CS 348 Lectures 15-16
Query Optimization
Semih Salihoğlu
Nov 9-11 2021
Announcements

➢ Prof. Thomas Neumann DSG Seminar (Don’t miss!)
➢ Nov 15th at 10:30am
➢ Register here
➢ Send me an email if you want to join the student meeting

Data Systems Seminar Series
Monday, November 15, 2021 • 10:30 a.m. EST

Adaptive Join Order Optimization using Search Space Linearization

Join ordering is one of the core problems of query optimization, as differences in join order can affect the execution time of queries by orders of magnitudes. Unfortunately, the problem is NP hard in general, and real-world queries can join hundreds of relations, which makes exact solutions prohibitive expensive. In this talk we show how to tackle the join ordering problem by using a search space linearization technique. This adaptive optimization mechanism allows for a smooth transition from guaranteed optimality to a greedier approach, depending on the size of problem. In practice, a surprisingly large number of queries can be solved optimally or near optimally, with very low optimization times even for hundreds of relations.

Thomas Neumann is a full professor in the Department of Computer Science at the Technical University of Munich. His research interests are in the areas of database systems, query processing, and query optimization. In 2020, he received the Gottfried Wilhelm Leibniz Prize, which is considered the most important research award in Germany.

Please register • https://uwaterloo.zoom.us/meeting/register/tJAgdeCtzqzwJtCPTmloSuMv_lOJFuRkpqjL
Recall: Overview of Compilation Steps

Text
SELECT ... FROM ... WHERE...

AST
Parser
Root
SELECT ... TABLES ... CON.
Customer Product =
cid 5

Logical Plan
Optimizer
Project oid
Join C.cid=O.cid
Scan Tbl Order
100 < price
Scan Tbl Customer cid=3

Physical Plan Translator
Physical Plan
Project $5
HashJoin L.$2=R.$3
Filter $2=5
IndexScan cust.mkd

Logical Plan Translator
Query Executor
RESULT
cid
...

Binder
Normalizer
Outline For Today

1. Goal of Query Optimization and Overview of Techniques
2. Cost-based Optimization Principles & Cardinality Estimation
3. Cost-based DP Logical Join Plan Optimizer
4. Rule-based Optimizations/Transformations
5. Final Remarks on Query Optimization & Query Processing
Outline For Today

1. Goal of Query Optimization and Overview of Techniques
2. Cost-based Optimization Principles & Cardinality Estimation
3. Cost-based DP Logical Join Plan Optimizer
4. Rule-based Optimizations/Transformations
5. Final Remarks on Query Optimization & Query Processing
Goal of Query Optimization (1)

- Recall ultimately a *physical plan* executes to answer a query
- Given a query Q, many equivalent physical plans exist:
  1. Many equivalent logical plans exist

SELECT cid
FROM Customer C, Order O, Product P
WHERE C.cid = O.cid AND O.pid = P.pid
    AND P.name = BookA

Logical Plans:

L₁
Project cid
  └ Filter name=BookA
      └ Join C.cid=O.cid
          └ Join P.pid=O.pid
              └ Scan Customer
                  └ Scan Product
                          └ Scan Order

L₂
Project cid
  └ Filter name=BookA
      └ Join P.pid=O.pid
          └ Join C.cid=O.cid
              └ Scan Product
                  └ Scan Order
                          └ Scan Customer

L₃
Project cid
  └ Join P.pid=O.pid
      └ Scan Product
          └ Filter name=BookA
              └ Join C.cid=O.cid
                  └ Scan Order
                          └ Scan Customer
                          └ Scan Order
                                  └ Scan Customer
Goal of Query Optimization (1)

➢ Recall ultimately a physical plan executes to answer a query
➢ Given a query Q, many equivalent physical plans exist:
  1. Many equivalent logical plans exist
  2. Each logical plan can have many equivalent physical plans.

Physical Plans:

SELECT cid
FROM Customer C, Order O, Product P
WHERE C.cid = O.cid AND O.pid = P.pid
     AND P.name = BookA

$P_{1,1}$

Scan Product

Scan Order

Scan Customer

HashJoin C.cid = O.cid

HashJoin P.pid = O.pid

Filter name = BookA

Project cid

$P_{1,2}$

Scan Product

Scan Order

Sort(pid)

Scan Customer

HashJoin C.cid = O.cid

HashJoin P.pid = O.pid

MergeJoin P.pid = O.pid

Filter name = BookA

Project cid
Goal of Query Optimization (2)

- Ultimately: Given Q, pick the “best” physical plan for Q:
  - Best: often means fastest, could mean “cheapest”
  - DBMS developers are more humble:
    - Pick a reasonably good plan. Do not pick a very bad plan!
    - Example plan spectrum of join-heavy queries

![Diagram showing plan spectrum and performance metrics.]

*Mhedhbi, Salihoglu, Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins, VLDB 2019*
Overview of Query Opt. Techniques

1. Enumerate a logical plan space (often enumerates all join orders)

   Options for steps 1 & 2:
   i. Rule-based
   ii. Cost-based
   iii. Hybrid rule/cost-based

   (extended) relational algebraic expressions \[ L_1, L_2, \ldots, L_k \]

2. For one or more of \( L_i \), (optionally) enumerate a physical plan space:

   \[ P_{i,1}, P_{i,2}, \ldots, P_{i,t} \]

3. Pick the best \( P_{i,1} \)

   A common approach:
   - Step 1 is cost-based or hybrid
   - Step 2 is rule-based
Outline For Today

1. Goal of Query Optimization and Overview of Techniques
2. Cost-based Optimization Principles & Cardinality Estimation
3. Cost-based DP Logical Join Plan Optimizer
4. Rule-based Optimizations/Transformations
5. Final Remarks on Query Optimization & Query Processing
Cost-based Optimization Principles

- System R ('70s): First prototype relational DBMS from IBM

Patricia Selinger

Given each enumerated log/phy plan, e.g., \( L_i \), a \( \text{Cost}(L_i) = c_i \)

Cost is the estimate of the system for how good/bad \( L_i \) is.

Pick min cost plan

Access Path Selection in a Relational Database Management, Selinger et al. 1979, SIGMOD
Cost-based Optimization Principles (1)

- Naturally: cost definition is broken into costs of operators.
  
i.e: \( \text{Cost}(L_i) = \sum_j \text{cost}(o_j \in L_i) \)

- Example cost metrics or components:
  - # I/Os a plan will make
  - # tuples that will pass through operators
  - # runtime of algorithm \( o_j \) is running
    - e.g., nested loop join of \( R, S: |R|*|S| \)
  - Combination of above

- For any reasonable metric:
  - Need to estimate cardinality, i.e., size, of tuples \( o_j \) will process

\[\text{The (notorious) cardinality estimation problem!}\]
Cardinality Estimation

- Given a database
  1. \( D: R_1(A_{1,1}, \ldots, A_{1,m_1}), \ldots, R_n(A_{n,1}, \ldots, A_{n,m_n}) \)
  2. A (sub-) query \( Q \) (a relational algebra expression)

What is the \(|Q|\)?

- E.g:
  - \( \sigma_{name=BookA}(Product) \)?
  - \( Product \bowtie Order \)?
  - \( \sigma_{name=BookA}(Product \bowtie Order \bowtie Customer) \)?
2 High-level Card. Estimation Techniques

1. Sampling-based:
   ➢ While optimizing Q, sample relations to make an estimate

2. Summary/statistics-based:
   ➢ Use statistics about D to make estimates
   ➢ Possible statistics:
     ➢ $|R_i|$: size of each relation
     ➢ $|\pi_{A_j}(R_i)|$: # distinct values in column $A_j$
     ➢ Histograms: Distribution of values on $A_j$
     ➢ Also use known constraints:
       ➢ E.g: FK constraint from R to S: $|R \bowtie S| = |R|$
       ➢ 2 common simplification assumptions (no other good reason):
         (i) uniformity; (ii) independence
Example Statistics-based Estimation Techniques
Selections with Equality Predicates

- $Q: \sigma_{A=v} R$

Suppose the following information is available:

- Size of $R$: $|R|$
- Number of distinct $A$ values in $R$: $|\pi_A R|$

Assumptions:

1. Values of $A$ are uniformly distributed in $R$
2. $v \in |\pi_A R|$

$|Q| \approx \frac{|R|}{|\pi_A R|}$

Selectivity factor of $(A = v)$ is $\frac{1}{|\pi_A R|}$

Ex: $|Product| = 1000$, $|\pi_{name}(Product)| = 50$

$\sigma_{name=BookA Product}: 1000/50 = 20$
Conjunctive Predicates

- \( Q: \sigma_{A=u \land B=v} R \)

- Additional assumption:
  3. \((A = u) \) and \((B = v) \) are independent
    - Counter example: age and salary

- \(|Q| \approx \frac{|R|}{\pi_{AR} \cdot \pi_{BR}}\)
  - Reduce total size by all selectivity factors
  - Directly derived from standard probability rules:
    - \( \Pr(E_1) = p_1 \), and \( \Pr(E_2) = p_2 \) and \( E_1 \) and \( E_2 \) are independent:
      - \( \Pr(E_1 \land E_2) = p_1 \ast p_2 \)
      - Ex: \( \Pr(\text{heads} \land \text{dice}=6) = 1/2 \ast 1/6 = 1/12 \)
    - Ex: \(|Prod| = 1000, |\pi_{name}(Prod)| = 50, |\pi_{merchant}(Prod)| = 4 \)
      - \( \sigma_{name=BookA \land merchant=B\&N} Product: 1000/(50 \ast 4) = 5 \)
Negated and Disjunctive Predicates

- $Q: \sigma_{A \neq v} R$
  - $|Q| \approx |R| \cdot \left(1 - \frac{1}{|\pi_AR|}\right)$
  - Selectivity factor of $\neg p$ is $(1 - \text{selectivity factor of } p)$

- $Q: \sigma_{A=u \lor B=v} R$
  - $|Q| \approx |R| \cdot \left(\frac{1}{|\pi_AR|} + \frac{1}{|\pi_BR|}\right)$?
    - No! Tuples satisfying $(A = u)$ and $(B = v)$ are counted twice
    - Use only for $\sigma_{A=u \lor A=v} R$ (b/c then $A=u$ and $A=v$ are disjoint)
  - $|Q| \approx |R| \cdot \left(\frac{1}{|\pi_AR|} + \frac{1}{|\pi_BR|} - \frac{1}{|\pi_AR||\pi_BR|}\right)$
    - Inclusion-exclusion principle from probability

![Diagram showing the relationship between $A=u$, $B=v$, and $A=u \land B=v$.]
Range Predicates

- $\sigma_{A>u} R$
- Case 1: Suppose the DBMS knew actual projection values:
  - Then range queries are a generalization of $\sigma_{A=u \lor A=v} R$
  - $\sigma_{A>u} R = |Q| \approx |R| \cdot \left( \frac{|\#vals>u|}{|\pi_A R|} \right)$?
  - E.g: A was an int column and $|\pi_A R| = \{1, 2, 3, 4, 5\}$
    - $\sigma_{A>2} R = |R| * 3/5$
Case 2 of Range Predicates

➢ Case 2: We don’t know actual values
➢ Not enough information!
   ➢ Just pick a magic constant, e.g., $|Q| \approx |R| \cdot \frac{1}{3}$
➢ With more information
   ➢ Largest $R.A$ value: $\text{high}(R.A)$
   ➢ Smallest $R.A$ value: $\text{low}(R.A)$
   ➢ $|Q| \approx |R| \cdot \frac{\text{high}(R.A) - v}{\text{high}(R.A) - \text{low}(R.A)}$
➢ In practice: sometimes the second highest and lowest are used
   ➢ The highest and the lowest are often used by inexperienced database designer to represent invalid values!
Equi-Join of Two Relations (1)

- \( Q: R(A, B) \bowtie S(A, C) \)
- Assumption: containment of value sets:
  - Every tuple in the “smaller” relation (one with fewer distinct values for the join attribute) joins with a tuple in the other relation
  - That is, if \( |\pi_A R| \leq |\pi_A S| \) then \( \pi_A R \subseteq \pi_A S \)
  - Certainly not true in general
  - But holds in the common case of foreign key joins

- \(|Q| \approx \frac{|R| \cdot |S|}{\max(|\pi_A R|, |\pi_A S|)} \)
- Selectivity factor of \( R.A = S.A \) is \( \frac{1}{\max(|\pi_A R|, |\pi_A S|)} \)
Equi-Join of Two Relations (2)

Example:

\[
\begin{align*}
R & \\
| & A & B \\
a_1 & b_1 \\
a_1 & b_2 \\
a_1 & b_3 \\
a_1 & b_4 \\
\end{align*}
\]

\[
\begin{align*}
S & \\
| & A & C \\
a_1 & c_1 \\
a_1 & c_2 \\
a_2 & c_3 \\
a_2 & c_4 \\
\end{align*}
\]

\[
\pi_A R = \{a_1\}
\]

\[
\pi_A S = \{a_1, a_2\}
\]

\[
|Q| \approx \frac{|R| \cdot |S|}{\max(|\pi_A R|, |\pi_A S|)} = \frac{4 \times 4}{2} = 8 \text{ (correct)}
\]

If we had picked \( \min(|\pi_A R|, |\pi_A S|) \), then we’d over-estimate

Intuitively a fraction of tuples from the larger-domain table will join with each tuple from smaller-domain table (not vice versa)
Other Estimations Techniques

➢ Using similar ideas, we can estimate the size of projection, duplicate elimination, union, difference, aggregation (with grouping)

➢ Lots of assumptions and very rough estimation
  ➢ Accurate estimate is not needed
  ➢ Maybe okay if we overestimate or underestimate consistently

➢ In practice: Very very difficult but very important for the optimizer.
  ➢ B/c: ultimate goal is to help estimate costs of operators & plans
  ➢ If we badly underestimate an expression => may lead to bad plans
Example Poor Optimizer Choice

- Suppose cost(o_j): # input tuples processed.
- $\sigma_{p1}(R1) \bowtie R2 \bowtie R3$
- Suppose $\sigma_{p1}(R1) = 1M$ but DBMS underestimates as 1
- Suppose $|R2| = 1M$ and $|R3| = 1K$
- Suppose output of join has the size of the minimum input relation

Estimated Cost: 2
(ignoring in relations)

Estimated Cost: 1001
Example Poor Optimizer Choice

- Suppose \( \text{cost}(o_j) \): \# input tuples processed.
- \( \sigma_{p_1}(R1) \bowtie R2 \bowtie R3 \)
- Suppose \( \sigma_{p_1}(R1) = 1\text{M} \) but DBMS underestimates as 1
- Suppose \(|R2| = 1\text{M} \) and \(|R3| = 1\text{K} \)
- Suppose output of join has the size of the minimum

Actual Cost: \( 1\text{M} + 1\text{K} \)

Actual Cost: \( 2\text{K} \)
Outline For Today

1. Goal of Query Optimization and Overview of Techniques
2. Cost-based Optimization Principles & Cardinality Estimation
3. Cost-based DP Logical Join Plan Optimizer
4. Rule-based Optimizations/Transformations
5. Final Remarks on Query Optimization & Query Processing
Recall:

1. Enumerate a logical plan space (often enumerates all join orders)
   \[ L_1, L_2, \ldots, L_k \]

   A widely used optimization algorithm is to use dynamic programming:
   - Consider a join only query:
     
     ```
     SELECT *
     FROM R1 NATURAL JOIN R2 NATURAL JOIN \ldots\ldots\ldots NATURAL JOIN Rn
     ```
     
   - \( Q = R_1 \Join R_2 \Join \ldots \Join R_n \)
   - Note not-necessarily a `chain` query. It could be in any form, e.g:
     
     ```
     R_1(A, B) \Join R_2(B, C) \Join R_3(C, A) \Join R_4(A, B, C)
     ```
Plan Space

➢ In its most general form Plan Space = All possible join plan `trees`

➢ In practice: If possible you’d avoid plans that do Cartesian Products

➢ Thought experiment: What does optimal tree $L^*$ look like?
Optimal Sub-Join Tree Structure in L*

➢ In L*: What can we say about the sub-tree L_X starting from X?
➢ Must be the best plan for the sub-query Q_X = \( \bowtie_{\forall R_i \in X} R_i \)
  ➢ E.g: red-box must be the best plan for \( R_1 \bowtie R_2 \bowtie R_5 \) (o.w. just replace \( L_X \) with the best plan for \( Q_X: L_X^* \).
➢ Therefore can use dynamic programming algorithm to find join order.
Cost-based DP Join Plan Optimizer

Input Q: R1 \bowtie R2 \bowtie ... \bowtie Rn

Output Optimal Join Plan P:

OptPlans[]: a map that takes a sub-query Q_t and stores the already computed optimal plan:

for int t = 2 ... n // size of sub-queries
for each Q_t \subseteq Q with t relations

P*_{Qt}: // best plan found so far

for each ``split’’ X, Q_t – X:

P*_x = OptPlans[X]; P*_{Qt-X} = OptPlans[Q_t-X];

P_{Qt}: P*_x \bowtie P*_{Qt-X}; // Possible plan when split as X and Q_t-X

P*_Qt = \text{min cost of } P*_{Qt}, P_{Qt}

OptPlans[Q_t] = P*_Qt

where cardinality estimation of Q_t would happen

Optimization 1:
can enumerate over sub-queries that are ```connected``` to avoid Cartesian Products

Optimization 2:
enumerate only if X and Q_t-X have common attributes; otherwise the possible plan would Cartesian product
Example Chain-based Join Optimizer (A4)

➢ A4: specialized version of DP Join Optimizer on `chain queries`:

Q: \( R_0(A_0, A_1) \bowtie R_1(A_1, A_2) \bowtie \ldots \bowtie R_{n-1}(A_{n-1}, A_n) \)

➢ Opt 1: Do not need to enumerate any dis-connected sub-query:

➢ \( Q_{t1}: R_0(A_0, A_1) \bowtie R_2(A_2, A_3) \bowtie R_6(A_6, A_7) \)  ❌ No common attributes

\[ R_0 \quad R_2 \quad R_6 \]
\[ A_0 \quad A_1 \quad A_2 \quad A_3 \quad A_6 \quad A_7 \]

➢ \( Q_{t2}: R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie R_3(A_3, A_4) \)

\[ R_1 \quad R_2 \quad R_3 \]
\[ A_1 \quad A_2 \quad A_3 \quad A_4 \]  ✓

➢ Enumerate plans only for “consecutive”: \( R_i \bowtie R_{i+1} \bowtie \ldots \bowtie R_j \)

➢ Enumerate only j-i``split points” for each k: i…j-1:

➢ \( R_i \bowtie R_{i+1} \bowtie \ldots \bowtie R_k \) and \( R_{k+1} \bowtie R_{k+2} \bowtie \ldots \bowtie R_j \)
Simulation

Opt Plans for 1-size sub-queries $R_i$:

Opt Plan: $R_i$

cost: $|R_i|$

Opt Plans for 2-size sub-queries $R_i \bowtie R_{i+1}$:

Opt Plan: $\text{Join}$

$R_i \quad R_{i+1}$

cost: $c^*_{i,i+1}$

Opt Plans for 3-size sub-queries (using 1- and 2-size opt. plans):

Opt Plan: $\text{Join}$

$R_2$

$R_3 \quad R_4$

cost: $c^*_{2,3,4}$

Opt Plan: $\text{Join}$

$R_3 \quad R_4$

cost: $c^*_{3,4,5}$

...
Simulation

When computing plans for a 4-size sub-query: e.g., \( R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5 \):

- Opt Plan: \( \text{Join} \)
  
  \[
  \begin{align*}
  R_2 & \quad \text{Join} \quad R_3 \\
  \text{cost: } c^{*, 2,3} & \quad \text{Join} \\
  \end{align*}
  \]

- Opt Plan: \( \text{Join} \)
  
  \[
  \begin{align*}
  R_3 & \quad \text{Join} \quad R_4 \\
  \text{cost: } c^{*, 3,4} & \quad \text{Join} \\
  \end{align*}
  \]

- Opt Plan: \( \text{Join} \)
  
  \[
  \begin{align*}
  R_4 & \quad \text{Join} \quad R_5 \\
  \text{cost: } c^{*, 4,5} & \quad \text{Join} \\
  \end{align*}
  \]

Possible plans:

- \( R_2 \) as the split point
  
  \[
  \begin{align*}
  \text{Join} & \quad \text{Join} \quad R_3 \\
  R_2 & \quad \text{Join} \quad R_5 \\
  \end{align*}
  \]

- \( R_3 \) as the split point
  
  \[
  \begin{align*}
  \text{Join} & \quad \text{Join} \quad R_4 \\
  R_2 & \quad \text{Join} \quad R_3 \\
  \end{align*}
  \]

- \( R_4 \) as the split point
  
  \[
  \begin{align*}
  \text{Join} & \quad \text{Join} \quad R_5 \\
  R_2 & \quad \text{Join} \quad R_3 \\
  \end{align*}
  \]

If left/right child matters compare 2x more plans.
Outline For Today

1. Goal of Query Optimization and Overview of Techniques
2. Cost-based Optimization Principles & Cardinality Estimation
3. Cost-based DP Logical Join Plan Optimizer
4. Rule-based Optimizations/Transformations
5. Final Remarks on Query Optimization & Query Processing
Rule-based Transformations

- DP-based join optimizer algorithm only considered join-only queries
- What if there was a selection, projection, group-by aggregate etc?
- When possible we consider them as we enumerate plans but often in a *rule-based manner*
Example (1)

```
SELECT * 
FROM R1 NATURAL JOIN R2 NATURAL JOIN R3 NATURAL JOIN R4 
WHERE R1.A1 = "foo" AND R3.A3 = "bar"
``` 

➢ Intuitively instead of enumerating a plan for R1 we should enumerate a plan for relation: $\sigma_{A1=foo}(R1)$

➢ Similarly instead of R2, we should enumerate plans for $\sigma_{A3=bar}(R3)$

➢ Why?

➢ But not if the predicate was: R1.A1 = "foo" OR R3.A3 = "bar"

➢ What to enumerate is governed by algebraic laws

➢ This is an important advantage of implementing a query language that’s based on a formal algebra: i.e., relational algebra
Example (2)

```sql
SELECT *
FROM R1 NATURAL JOIN R2 NATURAL JOIN R3 NATURAL JOIN R4
WHERE R1.A1 = "foo" AND R3.A3="bar"
```

➢ In relational algebra:

\[\sigma_{A1=foo \land A3=bar} (R1 \bowtie R2 \bowtie R3 \bowtie R4) = (\sigma_{A1=foo} (R1) \bowtie R2 \bowtie \sigma_{A3=bar} (R3) \bowtie R4)\]

➢ The expression effectively joins these smaller relations:

- i. \[\sigma_{A1=foo} (R1)\]
- ii. \[R2\]
- iii. \[\sigma_{A3=bar} (R3)\]
- iv. \[R4\]

➢ What if WHERE clause was \(R1.A1 = \text{"foo" OR R3.A3="bar"}\)?

➢ Apply the predicate only for sub-queries with both R1 and R3.

➢ The above algebraic law is called: **pushing down selections**
Ex Algebraic Transformation Rules (1)

Will use pure rel. algebra notation but can use our logical plan notation:

- Convert $\sigma_p \times$ to/from $\bowtie_p \colon \sigma_p (R \times S) = R \bowtie_p S$
  
  - Example: $\sigma_{User.uid = Member.uid} (User \times Member) = User \bowtie Member$

- Merge/split $\sigma$'s: $\sigma_{p_1} (\sigma_{p_2} R) = \sigma_{p_1 \land p_2} R$
  
  - Example: $\sigma_{age > 20} (\sigma_{pop = 0.8} User) = \sigma_{age > 20 \land pop = 0.8} User$

- Merge/split $\pi$'s: $\pi_{L_1} (\pi_{L_2} R) = \pi_{L_1} R$, where $L_1 \subseteq L_2$
  
  - Example: $\pi_{age} (\pi_{age, pop} User) = \pi_{age} User$
Example In Logical Plan Notation

- Merge/split \( \sigma \)'s: \( \sigma_{p_1} (\sigma_{p_2} R) = \sigma_{p_1 \land p_2} R \)

- Example: \( \sigma_{\text{age}>20} (\sigma_{\text{pop}=0.8 \text{User}}) = \sigma_{\text{age}>20 \land \text{pop}=0.8 \text{User}} \)

Diagram:

```
  ...  
  /   
/     
Filter pop>0.8  ...  Filter age>20 \land pop>0.8  
/       
Filter age>20 
/         
User  
```

```
  ...  
  /   
/     
Filter pop>0.8  ...  Filter age>20 \land pop>0.8  
/       
User  
```
Push down/pull up \( \sigma \) (not predicate is a conjunction):

\[
\sigma_{p \land p_r \land p_s} (R \bowtie_{p'} S) = (\sigma_{p_r} R) \bowtie_{p \land p'} (\sigma_{p_s} S), \text{ where}
\]

- \( p_r \) is a predicate involving only \( R \) columns
- \( p_s \) is a predicate involving only \( S \) columns
- \( p \) and \( p' \) are predicates involving both \( R \) and \( S \) columns
- i.e., \( p \) an additional join predicate

Example:

\[
\sigma_{U1.name=U2.name \land U1.pop>0.8 \land U2.pop>0.8} (\rho_{U1 User} \bowtie_{U1.uid\neq U2.uid} \rho_{U2 User})
= \sigma_{pop>0.8} (\rho_{U1 User}) \bowtie_{U1.uid\neq U2.uid, U1.name=U2.name} (\sigma_{pop>0.8} (\rho_{U2 User}))
\]

Why should you always do this optimization?

- Selections are relatively cheap (e.g., compared to joins or group-by and aggregates) and can only reduce the number tuples processed.
Example In Logical Plan Notation

\[ \sigma_{U1.name=U2.name} \land U1.pop>0.8 \land U2.pop>0.8 \left( \rho_{U1.User} \bowtie_{U1.uid \neq U2.uid} \rho_{U2.User} \right) = \sigma_{pop>0.8} \left( \rho_{U1.User} \bowtie U1.uid \neq U2.uid, U1.name=U2.name \right) \left( \sigma_{pop>0.8} \left( \rho_{U2.User} \right) \right) \]
Ex Algebraic Transformation Rules (3)

- Push down $\pi$: $\pi_L(\sigma_p R) = \pi_L(\sigma_p(\pi_{L'} R))$, where
  - $L'$ is the set of columns referenced by $p$ that are not in $L$
  - Example:
    
    $\pi_{age}(\sigma_{pop>0.8 User}) = \pi_{age}(\sigma_{pop>0.8}(\pi_{age, pop} User))$
    
    - Not as important and effective as pushing $\sigma$
  
  - Many more (seemingly trivial) equivalences…
    
    - Can be systematically used to transform plans
Outline For Today

1. Goal of Query Optimization and Overview of Techniques
2. Cost-based Optimization Principles & Cardinality Estimation
3. Cost-based DP Logical Join Plan Optimizer
4. Rule-based Optimizations/Transformations
5. Final Remarks on Query Optimization & Query Processing
Query Optimizer and Cardinality Estimator: Brain of the DBMS

- **Ultimate Goal:** Pick a reasonable plan (i.e., one processing few tuples)

Query Processor and Storage: Skeleton

- They do actual data searching and computation

Several insights have emerged over the years in DBMS literature:

- Cost model is not very critical: keep a simple model (e.g., # tuples)
- Cardinality estimation: matters a lot
- But! Extremely difficult to integrate a good estimator. Always a hack with wild unrealistic assumptions here and there to make it implementable: magic constants, uniformity assumptions, independence assumptions etc.

My advice: Optimizer is important but keep it simple.

- Do not be complacent on the query processor and storage! Work very hard on these and optimize relentlessly!
Final Remarks (2)

- CS 448: Database Systems Implementation
  - Gets into many more details about the internals of query processing and optimization and other DBMS components!
  - A4’s programming question is meant to give you a glimpse of CS 448 assignments.

Query Optimization Through the Looking Glass, and What We Found Running the Join Order Benchmark, Leis et al., VLDBJ, 2018