Datenbanksysteme II: Query Optimization

Ulf Leser
5 Layer Architecture

- Data Model
- Logical Access
- Data Structures
- Buffer Management
- Operating System
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 query plans are *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

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

• Interpret query „from left to right“
  - 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

- Algebraic expression can be rewritten
  - \( \pi_{\text{name,adr}}(C \bowtie_{O.\text{c_name}=C.\text{name}}(\sigma_{O.\text{product}=\text{coffee}}(O))) \)

- Compute selection on O
  - Reads: 1.000, writes: 50/10 = 5

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

- Compute projection
  - Reads: 17, writes: 50/50 \approx 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: $50/3 \approx 17$

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

- **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
    - 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 \(~8.000\)
- Conclusion: DB should invest some time in optimization
Steps in Optimization

- Parsing, view expansion, \textbf{subquery rewriting}
- Query minimization (maybe)
- Expression/tree generation
- Plan optimization
  - Term rewriting (logic optimization)
  - Cost estimation (cost-based optimization)
  - Plan instantiation (physical optimization)
  - Plan \textit{enumeration and pruning}
  - Note: \textbf{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

<table>
<thead>
<tr>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Address</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>O_id</td>
</tr>
<tr>
<td>C_name</td>
</tr>
<tr>
<td>P_Id</td>
</tr>
<tr>
<td>Date</td>
</tr>
<tr>
<td>Total_price</td>
</tr>
<tr>
<td>revenue</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Delivery</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
</tr>
<tr>
<td>O_ID</td>
</tr>
<tr>
<td>Date</td>
</tr>
<tr>
<td>Price</td>
</tr>
<tr>
<td>Quantity</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
</tr>
<tr>
<td>P_Name</td>
</tr>
<tr>
<td>Price</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
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)

```
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**
Correlated Subquery without Aggregation

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

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

\[
\begin{align*}
\text{SELECT DISTINCT} & \quad o.o\_id \\
\text{FROM} & \quad \text{order } o, \text{delivery } d \\
\text{WHERE} & \quad o.o\_id = d.o\_id \text{ AND } \\
& \quad \quad \quad d.date - o.date < 5
\end{align*}
\]
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
- Naïve computation again 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 q1 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 q1
      WHERE q1.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 q1
   WHERE q1.o_id = o.o_id)

SELECT o.o_id
FROM order o, q1
WHERE o.total_price != q1.tp
Impovements

- Inner query is computed only 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 “filter”
  SELECT name
  FROM good_business
  WHERE revenue > 5.000

- Optimization: Remove redundant conditions
  SELECT C.name
  FROM customer C, order O
  WHERE C.name = O.name AND O.revenue > 1.000 AND O.revenue > 5.000
```
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 with short query

```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
```

• Optimization: Remove redundant joins
Content of this Lecture

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

• Usually, we generate equivalent expressions by applying 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
  - $Cond, Cond1, Cond2$ are join conditions

• Rule 1: Join and product 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: Join and product are **associative**
  - $(E_1 \bowtie_{Cond1} E_2) \bowtie_{Cond2} E_3 \equiv E_1 \bowtie_{Cond1} (E_2 \bowtie_{Cond2} E_3)$
  - $(E_1 \times E_2) \times E_3 \equiv E_1 \times (E_2 \times E_3)$
For Projection and Selection

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

- Rule 5a. Exchange of projection and selection
  
  If $\text{Cond}$ contains only attributes $A_1, \ldots, A_n$, then:
  
  $$
  \pi_{\{A_1, \ldots, A_n\}} (\sigma_{\text{Cond}} (E)) \equiv \sigma_{\text{Cond}} (\pi_{\{A_1, \ldots, A_n\}} (E))
  $$

- Rule 5b. Exchange of projection and selection
  
  If $\text{Cond}$ contains only attributes $A_1 \ldots A_n$ and $B_1 \ldots B_m$ then:
  
  $$
  \pi_{\{A_1 \ldots A_n\}} (\sigma_{\text{Cond}} (E)) \equiv \pi_{\{A_1 \ldots A_n\}} (\sigma_{\text{Cond}} (\pi_{\{A_1 \ldots A_n, B_1 \ldots B_m\}} (E))
  $$
Joins and Projection/Selection

• Rule 6. Exchange of selection and join

  If $\text{Cond}$ contains only attributes of $E_1$, then:

  \[ \sigma_{\text{Cond}} ( E_1 \bowtie_{\text{Cond}_1} E_2 ) \equiv \sigma_{\text{Cond}} ( E_1 ) \bowtie_{\text{Cond}_1} E_2 \]

• Rule 7. Exchange of selection and union/difference

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

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

• Rule 8. Exchange of selection and natural join

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

• Rule 9. Exchange of projection and join:

\[
\text{If Cond contains only attributes } A_1 \ldots A_n, B_1 \ldots B_m \text{ and } A_1 \ldots A_n \text{ appear in } E_1, \text{ resp. } B_1 \ldots B_m \text{ in } E_2:\n\]
\[
\Pi \{ A_1, \ldots, A_n, B_1, \ldots, B_m \} ( E_1 \bowtie_{\text{Cond}} E_2 ) \equiv \\
\Pi \{ A_1, \ldots, A_n \} ( E_1 ) \bowtie_{\text{Cond}} \Pi \{ B_1, \ldots, B_m \} ( E_2 )
\]

• Rule 10. Exchange of projection and union:

\[
\text{If } A_1, \ldots, A_n \text{ are attributes appearing in } E_1 \text{ and } E_2, \text{ then:} \\
\Pi \{ A_1, \ldots, A_n \} ( E_1 \cup E_2 ) \equiv \\
\Pi \{ A_1, \ldots, A_n \} ( E_1 ) \cup \Pi \{ A_1, \ldots, A_n \} ( E_2 )
\]
Term Rewriting: Algebraic Optimization

• Usually there infinitely many rewrite steps
  – But not infinitely many rewritings
  – 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

• Option 1: Rule-based
  – Use heuristics for selecting order of rule application
  – Example: Push selections and projections down the tree
  – Very fast, already saves a lot
A Simple Rule-Based Optimizer

- Break complex selections into many simple selections
- Build simple projects from complex projections
- Push selects/proj ects as much down the tree as possible
- Replace selection and product with join
- Introduce add. projects as deep in the tree as possible
- Where ever possible, turn different sequential operations into one (conjunction of conditions, intersections of projections) to compute with one scan
  - Series of selections / projections right over a leaf
- Ignored here: Indexes, GROUP-BY, etc.
Limitations

- Problem: Join order
- Option 2: Cost-based rule selection
  - Estimate effect of operations
  - Find rewriting with the least intermediate result sizes
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

\[
\begin{align*}
\Pi & \quad \sigma \quad \times \\
\text{Name, Account#, Savings} & \quad \text{"Bond"} \leq \text{Name} \quad \text{Name} \leq \text{"Carter"} \quad \text{Address} = \text{"World"} \\
\times & \quad \text{Transaction} = \text{"Withdraw"} \quad \text{Amount} > \$1,000,000 \\
\times & \quad \text{C.Account#} = \text{A.Account#} \\
\times & \quad \text{C.Account#} = \text{J.Account#} \\
\times & \quad \text{Journal}
\end{align*}
\]
Breaking and Pushing Selections

\[ \Pi \sigma \times \sigma \sigma \times \sigma \]

- Name, Account#, Savings
- C.Account# = J.Account#
- C.Account# = A.Account#
- "Bond" \( \leq \) Name
  - Name \( \leq \) "Carter"
  - Address = "World"
- Transac = "Withdraw"
- Amount > 1000000

CUSTOMER
ACCOUNT
Journal
Introduce Joins

- CUSTOMER
  - Name, Account#, Savings
  - "Bond" ≤ Name
  - Name ≤ "Carter"
  - Address = "World"

- ACCOUNT
  - C.Account# = A.Account#

- Journal
  - Transac = "Withdraw"
  - Amount > 1000000

- ACCOUNT
  - C.Account# = J.Account#

- CUSTOMER
  - Π

- ACCOUNT
  - σ

- Journal
  - σ
Pushing Projections

CUSTOMER

ACCOUNT

Journal

Name, Account#, Address

Name, Account#, Savings
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
  ⋈
  movie
  σ
  year=99
  ⋈
  movie
  σ
  year=99
  ⋈
  movie
  σ
  year=99
  ⋈
  movie
  σ
  year=99

  SELECT m.title, a.name
  FROM movies99 m, actsin a
  WHERE m.title=a.title AND m.year=a.year
Another 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

π_{s.semester}

σ_{p.name = Sokrates and ⋯}

professor

vorlesung

student

hoeren
Break Up Selections

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

\[ \sigma_{p.\text{PersNr}=v.\text{gelesenVon}} \]
\[ \sigma_{v.\text{VorNr}=h.\text{VorNr}} \]
\[ \sigma_{s.\text{MatrNr}=h.\text{MatrNr}} \]
\[ \sigma_{p.\text{Name} = 'Sokrates'} \]
Push Selections

\[ \pi_{s,Semester} \]

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

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

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

\[ \sigma_{p,\text{Name} = \text{	extquoteleft}Sokrates	extquoteright} \]

\[ \pi_{s,Semester} \]
Rewrite Product+Selection into Joins

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

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

\[\pi_{h.\text{MatrNr}}\]
Order of Joins: Indistinguishable

\[
\pi_s.Semester \quad \sigma_{p.Name = 'Sokrates'} \quad \pi_s.Semester
\]

\[
s \ni \quad \sigma_{p.Name = 'Sokrates'} \quad p \ni \quad \pi_s.Semester
\]

\[
\times \quad v.VorlNr=h.VorlNr \quad \times \quad s.MatrNr=h.MatrNr \ni \quad v.VorlNr=h.VorlNr
\]

\[
\times \quad p.PersNr=v.gelesenVon \quad \times \quad s.MatrNr=h.MatrNr \ni \quad p.PersNr=v.gelesenVon
\]
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 (4000 “hören” tuples)

• Note: Pipelining makes consequences less severe
  - No additional IO, but still additional computation
Join Order – Does it Matter?

- Compute $\sigma_{\text{Sokrates}}(P) \Join (V \Join (S \Join H)))$
  - 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: $4000 + 4000 + 1 + 200$

- Compute $S \Join (H \Join (\sigma_{\text{Sokrates}}(P) \Join V))$
  - Inner join selects $4$ tuples
  - Next join generates $50 \times 4 = 200$ tuples
  - Last join: No change
  - (Intermediate) result sizes: $1 + 4 + 200 + 200$
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
- From an implementation perspective, execution times of different orders can differ tremendously
- Holds only for more than 2 relations
  - E.g. $R \bowtie (S \bowtie T) \equiv (S \bowtie R) \bowtie T$
- Given $n$ joins, there are $n!$ possible orders
  - Depending on join predicates, many orders may involve cross-products
Left/Right-deep versus Bushy Trees

- There is one left-deep tree topology, but still $n!$ orders.
- There are many more than $n!$ binary trees with $n$ leaves, and for each $n!$ possible orders (some symmetric cases).

Left-deep join tree

Bushy join tree
Choosing a Join Order

- Typical heuristic for pruning the search space
  - Consider only left-deep trees (can be pipelined efficiently)
  - Usually generates among the best plans
- But there are still $n!$ possible orders
- Find best using dynamic programming
  - Generate plans bottom up: Plans for pairs, triples, ...
  - For each concrete join group, keep only best plan
  - Use these to enumerate possibilities for larger join groups
  - Still very expensive problem
- Use additional heuristics
  - Prune plans containing a cross product
  - Prune plans much worse than current best plan
  - ...
Join Groups

- There are \((\binom{n}{i})\) join groups with \(i\) elements
  - That’s enough to make the problem costly
Details

- Create a table containing for each join group
  - 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 = size of join group
  - i=1: Consider every relation in isolation
    - Size = Size of relation
    - Cost = 0 (assumption here – no access)
  - i=2: Consider each relation pair
    - Size: Estimated size of “joining” both relations (might be product)
    - Cost = 0 (no intermediate result so far due to previous assumption)
    - Fix an order (e.g.: smaller relation as inner relation)
      - This order will never change again
  - i=3: Consider each pair in triple and join with third relation
    - Consider only chosen order for pairs involved
      - ...

Example 1

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

```
Example 1

<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>R × T</td>
<td>R × U</td>
<td>S × T</td>
<td>S × U</td>
<td>T × 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 × T) × R</td>
<td>(R × S) × U</td>
<td>(T × U) × R</td>
<td>(T × U) × 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 ∙ T) ∙ R</td>
<td>(R ∙ S) ∙ U</td>
<td>(T ∙ U) ∙ R</td>
<td>(T ∙ U) ∙ S</td>
</tr>
</tbody>
</table>

Plan 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 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: }
14: \text{return} \( \text{optPlan}(\{R_1, \ldots, R_n\}) \)
Dynamic Programming

• DP here is a heuristic
  – Assumption: Any subplan of an optimal plan is optimal
  – True for computing shortest paths, edit distance, knapsack, …
• 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
• Solution
  – 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 an optimal solution is a common computer science problem
  - Exhaustive search
    • Guarantees optimal result, but often too expensive
    • DP query optimizer: optimal for left-deep join order without sorting
  - Heuristic method
    • Greedy/Hill-Climbing: only use one alternative for further search
  - Branch-and-Bound
    • Search i levels exhaustively, choose k alternatives 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
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
- Nur 3! beinhalten kein Kreuzprodukt
Heuristiken

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

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

<table>
<thead>
<tr>
<th>Zwischenergebnis</th>
<th>1. Join (M / 15)</th>
<th>6.666.666</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2. Join (</td>
<td>J_1</td>
</tr>
<tr>
<td></td>
<td>3. Join (</td>
<td>J_2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Zwischenergebnis</th>
<th>1. Join (M / 50)</th>
<th>2.000.000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2. Join (</td>
<td>J_1</td>
</tr>
<tr>
<td></td>
<td>3. Join (</td>
<td>J_2</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 (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>

- 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) **Bitmapindexte** lässt kartesisches Produkt weniger vorteilhaft erscheinen
- Phasen
  1. Berechnung aller FKs in Faktentabelle gemäß Dimensionsbedingungen einzeln für jede Dimension
  2. Anlegen/laden von Join-Bitmapindexten auf allen FK Attributen der Faktentabelle
  3. Merge (AND) aller Bitmapindexte
  4. Direkter Zugriff auf Faktentabelle über TID
  5. Join nur der selektierten Fakten mit Dimensionstabellen zum Zugriff auf Dimensionswerte
- Zwischenergebnisse sind nur (komprimierte) Bitlisten
**Gesamtplan**

<table>
<thead>
<tr>
<th>Phase 1</th>
<th>Phase 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SELECT STATEMENT</strong></td>
<td><strong>SELECT STATEMENT</strong></td>
</tr>
<tr>
<td><strong>SORT GROUP BY</strong></td>
<td><strong>SORT GROUP BY</strong></td>
</tr>
<tr>
<td><strong>HASH JOIN</strong></td>
<td><strong>HASH JOIN</strong></td>
</tr>
<tr>
<td><strong>TABLE ACCESS FULL</strong></td>
<td><strong>TABLE ACCESS FULL</strong></td>
</tr>
<tr>
<td><strong>HASH JOIN</strong></td>
<td><strong>HASH JOIN</strong></td>
</tr>
<tr>
<td><strong>TABLE ACCESS FULL</strong></td>
<td><strong>TABLE ACCESS FULL</strong></td>
</tr>
<tr>
<td><strong>PARTITION RANGE ALL</strong></td>
<td><strong>PARTITION RANGE ALL</strong></td>
</tr>
<tr>
<td><strong>TABLE ACCESS BY LOCAL INDEX ROWID</strong></td>
<td><strong>TABLE ACCESS BY LOCAL INDEX ROWID</strong></td>
</tr>
<tr>
<td><strong>BITMAP CONVERSION TO ROWIDS</strong></td>
<td><strong>BITMAP CONVERSION TO ROWIDS</strong></td>
</tr>
<tr>
<td><strong>BITMAP AND</strong></td>
<td><strong>BITMAP AND</strong></td>
</tr>
<tr>
<td><strong>BITMAP INDEX SINGLE VALUE</strong></td>
<td><strong>BITMAP INDEX SINGLE VALUE</strong></td>
</tr>
<tr>
<td><strong>BITMAP INDEX RANGE SCAN</strong></td>
<td><strong>BITMAP INDEX RANGE SCAN</strong></td>
</tr>
<tr>
<td><strong>BITMAP INDEX RANGE SCAN</strong></td>
<td><strong>BITMAP INDEX RANGE SCAN</strong></td>
</tr>
<tr>
<td><strong>BITMAP MERGE</strong></td>
<td><strong>BITMAP MERGE</strong></td>
</tr>
<tr>
<td><strong>BITMAP KEY ITERATION</strong></td>
<td><strong>BITMAP KEY ITERATION</strong></td>
</tr>
<tr>
<td><strong>BUFFER SORT</strong></td>
<td><strong>BUFFER SORT</strong></td>
</tr>
<tr>
<td><strong>TABLE ACCESS FULL</strong></td>
<td><strong>TABLE ACCESS FULL</strong></td>
</tr>
<tr>
<td><strong>BUFFER SORT</strong></td>
<td><strong>BUFFER SORT</strong></td>
</tr>
<tr>
<td><strong>TABLE ACCESS FULL</strong></td>
<td><strong>TABLE ACCESS FULL</strong></td>
</tr>
<tr>
<td><strong>BITMAP MERGE</strong></td>
<td><strong>BITMAP MERGE</strong></td>
</tr>
<tr>
<td><strong>BITMAP KEY ITERATION</strong></td>
<td><strong>BITMAP KEY ITERATION</strong></td>
</tr>
<tr>
<td><strong>BUFFER SORT</strong></td>
<td><strong>BUFFER SORT</strong></td>
</tr>
<tr>
<td><strong>TABLE ACCESS FULL</strong></td>
<td><strong>TABLE ACCESS FULL</strong></td>
</tr>
<tr>
<td><strong>BITMAP MERGE</strong></td>
<td><strong>BITMAP MERGE</strong></td>
</tr>
<tr>
<td><strong>BITMAP KEY ITERATION</strong></td>
<td><strong>BITMAP KEY ITERATION</strong></td>
</tr>
<tr>
<td><strong>BUFFER SORT</strong></td>
<td><strong>BUFFER SORT</strong></td>
</tr>
<tr>
<td><strong>TABLE ACCESS FULL</strong></td>
<td><strong>TABLE ACCESS FULL</strong></td>
</tr>
<tr>
<td><strong>BITMAP MERGE</strong></td>
<td><strong>BITMAP MERGE</strong></td>
</tr>
<tr>
<td><strong>BITMAP KEY ITERATION</strong></td>
<td><strong>BITMAP KEY ITERATION</strong></td>
</tr>
<tr>
<td><strong>BUFFER SORT</strong></td>
<td><strong>BUFFER SORT</strong></td>
</tr>
<tr>
<td><strong>TABLE ACCESS FULL</strong></td>
<td><strong>TABLE ACCESS FULL</strong></td>
</tr>
</tbody>
</table>

**Location**
- Time

**Time**
- Product

**Product**
- Sales

**Sales**
- Sales_L_BJIX

**Sales_L_BJIX**
- Product

**Product**
- Sales_P_BIX

**Sales_P_BIX**
- Time

**Time**
- Sales_TIME_BIX

**Sales_TIME_BIX**