Advanced Query Optimization

Andreas Meister
Otto-von-Guericke University Magdeburg
Summer Term 2018
Why do we need query optimization?
SQL - TPC-H - Query 5

```
SELECT  N_NAME,  SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT))
FROM    CUSTOMER, ORDERS, LINEITEM,
        SUPPLIER, NATION, REGION
WHERE   C_CUSTKEY = O_CUSTKEY AND L_ORDERKEY = O_ORDERKEY
        AND L_SUPPKEY = S_SUPPKEY
        AND C_NATIONKEY = S_NATIONKEY
        AND S_NATIONKEY = N_NATIONKEY
        AND N_REGIONKEY = R_REGIONKEY
        AND R_NAME = 'ASIA'
        AND O_ORDERDATE >= '1994-01-01'
        AND O_ORDERDATE < '1995-01-01'
GROUP BY N_NAME
ORDER BY REVENUE DESC
```
SQL - TPC-H - Query 5

```sql
SELECT N_NAME, SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT))
FROM CUSTOMER, ORDERS, LINEITEM, SUPPLIER, NATION, REGION
WHERE C_CUSTKEY = O_CUSTKEY AND L_ORDERKEY = O_ORDERKEY
    AND L_SUPPKEY = S_SUPPKEY
    AND C_NATIONKEY = S_NATIONKEY
    AND S_NATIONKEY = N_NATIONKEY
    AND N_REGIONKEY = R_REGIONKEY
    AND R_NAME = 'ASIA'
    AND O_ORDERDATE >= '1994-01-01'
    AND O_ORDERDATE < '1995-01-01'
GROUP BY N_NAME
ORDER BY REVENUE DESC
```

How would you execute this query?
Challenges

SQL

- Declarative (What, not how!)

Options are based on properties of database:

- Data type
- Data distribution

Why do we not choose an option randomly?
Challenges

SQL

- Declarative (What, not how!)

Plenty options to chose from:

- Operator variants (Joins: NL, BNL, Hash, Sort-Merge, ...)
- Table access: Index vs Scans
- Execution: Operator order
- ...
Challenges

SQL

- Declarative (What, not how!)

Plenty options to chose from:

- Operator variants (Joins: NL, BNL, Hash, Sort-Merge, ...)
- Table access: Index vs Scans
- Execution: Operator order
- ...

Options are based on properties of database:

- Data type
- Data distribution
- ...

Andreas Meister

Advanced Query Optimization
Challenges

SQL

- Declarative (What, not how!)

Plenty options to chose from:

- Operator variants (Joins: NL, BNL, Hash, Sort-Merge, ...)
- Table access: Index vs Scans
- Execution: Operator order
- ...

Options are based on properties of database:

- Data type
- Data distribution
- ...

Why do we not chose an option randomly?
Challenge

Because the efficiency depends on the choice.

Adapted from [Neumann, 2014]
How is efficiency ensured?
Query Processing - Phases

- SQL-Query
- Translation & View resolving
- Standardization & Simplification
- Optimization
- Code-Generation
- Execution
- Plan parameterization
- Access plan
- Code
- Access plan
- Translation time
- Runtime

Adapted from [Saake et al., 2012]
Query Processing - Translation & View Resolving

- Simplification of arithmetic expressions
- Resolve sub-queries
- Insertion of view definitions

\[
\sum (L_{\text{EXTENDEDPRICE}} \times (1 - L_{\text{DISCOUNT}})) \cdot \text{N\_NAME}
\]

\[\sigma_{\text{C\_CUSTKEY} = \text{O\_CUSTKEY} \ AND \ L\_ORDERKEY = \text{O\_ORDERKEY} \ AND \ L\_SUPPKEY = \text{S\_SUPPKEY} \ AND \ C\_NATIONKEY = \text{S\_NATIONKEY} \ AND \ S\_NATIONKEY = \text{N\_NATIONKEY} \ AND \ N\_REGIONKEY = \text{R\_REGIONKEY} \ AND \ R\_NAME = \text{ASIA}\} \ AND \ O\_ORDERDATE \geq \text{'1994-01-01'} \ AND \ O\_ORDERDATE < \text{'1995-01-01'}
\]
Query Processing - Standardization & Simplification

- Expressions:
  - Conjunctive normal form:
    \((p_{11} \lor \cdots \lor p_{1n}) \land \cdots \land (p_{m1} \lor \cdots \lor p_{mn})\)
  - Disjunctive normal form:
    \((p_{11} \land \cdots \land p_{1n}) \lor \cdots \lor (p_{m1} \land \cdots \land p_{mn})\)

- Query: Unnesting

\[\sigma \text{Condition} \bowtie R_1 \in R_2\]
Query Processing - Optimization

- Goal: Efficient query execution
Query Processing - Optimization

- Goal: Efficient query execution

- Three phases:
  - Logical optimization
  - Physical optimization
  - Cost-based selection
Query Processing - Optimization

• Goal: Efficient query execution

• Three phases:
  • Logical optimization
  • Physical optimization
  • Cost-based selection

• Method: Use available information about
  • Data (distribution, type, · · ·)
  • Database system (algorithms, processors, · · ·)
  • Query (operators, restrictions, · · ·)
Query Processing - Logical Optimization

- Apply optimization rules

\[
\sum (L_{\text{EXTENDEDPRICE}} \times (1 - L_{\text{DISCOUNT}})) \times N_{\text{NAME}}
\]

\[
\sigma_{R_{\text{NAME}} = 'ASIA'}
\]

\[
\bowtie_{L_{\text{SUPPKEY}} = S_{\text{SUPPKEY}}}
\]

\[
\bowtie_{L_{\text{ORDERKEY}} = O_{\text{ORDERKEY}}}
\]

\[
\bowtie_{C_{\text{CUSTKEY}} = O_{\text{CUSTKEY}}}
\]

\[
\sigma_{O_{\text{ORDERDATE}} \geq '1994-01-01'}
\]

\[
\text{AND } O_{\text{ORDERDATE}} < '1995-01-01'
\]
Query Processing - Algorithm

- Simple optimization algorithm
  - Resolve complex selection predicate, if applicable resolving of $\neg$ and $\lor$
  - Remove redundant operators
  - Pushing down selections as near as possible to the leaf
  - Resolve cross joins
  - Pushing projections in leaf direction
- Single steps will be executed in the stated order until no replacement can be performed
Query Processing - Physical Optimization
Consider:
• Storage information (Indexes, page size, · · ·)
• Algorithms
• Processors
• · · ·
Query Processing - Cost-Based Selection

Often combined with physical optimization

Adapted from [Saake et al., 2012]
Query Processing - Plan Parametrization

Prepared Statements:
- Optimize once
- Execute multiple times
- Configuration via variables
  → Replace variables with values
Query Processing - Code Generation

- Compile query into executable code

TPC-H Query 1 with different vector size [Zukowski et al., 2005]
Query Processing - Execution

- Planed all details
- Execute the code
Query Processing - Execution

- Planed all details
- Execute the code

What could possibly go wrong?
Query Processing - Execution

- Planed all details
- Execute the code

What could possibly go wrong?

In practice:
- Wrong assumptions
- Bad estimates
- ...
Query Processing - Execution

- Planed all details
- Execute the code

What could possibly go wrong?

In practice:
- Wrong assumptions
- Bad estimates
- ...

→ Consider runtime information to adapt query-execution
How does the optimization work?
Cost-based Optimization

query

span search space

equivalent plans

search strategy

"best" plan

cost estimations

transformation rules

Adapted from [Saake et al., 2012]
SQL - TPC-H - Query 5

```sql
SELECT N_NAME, SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT))
FROM CUSTOMER, ORDERS, LINEITEM,
     SUPPLIER, NATION, REGION
WHERE C_CUSTKEY = O_CUSTKEY AND L_ORDERKEY = O_ORDERKEY
  AND L_SUPPKEY = S_SUPPKEY
  AND C_NATIONKEY = S_NATIONKEY
  AND S_NATIONKEY = N_NATIONKEY
  AND N_REGIONKEY = R_REGIONKEY
  AND R_NAME = 'ASIA'
  AND O_ORDERDATE >= '1994-01-01'
  AND O_ORDERDATE < '1995-01-01'
GROUP BY N_NAME
ORDER BY REVENUE DESC
```
SQL - TPC-H - Query 5

```sql
SELECT N_NAME, SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT))
FROM CUSTOMER, ORDERS, LINEITEM,
     SUPPLIER, NATION, REGION
WHERE C_CUSTKEY = O_CUSTKEY AND L_ORDERKEY = O_ORDERKEY
  AND L_SUPPKEY = S_SUPPKEY
  AND C_NATIONKEY = S_NATIONKEY
  AND S_NATIONKEY = N_NATIONKEY
  AND N_REGIONKEY = R_REGIONKEY
  AND R_NAME = 'ASIA'
  AND O_ORDERDATE >= '1994-01-01'
  AND O_ORDERDATE < '1995-01-01'
GROUP BY N_NAME
ORDER BY REVENUE DESC
```

In which order should the join be performed?
Join-Order Optimization

Why consider join-order optimization?

Adapted from [Neumann, 2014]
Join-Order Optimization - Challenge

Goal:
Find an efficient join order
Join-Order Optimization - Challenge

Goal:
Find an efficient join order

Challenge:
- NP-complete problem
- Limited time
Join-Order Optimization - Example

CUSTOMER (C)  ORDERS (O)  LINEITEM (L)  SUPPLIER (S)  NATION (N)  REGION (R)

((L⋈S)⋈C)  ((L⋈S)⋈O)  ((L⋈S)⋈N)  ((L⋈S)⋈R)

((((L⋈S)⋈O)⋈N)⋈C)  ((((L⋈S)⋈O)⋈N)⋈R)


TPC-H Q5

This is not even all!
Join-Order Optimization - Example

This is not even all!
Join-Order Optimization - Tree Form

Deep Trees

Left-Deep Trees

Right-Deep Trees
Join-Order Optimization - Tree Form

More general tree forms:

Zick-Zack Trees

Bushy Trees
Join-Order Optimization - Challenge

Number of equivalent options:

- Left (or Right) Deep-Trees:
  - $n!$
  - For 10 tables: $3.628.800$
Join-Order Optimization - Challenge

Number of equivalent options:

- Left (or Right) Deep-Trees:
  - \( n! \)
  - For 10 tables: 3,628,800
- Bushy Trees:
  - \( S(n) \cdot n! \) variants
  - \[
    S(1) = 1 \\
    S(n) = \sum_{i=1}^{n-1} S(i)S(n - i)
  \]
  - For 10 tables: 17,643,225,600
Join-Order Optimization - Topology

Number of valid trees depends on the query topology:

- **linear**
  - $r_1 \rightarrow r_2 \rightarrow r_3 \rightarrow \ldots \rightarrow r_n$

- **cyclic**
  - $r_1 \rightarrow r_2 \rightarrow r_3 \rightarrow \ldots \rightarrow r_n$

- **star**
  - $r_1 \rightarrow r_2 \rightarrow r_3 \rightarrow \ldots \rightarrow r_n$

- **clique**
  - $r_1 \rightarrow r_2 \rightarrow r_3 \rightarrow \ldots \rightarrow r_n$

Considering cross-joins $\rightarrow$ clique
Join-Order Optimization - Approaches

- Deterministic
  - Greedy
  - Exhaustive search
- Hybrid
- Randomized
  - Transformation
  - Sampling
  - Genetic algorithms
Join-Order Optimization - Deterministic Approaches

- Same input $\rightarrow$ Same output
Join-Order Optimization - Deterministic Approaches

- Same input → Same output

- Greedy:
  - Short runtime
  - Simple heuristics
  - Neither efficiency nor optimality guaranteed
Join-Order Optimization - Deterministic Approaches

- Same input $\rightarrow$ Same output

- Greedy:
  - Short runtime
  - Simple heuristics
  - Neither efficiency nor optimality guaranteed

- Exhaustive Search:
  - Guarantee optimal results
  - Long runtime
  - Only applicable to simple queries
Join-Order Optimization - Randomized Approaches

- Same input $\rightarrow$ Different outputs
- No optimality guaranteed, but often efficient results
Join-Order Optimization - Randomized Approaches

- Same input $\rightarrow$ Different outputs
- No optimality guaranteed, but often efficient results
- Transformation:
  - Selection of initial result
  - Transformation of current result
Join-Order Optimization - Randomized Approaches

- Same input → Different outputs
- No optimality guaranteed, but often efficient results

Transformation:
- Selection of initial result
- Transformation of current result

Sampling:
- Randomly choosing candidates
- Best solution is stored
Join-Order Optimization - Randomized Approaches

- Same input → Different outputs
- No optimality guaranteed, but often efficient results

Transformation:
- Selection of initial result
- Transformation of current result

Sampling:
- Randomly choosing candidates
- Best solution is stored

Genetic algorithms:
- Selection of initial solution pool
- Creating new solutions based on solution pool
Join-Order Optimization - Hybrid Approaches

Advantages deterministic approaches:
- Predictable
- Ensure optimality

Advantages randomized approaches:
- Suitable for complex optimization problems
- Limited runtime

Hybrid approaches:

Combine both deterministic and randomized approaches
Join-Order Optimization - Approaches

- Deterministic
  - Greedy
  - Exhaustive search

- Randomized
  - Transformation
  - Sampling
  - Genetic algorithms

Hybrid

In specific: Dynamic programming
Join-Order Optimization - Dynamic Programming

Algorithm 1: $D_{PSIZE}$ [Selinger et al., 1979]

Input: Join query $Q$ with $n$ tables $T = \{T_1, \ldots, T_n\}$
Output: an optimal bushy join tree

1. foreach $T_i \in T$ do
   2. optimalPlan($T_i$) = $T_i$ ;
3. for $s = 2$ to $n$ do
   4. for $s_l = 1$ to $s - 1$ do
      5. $s_r = s - s_l$ ;
      6. foreach $S_l \subset T : |S_l| = s_l$ do
         7. foreach $S_r \subset T : |S_r| = s_r$ do
            8. if $S_l \cap S_r \neq \emptyset$ then continue;
            9. if $S_l$ not connected to $S_r$ then continue;
           10. optimal-left-plan = optimalPlan($S_l$);
           11. optimal-right-plan = optimalPlan($S_r$);
           12. current-plan = createJoinTree(optimal-left-plan,
                                            optimal-right-plan);
           13. if cost(optimalPlan($S_l \cup S_r$)) > cost(current-plan) then
               14. optimalPlan($S_l \cup S_r$) = current-plan ;
5. return optimalPlan($T$) ;
Dynamic Programming - $DP_{SIZE}$
Dynamic Programming - $DP_{SIZE}$

![Diagram showing relationships between tables T1, T2, T3, T4 and T1,2, T3,4]
Dynamic Programming - $\mathcal{DP}_{\text{SIZE}}$

Andreas Meister
Advanced Query Optimization
Last Change: April 23, 2018
Dynamic Programming - $DP_{SIZE}$

Andreas Meister

Advanced Query Optimization

Last Change: April 23, 2018
Dynamic Programming - $DP_{SIZE}$

- Evaluation of $DP_{SIZE}$ only based on number of tables
  - Not only valid...:
    - $(T_1, T_2, T_3) \bowtie (T_4)$
    - $(T_1, T_2, T_4) \bowtie (T_3)$
    - $(T_1, T_3, T_4) \bowtie (T_2)$
    - $(T_2, T_3, T_4) \bowtie (T_1)$

Solution: Enumerate calculations ($DP_{SUB}$)
Dynamic Programming - $DP_{SIZE}$

- Evaluation of $DP_{SIZE}$ only based on number of tables
  → Not only valid...:
    - $(T_1, T_2, T_3) \bowtie (T_4)$
    - $(T_1, T_2, T_4) \bowtie (T_3)$
    - $(T_1, T_3, T_4) \bowtie (T_2)$
    - $(T_2, T_3, T_4) \bowtie (T_1)$

  .., but also invalid entries are evaluated:
    - $(T_1, T_2, T_3) \bowtie (T_1)$
    - $(T_1, T_2, T_3) \bowtie (T_2)$
    - $(T_1, T_2, T_3) \bowtie (T_3)$
    - ...
Dynamic Programming - $DP_{SIZE}$

- Evaluation of $DP_{SIZE}$ only based on number of tables
  → Not only valid...:
  
  - $(T_1, T_2, T_3) \sqsubset (T_4)$
  - $(T_1, T_2, T_4) \sqsubset (T_3)$
  - $(T_1, T_3, T_4) \sqsubset (T_2)$
  - $(T_2, T_3, T_4) \sqsubset (T_1)$

  .., but also invalid entries are evaluated:
  
  - $(T_1, T_2, T_3) \sqsubset (T_1)$
  - $(T_1, T_2, T_3) \sqsubset (T_2)$
  - $(T_1, T_2, T_3) \sqsubset (T_3)$
  - ... 

Solution: Enumerate calculations ($DP_{SUB}$)
Dynamic Programming - $DP_{SUB}$

- Idea: Use integer representation of solutions

- Table $T_i$ available $\rightarrow$ i-th bit set
  - $1 \equiv (T_1)$
  - $2 \equiv (T_2)$
  - $3 \equiv (T_1, T_2)$
  - $4 \equiv (T_3)$
  - $5 \equiv (T_1, T_3)$
  - $\ldots$
  - $15 \equiv (T_1, T_2, T_3, T_4)$
Algorithm 2: $DP_{SUB}$ [Vance and Maier, 1996]

**Input**: Join query $Q$ with $n$ tables $T = \{T_1, \ldots, T_n\}$

**Output**: an optimal bushy join tree

1. foreach $T_i \in T$ do
   2. $\text{optimalPlan}(T_i) = T_i$
   3. for $k = 2$ to $n$ do
      4. for $S = 2^{k-1} + 1$ to $2^k - 1$ do
         5. foreach $S_l \subset S$ do
            6. $\text{optimal-left-plan} = \text{optimalPlan}(S_l)$;
            7. if $\text{optimal-left-plan} = \emptyset$ then continue;
            8. $S_r = S - S_l$;
            9. $\text{optimal-right-plan} = \text{optimalPlan}(S_r)$;
            10. if $\text{optimal-right-plan} = \emptyset$ then continue;
            11. if $\text{optimal-left-plan}$ not connected to $\text{optimal-right-plan} \neq \emptyset$ then continue;
            12. current-plan = createJoinTree(optimal-left-plan, optimal-right-plan);
            13. if $\text{cost(optimalPlan}(S)) > \text{cost(current-plan)}$ then
                14. $\text{optimalPlan}(S) = \text{current-plan}$;
      15. return $\text{optimalPlan}(2^n - 1)$;
Dynamic Programming - $DP_{SUB}$

- $n=2$:
  - $3 \equiv (T_1, T_2)$
Dynamic Programming - $DP_{SUB}$

- $n=2$:
  - $3 \equiv (T_1, T_2)$
- $n=3$:
  - $5 \equiv (T_1, T_3)$
  - $6 \equiv (T_2, T_3)$
  - $7 \equiv (T_1, T_2, T_3)$
Dynamic Programming - $DP_{SUB}$

- $n=2$:
  - $3 \equiv (T_1, T_2)$

- $n=3$:
  - $5 \equiv (T_1, T_3)$
  - $6 \equiv (T_2, T_3)$
  - $7 \equiv (T_1, T_2, T_3)$

- $n=4$:
  - $9 \equiv (T_1, T_4)$
  - $10 \equiv (T_2, T_4)$
  - $11 \equiv (T_1, T_2, T_4)$
  - $12 \equiv (T_3, T_4)$
  - $13 \equiv (T_1, T_3, T_4)$
  - $14 \equiv (T_2, T_3, T_4)$
  - $15 \equiv (T_1, T_2, T_3, T_4)$
Dynamic Programming - $DP_{SUB}$

- Splitting integers determines calculations
- e.g. solution $s\ 13 \equiv (T_1, T_3, T_4)$:
  - $1\ -\ 12$
  - $4\ -\ 9$
  - $5\ -\ 8$
  - ...
Dynamic Programming - $DP_{SUB}$

- Splitting integers determines calculations
- e.g. solution $s \equiv (T_1, T_3, T_4)$:
  - 1 - 12
  - 4 - 9
  - 5 - 8
  - ...

- Implementation:
  1. Determine the first left join partner $l$ (least significant bit of $s$ using Compiler or DeBruijn)
  2. Determine first right join partner ($s - l$)
  3. Determine next left join partner
     \[ l = s \& (l - s) \]
  4. Repeat step 2-3 until $l \equiv s$
Dynamic Programming - $DP_{SUB}$

Remember: Valid combinations based on query topology:

- **linear**
- **cyclic**
- **star**
- **clique**

$DP_{SUB}$ always enumerate all combinations
For non-cliques also unneeded combinations are evaluated
Dynamic Programming - $DP_{CCP}$

- Idea: Enumerate calculations based on query
Dynamic Programming - $DP_{CCP}$

• Idea: Enumerate calculations based on query

• Approach:
  • Enumerate tables of query in a breath-first-manner (Avoid duplicate calculations)
  • Determine connected sub-graphs within query
  • For sub-graphs: Determine complements (joinable connected-subgraphs)
Algorithm 3: $DP_{CCP}$ [Moerkotte and Neumann, 2006]

Input: Join query $Q$ with $n$ tables $T = \{T_1, \ldots, T_n\}$
Output: an optimal bushy join tree

1. foreach $T_i \in T$ do
   2. optimalPlan($T_i$) = $i$
   3. $csgs$ = enumerateCSG($Q$);

4. foreach $S_l \in csgs$ do
   5. $cmps$ = enumerateCMP($Q, S_l$);

6. foreach $S_r \in cmps$ do
   7. optimal-left-plan = optimalPlan($S_l$);
   8. optimal-right-plan = optimalPlan($S_r$);
   9. current-plan = createJoinTree(optimal-left-plan, optimal-right-plan);
   10. if cost(optimalPlan($S_l \cup S_r$)) > cost(current-plan) then
        11. optimalPlan($S_l \cup S_r$) = current-plan ;
   12. current-plan = createJoinTree(optimal-right-plan, optimal-left-plan);
   13. if cost(optimalPlan($S_l \cup S_r$)) > cost(current-plan) then
        14. optimalPlan($S_l \cup S_r$) = current-plan ;

15. return optimalPlan(R) ;
Dynamic Programming - Overview

- $DP_{SIZE}$
  - Good approach for simple optimization problems
  - Inefficient (invalid join partners), for more complex optimization problems

- $DP_{SUB}$
  - Good for optimizing cliques
  - Based on enumeration of all possible combinations, inefficient for other topologies

- $DP_{CCP}$
  - Only needed join pairs are evaluated based on enumeration
  - Enumeration poses overhead for simple optimization problems
Problems with traditional DP Approaches

- Serial execution of calculations
- Independent calculations available

\[ |S| = 2: R1-R2; R1-R3; R1-R4 \]

- Current multi-core CPUs offer parallel execution
  → Parallel DP-approach needed for current hardware

Adapted from [Han et al., 2008]
Parallelized Dynamic Programming - \textit{PDP}_{SVA}

\textbf{Idea:} Partition independent calculations \cite{Han2008}

\begin{itemize}
\item \textit{QS}: Qualifier set \mid \textit{PlanList}: Optimal query execution plan
\item \(q_1, \ldots, q_4\): Qualifier for relations \mid \(P_1, \ldots, P_4\): QS with 1, \ldots, 4 relations
\end{itemize}
Algorithm 4: $PDP_{SVA}$ [Han et al., 2008]

**Input**: Join query $Q$ with $n$ tables $T = \{T_1, \ldots, T_n\}$

**Output**: an optimal bushy join tree

1. foreach $T_i \in T$ do
2.    optimalPlan($T_i$) = $T_i$;
3. for $s = 2$ to $n$ do
4.    SSDVs = AllocateSearchSpace($S, m$);
5.    for $i = 1$ to $MAX\_THREAD\_ID$ do
6.        threadPool.SubmitJob(MutiplePlanJoin(SSDV$s[i], S));
7.        ThreadPool.sync();
8.    MergeAndPrunePlanPartitions($S$);
9.    for $i = 1$ to $MAX\_THREAD\_ID$ do
10.       threadPool.SubmitJob(BuildSkipVectorArray($i$));
11.      ThreadPool.sync();
12. return optimalPlan($R$);
Parallelized Dynamic Programming - $PDP_{SVA}$

**Idea:** Distribute over different threads [Han et al., 2008]

\[ P_1 \bowtie_{\theta} P_3 \]
\[ P_2 \bowtie_{\theta} P_2 \]

Adapted from [Han et al., 2008]
Allocation Schemata

- **Search space:** \( \left\lfloor \frac{s}{2} \right\rfloor \sum_{\text{smallSZ}=1} \left( |P_{\text{smallSZ}}| \times |P_{S-\text{smallSZ}}| \right) \)

- **Total Sum Allocation:**
Divide the search space in m (number of threads) smaller parts and distribute them equally over the m threads

Adapted from [Han et al., 2008]
Allocation Schemata /2

- **Equi-Depth Allocation:**
  Equally distribute each \((|P_{smallSZ}| \times |P_{S-smallSZ}|)\) over all threads

\[
P_1 \bigotimes \theta \bigotimes P_3
\]

\[
P_2 \bigotimes \theta \bigotimes P_2
\]

Adapted from [Han et al., 2008]
Allocation Schemata /3

• **Round-Robin Outer Allocation:**
Randomly distribute join pairs \((t_i, t'_j)\) to thread \((i \mod m)\)

\[
P_1 \Join_{\theta} P_3 \begin{cases} 
(q_1, q_1 q_2 q_3) & (q_1, q_1 q_2 q_4) & (q_1, q_1 q_3 q_4) \\
(q_2, q_1 q_2 q_3) & (q_2, q_1 q_2 q_4) & (q_2, q_1 q_3 q_4) \\
(q_3, q_1 q_2 q_3) & (q_3, q_1 q_2 q_4) & (q_3, q_1 q_3 q_4) \\
(q_4, q_1 q_2 q_3) & (q_4, q_1 q_2 q_4) & (q_4, q_1 q_3 q_4) \\
\end{cases}
\]

thread 1

\[
P_2 \Join_{\theta} P_2 \begin{cases} 
(q_1 q_2, q_1 q_2) & (q_1 q_2, q_1 q_3) & (q_1 q_2, q_1 q_4) \\
(q_1 q_3, q_1 q_2) & (q_1 q_3, q_1 q_3) & (q_1 q_3, q_1 q_4) \\
(q_1 q_4, q_1 q_2) & (q_1 q_4, q_1 q_3) & (q_1 q_4, q_1 q_4) \\
\end{cases}
\]

thread 2

Adapted from [Han et al., 2008]
**Allocation Schemata /4**

- **Round-Robin Inner Allocation:**
  Randomly distribute join pairs \((t_i, t'_j)\) to thread \((j \mod m)\)

\[
\begin{align*}
P_1 \otimes \theta \ P_3 \left\{ 
\begin{array}{lll}
(q_1, q_1 q_2 q_3) & (q_1, q_1 q_2 q_4) & (q_1, q_1 q_3 q_4) \\
(q_2, q_1 q_2 q_3) & (q_2, q_1 q_2 q_4) & (q_2, q_1 q_3 q_4) \\
(q_3, q_1 q_2 q_3) & (q_3, q_1 q_2 q_4) & (q_3, q_1 q_3 q_4) \\
(q_4, q_1 q_2 q_3) & (q_4, q_1 q_2 q_4) & (q_4, q_1 q_3 q_4)
\end{array}
\right.
\end{align*}
\]

\[
\begin{align*}
P_2 \otimes \theta \ P_2 \left\{ 
\begin{array}{lll}
(q_1 q_2, q_1 q_2) & (q_1 q_2, q_1 q_3) & (q_1 q_2, q_1 q_4) \\
(q_1 q_3, q_1 q_2) & (q_1 q_3, q_1 q_3) & (q_1 q_3, q_1 q_4) \\
(q_1 q_4, q_1 q_2) & (q_1 q_4, q_1 q_3) & (q_1 q_4, q_1 q_4)
\end{array}
\right.
\end{align*}
\]

Adapted from [Han et al., 2008]
Storage of allocation information

- Store distribution information in the search space description vector (SSDV)

- SSDV-Entry: \( \langle \text{smallSZ}, [\text{stOutIdx}, \text{stBlkIdx}, \text{stBlkOff}], [\text{endOutIdx}, \text{endBlkIdx}, \text{endBlkOff}], \text{outInc}, \text{inInc} \rangle \)

- smallSZ: Identifier for join of \(|P_{\text{smallSZ}}| \times |P_{\text{S-smallSZ}}|\)

- stOutIdx: Start index of outer tuple

- stBlkIdx: Start block index

- stBlkOff: Offset of inner tuple within block

- endOutIdx: End index of outer tuple

- endBlkIdx: End block index

- endBlkOff: Offset of end inner tuple within block

- outInc: Step size for outer loop

- inInc: Step size for inner loop
Storage of allocation information

- Store distribution information in the search space description vector (SSDV)

- SSDV-Entry: \( \langle \text{smallSZ}, [\text{stOutIdx}, \text{stBlkIdx}, \text{stBlkOff}], [\text{endOutIdx}, \text{endBlkIdx}, \text{endBlkOff}], \text{outInc}, \text{inInc} \rangle \)
  - **smallSZ**: Identifier for join of \((|P_{\text{smallSZ}}| \times |P_{S-\text{smallSZ}}|)\)
  - **stOutIdx**: Start index of outer tuple
  - **stBlkIdx**: Start block index
  - **stBlkOff**: Offset of inner tuple within block
  - **endOutIdx**: End index of outer tuple
  - **endBlkIdx**: End block index
  - **endBlkOff**: Offset of end inner tuple within block
  - **outInc**: Step size for outer loop
  - **inInc**: Step size for inner loop
Storage of allocation information - example

- 1 Block:
  - Thread 1 - SSDV-Entry:
    \{⟨1, [1, 1, 1], [4, 1, 1], 1, 1⟩, ⟨2, [−1, −1, −1], [−1, −1, −1], 1, 1⟩\}
  - Thread 2 - SSDV-Entry:
    \{⟨1, [4, 1, 2], [4, 1, 3], 1, 1⟩, ⟨2, [1, 1, 1], [3, 1, 3], 1, 1⟩\}

Adapted from [Han et al., 2008]
Parallelized Dynamic Programming - $PDP_{SVA}$

Algorithm 5: MultiplePlanJoin [Han et al., 2008]

**Input:** SSDV, S

1. for $i=1$ to $\lfloor S/2 \rfloor$ do
   2. PlanJoin(SSDV[i],S)
Algorithm 6: PlanJoin [Han et al., 2008]

Input: ssdvElmt, S
1  smallSZ = ssdvElmt.smallSZ; largeSZ = S-smallSZ;
2  for blkIdx = ssdvElmt.stBlkIdx to ssdvElmt.endBlkIdx do
3      blk = blkIdx-th block in $P_{largeSZ}$;
4      $\langle stOutIdx, endOutIdx \rangle = GetOuterRange(ssdvElmt, blkIdx);$;
5      for $t_o=P_{smallSZ}[stOutIdx]$ to $P_{smallSZ}[endOutIdx]$ step by ssdvElmt.outInc do
6          $\langle stBlkOff, endBlkOff \rangle =$
7          GetOffsetRangeInBlk(ssdvElmt,blkIdx,offset of $t_o$);
8          for $t_i = blk[stBlkOff]$ to $blk[endBlkOff]$ step by ssdvElmt.inInc do
9              if $t_o.QS \cap t_i.QS \neq \emptyset$ then
10                 continue;
11              if not ($t_o.QS$ connected to $t_i.QS$) then
12                 continue;
13          Resulting plans = CreateJoinPlans($t_o, t_i$);
14          PrunePlans($P_S, ResultingPlans$);
Storage of allocation information - example

- 1 Block:
  - Thread 1 - SSDV-Entry:
    \{\langle 1, [1, 1, 1], [4, 1, 1], 1, 1 \rangle, \langle 2, [-1, -1, -1], [-1, -1, -1], 1, 1 \rangle\}
  - Thread 2 - SSDV-Entry:
    \{\langle 1, [4, 1, 2], [4, 1, 3], 1, 1 \rangle, \langle 2, [1, 1, 1], [3, 1, 3], 1, 1 \rangle\}

Adapted from [Han et al., 2008]
Parallelizing problems

- Only a small amount of combinations of different join sets are valid

(a) # of disjoint filter calls.  
(b) Selectivities.

Adapted from [Han et al., 2008]
Skip Vector Arrays

**Problem:** High number of invalid combination of qualifier sets
Skip Vector Arrays

**Problem:** High number of invalid combination of qualifier sets

**Idea:**
- Increase performance by skipping unnecessary combinations of join sets
- Store additional skipping information to efficiently determine the next join sets
Skip Vector Arrays

<table>
<thead>
<tr>
<th>P_1</th>
<th>QS</th>
<th>PlanList</th>
<th>SV</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>q_1</td>
<td>...</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>q_2</td>
<td>...</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>q_3</td>
<td>...</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>q_4</td>
<td>...</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>q_5</td>
<td>...</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>q_6</td>
<td>...</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>q_7</td>
<td>...</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>q_8</td>
<td>...</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P_3</th>
<th>QS</th>
<th>PlanList</th>
<th>SV</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>q_1q_2q_3</td>
<td>...</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>q_1q_2q_4</td>
<td>...</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>q_1q_2q_5</td>
<td>...</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>q_1q_2q_6</td>
<td>...</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>q_1q_3q_4</td>
<td>...</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>q_1q_4q_7</td>
<td>...</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>q_1q_4q_8</td>
<td>...</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>q_2q_5q_6</td>
<td>...</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>q_4q_7q_8</td>
<td>...</td>
<td>10</td>
</tr>
</tbody>
</table>
Skip Vector Arrays

Equi-depth partitioning & building SVAs

Adapted from [Han et al., 2008]
Parallelized Dynamic Programming - $PDP_{SVA}$

- Based on $DP_{SIZE}$
  → Same drawback invalid calculations
- Parallelization strategies for $DP_{SUB}$ and $DP_{CCP}$ needed
Parallelized Dynamic Programming - $PDP_{SVA}$

- Based on $DP_{SIZE}$
  - Same drawback invalid calculations
- Parallelization strategies for $DP_{SUB}$ and $DP_{CCP}$ needed
- Idea: Use produce-consumer-model

![Diagram of Producer and Consumers](producer_consumer_diagram.png)
Algorithm 7: $DPE_{GENERIC}$ [Han and Lee, 2009]

**Input**: Join query $Q$ with $n$ tables $T = \{T_1, \ldots, T_n\}$

**Output**: an optimal bushy join tree

1. EnumerationBuffer $B_c$, $B_p$
2. Hash-Table Memo;
3. `partial_order = buildPartialOrder(R);`
4. `e = parseCalculations(partial_order, Memo, B_p, MAX_ENUM_CNT);`
5. **while** $e \neq NO\_MORE\_PAIR$ **do**
   6. `switchBuffers() // $B_c = B_p$ and $B_p = B_c$;`
   7. **for** $i = 0$ to $MAX\_THREAD\_ID - 1$ **do**
      8. `threadPool.SubmitJob(GenerateQEPs(B_c,Memo));`
      9. `e = parseCalculations(partial_order, Memo, B_p, MAX_ENUM_CNT);`
      10. `GenerateQEPs(B_c,Memo));`
   11. `threadPool.sync();`
6. **return** Memo(R);
Parallelized Dynamic Programming - $DPE_{\text{GENERIC}}$

- Characteristics:
  - Parallel
  - Producer-Consumer model
  - Double queue (Producer + Consumer)

- Parameters:
  - Maximal queue size
  - Partial order
  - Enumeration ($DP_{\text{SIZE}}, DP_{\text{SUB}}, DP_{\text{CCP}}$)
Parallelized Dynamic Programming - \( DPE_{\text{GENERIC}} \)

Partial Order: Grouping based on resulting quantifier set

Adapted from [Han and Lee, 2009]

Problem: Only few calculations are grouped together
Parallelized Dynamic Programming - $DPE_{GENERIC}$

Partial Order: Grouping based on size of resulting quantifier set

SRQ $S_1$:
- $(q_1, q_2)$
- $(q_1, q_3)$
- $(q_1, q_4)$
- $(q_2, q_3)$
- $(q_2, q_4)$
- $(q_3, q_4)$

SRQ $S_2$:
- $(q_1, q_2, q_3)$
- $(q_1, q_2, q_4)$
- $(q_1, q_3, q_4)$
- $(q_2, q_3, q_4)$
- $(q_3, q_1, q_4)$
- $(q_3, q_2, q_4)$

SRQ $S_3$:
- $(q_1, q_2, q_3, q_4)$
- $(q_2, q_1, q_3, q_4)$
- $(q_3, q_1, q_2, q_4)$
- $(q_4, q_1, q_2, q_3)$
- $(q_1, q_2, q_3)$
- $(q_1, q_2, q_4)$

SRQ $S_4$:
- $(q_1, q_2, q_3, q_4)$
- $(q_2, q_1, q_3, q_4)$
- $(q_3, q_1, q_2, q_4)$
- $(q_4, q_1, q_2, q_3)$
- $(q_1, q_2)$
- $(q_1, q_3)$
- $(q_1, q_4)$
- $(q_2, q_3)$
- $(q_2, q_4)$
- $(q_3, q_4)$

Adapted from [Han and Lee, 2009]

Problem: Only few independent calculations are available
Parallelized Dynamic Programming - \( DPE_{\text{GENERIC}} \)

Partial Order: Grouping based on size of larger quantifier set

Adapted from [Han and Lee, 2009]
Parallelized Dynamic Programming - $\textbf{DPE}_{\text{ GENERIC}}$

- Each element in partial: own queue

- Insert calculations into corresponding queue
- Consumer iterate over all available queues and pull available calculations
- Access need to be synchronized

Adapted from [Han and Lee, 2009]
Parallelized Dynamic Programming - \( DPE_{\text{GENERIC}} \)

- While consumer evaluate available calculations, the producer creates new calculations
- Number of calculations is predefined

Adapted from [Han and Lee, 2009]
Parallelized Dynamic Programming - $DPE_{GENERIC}$

- Sort calculations based on dependencies
  → Better parallelization

Adapted from [Han and Lee, 2009]
Parallelized Dynamic Programming - $DPE_{\text{generic}}$

- Sort calculations based on dependencies
  $\rightarrow$ Better parallelization

Threading Across Dependencies

What happens after a thread pulled a calculation?

Adapted from [Han and Lee, 2009]
Parallelized Dynamic Programming - \textbf{DPE}_{\text{GENERIC}}

- Processing similar to sequential execution:
  - Fetch information of join-partners
  - Evaluate costs

\begin{itemize}
  \item Problem: Synchronization needed
\end{itemize}

Adapted from [Han and Lee, 2009]
Parallelized Dynamic Programming - $DPE_{GENERIC}$

- Problem: Synchronization needed
- Solution:
  - Group calculations into equivalence classes
  - Link entries directly to entries of the hash table

Adapted from [Han and Lee, 2009]
So what is the best approach?
Dynamic Programming - Evaluation

Linear queries - simple cost function:

<table>
<thead>
<tr>
<th>#Tables</th>
<th>Runtime (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10^2</td>
</tr>
<tr>
<td>3</td>
<td>10^3</td>
</tr>
<tr>
<td>4</td>
<td>10^4</td>
</tr>
<tr>
<td>5</td>
<td>10^5</td>
</tr>
<tr>
<td>6</td>
<td>10^6</td>
</tr>
<tr>
<td>7</td>
<td>10^7</td>
</tr>
<tr>
<td>8</td>
<td>10^8</td>
</tr>
<tr>
<td>9</td>
<td>10^9</td>
</tr>
<tr>
<td>10</td>
<td>10^10</td>
</tr>
<tr>
<td>11</td>
<td>10^11</td>
</tr>
<tr>
<td>12</td>
<td>10^12</td>
</tr>
<tr>
<td>13</td>
<td>10^13</td>
</tr>
<tr>
<td>14</td>
<td>10^14</td>
</tr>
<tr>
<td>15</td>
<td>10^15</td>
</tr>
<tr>
<td>16</td>
<td>10^16</td>
</tr>
<tr>
<td>17</td>
<td>10^17</td>
</tr>
<tr>
<td>18</td>
<td>10^18</td>
</tr>
<tr>
<td>19</td>
<td>10^19</td>
</tr>
<tr>
<td>20</td>
<td>10^20</td>
</tr>
</tbody>
</table>

Andreas Meister
Advanced Query Optimization
Last Change: April 23, 2018
Dynamic Programming - Evaluation

Star queries - simple cost function:

![Graph showing runtime vs. number of tables for different algorithms: DP_SIZE, DP_SUB, DP_CCP, PDP_SVA, DPE_GENERIC. The x-axis represents the number of tables ranging from 2 to 20, and the y-axis represents runtime in ns ranging from 10^2 to 10^12.]
Dynamic Programming - Evaluation

Clique queries - simple cost function:

\[
\begin{array}{c|cccccc}
\text{#Tables} & DP_{SIZE} & DP_{SUB} & DP_{CCP} & PDP_{SVA} & DPE_{GENE} \\
2 & 10^2 & 10^2 & 10^2 & 10^2 & 10^2 \\
3 & 10^4 & 10^4 & 10^4 & 10^4 & 10^4 \\
4 & 10^6 & 10^6 & 10^6 & 10^6 & 10^6 \\
5 & 10^8 & 10^8 & 10^8 & 10^8 & 10^8 \\
6 & 10^{10} & 10^{10} & 10^{10} & 10^{10} & 10^{10} \\
7 & 10^{12} & 10^{12} & 10^{12} & 10^{12} & 10^{12} \\
8 & 10^{14} & 10^{14} & 10^{14} & 10^{14} & 10^{14} \\
9 & 10^{16} & 10^{16} & 10^{16} & 10^{16} & 10^{16} \\
10 & 10^{18} & 10^{18} & 10^{18} & 10^{18} & 10^{18} \\
11 & 10^{20} & 10^{20} & 10^{20} & 10^{20} & 10^{20} \\
12 & 10^{22} & 10^{22} & 10^{22} & 10^{22} & 10^{22} \\
13 & 10^{24} & 10^{24} & 10^{24} & 10^{24} & 10^{24} \\
14 & 10^{26} & 10^{26} & 10^{26} & 10^{26} & 10^{26} \\
15 & 10^{28} & 10^{28} & 10^{28} & 10^{28} & 10^{28} \\
\end{array}
\]
Dynamic Programming - Evaluation
Linear queries - complex cost function:

\[ DP_{SIZE} \quad DP_{SUB} \quad DP_{CCP} \quad PDP_{SVA} \quad DPE_{GENE} \]

Complex

Runtime (ns)

\[ \begin{align*}
10^{12} & \quad 10^{11} & \quad 10^{10} & \quad 10^{9} & \quad 10^{8} & \quad 10^{7} & \quad 10^{6} & \quad 10^{5} & \quad 10^{4} & \quad 10^{3} & \quad 10^{2} \\
2 & \quad 3 & \quad 4 & \quad 5 & \quad 6 & \quad 7 & \quad 8 & \quad 9 & \quad 10 & \quad 11 & \quad 12 & \quad 13 & \quad 14 & \quad 15 & \quad 16 & \quad 17 & \quad 18 & \quad 19 & \quad 20
\end{align*} \]

#Tables
Dynamic Programming - Evaluation

Star queries - complex cost function:

![Graph showing runtime comparison between different algorithms like DP_{SIZE}, DP_{SUB}, DP_{CCP}, PDP_{SVA}, and DPE_{GENE} for varying number of tables. The x-axis represents the number of tables ranging from 2 to 20, and the y-axis represents runtime in nanoseconds logarithmically scaled from $10^2$ to $10^{12}$.

Andreas Meister
Advanced Query Optimization
Last Change: April 23, 2018
76/90
Dynamic Programming - Evaluation

Clique queries - complex cost function:

\[ \begin{array}{c|l}
\text{#Tables} & \text{Runtime (ns)} \\
\hline
2 & 1 \times 10^2 \\
3 & 1 \times 10^4 \\
4 & 1 \times 10^6 \\
5 & 1 \times 10^8 \\
6 & 1 \times 10^{10} \\
7 & 1 \times 10^{12} \\
\end{array} \]

![Graph showing runtime vs. number of tables for different algorithms](image-url)

- \( D_{\text{SIZE}} \)
- \( D_{\text{SUB}} \)
- \( D_{\text{CCP}} \)
- \( P_{\text{PDP}_{\text{SVA}}} \)
- \( D_{\text{PDE}_{\text{GEN}}} \)
Dynamic Programming - Evaluation

- There is no best approach:
  - Sequential approaches good for simple problems
  - Parallel approaches good for complex problems
Dynamic Programming - Evaluation

• There is no best approach:
  • Sequential approaches good for simple problems
  • Parallel approaches good for complex problems

• Simple problems:
  • Simple cost function
  • Small number of tables
Dynamic Programming - Evaluation

- There is no best approach:
  - Sequential approaches good for simple problems
  - Parallel approaches good for complex problems

- Simple problems:
  - Simple cost function
  - Small number of tables

- Complex problems:
  - Complex cost function
  - Large number of tables
How are cost calculated?
Cost estimation

- Cost-based optimization needs cost estimations

- Cost-estimation directly influence optimization
Cost estimation

- Cost-based optimization needs cost estimations
- Cost-estimation directly influence optimization

- Challenge:
  - Accurate estimations
  - Time limit
Cost estimation

- Cost-estimation directly influence optimization:
  - Good estimation $\rightarrow$ good optimization results
  - Bad estimations $\rightarrow$ unpredictable optimization results
Cost estimation

- Cost-estimation directly influence optimization:
  - Good estimation $\rightarrow$ good optimization results
  - Bad estimations $\rightarrow$ unpredictable optimization results

- Challenge:
  - Accurate estimations
  - Time limit
Cost estimation

- Cost-estimation directly influence optimization:
  - Good estimation $\rightarrow$ good optimization results
  - Bad estimations $\rightarrow$ unpredictable optimization results

- Challenge:
  - Accurate estimations
  - Time limit

- Components:
  - Cardinality estimation:
    - Statistics + Formulas
    - Histograms
    - Parametric approaches
    - Sample-based approaches
  - Cost function
Cardinality estimation - Statistics + Formulas

• Idea: Estimate selectivity of operators

\[ |\sigma_F(R(R))| = sel(F, R) \cdot |r| \]
Cardinality estimation - Statistics + Formulas

• Idea: Estimate selectivity of operators

\[ |\sigma_F(R(R))| = sel(F, R) \cdot |r| \]

• Estimation (for interpolateable, arithmetic values):

\[
\begin{align*}
  sel(A = v, R) &= \frac{1}{val_{A,r}} \\
  sel(A < v, R) &= \frac{v - A_{min}}{A_{max} - A_{min}} \\
  sel(A > v, R) &= \frac{A_{max} - v}{A_{max} - A_{min}} \\
  sel(A \text{ between } v_1 \text{ and } v_2, R) &= \frac{v_2 - v_1}{A_{max} - A_{min}}
\end{align*}
\]

• Further cost formulas presented in [Saake et al., 2012]
Cardinality estimation - Histograms

Adapted from [Saake et al., 2012]
Cardinality estimation - Parametric approaches
Gaussian Distribution Cluster

Adapted from [Böhm et al., 2005]
Cardinality estimation - Sample-based approaches

Kernel density estimator

Adapted from [Heimel and Markl, 2012]
Cardinality estimation - Cost function

- Input:
  - Cardinality estimation
  - Query information
  - Database information

- Output: Cost estimation (artificial, time, ⋯)
Cardinality estimation - Cost function

• Input:
  • Cardinality estimation
  • Query information
  • Database information

• Output: Cost estimation (artificial, time, ⋯)

• Considered aspects:
  • Disk accesses
  • CPU consumption
  • Memory consumption
  • Network traffic
  • Execution time
  • Cache misses
  • ⋯
Cardinality estimation - Cost function

- Input:
  - Cardinality estimation
  - Query information
  - Database information

- Output: Cost estimation (artificial, time, ...) 

- Considered aspects:
  - Disk accesses
  - CPU consumption
  - Memory consumption
  - Network traffic
  - Execution time
  - Cache misses
  - ...

- Complex is not always better! [Leis et al., 2015]
Is this all?
Outlook

- Modern hardware
- Cost estimation
- New optimization approaches:
  - Ant-Colony optimization
  - Genetic algorithms
  - Mixed integer linear programming
  - ...
- Further optimization problems:
  - Physical Database Design
    - Partitioning
    - Index
    - Materialized views
  - Self-Tuning
  - ...
Wrap-up

- Query Optimization
- Join-Order Optimization
- Dynamic Programming
  - Sequential
  - Parallel
- Cost estimation
Wrap-up

- Query Optimization
- Join-Order Optimization
- Dynamic Programming
  - Sequential
  - Parallel
- Cost estimation

There is no best approach
Join us!

- Possible collaboration:
  - Thesis
    www.dbse.ovgu.de/Thesis_Jobs.html
  - Scientific Team Project: Modern Database Technologies
    http://www.dbse.ovgu.de/Lehre/Scientific+Team+Project.html

- Requirements:
  - Motivation
  - (Programming skills)
References I

Selectivity Estimation of High Dimensional Window Queries via Clustering.
In Advances in Spatial and Temporal Databases, volume 3633 of Lecture Notes in Computer Science, pages 1–18. Springer.

Parallelizing Query Optimization.

Dependency-aware Reordering for Parallelizing Query Optimization in Multi-core CPUs.
SIGMOD, pages 45–58. ACM.

A First Step Towards GPU-assisted Query Optimization.
In ADMS.

How Good Are Query Optimizers, Really?

Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees Without Cross Products.
VLDB, pages 930–941. VLDB Endowment.
References II

Engineering High-Performance Database Engines. 

*Datenbanken: Implementierungstechniken.*
mitp-Verlag, Bonn, 3 edition.

Access Path Selection in a Relational Database Management System. 
*SIGMOD*, pages 23–34. ACM.

Rapid Bushy Join-order Optimization with Cartesian Products. 
*SIGMOD*, pages 35–46. ACM.

Monetdb/x100 - A DBMS in the CPU cache. 