Datenbanksysteme II: Query Optimization

Ulf Leser
5 Layer Architecture

- Data Model
- Logical Access
- Data Structures
- Buffer Management
- Operating System

We are here
Content of this Lecture

- Introduction
- Rewriting Subqueries
- Algebraic Term Rewriting
- Optimizing Join Order
- Plan Enumeration
- A counter-example
Is Optimization Worth It?

- **Goal:** Find cheapest way to compute a query result
  - Generate and judge different physical plans to answer the query
  - All QEPs must be semantically equal

- **Optimization costs time**
  - Some steps are exponential
    - E.g. join order: 10 joins - potentially $3^{10}$ steps
  - Finding the best plan might take more time than executing an arbitrary plan
    - And usually we don’t even find the best plan

- **Why bother?**
Example

```
SELECT C.name, C.address
FROM   customer C, order O
WHERE  C.name = O.c_name AND
       O.product = "coffee"
```

- **Assumptions**
  - 1:n relationship between C and O
  - |C|=100, 5 tuples per block, b(C)=20
  - |O|=10,000, 10 tuples per block, b(O) = 1,000
  - Result size: 50 tuples
  - Intermediate results
    - (C.name, C.address): 50 per block
    - Join result (C,O) with full tuples: 3 per block
  - Small main memory
First Attempt

- Translate in relational algebra expression
  \[ \pi_{name,adr}(\sigma_{O.C_name=C.name \land O.product='coffee'} (C \times O)) \]

- Interpret query "from inner to outer"
  - No optimization at all
  - Full materialization of intermediate results (no buffering, no pipelining)
Cost

- **Compute cross-product**
  - Reads: $b(C) \times b(O) = 20,000$
  - Writes: $100 \times 10,000 / 3 \approx 333,000$

- **Compute selections**
  - Reads: 333,000
  - Writes: $50 / 3 \approx 17$

- **Compute projection**
  - Reads: 17
  - Writes: $50 / 50 \approx 1$

- **Altogether: \( \approx 686,000 \) IO**
  (and 333,000 blocks required on disk)
Use Term Rewriting

• Rewrite into: \( \pi_{\text{name, adr}} (C \bowtie_{O.C_{\text{name}}=C._{\text{name}}} (\sigma_{O._{\text{product}}=\text{coffee'}}(O))) \)

• Compute selection on O
  - Reads: 1,000, writes: 50/10 = 5

• Compute join using BNL
  - Reads: 5 + b(C)*5 = 105
  - Writes: 50/3 ~ 17

• Compute projection
  - Reads: 17, writes: 50/50 ~ 1

• Altogether: 1.145
  (requiring 17 blocks on disk)

• Maybe there is an ever better way?
Better Plan

- **Push projection**
  - $\pi_{\text{name,adr}} (\pi_{\text{name,adr}} (C) \bowtie_{O.\text{c_name}=C.\text{name}} (\sigma_{O.\text{product}=\text{coffee}} (O)))$

- **Compute selection on O**
  - Reads: 1,000, writes: $50/10 = 5$

- **Compute projection on C**
  - Reads b(C)=20, writes $100 / 50 = 2$

- **Compute join using nested loop**
  - Reads: $2 + 2*5 = 12$, writes: $50/3 \sim 17$

- **Compute projection**
  - Reads: 17, writes: $50/50 \sim 1$

- **Altogether: 1.080** (requiring 17 blocks on disk)
Even Better – Use Indexes

- Indexes on (O.product, O.C_name) and (C.name, C_address)
- Compute **selection on O using index**
  - Reads: Roughly between 5 and 10
    - Height of index plus consecutive blocks for 50 TIDs with product=‘coffee’
    - Number of blocks depends on fill degree of B-tree
    - Assume 10 pointer in an index node: height = 4
  - Writes: 50/10 = 5
- **Sort** intermediate result
  - Read and writes: \( \sim 5 \times \log(5) \sim 15 \)
    - Very conservative estimation
  - Result has 5 blocks
Even Better – Use Indexes

• ...

• Compute join
  - Reads: 20 + 5 = 25
    • Using sort-merge – read C.name in sorted order using index
  - Writes: 50/3 ~ 17

• Compute projection
  - Reads: 17, writes: 50/50 ~ 1

• Altogether: *between 85 and 90*
  (requiring 17 blocks on disk)

• Even better?
Comparison

<table>
<thead>
<tr>
<th></th>
<th>Read/Write</th>
<th>Temp space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>687.000</td>
<td>333.000</td>
</tr>
<tr>
<td>Optimized, no index</td>
<td>1.080</td>
<td>17</td>
</tr>
<tr>
<td>With index</td>
<td>85-90</td>
<td>17</td>
</tr>
</tbody>
</table>

- Reduction by a factor of $\sim8.000$
- Conclusion: DB should invest some time in optimization
Steps in Optimization

• Parsing, view expansion, subquery rewriting
• Query minimization (maybe)
• Expression/tree generation
• Plan optimization
  - Algebraic term rewriting (logic optimization)
  - Cost estimation (cost-based optimization)
  - Plan instantiation (physical optimization)
  - Plan enumeration and pruning
  - Note: Steps are interleaved
• Selection of best plan
• Code generation (compilation or interpretation)
Content of this Lecture

- Introduction
- **Rewriting Subqueries**
- Algebraic Term Rewriting
- Optimizing Join Order
- Plan Enumeration
- A counter-example
Subquery Rewriting

• No equivalent in relational algebra: IN, EXISTS, ALL
  - Generate subtrees during parsing
  - For optimization, a single tree with only relational operations is easier to handle
  - But: Transformation not always easy, not always advantageous

• We look at four cases of IN
  - Uncorrelated without aggregation
  - Uncorrelated with aggregation
  - Correlated without aggregation
  - Correlated with aggregation

• See literature for EXISTS, ALL, MINUS, INTERSECT, …
Example

### Order
- O_id
- C_name
- P_Id
- Date
- Total_price
- revenue

### Delivery
- Id
- O_ID
- Date
- Price
- Quantity

### Product
- Id
- P_Name
- Price
- ...

### Customer
- Name
- Address
- ...

### Diagram Description
- The diagram represents a database schema involving tables for orders, deliveries, products, and customers.
- Each table is connected by relationships indicating the data dependencies.
- The Order table is linked to the Customer and Delivery tables.
- The Delivery table is linked to the Order table.
- The Product table is linked to the Order table.
- The diagram illustrates how data is stored and related in a database system.
Uncorrelated Subquery without Aggregation

```
SELECT o_id
FROM order
WHERE p_id IN (SELECT id
    FROM product
    WHERE price < 1)
```

- **Option 1:** Compute subquery and **materialize result**
  - Advantageous if subquery appears more than once

- **Option 2:** Rewrite into join
  - Allows global optimization (i.e. index join)
  - Be careful with **duplicates**
    - Assuming id is PK of P, example is fine
    - Otherwise, we need to introduce a DISTINCT

```
SELECT o.o_id
FROM   order o, product p
WHERE  o.p_id = p.id AND
       p.price < 1
```
Uncorrelated Subquery with Aggregation

```sql
SELECT o_id
FROM order
WHERE p_id IN (SELECT max(id)
                FROM product)
```

- (Only) option: Compute subquery and materialize result
- Rewriting not possible
- Other way of expression this: User-defined table functions
  - This would allow formulation as join
  - But overall even harder to optimize
- Third way: Use view (two queries)
Correlated Subquery without Aggregation

\[
\begin{align*}
\text{SELECT} & \quad o.o_id \\
\text{FROM} & \quad \text{order } o \\
\text{WHERE} & \quad o.o_id \text{ IN } (\text{SELECT } d.o_id \\
& \quad \text{FROM } \text{delivery } d \\
& \quad \text{WHERE } d.o_id = o.o_id \text{ AND } \\
& \quad d.date-o.date<5)
\end{align*}
\]

- Subquery materialization not possible
- **Naïve** computation requires one execution of subquery for each tuple of outer query
- Solution: **Rewrite into join**
  - Again: Caution with duplicates
    (if o:d is 1:n, DISTINCT required)
Correlated Subquery with Aggregation

```sql
SELECT o.o_id
FROM order o
WHERE o.total_price != (SELECT sum(price*quantity)
                        FROM delivery d
                        WHERE d.o_id = o.o_id)
```

- Materialization not easily possible
  - Note that there is only one join condition
- Rewrite into join not possible
- Naïve computation requires one execution of subquery for each tuple of outer query
- Solution: Rewrite into two queries
Correlated Subquery with Aggregation

```
SELECT o.o_id
FROM order o
WHERE o.total_price != (SELECT sum(price*quantity)
                        FROM delivery d
                        WHERE d.o_id = o.o_id)
```

- **New inner query**

```
CREATE VIEW all_sums AS
SELECT o_id, sum(price*quant) as tp
FROM delivery
GROUP BY o_id
```

- **New outer query**

```
SELECT o.o_id
FROM order o
WHERE o.total_price !=
    (SELECT tp
     FROM all_sums
     WHERE all_sums.o_id = o.o_id)
```
Can be Combined

SELECT o_id, sum(price*quant) as tp
FROM delivery
GROUP BY o_id

SELECT o.o_id
FROM order o
WHERE o.total_price !=
(SELECT tp
  FROM all_sums
  WHERE all_sums.o_id = o.o_id)

SELECT o.o_id
FROM order o, all_sums
WHERE o.total_price != all_sums.tp
**Improvements**

- Inner query can be computed and materialized once
- Inner query will use (efficient) full table scan instead of multiple queries with condition on join attribute
Query Minimization 1

• Especially important when views are involved or queries are created automatically

CREATE VIEW good_business
SELECT C.name, O.O_id, O.revenue
FROM customer C, order O
WHERE C.name = O.name AND O.revenue>1.000

- Find very good customers using view as first filter

SELECT name
FROM good_business
WHERE revenue>5.000

SELECT C.name
FROM customer C, order O
WHERE C.name = O.name AND O.revenue>1.000 AND O.revenue>5.000

• Goal: Remove redundant conditions
Query Minimization 2

- Especially important when **views are involved** or queries are created automatically

```sql
CREATE VIEW good_business
SELECT  C.name, O.O_id, O.revenue
FROM    customer C, order O
WHERE   C.name = O.name AND O.revenue > 1.000
```

- Find goods from good businesses

```sql
SELECT G.name, O.good
FROM    good_busi G, order O
WHERE   G.o_id = O.o_id
```

```sql
SELECT C.name, O2.good
FROM    custom C, ord O1, ord O2
WHERE   C.name = O1.name AND O1.revenue > 1000 AND O1.o_id = O2.o_id
```

- Remove **redundant joins**
Content of this Lecture

- Introduction
- Rewriting Subqueries
- Algebraic Term Rewriting
- Optimizing Join Order
- Plan Enumeration
- A counter-example
**Equivalence of Relational Algebra Expressions**

- **Definition**
  
  Let $E_1$ und $E_2$ be two relational algebra expressions over a schema $S$. $E_1$ and $E_2$ are called **equivalent** iff
  
  - $E_1$ and $E_2$ contain the same relations $R_1 \ldots R_n$
  - For any instances of $S$, $E_1$ and $E_2$ compute the same result

- We **generate equivalent expressions** by applying certain rewrite rules

- We will see some rules (there exist more: literature)
Rules for Joins and Products

• Assume
  - $E_1, E_2, E_3$ relational expressions
  - $\text{Cond}, \text{Cond1}, \text{Cond2}$ are join conditions

• Rule 1: Joins and Cartesian-products are commutative
  $$E_1 \bowtie_{\text{Cond}} E_2 \equiv E_2 \bowtie_{\text{Cond}} E_1$$
  $$E_1 \times E_2 \equiv E_2 \times E_1$$

• Rule 2: Joins and Cartesian-products are associative
  $$\left( E_1 \bowtie_{\text{Cond1}} E_2 \right) \bowtie_{\text{Cond2}} E_3 \equiv E_1 \bowtie_{\text{Cond1}} \left( E_2 \bowtie_{\text{Cond2}} E_3 \right)$$
  Requirement: $E_3$ joins with $E_2$ (and not with $E_1$)
  $$\left( E_1 \times E_2 \right) \times E_3 \equiv E_1 \times \left( E_2 \times E_3 \right)$$
For Projection and Selection

• Assume
  - $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$ be attributes of $E$
  - $Cond1$ and $Cond2$ conditions on $E$

• Rule 3: Cascading projections
  If $A_1, \ldots, A_n \supseteq B_1, \ldots, B_m$, then
  $$\Pi \{B_1, \ldots, B_m\} (\Pi \{A_1, \ldots, A_n\} (E)) \equiv \Pi \{B_1, \ldots, B_m\} (E)$$

• Rule 4: Cascading selections
  $$\sigma_{Cond1} (\sigma_{Cond2} (E)) \equiv \sigma_{Cond2} (\sigma_{Cond1} (E))$$
  $$\equiv \sigma_{Cond1 \text{ and } Cond2} (E)$$
For Projection and Selection

- Assume
  - A₁, . . . , Aₙ and B₁ , . . . , Bₘ be attributes of E
  - Cond₁ und Cond₂ conditions on E

- Rule 5a. Exchange of projection and selection

\[ \pi_{\{A₁,...,Aₙ\}}(\sigma_{\text{Cond}}(E)) \equiv \sigma_{\text{Cond}}(\pi_{\{A₁,...,Aₙ\}}(E)) \]

Requirement: Cond contains only attributes A₁, . . . , Aₙ

- Rule 5b. Injection of projection

\[ \pi_{\{A₁...Aₙ\}}(\sigma_{\text{Cond}}(E)) \equiv \pi_{\{A₁...Aₙ\}}(\sigma_{\text{Cond}}(\pi_{\{A₁...Aₙ, B₁...Bₘ\}}(E))) \]

Requirement: Cond contains only attributes A₁..Aₙ and B₁..Bₘ
Joins and Projection/Selection

• Rule 6. Exchange of selection and join

\[ \sigma_{\text{Cond}} ( E_1 \bowtie_{\text{Cond}1} E_2 ) \equiv \sigma_{\text{Cond}} ( E_1 ) \bowtie_{\text{Cond}1} E_2 \]

Requirement: Cond contains only attributes of E1

• Rule 7. Exchange of selection and union/difference

\[ \sigma_{\text{Cond}} ( E_1 \cup E_2 ) \equiv \sigma_{\text{Cond}} ( E_1 ) \cup \sigma_{\text{Cond}} ( E_2 ) \]
\[ \sigma_{\text{Cond}} ( E_1 - E_2 ) \equiv \sigma_{\text{Cond}} ( E_1 ) - \sigma_{\text{Cond}} ( E_2 ) \]
Joins and Projection/Selection

• Rule 9. Exchange of projection and join:

\[ \Pi_{\{A_1, \ldots, A_n, B_1, \ldots, B_m\}} (E_1 \bowtie_{\text{Cond}} E_2) \equiv \Pi_{\{A_1, \ldots, A_n\}} (E_1) \bowtie_{\text{Cond}} \Pi_{\{B_1, \ldots, B_m\}} (E_2) \]

Requirement: Cond contains only attributes \(A_1\ldots A_n\), \(B_1\ldots B_m\) and \(A_1\ldots A_n\) appear in \(E_1\), resp. \(B_1\ldots B_m\) in \(E_2\).

• Rule 10. Exchange of projection and union:

\[ \Pi_{\{A_1, \ldots, An\}} (E_1 \cup E_2) \equiv \Pi_{\{A_1, \ldots, An\}} (E_1) \cup \Pi_{\{A_1, \ldots, An\}} (E_2) \]
Example

• Query on CUSTOMER database

```sql
SELECT Name, Account#, Savings
FROM customer C, account A, journal J
WHERE "Bond" ≤ Name ≤ "Carter" and
  Address = "World" and
  Transaction = "Withdraw" and
  Amount > 1,000,000 and
  C.Account# = A.Account# and
  C.Account# = J.Account#
```
Initial Operator Tree

\[ \Pi \]

\[ \sigma \]

"Bond" \leq Name
Name \leq "Carter"
Address = "World"
Transaction = "Withdraw"
Amount > $1,000,000
C.Account# = A.Account#
C.Account# = J.Account#

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]

\[ \times \]
Breaking and Pushing Selections

\[ \Pi \text{ Name, Account#, Savings} \]

\[ \sigma \text{ C.Account#=J.Account#} \]

\[ \times \text{ C.Account#=A.Account#} \]

\[ \sigma \text{ "Bond"\leq Name \leq "Carter" \ Address="World"} \]

\[ \times \text{ ACCOUNT} \]

\[ \sigma \text{ Transac="Withdraw" Amount>1000000} \]

\[ \sigma \text{ CUSTOMER} \]

\[ \text{Journal} \]
Introduce Joins

\[
\begin{align*}
\Pi \quad & Name, Account#, Savings \\
\bowtie & C.\text{Account#} = A.\text{Account#} \\
\bowtie & "Bond" \leq \text{Name} \\
\bowtie & Name \leq "\text{Carter}" \\
\bowtie & Address = "World" \\
\sigma & C.\text{Account#} = J.\text{Account#} \\
\sigma & Transac = "Withdraw" \\
\sigma & \text{Amount} > 1000000
\end{align*}
\]
Pushing Projections

Diagram:

- CUSTOMER (Π)
- ACCOUNT (Π)
- Journal (Π)
- Account# (σ)
- Name, Account#, Savings (Π)
- Name, Account# (Π)
- Name, Account#, Address (Π)

Projected relations:

1. NAME, ACCOUNT#, SAVINGS
2. NAME, ACCOUNT#
3. NAME, ACCOUNT#, ADDRESS

Operations:

- Π (Projection)
- σ (Selection)

Note: The diagram illustrates the process of pushing down projections in a database schema.
Caution

• Sometimes, **pushing up selections** is good
  - Especially for conditions on join attributes

• Example

  CREATE VIEW movies99 AS
  SELECT title, year, studio
  FROM   movie WHERE year=1999

  ⋈

  SELECT m.title, a.name
  FROM   movies99 m, actsin a
  WHERE  m.title=a.title AND
         m.year=a.year
• Usually there infinitely many rewrite steps
  – But not infinitely many different plans
  – Rewritings often go back and forth
• General heuristic: Minimize intermediate results
  – Less IO if materialization is necessary
  – Less input for operations that are higher in the plan
• Option1: Rule-based
  – Use heuristics for selecting order of rule application
  – Based on experience – rules that are beneficial in most cases
  – Simple to implement, fast optimizer
  – But: Unusual queries lead to bad plans
A Simple Rule-Based Optimizer

• **Down – break and push down**
  - Break combined selections into many simple selections
  - Break combined projections into many simple projections
  - Push selects/projects as much **down the tree** as possible
  - Introduce add. projections as deep in the tree as possible

• **Up – merge operations**
  - Replace selection and Cartesian product with join
  - Merge simple selections into combined selections
  - Merge simple projections into combined projections

• **Physical**
  - If there is a condition on an indexed attribute – use the index
    - Conflicts with break / merge patterns
  - For a join over PK-FK relationships: Use **sort-merge**
  - Other joins: Use hash join
Another Example

```sql
SELECT s.Semester
FROM   student s, hoeren h,
       vorlesung v, professor p
WHERE  p.name = "Sokrates" and
       v.gelesenvon = p.persnr and
       v.vorlnr = h.vorlnr and
       h.matrnr = s.matrnr
```
Break Up Selections

\[ \pi_{s.\text{Semester}} \sigma_{p.\text{Name} = \text{´Sokrates´ and}} \ldots \]

\[ \sigma_{p.\text{PersNr}=v.\text{gelesenVon}} \sigma_{v.\text{VorlNr}=h.\text{VorlNr}} \sigma_{s.\text{MatrNr}=h.\text{MatrNr}} \sigma_{p.\text{Name} = \text{´Sokrates´}} \]

Push Selections

\[ \pi_{s.\text{Semester}} \]

\[ \sigma_{p.\text{PersNr}=v.\text{gelesenVon}} \]

\[ \sigma_{v.\text{VorlNr}=h.\text{VorlNr}} \]

\[ \sigma_{s.\text{MatrNr}=h.\text{MatrNr}} \]

\[ \sigma_{p.\text{Name} = '\text{Sokrates}' } \]

\[ \pi_{s.\text{Semester}} \]

\[ \sigma_{p.\text{PersNr}=v.\text{gelesenVon}} \]

\[ \sigma_{v.\text{VorlNr}=h.\text{VorlNr}} \]

\[ \sigma_{s.\text{MatrNr}=h.\text{MatrNr}} \]

\[ \sigma_{p.\text{Name} = '\text{Sokrates}' } \]

\[ s \quad h \quad v \]

\[ p \]

Rewrite Product+Selection into Joins

\[ \pi_{s.\text{Semester}} \]
\[ \sigma_{p.\text{PersNr}=v.\text{gelesenVon}} \]
\[ \sigma_{v.\text{VorlNr}=h.\text{VorlNr}} \]
\[ \sigma_{s.\text{MatrNr}=h.\text{MatrNr}} \]

\[ \bowtie \]

\[ \pi_{s.\text{Semester}} \]
\[ \sigma_{p.\text{Name} = 'Sokrates'} \]
\[ \sigma_{v.\text{VorlNr}=h.\text{VorlNr}} \]
\[ \sigma_{s.\text{MatrNr}=h.\text{MatrNr}} \]
Introduce Additional Projections

\[
\begin{align*}
\pi_{s.\text{Semester}} & \quad \bowtie \quad \sigma_{p.\text{Name} = '\text{Sokrates}'} \quad \bowtie \quad \pi_{\text{MatrNr}, \text{semester}} \\
\bowtie_{v.\text{VorlNr}=h.\text{VorlNr}} & \quad \bowtie_{s.\text{MatrNr}=h.\text{MatrNr}} \\
\bowtie_{p.\text{PersNr}=v.\text{gelesenVon}} & \\
\pi_{s.\text{Semester}} & \quad \bowtie \quad \sigma_{p.\text{Name} = '\text{Sokrates}'} \quad \bowtie \quad \pi_{\text{MatrNr}, \text{vorlNr}} \\
\bowtie_{v.\text{VorlNr}=h.\text{VorlNr}} & \quad \bowtie_{s.\text{MatrNr}=h.\text{MatrNr}} \\
\end{align*}
\]
Limitations

• Rule-based optimization is **data-independent**
  - Optimal selection of **operators** impossible without estimates about size of results (cardinality, width)
  - No rules for order of **join processing**
  - Rules are partly contradictory
    • E.g. Conjunctive selections and composite indexes
  - Benefit of indexes depends on selectivity
  - ...

• Option 2: **Cost-based optimization**
  - Estimate effect of rewritings on size of intermediate results (SIR)
  - Different optimization goals
    • Greedy: Chose next rewrite with greatest saving in SIR
    • Global: Chose plan with overall smallest SIR
    • Bound: Chose plan with **smallest maximal SIR**
Order of Joins: Indistinguishable
Join Order – Does it Matter?

- Assume uniform distributions
  - There are 1,000 students, 20 professors, 80 courses
  - Each professor gives 4 courses
  - Each student listens to 4 courses
  - Each course is followed by 50 students (4,000 “hören” tuples)
Join Order – Does it Matter?

- **Compute** $\sigma_{\text{Sokrates}}(P) \bowtie (V \bowtie (S \bowtie H))$
  - Inner join: $1000 \times 4 = 4000$ tuples
  - Next join: Again 4000 tuples
  - Last join selects only 1/20 of intermediate results = 200
  - Intermediate result sizes: $4000 + 4000 + 1 = 8001$

- **Compute** $S \bowtie (H \bowtie (\sigma_{\text{Sokrates}}(P) \bowtie V))$
  - Inner join selects 4 tuples
  - Next join generates $50 \times 4 = 200$ tuples
  - Last join: No change
  - Intermediate result sizes: $1 + 4 + 200 = 205$
Content of this Lecture

- Introduction
- Rewriting Subqueries
- Algebraic Term Rewriting
- Optimizing Join Order
- Plan Enumeration
- A counter-example
Optimizing Join Order

- From the relation algebra perspective, join is **associative and commutative** - reordering doesn’t change result
- But execution times of different orders differ tremendously
  - If there are at least two joins, e.g. $R \bowtie (S \bowtie T) \equiv (S \bowtie R) \bowtie T$
- **Join versus cross-product**
  - Depending on join conditions, many orders involve **intermediate cross-products**
  - Most join-order algorithms **disregard any plan** containing a cross-product - which heavily reduces the search space
  - In the following, we assume that no order involves a cross-product
- Given $n$ relations, there are $n!$ **possible orders**
Query Types

- \(((S \bowtie R) \bowtie T) \bowtie L\)
- \(((S \bowtie L) \bowtie R) \bowtie T\)
- ...

- \(((P \bowtie G) \bowtie T) \bowtie B\)
- \((P \bowtie G) \bowtie (T \bowtie B)\)
- ...

...
Left/Right-deep versus Bushy Join Trees

- There is one left-deep tree topology, but still $O(n!) \text{ orders}$
- There are $(2n-3)!/(2^{n-2}(n-2)!) \text{ unordered binary trees with n leaves, and for each } O(n!) \text{ orders}$
  - Some are equivalent
Choosing a Join Order

- Typical first heuristic: Consider **only left-deep trees**
  - Can be **pipelined efficiently**
  - Usually generates among the best plans
- But there are still $O(n!)$ possible orders
- Second Heuristic: Use **dynamic programming with pruning**
  - Generate plans bottom up: Plans for pairs, triples, ...  
  - For each join group, keep only best plan
  - Use these to enumerate possibilities for larger join groups
  - Prune all subplans containing a cross product
Join Groups

- There are \((n \text{ over } i)\) join groups with \(i\) elements
Details

• Create a table containing for each join group
  - [Prune if this would involve a Cartesian product]
  - Estimated size of result (how: later)
  - **Optimal cost** for computing this group
    • For now, we simply take sum of sizes of intermediate results so far
  - **Optimal plan** for computing this group
Induction

- Induction over plan length = sizes of join groups
  - $i=1$: Consider every relation in isolation
    - Size = Size of relation
    - Cost = 0 (access costs are fixed for all plans anyway)
      - Not true is pushing of selections is considered
  - $i=2$: Consider each pair of relations
    - Size: Estimated size of “joining” both relations (might be product)
    - Cost = 0 (no intermediate result so far due to previous assumption)
    - Fix join method to use (e.g.: BNL with smaller relation as inner relation)
      - This method will never change again
  - $i=3$: Consider each pair in each triple and join with third relation
    - Consider only chosen methods for pairs involved
Example 1

- We join four relations R, S, T, U
- Four join conditions

\[
\begin{array}{c|c|c|c|c}
 & \{R\} & \{S\} & \{T\} & \{U\} \\
\hline
\text{Kardinalität} & 1000 & 1000 & 1000 & 1000 \\
\text{Kosten} & 0 & 0 & 0 & 0 \\
\text{Optimaler Plan} & \text{scan}(R) & \text{scan}(S) & \text{scan}(T) & \text{scan}(U) \\
\end{array}
\]
Example 2

<table>
<thead>
<tr>
<th></th>
<th>{R, S}</th>
<th>{R, T}</th>
<th>{R, U}</th>
<th>{S, T}</th>
<th>{S, U}</th>
<th>{T, U}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kardinalität</td>
<td>5000</td>
<td>1M</td>
<td>10000</td>
<td>2000</td>
<td>1M</td>
<td>1000</td>
</tr>
<tr>
<td>Kosten</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>opt. Plan</td>
<td>R \bowtie S</td>
<td>R \bowtie T</td>
<td>R \bowtie U</td>
<td>S \bowtie T</td>
<td>S \bowtie U</td>
<td>T \bowtie U</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>{R, S, T}</th>
<th>{R, S, U}</th>
<th>{R, T, U}</th>
<th>{S, T, U}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kardinalität</td>
<td>10000</td>
<td>50000</td>
<td>10000</td>
<td>2000</td>
</tr>
<tr>
<td>Kosten</td>
<td>2000</td>
<td>5000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>opt. Plan</td>
<td>(S \bowtie T) \bowtie R</td>
<td>(R \bowtie S) \bowtie U</td>
<td>(T \bowtie U) \bowtie R</td>
<td>(T \bowtie U) \bowtie S</td>
</tr>
</tbody>
</table>

Prune products

Better than \(S \bowtie (T \bowtie R)\) and \((R \bowtie S) \bowtie T\)
Example 3

<table>
<thead>
<tr>
<th></th>
<th>{R,S,T}</th>
<th>{R,S,U}</th>
<th>{R,T,U}</th>
<th>{S,T,U}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kardinalität</td>
<td>10000</td>
<td>50000</td>
<td>10000</td>
<td>2000</td>
</tr>
<tr>
<td>Kosten</td>
<td>2000</td>
<td>5000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>opt. Plan</td>
<td>((S \bowtie T) \bowtie R)</td>
<td>((R \bowtie S) \bowtie U)</td>
<td>((T \bowtie U) \bowtie R)</td>
<td>((T \bowtie U) \bowtie S)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Plan</th>
<th>Kosten</th>
</tr>
</thead>
<tbody>
<tr>
<td>((S \bowtie T) \bowtie R) \bowtie U\</td>
<td>12k</td>
</tr>
<tr>
<td>((R \bowtie S) \bowtie U) \bowtie T\</td>
<td>55k</td>
</tr>
<tr>
<td>((T \bowtie U) \bowtie R) \bowtie S\</td>
<td>11k</td>
</tr>
<tr>
<td>((T \bowtie U) \bowtie S) \bowtie R\</td>
<td>3k</td>
</tr>
</tbody>
</table>

(Hopefully) optimal left-deep plan
Algorithm

Input: SPJ query $q$ on relations $R_1, \ldots, R_n$
Output: A query plan for $q$

1: for $i = 1$ to $n$ do {
2:     $\text{optPlan}\{R_i\} = \text{accessPlans}(R_i)$
3:     $\text{prunePlans}(\text{optPlan}\{R_i\})$
4: }
5: for $i = 2$ to $n$ do {
6:     for all $S \subseteq \{R_1, \ldots, R_n\}$ such that $|S| = i$ do {
7:         $\text{optPlan}(S) = \emptyset$
8:         for all $O$ such that $S \cup X = O$
9:             $\text{optPlan}(S) = \text{optPlan}(S) \cup \text{joinPlans}(\text{optPlan}(O), X)$
10:        $\text{prunePlans}(\text{optPlan}(S))$
11:    }}
12: }
13: return $\text{optPlan}\{R_1, \ldots, R_n\}$
Dynamic Programming

• DP here is a heuristic
  - Assumption of DP: **Any subplan of an optimal plan is optimal**
  - True for computing shortest paths, edit distance, …

• But not true **for join-order**
  - Using a sort-merge join early in a plan might not be optimal for this particular join group - but result is sorted
  - **Later joins can profit** and also use sort-merge without sorting one intermediate relation again
  - Optimal plan might involve Cartesian product

• Solution (for sort order)
  - Keep different “optimal” plans for each join group
  - **System R**: One “optimal” plan per **interesting sort order**
Content of this Lecture

- Introduction
- Rewriting Subqueries
- Algebraic Term Rewriting
- Optimizing Join Order
- Plan Enumeration
- A counter-example
Ingredients

• We can evaluate different access paths for a single relation
• We can generate various equivalent relational algebra terms for computing a query
• We can optimize join order
  – Given selectivity estimates
• Query optimization =
  Search space (space of all possible plans) +
  Search strategy (algorithm to enumerate plans) +
  Cost functions for pruning plans (still missing)
Search Strategies

• Searching a huge search space for a good (optimal) solution is a common computer science problem
  - Exhaustive search
    • Guarantees optimal result, but often too expensive
  - Heuristic method
    • Greedy/Hill-Climbing: only use one alternative for further search
  - Genetic optimization
    • Generate some good plans
    • Build combinations
  - Simulated annealing
  - …

Content of this Lecture

- Introduction
- Rewriting Subqueries
- Algebraic Term Rewriting
- Optimizing Join Order
- Plan Enumeration
- A counter-example
Typische Anfrage gegen Star Schema
- Aggregation und Gruppierung
- Bedingungen auf den Werten der Dimensionstabellen
- Joins zwischen Dimensions- und Faktentabelle
Beispielquery


```
SELECT T.year, sum(amount*price)
FROM Sales S, Product P, Time T, Localization L
WHERE P.pg_name='Wasser' AND
  P.product_id = S.product_id AND
  T.day_id = S.day_id AND
  T.month = '1' AND
  L.shop_id = S.shop_id AND
  L.region_name='Berlin'
GROUP BY T.year
```
**Anfrageplanung**

- Anfrage enthält 3 Joins über 4 Tabellen
- Zunächst 4! left-deep join trees
  - Aber: Nicht alle Tabellen sind mit allen gejoined
- Nur 3! beinhalten **kein Kreuzprodukt**

![Diagram](attachment://diagram.png)
Heuristiken

- Typisches Vorgehen
  - Auswahl des Planes nach Größe der Zwischenergebnisse
  - Keine Beachtung von Plänen, die kartesisches Produkt enthalten

Kartesisches Produkt

\[ \sigma_{\text{pg\_name}=',\text{Wasser}'}, \sigma_{\text{region\_name}=',\text{Berlin}'}, \sigma_{\text{year \ in \ (1997,1998, 1999)}}, \sigma_{\text{month}=1} \]

\[ \times \]

\[ \sigma_{\text{Sales}}, \sigma_{\text{Product}}, \sigma_{\text{Location}}, \sigma_{\text{Time}} \]
Abschätzung von Zwischenergebnissen

Annahmen
- \( M = |S| = 100,000,000 \)
- 20 Verkaufstage pro Monat
- Daten von 10 Jahren
- 50 Produktgruppen a 20 Produkten
- 15 Regionen a 100 Shops
- Gleichverteilung aller Verkäufe

Größe des Ergebnis
- Selektivität Zeit
  - 60 Tage:
    \( \frac{M}{(20 \times 12 \times 10)} \times 3 \times 20 \)
- Selektivität 'Wasser'
  - 20 Produkte
    \( \frac{M}{(20 \times 50)} \times 20 \)
- Selektivität 'Berlin'
  - 100 Shops
    \( \frac{M}{(15 \times 100)} \times 100 \)
- Gesamt
  - 3.333 Tupel
  - Selektivität: 0,00003%
Left-deep Pläne

### Zwischenergebnis

<table>
<thead>
<tr>
<th>Schritt</th>
<th>Operation</th>
<th>Ergebnis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Join</td>
<td>((M / 15))</td>
<td>6.666.666</td>
</tr>
<tr>
<td>2. Join</td>
<td>((\lvert J_1 \rvert \cdot 3/120))</td>
<td>166.666</td>
</tr>
<tr>
<td>3. Join</td>
<td>((\lvert J_2 \rvert / 50))</td>
<td>3.333</td>
</tr>
</tbody>
</table>

### Zwischenergebnis

<table>
<thead>
<tr>
<th>Schritt</th>
<th>Operation</th>
<th>Ergebnis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Join</td>
<td>((M / 50))</td>
<td>2.000.000</td>
</tr>
<tr>
<td>2. Join</td>
<td>((\lvert J_1 \rvert \cdot 3/120))</td>
<td>50.000</td>
</tr>
<tr>
<td>3. Join</td>
<td>((\lvert J_2 \rvert / 15))</td>
<td>3.333</td>
</tr>
</tbody>
</table>
Plan mit kartesischen Produkten

<table>
<thead>
<tr>
<th></th>
<th>/Zwischenergebnis/</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Time x Location</td>
<td>6.000</td>
</tr>
<tr>
<td>(3*20 * 100)</td>
<td></td>
</tr>
<tr>
<td>2. ... x Product</td>
<td>120.000</td>
</tr>
<tr>
<td>(</td>
<td>P_1</td>
</tr>
<tr>
<td>3. ... ⋈ Sales</td>
<td>3.333</td>
</tr>
</tbody>
</table>

- Es gibt mehr „Zellen“ als Verkäufe
- Nicht an jedem Tag wird jed. Produkt in jed. Shop verkauft
STAR Join in Oracle (v7)

- **STAR Join Strategie in Oracle v7**
  - Kartesisches Produkt aller Dimensionstabellen
  - Zugriff auf *Faktentabelle über Index*
    - Hohe Selektivität für Anfrage wichtig
    - Zusammengesetzter Index auf allen FKs muss vorhanden sein
    - Sonst „nur“ kleinere Zwischenergebnisse, aber trotzdem teurer Scan

- **Aber: Nicht immer gut**
  - Daten für 3 Monate, 10 Jahre, 5 Regionen, 10 Produktgruppen
  - Größe des *kartesischen Produkts*:
    \[3 \times 20 \times 10 \times 5 \times 100 \times 10 \times 20 = 60.000.000\]
STAR Join in Oracle 8i – 9i

- Möglichkeit der (komprimierten) Bitmapindexe lässt kartesisches Produkt weniger vorteilhaft erscheinen

- Phasen
  1. Berechnung aller FKs in Faktentabelle gemäß Dimensionsbedingungen einzeln für jede Dimension
  2. Anlegen/laden von Join-Bitmapindexen auf allen FK Attributen der Faktentabelle
  3. Merge (AND) aller Bitmapindexe
  4. Direkter Zugriff auf Faktentabelle über TID
  5. Join nur der selektierten Fakten mit Dimensionstabellen zum Zugriff auf Dimensionswerte

- Zwischenergebnisse sind nur (komprimierte) Bitlisten
Phase 1

Phase 2

SELECT STATEMENT
SORT GROUP BY
HASH JOIN
TABLE ACCESS FULL
HASH JOIN
TABLE ACCESS FULL
HASH JOIN
TABLE ACCESS FULL
PARTITION RANGE ALL
TABLE ACCESS BY LOCAL INDEX ROWID
BITMAP CONVERSION TO ROWIDS
BITMAP AND
BITMAP INDEX SINGLE VALUE
BITMAP MERGE
BITMAP KEY ITERATION
BUFFER SORT
TABLE ACCESS FULL
BITMAP INDEX RANGE SCAN
BITMAP MERGE
BITMAP KEY ITERATION
BUFFER SORT
TABLE ACCESS FULL
BITMAP INDEX RANGE SCAN

LOCATION
TIME
PRODUCT
SALES

SALES_L_BJIX
SALES_P_BIX
SALES_TIME_BIX