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 assess different physical plans to answer the query
  – All plans must be semantically equal

• Optimization itself costs time
  – Some steps have exponential complexity
    • E.g. join order: 10 joins – potentially $\sim 3^{10}$ steps
  – Finding the best plan might take more time than executing an arbitrary plan
    • And usually we don’t find the best plan anyway

• Why bother?
Example

```sql
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
  \[ \pi_{\text{name,adr}}(\sigma_{O.C\_name=C\_name \land O\_product='coffee'}(C \times O)) \]

• Interpret query „from inner to outer“
  – No optimization yet

• Assume materialization of intermediate results
  – No caching, no pipelining
Cost

- Compute cross-product (block-nested-loop)
  - 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: $\sim 686,000$ IO
  - and 333,000 blocks temp space required on disk
Query Rewriting

- Rewrite into: \[ \pi_{\text{name, adr}} (C \bowtie_{O.c\_name=C.name} (\sigma_{O.product='coffee'}(O))) \]
- Compute selection on \( O \)
  - Reads: 1,000, writes: \( 50/10 = 5 \)
- **Compute join** using BNL
  - Reads: \( 5 + b(C) \times 5 = 105 \)
  - Writes: \( 50/3 \approx 17 \)
- Compute projection
  - Reads: 17, writes: \( 50/50 \approx 1 \)
- **Altogether**: 1.145
  - 17 blocks temp space
- Maybe there is an ever better way?
Better Plan

• **Push projection**
  - \( \pi_{\text{name,adr}}(\pi_{\text{name,adr}}(C) \bowtie_{O.c\_name=C.name}(\sigma_{O.product='coffee'}(O))) \)

• **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 ~ 17

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

• **Altogether:** 1.080
  - 17 blocks temp space
Even Better – Use Indexes

- Assume indexes on \((O.\text{product}, O.\text{C\_name})\) and on \((\text{C\_name}, \text{C\_address})\)

- Compute **selection on O using index**
  - Reads: Roughly between 5 and 10 blocks
    - 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: \(\sim 5*\log(5) \sim 15\)
    - 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  
    - No access to data blocks necessary  
  - Writes: 50/3 ~ 17  
- Compute projection  
  - Reads: 17, writes: 50/50 ~ 1  
- Altogether: between 85 and 90  
  (requiring 17 blocks on disk)
**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 $\sim 8.000$
- DB should invest some time in optimization
Steps in Optimization

- Parsing, view expansion, **subquery rewriting**
- **Query minimization** (maybe)
- Generation of query tree
- Plan optimization
  - **Algebraic query rewriting** (logic optimization)
  - **Cost estimation** (cost-based optimization)
  - Plan instantiation (physical optimization)
  - Plan enumeration and pruning
  - Note: These steps are executed in an **interleaved fashion**
- Selection of best plan
- Code generation (compilation or interpretation)
Content of this Lecture

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

- No equivalent in relational algebra: IN, EXISTS, ALL, MINUS, INTERSECT, UNION ...
  - Generate subtrees with non-relational root node
  - For optimization, a fully relational tree is easier to handle
  - But: Transformation not always easy, not always advantageous

- We look at four cases of IN
  - A subquery \( p \) is called correlated if it refers to a variable declared in the outer query
  - Uncorrelated without aggregation
  - Uncorrelated with aggregation
  - Correlated without aggregation
  - Correlated with aggregation

- See literature for other predicates
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 (hence order:product is 1:n), example is fine
    • Otherwise, we need to **introduce a DISTINCT**

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

```
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 expressing such functionality: User-defined table functions
  - This would allow formulation as join
  - But even harder to optimize
- Third way: Use view (two queries)
  - Optimization problem does not change
Correlated Subquery without Aggregation

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

- For correlated sqs, isolated materialization is impossible
- 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 not 1:n, DISTINCT required)
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)
```

- Materialization not possible (correlation)
- Rewrite into join not possible (aggregation)
- Naïve computation requires one execution of subquery for each tuple of outer query
- Solution: Rewrite into two queries
  - That are optimized in isolation
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)
```

• Query 1
  - Computes inner query result for all tuples of o
  - Can be materialized

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

• Query 2

```
SELECT o.o_id
FROM order o, all_sums
WHERE o.total_price != all_sums.tp
```
Subquery rewriting Wrap-Up

- Some subqueries with IN can be rewritten in single SPJ queries, some not
  - A syntactical rewrite is always possible using views
  - This doesn’t help the optimizer, but the developer
- Same holds true for other “unusual” predicates: EXISTS, NOT IN, INTERSECT, UNION
  - Many detailed rules; see, e.g., Seshadri et al. (1996). Complex query decorrelation. ICDE; Elhemali et al. (2007). Execution strategies for SQL subqueries. SIGMOD
- Special problems occur, when subqueries appear multiple times in a single query
  - Syntax: Use “WITH” predicate
  - Optimization: Detection of redundant query fragments
Content of this Lecture

- Introduction
- Rewriting Subqueries
- Query Minimization
- Algebraic Term Rewriting
- Optimizing Join Order
- Plan Enumeration
- A counter-example
Query Minimization 1

- Especially important when views are involved or queries are created programmatically

```
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

-- Optimization: Remove redundant condition
```
Query Minimization 2

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

```
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

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

```
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
```

• Optimization: Remove redundant joins
Techniques (sketch)

- Group conjunctive conditions with constants per attribute and compute **minimal intervals** (or find contradictions)
  - Different techniques for OR, XOR, NOT
- Equi-Joins: Build join graph, compute transitive closure, and find **minimal spanning tree**
  - Be careful with join attributes – must all be the same
  - “Minimal” already assumes a cost estimate (later)
  - Different MST’s – different logical plans – different optimized plans – different runtimes
- **Theta-Joins**: Translate into propositional logical formula and test for soundness
- ...
- [I don’t think that real systems do much of this]
Content of this Lecture

- Introduction
- Rewriting Subqueries
- Query Minimization
- **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 $R_1$ . . . $R_n$
  
  – For any instances of $S$, $E_1$ and $E_2$ compute the same result

• Optimizers generate equivalent expressions by applying provably correct rewrite rules
  
  – Testing if two query are equivalent is a different topic

• We look at a dozen of such rules
  
  – There exist more (see literature)
Rules for Joins and Products

- Assume
  - $E_1$, $E_2$, $E_3$ are relational expressions (queries)
  - $Cond$, $Cond1$, $Cond2$ are (equi-)join conditions

- Rule 1: Joins and Cartesian-products are **commutative**
  \[
  E_1 \bowtie_{Cond} E_2 \equiv E_2 \bowtie_{Cond} E_1 \\
  E_1 \times E_2 \equiv E_2 \times E_1
  \]

- Rule 2: Joins and Cartesian-products are **associative**
  \[
  (E_1 \bowtie_{Cond1} E_2) \bowtie_{Cond2} E_3 \equiv E_1 \bowtie_{Cond1} (E_2 \bowtie_{Cond2} E_3)
  \]
  Requirement: $E_3$ joins with $E_2$ (and not with $E_1$)
  \[
  (E_1 \times E_2) \times E_3 \equiv E_1 \times (E_2 \times E_3)
  \]
Projections and Selections

• Assume
  – $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$ are attributes of $E$
  – $Cond1$ and $Cond2$ are 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
  $$\sigma_{\text{Cond1}} (\sigma_{\text{Cond2}} (E)) \equiv \sigma_{\text{Cond2}} (\sigma_{\text{Cond1}} (E))$$
  $$\equiv \sigma_{\text{Cond1 and Cond2}} (E)$$
Projections and Selections Part 2

• Assume
  – $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$ are attributes of $E$
  – $\text{Cond1}$ and $\text{Cond2}$ are conditions on $E$

• Rule 5a. **Exchange** of projection and selection

$$
\pi_{\{A_1, \ldots, A_n\}} (\sigma_{\text{Cond}}(E)) \equiv \sigma_{\text{Cond}} (\pi_{\{A_1, \ldots, A_n\}}(E))
$$

Requirement: $\text{Cond}$ contains only attributes $A_1, \ldots, A_n$

• Rule 5b. **Injection** of projection

$$
\pi_{\{A_1 \ldots A_n\}} (\sigma_{\text{Cond}}(E)) \equiv \pi_{\{A_1 \ldots A_n\}} (\sigma_{\text{Cond}} (\pi_{\{A_1 \ldots A_n, B_1 \ldots B_m\}}(E)))
$$

Requirement: $\text{Cond}$ contains only attributes $A_1 \ldots A_n$ and $B_1 \ldots B_m$
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: \text{Cond} contains only attributes of \( E_1 \)

- 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 $E_1$ and $B_1 \ldots B_m$ appear in $E_2$

- Rule 10. Exchange of projection and union:

$$\Pi_{\{A_1, \ldots, A_n\}}(E_1 \cup E_2) \equiv \Pi_{\{A_1, \ldots, A_n\}}(E_1) \cup \Pi_{\{A_1, \ldots, A_n\}}(E_2)$$
Special Case: Cartesian Product versus Joins

- “Pure” relational algebra has no joins
- The merge of a Cartesian Product and a join condition into a join operator actually is a physical optimization
  - Replace implementation of two operators by one
  - But so common that it is applied on the logical level
- Trick: Define join operator and define a rewrite rule

Rule 11: Turn Cartesian Products and \( cond \) into join

\[
\sigma_{cond} \left( E_1 \times E_2 \right) \equiv E_1 \bowtie_{cond} E_2
\]
Example

- Query on a CUSTOMER database

```
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

Name, Account#, Savings

π

σ

×

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

×

customer

account

journal
Breaking and Pushing Selections

\[ \Pi_{\text{Name, Account#, Savings}} \]
\[ \sigma_{\text{C.Account#=J.Account#}} \]
\[ \times_{\text{C.Account#=A.Account#}} \]
\[ \sigma_{\text{"Bond"\leq\text{Name}, Name\leq\"Carter", Address=\"World"}} \]
\[ \times_{\text{\sigma}} \]
\[ \Pi \]
\[ \sigma_{\text{Transac=\"Withdraw\", Amount>1000000}} \]
\[ \times_{\text{ACCOUNT}} \]
\[ \sigma_{\text{CUSTOMER}} \]
\[ \times_{\text{Journal}} \]
Introduce Joins

\[ \Pi \sigma \sigma \bowtie \sigma \bowtie \Pi \]
Pushing Projections

CUSTOMER

ACCOUNT

Journal

Name,Account#, Savings

Name,Account#

Name,Account#, Address

Ulf Leser: Implementation of Database Systems
Caution

- Sometimes, **pushing up selections temporarily** is beneficial
  - Especially for conditions on join attributes
- **Example** (assume both actsin and movie have a year attribute)

\[
\text{CREATE VIEW movies99 AS SELECT title, year, studio FROM movie WHERE year=1999 JOIN movie ON σ(year=99)} \]

\[
\text{SELECT m.title, a.name FROM movies99 m, actsin a WHERE m.title=a.title AND m.year=a.year} \]

- If this tree is generated in first place ...
Term Rewriting: Algebraic Optimization

• Usually there are infinitely many rewrite steps
  – But not infinitely many different plans
  – Rewritings may go back and forth

• General heuristic: **Minimize intermediate results**
  – Less IO if materialization is necessary
  – Less work for operations that are higher in the plan

• Option 1: **Rule-based**
  – Old school, simple

• Option 2: **Cost-Based**
  – State-of-the-art, more complex
Rule Based Query Optimization (RBO)

- Goal: Find a finite order in which rewrite steps are applied such that the final plan is faster than the original plan

- Rule-based optimization
  - Consider all rules regardless of the concrete database instance
  - Use heuristics for prioritizing rewrite rule
  - Based on experience – rules that are beneficial in most cases
  - Simple to implement, fast optimization
  - But: Most real instances lead to non-optimal plans
    - Though hopefully still better than the original plan
A Simple Rule-Based Optimizer

• First down: Break and push down conditions
  – Break conjunctive selections into sets of atomic selections
  – Break combined projections into atomic projections
  – Push selects/projects as deep down the tree as possible

• Then up: Merge operations
  – Replace selection and Cartesian product with join
  – Merge neighboring atomic selections into combined selections
  – Merge neighboring atomic projections into combined projections

• Avoid Cartesian Products (if possible)
  – Choose other join order, start optimization again

• Finally physical: Choose concrete implementations
  – If there is a condition on an indexed attribute – use the index
  – For a join over PK-FK relationships: Use sort-merge
  – Other joins: Use hash join
Example

```
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

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

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

\[ \times \]

\[ \times \]

\[ p \]

\[ v \]

\[ h \]

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

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

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

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

\[
\pi_s.\text{Semester} \quad \sigma_p.\text{PersNr}=v.\text{gelesenVon} \quad \sigma_v.\text{VorlNr}=h.\text{VorlNr} \quad \sigma_s.\text{MatrNr}=h.\text{MatrNr} \quad \sigma_p.\text{Name} = '\text{Sokrates}'
\]

\[
\pi_s.\text{Semester} \quad \sigma_p.\text{PersNr}=v.\text{gelesenVon} \quad \sigma_v.\text{VorlNr}=h.\text{VorlNr} \quad \sigma_p.\text{Name} = '\text{Sokrates}'
\]

\[
\sigma_s.\text{MatrNr}=h.\text{MatrNr}
\]
Rewrite Product+Selection into Joins

\[ \pi_{s.\text{Semester}} \]
\[ \sigma_{p.\text{PersNr}=v.\text{gelesenVon}} \]
\[ \sigma_{v.\text{VorlNr}=h.\text{VorlNr}} \]
\[ \sigma_{s.\text{MatrNr}=h.\text{MatrNr}} \]
\[ \sigma_{p.\text{Name} = 'Sokrates'} \]
\[ \sigma_{v.\text{VorlNr}=h.\text{VorlNr}} \]
\[ \sigma_{s.\text{MatrNr}=h.\text{MatrNr}} \]
\[ \pi_{s.\text{Semester}} \]
Break and Push Projections

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

\[ \pi_{\text{MatrNr,semester}} \]
\[ \pi_{\text{MatrNr,vorlNr}} \]
Limitations

- RBO is **data-independent**
- Optimal selection of **operators** impossible without estimates about size of results (cardinality, width)
  - Best index, best join method, best join order – all depend on the concrete input and output of an operation
- No rules for order of **join processing**
- Rules are partly contradictory
  - E.g. Conjunctive selections and composite indexes
Order of Joins: Indistinguishable

\[
\pi_{s.\text{Semester}} \leftarrow p.\text{PersNr}=v.\text{gelesenVon} \\
\sigma_{p.\text{Name} = 'Sokrates'} \\
\pi_{s.\text{Semester}} \leftarrow s.\text{MatrNr}=h.\text{MatrNr} \\
\sigma_{v.\text{VorlNr}=h.\text{VorlNr}} \\
\pi_{s.\text{Semester}} \leftarrow v.\text{VorlNr}=h.\text{VorlNr} \\
\pi_{s.\text{Semester}} \leftarrow v.\text{VorlNr}=h.\text{VorlNr} \\
\pi_{s.\text{Semester}} \leftarrow p.\text{PersNr}=v.\text{gelesenVon} \\
\sigma_{p.\text{Name} = 'Sokrates'} \\
\pi_{s.\text{Semester}} \leftarrow s.\text{MatrNr}=h.\text{MatrNr} \\
\pi_{s.\text{Semester}} \leftarrow v.\text{VorlNr}=h.\text{VorlNr} \\
\pi_{s.\text{Semester}} \leftarrow p.\text{PersNr}=v.\text{gelesenVon} \\
\sigma_{p.\text{Name} = 'Sokrates'}
\]
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: $4000 + 4000 + 200 = 8200$

- **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: $4 + 200 + 200 = 404$
Cost-Based Query Optimization (CBO)

- **Goal**: Find the plan that is **cheapest among all possible plans** given a **cost model**
  - “Possible” – Created by a finite sequence of applying rewrite rules
- **Cost-based optimization**
  - Use a clever algorithm to enumerate all possible plans
  - Estimate effect of all individual rewritings regarding your cost model
  - Use this to compute a cost per plan
  - Prune parts of the search space wherever possible
  - Choose cheapest
- **Variations: Other optimization goals**
  - Global: Chose plan with smallest sum of intermediate results
  - Bound: Chose plan with **smallest maximal** intermediate result
Enumerating Query Plans

- Assume a plan $P$ of size $p = |P|$ with $j$ joins
  - Size: Number of predicates in the plan
- Rewritings may ...
  - Merge / break predicates
    - Creates $p \pm c$ plans for some constant $c$
  - Move predicates up/down the tree
    - Creates $p$ different plans per predicate
  - Change order of joins (or Cartesian products)
    - Need to consider concrete join predicates
    - Creates in worst case more than $j!$ different plans (see later)
- Typical plan enumeration strategy
  - Push predicates as deep as possible
  - Find best join order
Content of this Lecture

- Introduction
- Rewriting Subqueries
- Query Minimization
- 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
- Join versus Cartesian 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 Cartesian Product
Query Types

- **Star join**
  - \(((S \bowtie R) \bowtie T) \bowtie L\)
  - \(((S \bowtie L) \bowtie R) \bowtie T\)
  - ...

- **Chain join**
  - \(((P \bowtie G) \bowtie T) \bowtie B\)
  - \((P \bowtie G) \bowtie (T \bowtie B)\)
  - ...

---

Ulf Leser: Implementation of Database Systems 57
Left/Right-deep versus Bushy Join Trees

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

• Typical first heuristic: Consider only left-deep trees
  – Used, for instance, in Oracle
  – Can be pipelined efficiently
  – Usually generates among the best plans
  – But suboptimal if parallel execution is possible
• 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 Cartesian Product
  – Still is a heuristic - later
Join Groups

- There are \((n \over i)\) join groups with \(i\) elements
• Create a table containing for each join group
  – [Prune if this would involve a Cartesian product]
  – Estimated size of result (how: later)
  – **Optimal (minimal) cost** for computing this group
    • For now, we simply take sum of sizes of intermediate results in the subtree representing this group
  – **Optimal plan** for computing this group

• **We here assume that**
  – There are no further selections
  – There are no indexes (and hence no index-join)
Induction

• Induction over sizes of join groups
  – i=1: Consider every relation in isolation
    • Size = Size of relation
    • Cost = 0 (access costs of leaf nodes are identical for all plans)
  – i=2: Consider each pair of relations
    • Remove if there is no join condition
    • Size: Estimated size of join result
    • 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 optimal methods for all pairs involved
    • ...
Example 1

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

<table>
<thead>
<tr>
<th></th>
<th>{R}</th>
<th>{S}</th>
<th>{T}</th>
<th>{U}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kardinalität</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>Kosten</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Optimaler Plan</td>
<td>scan(R)</td>
<td>scan(S)</td>
<td>scan(T)</td>
<td>scan(U)</td>
</tr>
</tbody>
</table>
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 ⊙ S</td>
<td>T ⊙ R</td>
<td>R ⊙ U</td>
<td>S ⊙ T</td>
<td>S ⊙ U</td>
<td>T ⊙ U</td>
</tr>
</tbody>
</table>

Prune CPs

Better than \( S \bowtie (T \times R) \) 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 ⊗ T) ⊗ R</td>
<td>(R ⊗ S) ⊗ U</td>
<td>(T ⊗ U) ⊗ R</td>
<td>(T ⊗ U) ⊗ S</td>
</tr>
</tbody>
</table>

#### Plan and Kosten

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

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

• DP is a heuristic for join order optimization
  – 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
  – Truly optimal plan might involve Cartesian Products (example later)

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

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

- 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
- Star-Join: Nur 3! beinhalten kein Kreuzprodukt
Heuristiken

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

```
σ region_name='Berlin'
σ pg_name='Wasser'
```

Kartesisches Produkt

```
σ month=1
```

```
Product
Location
Sales
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: \( (M / (20*12*10)) * 3*20 \)
- Selektivität 'Wasser'
  - 20 Produkte \( (M / (20*50)) * 20 \)
- Selektivität 'Berlin'
  - 100 Shops \( (M / (15*100)) * 100 \)
- Gesamt
  - 3.333 Tupel
- Selektivität: 0,00003%

```sql
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
```
Left-deep Pläne

Zwischenergebnis

1. Join (M / 15) | 6.666.666
2. Join (|J₁|*3/120) | 166.666
3. Join (|J₂|/50) | 3.333

Zwischenergebnis

1. Join (M / 50) | 2.000.000
2. Join (|J₁|*3/120) | 50.000
3. Join (|J₂|/ 15) | 3.333
Plan mit kartesischen Produkten

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

- Wie optimiert man Star-Joins?
- Siehe Modul „Data Warehousing“