Datenbanksysteme II: Implementation of Database Systems

Query Optimization

Material von
Prof. Johann Christoph Freytag
Prof. Kai-Uwe Sattler
Prof. Alfons Kemper, Dr. Eickler
Prof. Hector Garcia-Molina
• Oracle Schulung, 25-29.7.2005
  - Cluster und Ausfallsicherheit, Analytische & statistische Funktionen, Warehouse Builder, Oracle OLAP, ...
  - Anmeldung bis 1.7.2005
    • Bei mispel@informatik.hu-berlin.de
  - Verlosung, verbindliche Anmeldung, Nachrückliste
Sort-Merge Join

• How does it work?
  - Sort both relations on join attribute(s)
  - Merge both sorted relations

• Caution if duplicates exist
  - The result size still is $|R| \times |S|$ in worst case
  - If there are $r/s$ tuples with value $x$ in the join attribute in R / S, we need to output $r \times s$ tuples for $x$
  - So what is the worst case??

• Example

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>A1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>A2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>A4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>A3</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>1</td>
<td>C1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>C3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>C5</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>C2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>C4</td>
</tr>
</tbody>
</table>
Better than Blocked-Nested-Loop?

- Assume $b(R)=10,000$, $b(S)=2,000$, $m=500$
  - BNL costs 42,000
    - With $S$ as outer relation
  - SM: $10,000 + 2,000 + 4 \times 10,000 + 4 \times 2,000 = 60,000$
  - Improved SM: 36,000

- Assume $b(R)=1,000,000$, $b(S)=1,000$, $m=500$
  - BNL costs $1000 + 1,000,000 \times 1000/500 = 2,001,000$
  - SM: $1,000,000 + 1,000 + 6 \times 1,000,000 + 4 \times 1,000 = 7,005,000$
  - Improved SM: 5,003,000

- When is SM better than BNL??
  - Consider improved version with
    - $2 \times b(R) \times \text{ceil} \left( \log_m(b(R)) \right) + 2 \times b(S) \times \text{ceil} \left( \log_m(b(S)) \right) - b(R) - b(S) \sim$
    - $2 \times b(R) \times \log_m(b) + 2 \times b(S) \times \log_m(S) - b(R) - b(S) \sim$
    - $b(R) \times (2 \times \log_m(b) - 1) + b(S) \times (2 \times \log_m(S) - 1)$
  - In most cases, this means $3 \times (b(S) + b(R))$
Comparison 3

- $b(R)=1,000,000$, $b(S)=50,000$, $m$ between 500 and 90,000

- BNL very sensible to small memory sizes
Hash Join Cost

- **Hash phase** costs $2 \times b(R) + 2 \times b(S)$
- **Merge phase** costs $b(R) + b(S)$
- **Total**: $3 \times (b(R) + b(S))$
  - Under what assumption??
Hash Join with Large Tables

- Merge phase assumes that 2 entire buckets can be held in memory
  - Thus, we roughly assume that $2 \times b(R)/m < m$ (if $b(R) \approx b(S)$)
  - Note: Merge phase of sorting only requires 2 blocks (or more for more runs), hashing requires 2 buckets to be loaded
- What if $b(R) > m^2/2$ ??
  - We need to create smaller buckets
  - Partition $R/S$ such that each partition hopefully has buckets smaller than $m^2/2$
  - Compute buckets for all partitions in both relations
  - Merge in cross-product manner
    - $P_{R,1}$ with $P_{S,1}$, $P_{S,2}$, ..., $P_{S,n}$
    - $P_{R,2}$ with $P_{S,1}$, $P_{S,2}$, ..., $P_{S,n}$
    - ...
    - $P_{R,m}$ with $P_{S,1}$, $P_{S,2}$, ..., $P_{S,n}$
- Actually, it suffices if either $b(R)$ or $b(S)$ is small enough
  - Chose the smaller relation as driver (outer relation)
  - Load one bucket into main memory
  - Load same bucket in other relation block by block and filter tuples
Cost

• Assume \( b(R) = b(S) = b \)
• How many partitions (p) do we need?
  – Goal: For each partition \( P \), \( b(P) < m^2/2 \)
  – Hence: \( b/p \approx m^2/2 \), or \( p \approx 2b/m^2 \)
• In each partition, there are (still) \( m \) buckets of size \( \sim m/2 \)
• Hash/partition phase: \( 2b + 2b \)
• Merge phase: \( b + p \cdot m \cdot p \cdot m/2 = b + p^2 \cdot m^2/2 = b + 2b^2/m^2 \)
  – There are \( p \cdot m \) buckets in outer relation
  – For each bucket of outer relation, we have to read \( p \) buckets of inner relation, each of size \( m/2 \)
Alternative

- Accept overly large buckets
- Perform blocked-nested loop for each pair of buckets
- There are \( m \) buckets, each of size \( n = b/m (\geq m/2) \)
- Hash/partition phase: \( 2b + 2b \)
- BNL phase: \( m \times (n + n^2/n/m) = m \times (b/m + b^2/m^3) = b + b^2/m^2 \)
  - There are \( m \) bucket pairs
  - For each, we perform blocked nested loop over two buckets of size \( n \)
- Note: Since in fact only one relation must be small enough, the cross-product large hash join has app. the same cost
Comparing Hash Join and Sort-Merge Join

• If enough memory provided, both require approximately the same number of IO
  – $3 \times (b(R) + b(S))$
  – Hybrid-hash join improves slightly
• SM generates sorted results – sort phase of other joins in query plan can be dropped
  – Advantage propagates up the tree
• HJ does not need to perform $O(n\log(n))$ sorting in main memory
• HJ requires that only one relation is “small enough”, SM needs two small relations
  – Because both are sorted independently
• HJ depends on roughly equally sized buckets
  – Otherwise, performance might degrade due to unexpected paging
  – To prevent, estimate $k$ more conservative and do not fill $m$ completely
  – Some memory remains unused
• Both can be tuned to generate mostly sequential IO
Index Join

• Assume we have an index “B_Index” on one join attribute
• Choose indexed relation as inner relation
• Index join

\[
\text{FOR EACH } r \text{ IN } R \text{ DO } \quad X = \{ \text{SEARCH} (S.B\_Index, <r.B>) \} \\
\text{FOR EACH TID } i \text{ in } X \text{ DO } \quad s = \text{READ} (S, i) \text{ ; output } (r \bowtie s).
\]

• Actually, this is a one block-nested loop with index access
  - Using BNL possible (and of course better)
Index Join Cost

- **Assumptions**
  - R.B is foreign key referencing S.B
  - Every tuple from R has one or more join tuples in S
- **Let** \( v(X,B) \) **be the number of different values of attribute B in relation X**
  - Each value in S.B appears \( v \approx b(S)/v(S,B) \) times
- **For each** \( r \in R \), **we read all tuples with given value in S**
- **Total cost**: \( b(R) + |R| \times (\log_k(|S|) + v/k + v) \)
  - Outer relation read once
  - Find value in B*, read all matching TIDs (with block size k), access S for each TID

- **Other way round**: Assume that S.B is foreign key for R.B
  - Some tuples of R will have no join partner in S
  - Assume only \( r \in R \) tuples have partner
- **Total cost**: \( b(R) + r \times (\log_k(|S|) + v/k + v) \)
  - No real change
Semi Join

• Consider queries such as
  - \(\text{SELECT DISTINCT } R.* \text{ FROM } S,R \text{ WHERE } R.B=S.B\)
  - \(\text{SELECT } R.* \text{ FROM } R \text{ WHERE } R.B \text{ IN (SELECT } S.B \text{ FROM } S)\)
  - \(\text{SELECT } R.* \text{ FROM } R \text{ WHERE } R.B \text{ IN ( ...)}\)

• What’s special?
  - No values from \(S\) are requested in result
  - \(S\) (or inner query) acts as filter on \(R\)

• Semi-Join \(R \bowtie S\)
  - Very important for distributed databases
    - Accessing data becomes even more expensive
    - Idea: First ship only join attribute values, compute Semi-Join, then retrieve rest of matching tuples
    - Technique also can be used for very large tuples and small result sizes
      - First project on attribute values
      - Intersect lists, probe into tables and load data
      - Question: How do we know the sizes of intermediate result?
Implementing Semi-Join

- **Using blocked-nested-loop join**
  - Chose filtered relation as outer relation
  - Perform BNL
  - Whenever partner for R.B is found, exit inner loop

- **Using sort-merge join**
  - Sort R
  - Sort join attribute values from S, removing duplicates on the way
  - Perform merge phase as usual
  - Very effective if v(S,B) is small
5 Schichten Architektur

Wir sind hier:

- Mengenorientierter Zugriff
- Satzorientierter Zugriff
- Interne Satzschnittstelle
- Systempufferschnittstelle
- Dateischnittstelle
- Geräteschnittstelle

Datenmodellebene

Logischer Zugriff

Speicherstrukturen

Pufferverwaltung

Betriebssystem

Anfrageübersetzung, Zugriffspfadwahl, Zugriffskontrolle, Integritätskontrolle
Sortierung, Transaktionsverwaltung, Cursorverwaltung
Record Manager, Index Manager, Sperrverwaltung, Log / Recovery
Speichermanagement, Puffermanagement, Caching-Strategien
Externspeicherverwaltung
Content of this Lecture

• Steps in Query Optimization
• Algebraic Term Rewriting
  – A simple, heuristic, rule-based optimizer
• Optimizing Join Order
• Plan Enumeration
• Star-join - a counter-example
Is Optimization Worth It?

• Goal: Find cheapest way to compute query result
  – Generate and judge different physical plans to answer the query
  – All query plans are semantically equal
    • I.e., compute the same set of tuples

• Optimization costs time
  – Some steps are exponential
    • For instance, join order is exponential in the number of joins
    • 10 joins – potentially $3^{10}$ steps
  – Finding the best plan might take more time than executing an arbitrary plan
    • Usually, optimizers do not find the best plan
    • Heavy use of heuristics to prune search space

• Why bother?
Example

- **Consider query**
  - SELECT C.name, C.address
    FROM customer C, order O
    WHERE C.name = O.c_name AND
    O.good = "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
  - Minimal main memory
First Attempt

- Translate in relational algebra term
  - \( \pi_{\text{name}, \text{adr}} (\sigma_{O.\text{C_name}=C.\text{name} \land O.\text{good}=\texttt{coffee}} (C \times O)) \)

- Interpret query „from right to left“
  - Be as stupid as possible
  - Full materialization of intermediate results (no buffering, no pipelining)

- Compute cross-product
  - Reads: \( b(C) \times b(O) = 20.000 \)
  - Writes: \( 100 \times 10.000 / 3 \approx 333.000 \)

- Compute selection
  - Reads: 333.000
  - Writes: \( 50 / 3 \approx 17 \)

- Compute projection
  - Reads: 17
  - Writes: \( 17 / 50 \approx 1 \)

- Altogether: \( \approx 686.000 \) IO (and 330.000 blocks required on disk)
Use Term Rewriting

- Algebraic term can be rewritten
  \[ \pi_{\text{name}, \text{adr}} (C \bowtie \sigma_{O.\text{good}=\text{coffee}} (O)) \]

- Compute selection on O
  - Reads: 1,000
  - Writes: \( \frac{50}{10} = 5 \)

- Compute join using nested loop
  - Reads: \( 5 + b(C) \times 5 = 105 \)
  - Writes: \( \frac{50}{3} \approx 17 \)

- Compute projection
  - Reads: 17
  - Writes: \( \frac{17}{50} \approx 1 \)

- Altogether: \textbf{1.145} (requiring 17 blocks on disk)

- Maybe there is an ever better term??
  - Generally, there are quite many – which is the best?
Better Term Rewriting

- Push projection
  \[ \pi_{\text{name,adr}}(\pi_{\text{name,adr}}(C) \bowtie_{O.C_{\text{name}}=C_{\text{name}}} \sigma_{O.\text{good}=\text{coffee}}(O)) ) \]

- 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: \( 2 + 2 \times 5 = 12 \)
  - Writes: \( 50/3 \approx 17 \)
    - Actually, join tuples are smaller, more than 3 should fit on a block

- Compute projection
  - Reads: 17
  - Writes: \( 17/50 \approx 1 \)

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

- Assume indexes on O.good and C.name
- Compute selection on O using index
  - Reads: roughly between 5 and 50
    - Height of index plus consecutive blocks for 50 TIDs with good=‘coffee’
    - Assume 10 pointer in an index node: height = 4
  - Writes: 50/10 = 5
- Sort intermediate result
  - Read and writes: \( \sim 5 \cdot \log(5) \sim 15 \)
    - Very conservative estimation
  - Result has 5 blocks
- Compute join
  - Reads: 20 + 5 = 25
    - Using sort-merge – read C.name in sorted order using index
  - Writes: 50/3 \( \sim 17 \)
- Compute projection
  - Reads: 17
  - Writes: 17/50 \( \sim 1 \)
- Altogether: between 85 and 130 (requiring 17 blocks on disk)
  - Even better??
Comparison

<table>
<thead>
<tr>
<th>Variante der Ausführung</th>
<th>Lese- und Schreibzugriffe</th>
<th>Seiten für Zwischenergebnisse</th>
</tr>
</thead>
<tbody>
<tr>
<td>direkte Auswertung</td>
<td>ca. 687.000</td>
<td>ca. 333.000</td>
</tr>
<tr>
<td>optimierte Auswertung</td>
<td>ca. 1.140</td>
<td>17</td>
</tr>
<tr>
<td>Auswertung mit Index</td>
<td>min. 85</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>max. 130</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>85 bis 130</td>
<td>5 (plus sortieren)</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, subquery rewriting
- Query minimization
  - Remove implied predicates and redundant operations
- Term / tree generation
  - Note: many SQL operations have no “canonical” algebra equivalent
- Plan enumeration and pruning
  - Term rewriting (logic optimization)
  - Cost estimation (cost-based optimization)
  - Plan instantiation (physical optimization)
  - Pruning
- Selection of best plan
- Code generation (compilation or interpretation)
- Note: Some steps are interleaved
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 possible, not always advantageous

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

• Rewriting of EXISTS, ALL, (MINUS, INTERSECTION, …)
  - See literature
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
  - ...

The diagram represents a database schema with entities such as Order, Delivery, Product, and Customer, along with their attributes.
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
  - Subquery gets part of surrounding query
  - Allows “more 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
  - Advantageous if subquery appears more than once

- Rewriting not possible
Correlated Subquery without Aggregation

- SELECT o.o_id
  FROM order o
  WHERE o.o_id IN (SELECT d.o_id
                   FROM delivery d
                   WHERE d.o_id = o.o_id AND
                   d.date-o.date<5)
- 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)
- SELECT DISTINCT o.o_id
  FROM order o, delivery d
  WHERE o.o_id = d.o_id AND
  d.date-o.date<10
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)

- Again: 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)

- **Rewrite into two queries**
  - **First query q1**
    SELECT o_id, sum(price*quantity) 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 rewritten into join

- **Improvement**
  - Inner query is computed only once
  - q1 will use (efficient) full table scan instead of multiple queries with conditions
Query Minimization

- 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, and use view as “filter”
    - SELECT C.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
  - Get goods from very good businesses, and “save ink”
    - SELECT O.good
      FROM good_business G,order O
      WHERE G.o_id = O.o_id
    - SELECT o2.goods
      FROM customer C, order O1, order O2
      WHERE C.name=O1.name AND
      O1.o_id=O2.o_id

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

- Steps in Query Optimization
- Algebraic Term Rewriting
  - A simple, heuristic, rule-based optimizer
- Optimizing Join Order
- Plan Enumeration
- Star-join - a counter-example
Term Rewriting: Algebraic Optimization

• Definition
  - Let $E_1$ and $E_2$ be relational algebra expressions. $E_1$ and $E_2$ are equivalent iff
    - $E_1$ and $E_2$ contain the same relations $R_1 \ldots R_n$
    - For any instances of $R_1 \ldots R_n$, $E_1$ and $E_2$ compute the same result

• We will see some rules
  - There exist many more: see literature

• So many rules – which should we use for optimization?
  - General heuristic: Minimize intermediate results
    - Less IO if materialization is necessary
    - Less input for operations that are higher in the plan

• There are infinitely many rewrite steps
  - But not infinitely many rewritings

• Use simple heuristics for rule selection and application
  - Break operations and push selections and projections down the tree
  - Very fast, already saves a lot, but worse than cost-based optimization
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
  \[
  \sigma_{Cond1} (\sigma_{Cond2} (E)) \equiv \sigma_{Cond2} (\sigma_{Cond1} (E))
  \equiv \sigma_{Cond1 \text{ and } Cond2} (E)
  \]

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

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

• Rule 6. Exchange of projection and join operator

  If \( Cond \) contains only attributes of \( E_1 \), then:

  \[
  \sigma_{Cond} ( E_1 \bowtie_{Cond_1} E_2 ) \equiv \sigma_{Cond} ( E_1 ) \bowtie_{Cond_1} E_2
  \]

• Rule 7. Exchange of selection and union/difference

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

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

• Rule 8. Exchange of selection and natural join

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

- Rule 9. Exchange of projection and join:

  If \( \text{Cond} \) 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 \( E_2 \):

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

- Rule 10. Exchange of projection and union:

  If \( A_1, \ldots, A_n \) are attributes appearing in \( E_1 \) and \( E_2 \), then:

  \[
  \Pi_{\{ A_1, \ldots, A_n \}} ( E_1 \cup E_2 ) \equiv \\
  \Pi_{\{ A_1, \ldots, A_n \}} ( E_1 ) \cup \Pi_{\{ A_1, \ldots, A_n \}} ( E_2 )
  \]
A Simple Rule-Based Optimizer

- Use the following rules of thumb
  - Break complex selections into many simple selections
  - Break complex projections into many simple projections
  - Push selections and projections as much **down the tree** as possible
  - Replace selection and product with join
    - I.e.: move selections right over products and replace
  - Introduce additional projections as deep in the tree as possible
  - If possible, turn **diff. operations into one** to compute with one scan
    - Different selections or projections right over a leaf
    - Careful – some conditions could be evaluated using an index (which?)

- **Waste of potential**
  - Join order, cost and size estimations

- **Ignored:** Sorting, GROUP BY, DISTINCT
Example

- Query on 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#
```
\[
\sigma \left( \pi \text{Name, Account\#} \right) \land \left( \text{Address} = \text{“World”} \land \text{Transaction} = \text{“Withdraw”} \land \text{Amount} > 1,000,000 \land \text{C.Account\#} = \text{A.Account\#} \land \text{C.Account\#} = \text{J.Account\#} \right)
\]
Breaking and Pushing Selections

π (Name, Account#, Savings) 
σ ("C.Account# = J.Account#) 
× 
σ ("C.Account# = A.Account#) 
× 
σ ("Bond" ≤ Name ≤ "Carter" and Address = "World") 
× 
σ ("Transaction = "Withdraw" and Amount > $1,000,000")
"C.Account# = A.Account#"

"Bond" ≤ Name ≤ "Carter" and Address = "World"

Name, Account#, Savings

Π

C.Account# = J.Account#

"Transaction = "Withdraw" and Amount > 1,000,000"

σ

ACCOUNT

JOURNAL

σ

CUSTOMER

Introduce Joins
Caution

- We advised to always push down selections
- Sometimes, opposite is good
  - Especially for conditions on join attributes, with views involved
- 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`
select s.Semester
from Studenten s, hören h
    Vorlesungen v, Professoren p
where p.Name = "Sokrates" and
    v.gelesenVon = p.PersNr and
    v.VorlNr = h.VorlNr and
    h.MatrNr = s.MatrNr

Another Example
Break Up Selections

\[ \pi_s.\text{Semester} \]
\[ \sigma_p.\text{Name} = 'Sokrates' \text{ and } \cdots \]

\[ \sigma_v.\text{VorlNr}=h.\text{VorlNr} \]
\[ \sigma_s.\text{MatrNr}=h.\text{MatrNr} \]
\[ \sigma_p.\text{Name} = 'Sokrates' \]
Push Selections

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

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

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

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

\[ \sigma_{p.\text{Name} = 'Sokrates'} \]
Order of Joins: Indistinguishable Without Size Estimates

\[ \pi_{s.Semester} \]

\[ A_{p.PersNr=v.gelesenVon} \]

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

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

\[ A_{s.MatrNr=h.MatrNr} \]

\[ \pi_{s.Semester} \]

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

\[ A_{p.PersNr=v.gelesenVon} \]

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

\[ h \]

\[ s \]

\[ v \]

\[ p \]
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)

- **Compute** $\sigma_{Sokrates}(P) \bowtie (V \bowtie (S \bowtie H))$
  - Inner join: 1,000*4 = 4,000 tuples
  - Next join: Again 4,000 tuples
  - Last join selects only 1/20 of intermediate results = 200
  - (Intermediate) result sizes: 4,000 + 4,000 + 200

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

- **Note:** Pipelining makes consequences less severe
  - No additional IO, but still additional computation
Content of this Lecture

• Steps in Query Optimization
• Algebraic Term Rewriting
  – A simple, heuristic, rule-based optimizer
• Optimizing Join Order
• Plan Enumeration
• Star-join - a counter-example
Optimizing Join Order

- From the relation algebra perspective, join is associative and commutative: \( R \bowtie S \equiv S \bowtie R \)
- From an implementation point of view, they are not at all the same - *Execution time can differ tremendously*
- For >2 relations, e.g. \( R \bowtie (S \bowtie T) \equiv (S \bowtie R) \bowtie T \)
  - Result is the same
  - But *sizes of intermediate relations are different*
  - Large intermediate results are costly
    - Require main memory, get swapped to disk, more IO, ...
- Given \( n \) joins, there are \( n! \) possible orders
  - Usually, many require computation of cross-product
  - Cross-products are usually avoided where possible
    - Exception: Star-Join
Left/Right-deep versus Bushy Trees

- Topologies of a join tree
  - There is one left-deep tree topology, but still $n!$ orders
  - There are $>n!$ binary trees with $n$ leaves, and for each $n!$ possible orders
- Obviously, we cannot enumerate them all
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
• Hence: Topology fixed, but 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 $n$ over $i$ 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 over plan length = size of join group
  – i=1: Consider every relation in isolation
    • Size = Size of relation
    • Cost = 0 (assumption here)
  – i=2: Consider each relation pair
    • Size: Estimated size of “joining” both relations (might be product)
    • Cost = 0 (no int. res. 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

<table>
<thead>
<tr>
<th></th>
<th>{R}</th>
<th>{S}</th>
<th>{T}</th>
<th>{U}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
<td>0</td>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1000</td>
<td>0</td>
<td>1000</td>
</tr>
</tbody>
</table>

```
{R} {S} {T} {U}
1000 1000 1000 1000
0000
```

```
scan(R) scan(S) scan(T) scan(U)
```
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>

Prune products

Better than
S ⋈ (T ⋈ R) and (R ⋈ S) ⋈ T
Example 3

<table>
<thead>
<tr>
<th>opt. Plan</th>
<th>{(S \bowtie T) \bowtie R}</th>
<th>{(R \bowtie S) \bowtie U}</th>
<th>{(T \bowtie U) \bowtie R}</th>
<th>{(T \bowtie U) \bowtie S}</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>Plan</td>
<td>((S \bowtie T) \bowtie R) \bowtie U</td>
<td>((R \bowtie S) \bowtie U) \bowtie T</td>
<td>((T \bowtie U) \bowtie R) \bowtie S</td>
<td>((T \bowtie U) \bowtie S) \bowtie R</td>
</tr>
<tr>
<td>Kosten</td>
<td>12k</td>
<td>55k</td>
<td>11k</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: \quad optPlan(\{R_i\}) = accessPlans(R_i)
3: \quad prunePlans(optPlan(\{R_i\}))
4: 
5: for $i = 2$ to $n$ do 
6: \quad for all $S \subseteq \{R_1, \ldots, R_n\}$ such that $|S| = i$ do 
7: \quad \quad optPlan(S) = \emptyset
8: \quad \quad for all $O$ such that $|S| - |O| = 1$
9: \quad \quad \quad optPlan(S) = optPlan(S) \cup joinPlans(optPlan(O), O)
10: \quad \quad prunePlans(optPlan(S))
11: \quad \}
12: 
13: 
14: return optPlan(\{R_1, \ldots, R_n\})
Dynamic Programming

• DP 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
  - Recall sort-merge join
  - 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

• Steps in Query Optimization
• Algebraic Term Rewriting
  – A simple, heuristic, rule-based optimizer
• Optimizing Join Order
• Plan Enumeration
• Star-join - 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 all/some plans)
  +
  Cost functions for evaluating and pruning plans (still missing)
Search Strategies

• Searching a huge search space for optimal solution is a common computer science problem
  – AI: Planning = searching

• Strategies
  – 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, then choose k alternatives for further search
  – Simulated annealing
    • Generate a good plan
    • Improve iteratively, where “scope” of considered improvements shrink with time
  – Genetic optimization
    • Generate some good plans
    • Build combinations
• Typische Anfrage gegen Star Schema
  – Aggregation und Gruppierung
  – Bedingungen auf den Werten der Dimensionstabellen
  – Joins zwischen Dimensions- und Faktentabelle
Beispielquery


```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
```
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

```
σ
region_name='Berlin'
σ
σ
pg_name='Wasser'
σ
month=1

Sales

Product

Location

Time```

Heuristiken

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

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

Annahmen
- \( M = \mid S \mid = 100.000.000 \)
- 20 Verkaufstage pro Monat
- Daten von 10 Jahren
- 50 Produktgruppen a 20 Produkten
- 15 Regionen a 100 Shops
- Gleichverteilung aller Verkäufe

Größte 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>1. Join</th>
<th>Zweischenergebnis</th>
</tr>
</thead>
<tbody>
<tr>
<td>(M / 15)</td>
<td>6.666.666</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2. Join</th>
<th>Zweischenergebnis</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>J_1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3. Join</th>
<th>Zweischenergebnis</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>J_2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1. Join</th>
<th>Zweischenergebnis</th>
</tr>
</thead>
<tbody>
<tr>
<td>(M / 50)</td>
<td>2.000.000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2. Join</th>
<th>Zweischenergebnis</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>J_1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3. Join</th>
<th>Zweischenergebnis</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</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 jedes Produkt in jedem 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

• Weiterer Vorteil des kartesischen Produkts
  - „Berichtsform“: Auch leere Würfelzellen sollen ins Ergebnis
  - Werden durch das kartesische Produkt alle gebildet
  - Äquivalent zu Outer-Joins

• 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

• Neue STAR Join Strategie seit Oracle 8i
• 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
Gesamtplan

Phase 1

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

Phase 2

SELECT STATEMENT
SORT GROUP BY
  HASH JOIN
  TABLE ACCESS FULL
  HASH JOIN
  TABLE ACCESS FULL
  HASH JOIN
  TABLE ACCESS FULL
  PRODUCT
  PARTITION RANGE ALL
  TABLE ACCESS BY LOCAL INDEX ROWID
  SALES
  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