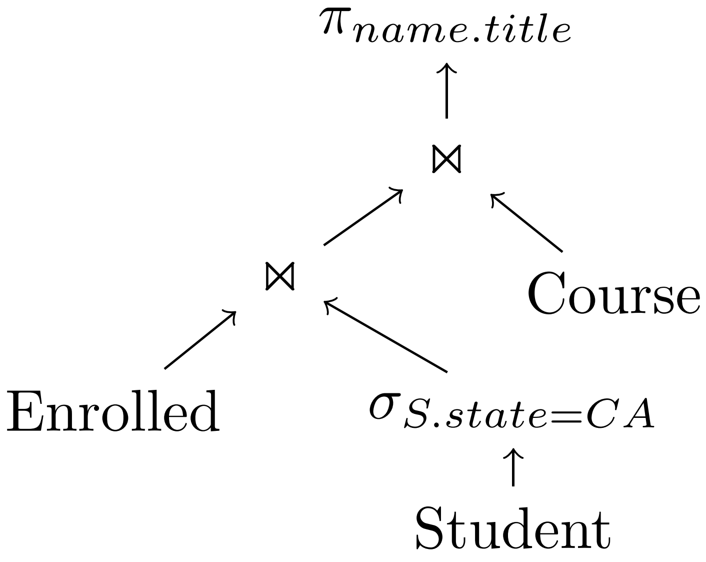
To load it:
For DuckDB:
    .read /path/to/demo.sql
For PostgreSQL:
    \i /path/to/demo.sql

SQL: A declarative language

- When writing SQL queries, we only express our high-level ideas.
- There can be different ways of evaluating the query.
  - *"Listing all students from CA and the courses they have enrolled in."*

```
SELECT name, title
FROM Student s, Course c, Enrolled e
WHERE s.sid = e.sid
AND c.cid = e.cid
AND s.state = 'CA';
```

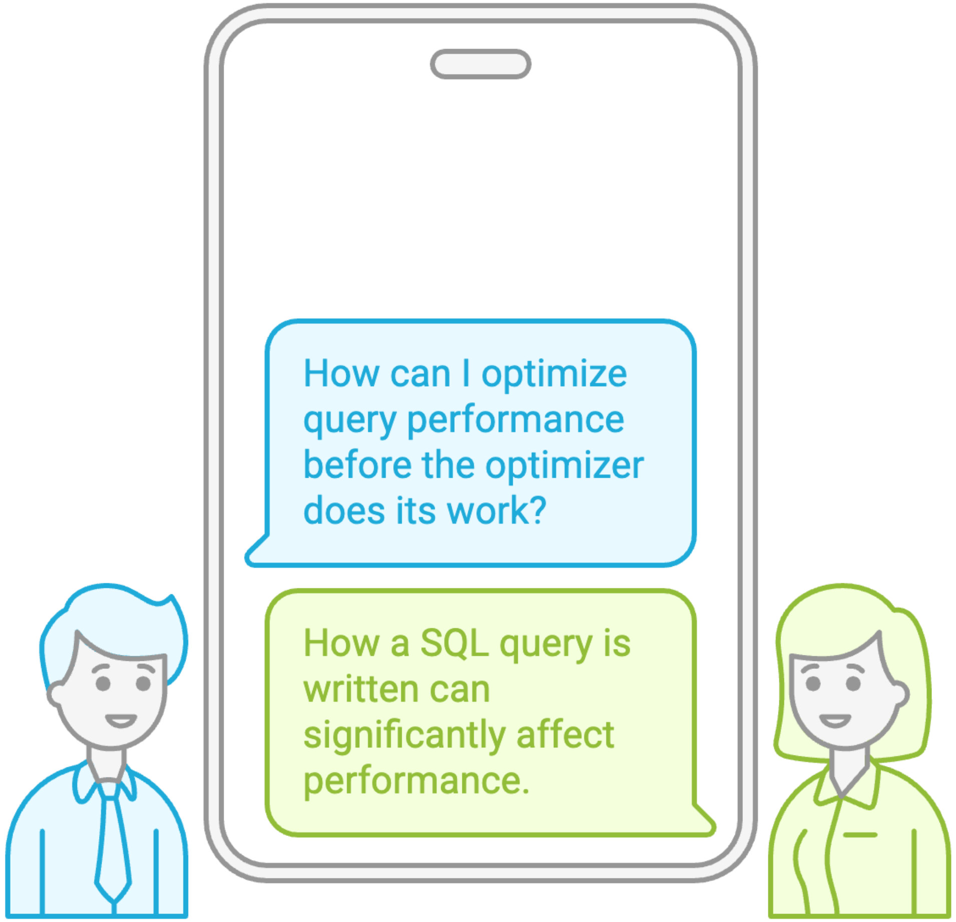




Before Optimization

The first step of optimization

- Ideally, the optimizer should do everything for you.
  - But that is not the case for current database systems.



An example:

Student(sid, name, state), Course(cid, title), Enrolled(sid, cid, grade)

- Suppose you want to find the students who have enrolled in all courses
- What will you do?
- ‘For all’ is hard to represent in SQL
- A direct translation: **Find the students for whom there are no course they have not enrolled in**.

```
SELECT sid
FROM Student s
WHERE NOT EXISTS (
      SELECT * FROM Course c
      WHERE NOT EXISTS (
            SELECT * FROM Enrolled e
            WHERE s.sid = e.sid AND c.cid = e.cid
));
```

It takes $O(n^2)$ time
Loop over all students and courses and check the Enrolled table for every possible combination.

How to do better?

Student(sid, name, state), Course(cid, title), Enrolled(sid, cid, grade)

- Suppose you want to find the students who have enrolled in all courses

- Another possible way: **Find the students whose enrolled course count matches the total number of courses in the Course table.**

```
SELECT sid
FROM Enrolled e
GROUP BY sid
HAVING count(*) = (SELECT count(*) FROM Course c);
```

Can be done in linear time $O(n)$

- **Writing a good SQL can reduce the complexity at the beginning.**

Query Optimization by Quantifier Elimination, Christoph Koch and Peter Lindner, *PODS 2024*

Some good practices you should know

- **Rule 1: Select Only Necessary Columns**
  - To avoid select * queries.
  - It is hard to find a query requiring every table column.
  - For some databases, data is stored in columnar format.
  - Selecting only required columns can significantly reduce the I/O cost.

Some good practices you should know

- **Rule 2: Remove redundant filter conditions and avoid functions in filter conditions**
  - For example, having both ”data >= 2025-01-01 and data <= 2025-12-31” and “YEAR(date) = 2025”
  - YEAR(date) = 2025 is redundant
  - Also, YEAR(date) = 2025 is not index-friendly; databases usually have indices on the range queries, but not for functions.

Some good practices you should know

- **Rule 3: Replace IN with EXISTS**
  - For some databases, the EXISTS clause often offers better performance.

```sql
SELECT name
FROM Student
WHERE state = 'CA'
 AND sid IN ( SELECT E.sid
     FROM Enrolled e, Course c
     WHERE e.cid = c.cid
     AND c.title = 'Database Systems'
     AND e.grade = 'A');
```

```sql
SELECT name
FROM Student s
WHERE EXISTS (SELECT 1
     FROM Enrolled e, Course c
     WHERE e.cid = c.cid
     AND s.sid = e.sid
     AND c.title = 'Database Systems'
     AND e.grade = 'A')
AND state = 'CA';
```

  - Some databases can optimize that for you (e.g., DuckDB) while some cannot (e.g., PostgreSQL)
  - Always use EXISTS if the right-hand side is a subquery.

Some good practices you should know

- **Rule 4: Replace unnecessary joins with semi-joins (EXISTS)**
  - Some join queries can be replaced with a semi-join if the output attributes are only located in one of the two relations.

```
SELECT DISTINCT S.name
FROM Student s, Enrolled e, Course c
WHERE S.state = 'CA'
AND C.title = 'Database Systems'
AND E.grade = 'A'
AND s.sid = e.sid AND c.cid = e.cid;
```

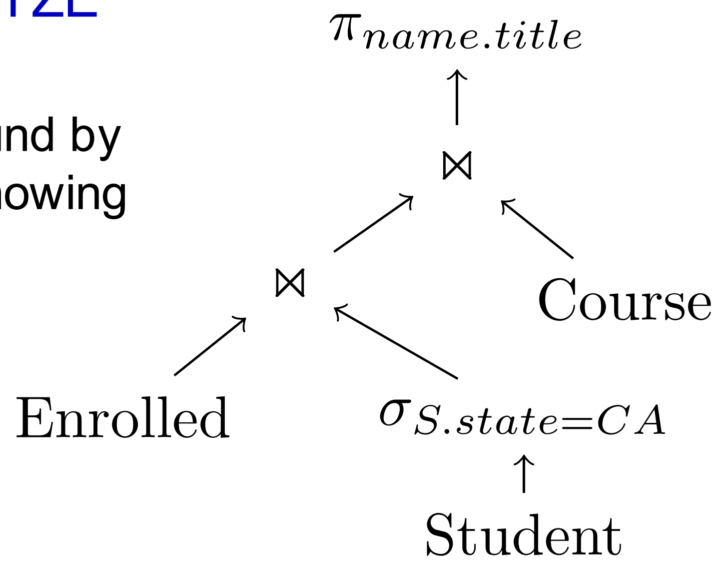
```
SELECT name
FROM Student s
WHERE EXISTS (SELECT 1
      FROM Enrolled e
      WHERE s.sid = e.sid
      AND e.grade = 'A'
      AND EXISTS (SELECT 1
            FROM Course c
            WHERE c.cid = e.cid
            AND c.title = 'Database Systems'))
AND S.state = 'CA';
```

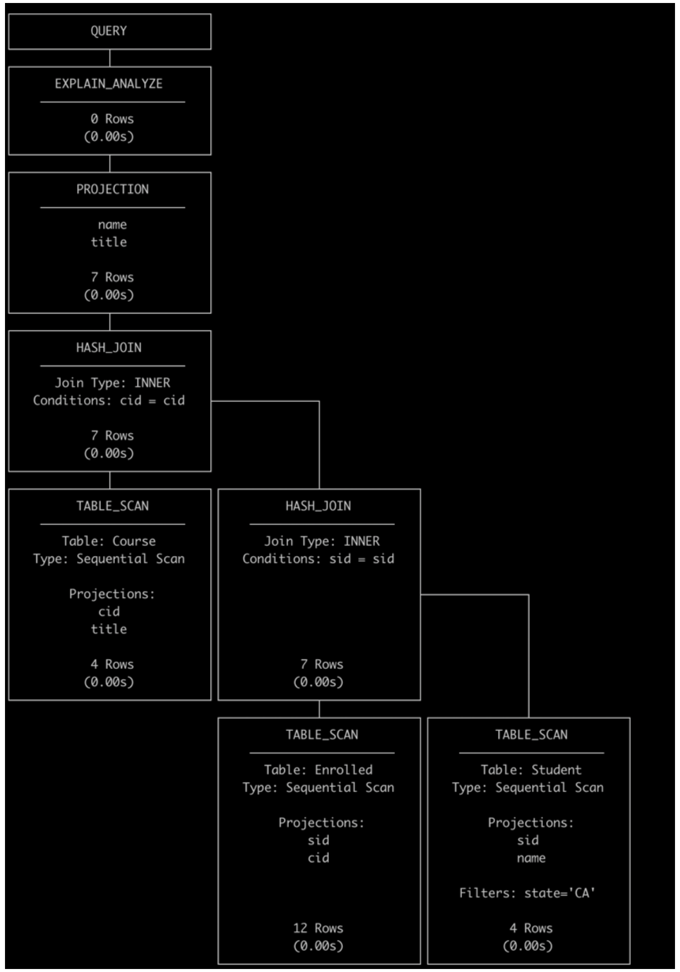
  - Avoid costly full join computation.

Viewing Query Evaluation Plans

```
EXPLAIN ANALYZE SELECT name, title
FROM Student s, Course c, Enrolled e
WHERE s.sid = e.sid
AND c.cid = e.cid
AND s.state = 'CA';
```

- Most databases support 'EXPLAIN <query>' to display the query execution plan.
  - Display plan chosen by query optimizer, along with cost estimation
- Some databases (e.g., PostgreSQL, DuckDB) support 'EXPLAIN ANALYZE <query>'
  - Shows actual runtime statistics found by running the query, in addition to showing the plan





Logical Plans and Rule-based Optimization

Logical Query Optimization

The parser first parses the SQL and generate abstract syntax tree.

The Binder first generate a logical plan for the input query

The planner then generate multiple equivalent plans for the given query

The optimizer calculate the cost for every equivalent plans and find the optimal one

- The logical plan corresponds to a relational algebra expression.
- We need to find the equivalent relational algebra expressions to find equivalent plans.

Transformation of Relational Expressions

- Two relational algebra expressions are said to be equivalent if the two expressions generate the same set of tuples on every legal database instance.
  - Note: order of tuples is irrelevant

- An **equivalence rule** says that expressions of two forms are equivalent.
  - Can replace the expression of the first form by the second, or vice versa

- It is actually hard to find all possible equivalent expressions
  - NP-hard problem

- **Practically**: Choose from a subset of all possible plans

Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.

$$\sigma_{\theta_1 \wedge \theta_2}(E) \equiv \sigma_{\theta_1}\left(\sigma_{\theta_2}(E)\right)$$

2. Selection operations are commutative.

$$\sigma_{\theta_1}\left(\sigma_{\theta_2}(E)\right) \equiv \sigma_{\theta_2}\left(\sigma_{\theta_1}(E)\right)$$

3. Only the last in a sequence of projection operations is needed, the others can be omitted.

$$\pi_{L_1}\left(\pi_{L_2}\left(\cdots\left(\pi_{L_n}(E)\right)\right)\right) \equiv \pi_{L_1}(E)$$

where $L_1 \subseteq L_2 \subseteq \cdots \subseteq L_n$

4. Join are commutative

$$E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$$

5. Natural join are associative

$$(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$$

Predicate Pushdown

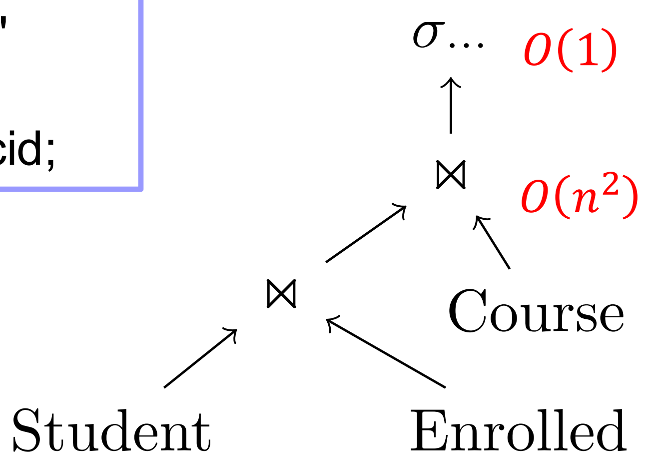
6. The selection operation can be distributed over the join operations if all the attributes in θ involve only the attributes of one of the expressions (E_1) being joined.

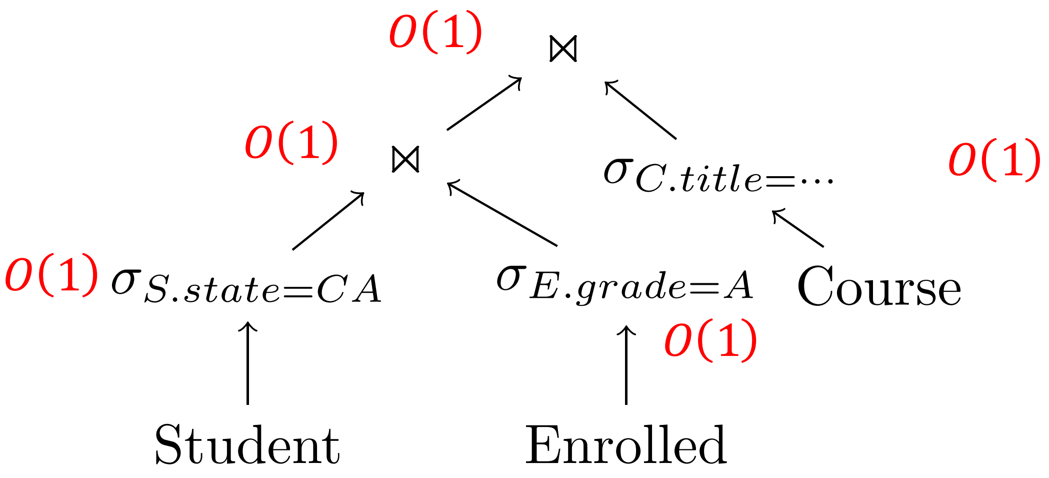
$$\sigma_\theta(E_1 \bowtie E_2) \equiv \left(\sigma_\theta(E_1)\right) \bowtie E_2$$

```sql
SELECT DISTINCT S.name
FROM Student s, Enrolled e, Course c
WHERE S.state = 'CA'
AND C.title = 'Database Systems'
AND E.grade = 'A'
AND s.sid = e.sid AND c.cid = e.cid;
```

$$\sigma_{S.state=CA \wedge C.title=Database\ Systems \wedge E.grade=A}(s \bowtie e \bowtie c)$$

$$\left(\sigma_{s.state=CA}(s)\right) \bowtie \left(\sigma_{e.grade=A}(e)\right) \bowtie \left(\sigma_{c.title=Database\ System}(c)\right)$$





Projection Pushdown

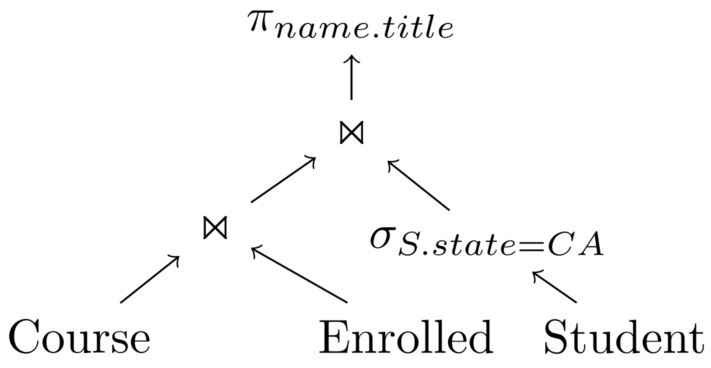
7. The projection operation distributes over the join operation as follows:

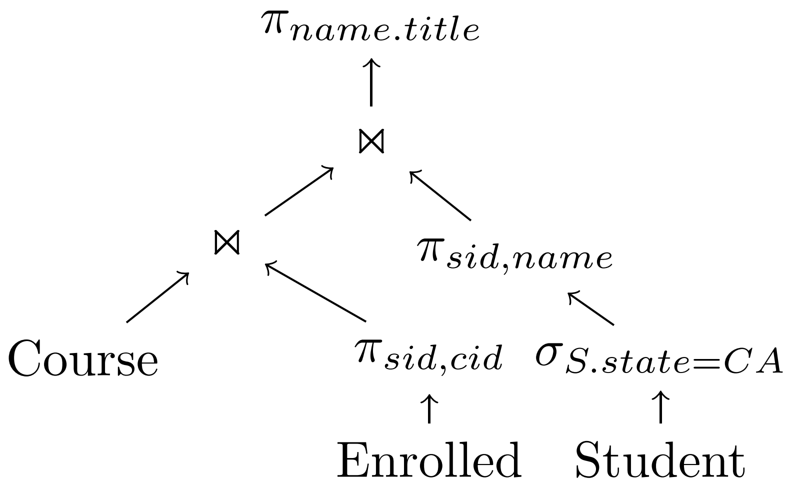
Assume L_1/L_2 only involves attributes from E_1/E_2, L_3 are the set of join attributes:

$$\pi_{L_1 \cup L_2}(E_1 \bowtie E_2) \equiv \pi_{L_1 \cup L_2}\left(\pi_{L_1 \cup L_3}(E_1) \bowtie \pi_{L_2 \cup L_3}(E_2)\right)$$

i.e., we first project all attributes in E_1/E_2 that are either not in the final output attributes, or the join attributes. After calculating the join, we remove all the non-output join attributes ($\pi_{L_1 \cup L_2}$)

```
SELECT name, title
FROM Student s, Course c, Enrolled e
WHERE s.sid = e.sid
AND c.cid = e.cid
AND s.state = 'CA';
```





Take-home exercise

- Can you use projection pushdown and the following rule

$$\left(E_1 \bowtie \left(\pi_{L_3} E_2\right)\right) \equiv E_1 \ltimes E_2$$

to find an equivalent rule for replacing joins with semi-joins?

$$\pi_{L_1}(E_1 \bowtie E_2) \equiv \left(\pi_{L_1} E_1\right) \ltimes E_2$$

  - L_1 only involves attributes from E_1
  - L_3 are the join attributes between E_1 and E_2

Heuristic Optimizations

- There are more rules (even rules that have not been discovered yet).
- These techniques **do not need** to examine data.
  - Predicate pushdown
  - Projection pushdown
- Idea: drop unused data as much as possible and as early as possible without affecting the efficiency
- Provide a much better starting point for the next stage of optimization.

Cost-based Optimization

Cost-based Query Optimization

- The efficiency of a query plan depends on multiple factors:
  - CPU time
  - I/O operations
  - Memory usage
  - Cache misses
- Cost Model: a weighted formula that combines all these factors:

$$c_1(CPU\ Ops) + c_2(I/O\ Ops) + \cdots$$

  - The constants $c_1, c_2, \cdots$ depend heavily on hardware
  - They are determined by the database system.
  - The formula can be simpler or more complicated.
- Also, heavily depends on the output size of each operator, which determine the number of CPU and I/O operations

Cost Estimation

- Need statistics of input relations.
  - E.g., number of tuples, sizes of tuples
- Need to estimate the statistics of expression results
  - Can work as the input of another expression
  - To do so, we require additional statistics
    - E.g., the number of distinct values for an attribute
    - Selectivity of a predicate conditions

How to Get Estimated Statistics

- Choice #1: Histograms
  - Maintain an occurrence count per value (or range of values) in a column
- Choice #2: Sketches
  - A probabilistic data structure that gives an approximate count for a given value
- Choice #3: Sampling
  - DMBS maintains a small subset of each table that it then uses to evaluate expressions to compute selectivity.
- Not covered in this lecture.
  - Let’s assume we have a perfect estimator that can always return the actual number.

Single-Relation Query Planning

- Pick the best access method.
  - Sequential Scan

  e.g. , Select * From R, which requires accessing all records
  - Binary Search (clustered indexes)

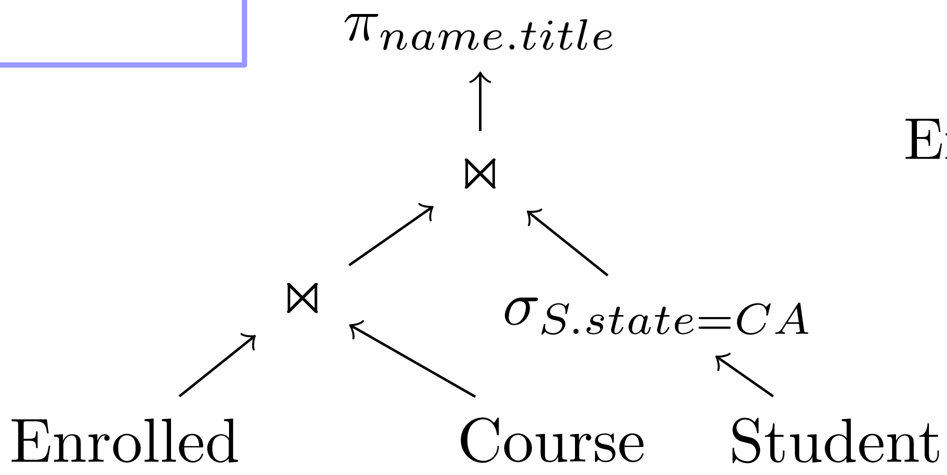
  e.g. , Range filter conditions like Select … From R Where R.x <= 10;
  - Index Scan

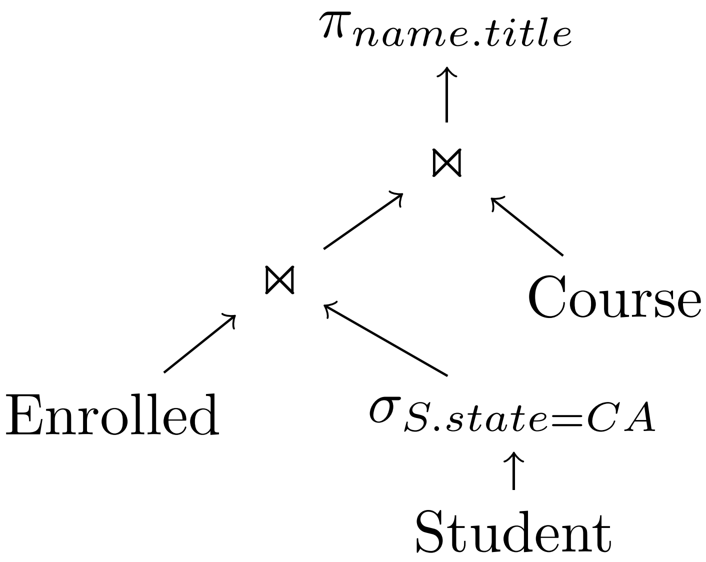
  e.g., Point filter conditions like Select … From R Where R.x = 'A';
- Predicate evaluation ordering
  - Apply the predicates with indexes first to avoid a sequential scan
  - Apply the most restricted predicate first
- Simple heuristics are often good enough for this

How to choose a better plan: Join Reordering

- Unlike predicate pushdown and projection pushdown, we cannot determine which relational expression is better after applying associative rules for multiple joins.

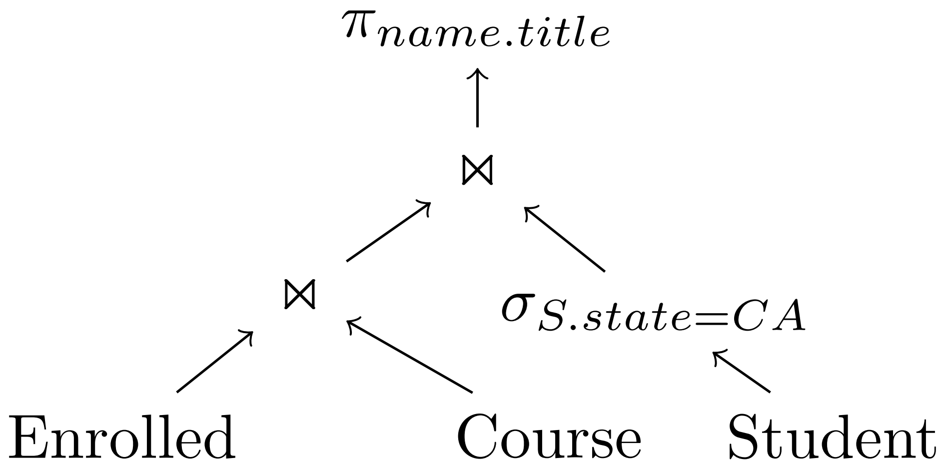
```
SELECT name, title
FROM Student s, Course c, Enrolled e
WHERE s.sid = e.sid
AND c.cid = e.cid
AND s.state = 'CA';
```





Join Reordering

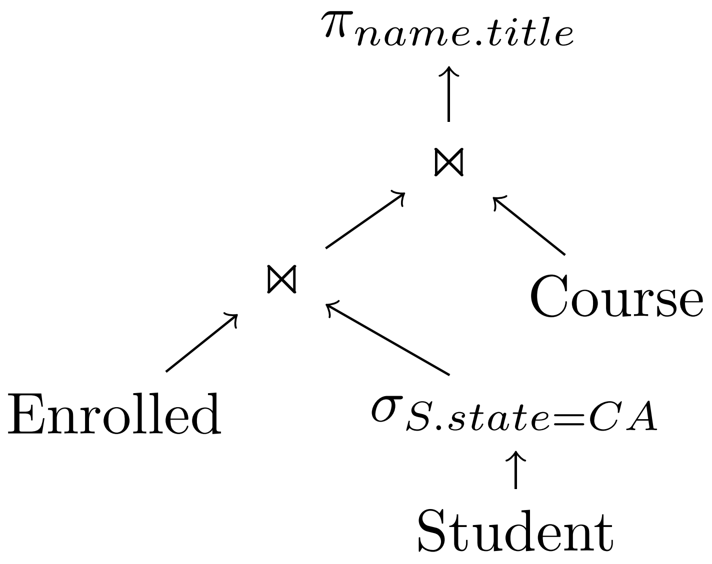
- Let's assume there are
  - 10000 records in the Enrolled relation
  - 50 records in the Course relation
  - 2000 records in the Student relation
  - Only 100 students are from CA
  - Every student enrolls in at most 10 courses



- Cost of the plan (The output size of each operation)
  - $Course \bowtie Enrolled$: returns 10000 records.
  - The filter predicate returns 100 records.
  - The final join returns at most 1000 records.

Join Reordering

- Let's assume there are
  - 10000 records in the Enrolled relation
  - 50 records in the Course relation
  - 2000 records in the Student relation
  - Only 100 students are from CA
  - Every student enrolls in at most 10 courses



- Cost of the plan (The output size of each operation)
  - The selective predicate returns 100 records.
  - The first join returns at most 1000 records.
  - The final join returns at most 1000 records.
- Assuming that generating one record requires a unit of time:
  - The first plan takes 11100 units
  - The second plan takes 2100 units

Join Reordering

- Consider a chain join query:

$$R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie \cdots \bowtie R_n(x_n, x_{n+1})$$

- There can be $O(4^n)$ different join orders (Catalan number)
  - With 10 relations, total 4862 plans
  - With 20 relations, more than 1.7 billion plans

Join Reordering (cont.)

- But there are a lot of duplicates for plans:

$$\Big(\big(R_1(x_1, x_2) \bowtie R_2(x_2, x_3)\big) \bowtie R_3(x_3, x_4)\Big) \bowtie \Big(\big(R_4(x_4, x_5) \bowtie R_5(x_5, x_6)\big) \bowtie R_6(x_6, x_7)\Big)$$

and

$$\Big(\big(R_1(x_1, x_2) \bowtie R_2(x_2, x_3)\big) \bowtie R_3(x_3, x_4)\Big) \bowtie \Big(R_4(x_4, x_5) \bowtie \big(R_5(x_5, x_6) \bowtie R_6(x_6, x_7)\big)\Big)$$

shares the same plan for evaluating the joins between R_1, R_2, R_3

- The problem has **overlapping sub-problems** and show **optimal sub-structure**.

Dynamic Programming!

Dynamic Programming for Join Ordering

- Let $cost[i,j]$ store the minimal cost for calculating chain query $R_i \bowtie \cdots \bowtie R_j$, with $plan[i,j]$ store the corresponding query plan. Assume the cost of calculating a join query is the size of the result.
  - When $i > j$, the problem is invalid
  - When $i = j$, return the relation R_i directly with the cost of $|R_i|$

- When calculating the optimal plan for chain query $R_i \bowtie \cdots \bowtie R_j$, we determine the position k for performing the last join
  - i.e., we calculate $R_i \bowtie \cdots \bowtie R_k$ and $R_{k+1} \bowtie \cdots \bowtie R_j$ first, and then calculate the join query
  $$(R_i \bowtie \cdots \bowtie R_k) \bowtie \left(R_{k+1} \bowtie \cdots \bowtie R_j\right)$$
  - There are totally $j - i$ different choices
- The cost of choosing k will be
$$cost[i,k] + cost[k+1,j] + \left|R_i \bowtie \cdots \bowtie R_j\right|$$

Bottom-up Procedure

- To calculate the optimal cost for $[i, j]$, we first calculate all $cost[l, m]$ with $i \leq l \leq m \leq j$ and $m - l < j - i$
- Then we try all possible k and keep only the optimal one.

```
Input: R_1, ..., R_n in chain order;
for i ← 1 to n do
    cost[i, i] ← |R_i|                          // single relation cost
end
// Outer loop, set segment length
for L ← 2 to n do
    // Middle loop, set the start index i
    for i ← 1 to n − L + 1 do
        j ← i + L − 1
        cost[i, j] ← ∞
        // Inner loop, set the split point
        for k ← i to j − 1 do
            c ← cost[i, k] + cost[k + 1, j] + |R_i ⋈ ··· ⋈ R_j|
            if c < cost[i, j] then
                cost[i, j] ← c;
                plan[i, j] ← k
            end
        end
    end
end
Output: cost[1, n] and query plan via plan
```

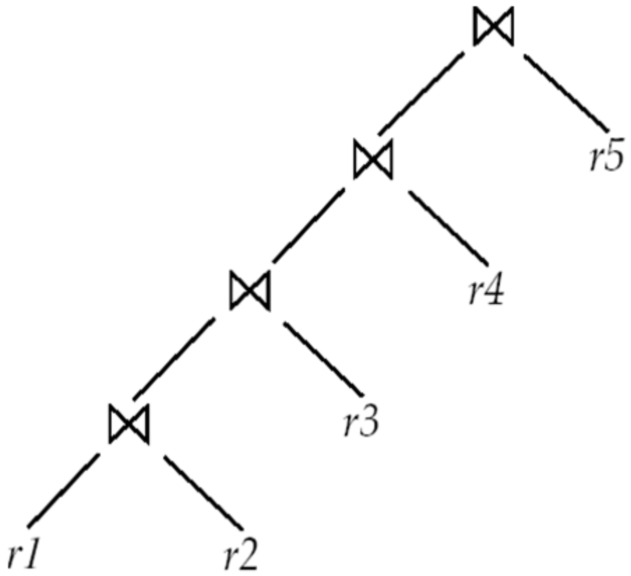
Complexity Analysis

- $O(n^2)$ memory cost
- $O(n^3)$ time complexity
  - When n = 20, the cost is 8000 instead of 1.7 billion.

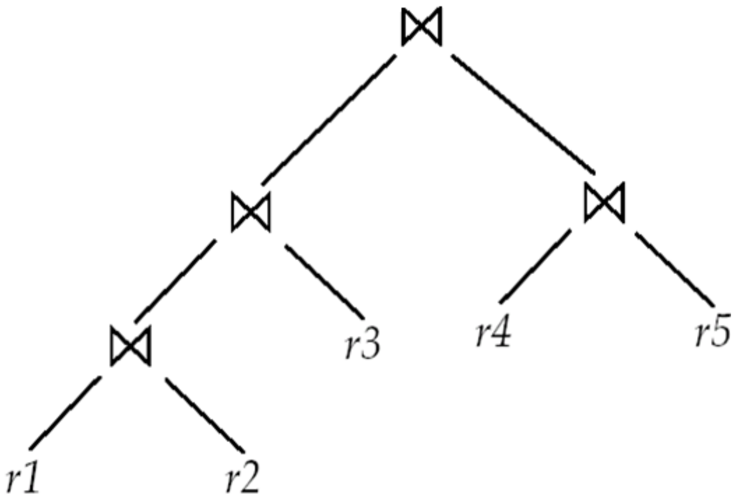
- It is still costly if n is large.

Left Deep Query Plans

- In left-deep query plans, the right-hand-side input for each join is a relation, not the result of an intermediate join



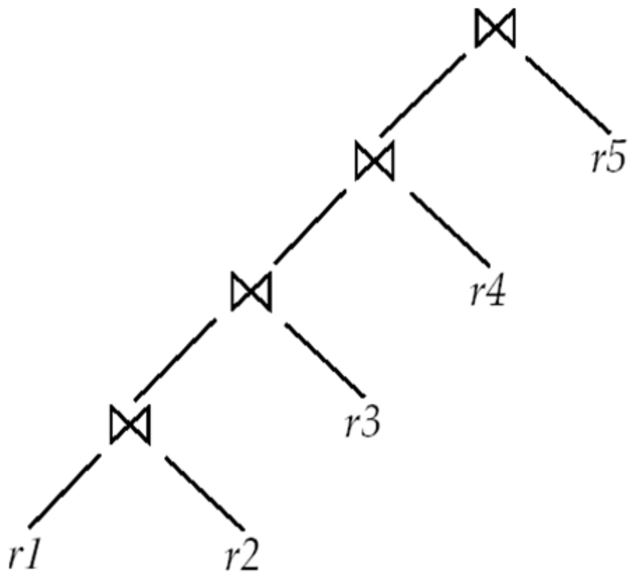
(a) Left-deep join tree



(b) Non-left-deep join tree

Left Deep Query Plans (cont.)

- If only left deep query plans are considered, the number of query plans is significantly reduced.
  - For a chain query, the right-most relation must be R_1 or R_n
  - If we also fix the left-most relation to be R_1, the query plan is uniquely determined.
  - For n = 5, the plan is $\left(\left((R_1 \bowtie R_2) \bowtie R_3\right) \bowtie R_4\right) \bowtie R_5$



Left Deep Query Plans (cont.)

- For calculating $cost[i, j]$, we only need to consider the right-most relation to be R_i or R_j
  - No need to choose split point k anymore.
  - Reduce a factor of n for time complexity.

```
Input: R_1, ..., R_n in chain order;
for i ← 1 to n do
|  cost[i,i] ← |R_i|                         // single relation cost
end
// Outer loop, set segment length
for L ← 2 to n do
    // Middle loop, set the start index i
    for i ← 1 to n − L + 1 do
        j ← i + L − 1
        // Choose R_i or R_j to be the right-most relation
        c_1 ← cost[i, j − 1] + cost[j, j] + |R_i ⋈ ··· ⋈ R_j|
        c_2 ← cost[i, i] + cost[i + 1, j] + |R_i ⋈ ··· ⋈ R_j|
        if c_1 < c_2 then
            cost[i, j] ← c_1
            plan[i, j] ← j
        else
            cost[i, j] ← c_2
            plan[i, j] ← i
    end
end
Output: cost[1, n] and query plan via plan
```

Conclusion

- Query optimization is critical for a database system.
  - SQL -> Logical Plan -> Physical Plan
- The optimization step:
  - Write good SQL if possible.
  - Rule-based optimization for filtering logical plans.
    - Finding equivalent relational expressions
  - Cost-based optimization is used to select the best logical and physical plan.
    - A dynamic programming-based algorithm to avoid plan recomputation
- What is missing:
  - Some equivalent rules (read Database System Concepts, Section 13.2.1, and finish the practice exercises)
  - The cost estimation methods (Section 13.3)

- If you like this and want to make cash money in the database industry, consider earning a PhD in the database team at NTU.

Reference

- Lecture Note 14: Query Planning and Optimization, 15-445/645 Database Systems (Fall 2023), Andy Pavlo, Jignesh Patel
- “The Alice book”, S. Abiteboul, R. Hull and V. Vianu, “Foundations of Databases.”
- “Database System Concepts”, Avi Silberschatz, Henry F. Korth, S. Sudarshan, 6th edition

- Some figures in this slide are from the textbook “Database System Concepts,” and some are AI-generated.