Datenbanksysteme II: Query Execution

Ulf Leser
Content of this Lecture

- Overview: Query optimization
- Relational operators
- Implementing (some) relational operators
- Query execution models
5 Layer Architecture

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

We are here
Query Optimization

- We have
  - Structured Query Language SQL
  - Relational algebra
  - How to access tuples in many ways (scan, index, ...)

- Now
  - Given a SQL query
  - Find a fast way and order of accessing tuples from different tables such that the answer to the query is computed
  - Usually, we won’t find the best way, but avoid the worst
  - Use knowledge about value distributions, access paths, query operators, IO cost, ... 
  - Compile a declarative query in an “optimal” executable program
Steps (Sketch)

• Translate query in a **logical query execution plan** (QEP)
  - Structured representation of a relational algebra expression
• Logical optimization: QEPs are rewritten in other, **semantically equivalent** and hopefully faster QEPs
  - E.g., selection is commutative: $\sigma_A(\sigma_B(expr)) = \sigma_B(\sigma_A(expr))$
• Physical optimization: For each (relational) operator in the query, we have **multiple possible implementations**
  - Table access: scan, indexes, sorted access through index, ...
  - Joins: Nested loop, sort-merge, hash, ...
• Query execution: Execute the best query plan found
Overview Optimization

SQL query
- parse
  - Parse tree
- convert
  - Logical QEP
    - estimate selectivities
      - Annotated QEP
        - Logical / physical Rewriting

Result
- execute
  - Best plan
    - Search space traversal
      - Many equivalent QEPs
Overview Optimization

SQL query
  ↓
parse
  ↓
Parse tree
  ↓
convert
  ↓
Logical QEP
  ↓
estimate selectivities
  ↓
Annotated QEP

Stat Store

Result
  ↑
execute
  ↑
Best plan
  ↑
Search space traversal
  ↑
Many equivalent QEPs
  ↑
Logical / physical Rewriting

Update statistics
Adaptive Optimization

SQL query
  ↓
parse
  ↓
Parse tree
  ↓
convert
  ↓
Logical QEP
  ↓
estimate selectivities
  ↓
Annotated QEP

Update selectivities
  ↓
execute
  ↓
Best plan
  ↓
Search space traversal
  ↓
Many equivalent QEPs
  ↓
Logical / physical Rewriting
  ↓
Annotated QEP

Plan adaptation
  ↓
Result
Example SQL query

```
SELECT title
FROM starsIn
WHERE starName IN (  
    SELECT name
    FROM movieStar
    WHERE birthdate LIKE '%%1960'
);  

(Find all movies with stars born in 1960)
```
SELECT title
FROM starsIn
WHERE starName IN (  
    SELECT name
    FROM movieStar
    WHERE birthdate LIKE '%1960'
);
SELECT title
FROM starsIn
WHERE starName IN (  
  SELECT name  
  FROM movieStar  
  WHERE birthdate LIKE '1960'  
);

Π_{title} (σ_{starName=name}(starsIn) \times σ_{birthdate}(movieStar))
Improved Logical Query Plan

\[ \Pi_{\text{title}} \sigma_{\text{starName}=\text{name}} \times \Pi_{\text{name}} \sigma_{\text{birthdate} \text{ LIKE} \ '%1960'} \text{MovieStar} \]

\[ \Pi_{\text{title}} \sigma_{\text{starName}=\text{name}} \times \Pi_{\text{name}} \sigma_{\text{birthdate} \text{ LIKE} \ '%1960'} \text{MovieStar} \]

**Question:** Push projection to StarsIn?
Physical Plan

Hash join

- Parameters: Join order, selectivity, memory size, size of attributes, ...

- sequential scan

  - StarsIn

- index scan

  - MovieStar

- Parameters: Selectivity, fragmentation of data file, size of tuples, ...
Content of this Lecture

- Overview: Query optimization
- Relational operators
- Implementing (some) relational operators
- Query execution models
Relational Operations: One Table

- In the following: Table means table or intermediate result
- **Selection** $\sigma$: WHERE clause
  - Read table and filter tuples based on condition
  - Possibility: Use index to access only the qualifying tuples
  - Selection never increases table length (selectivity)
  - Conjunctions, disjunction, equality, negation, ...
- **Projection** $\pi$: SELECT clause
  - Read table and manipulate columns
  - In SET semantic, also duplicates must be filtered
  - Projection usually decreases breadth of table
    - When not?
Relational Operations: One Table cont’d

- **Group-by**: Grouping and aggregation
  - Put all tuples with equal values in all grouping attributes into one bag; output one tuple per bag by aggregating values
  - Implementation by sorting or hashing

- **Distinct**: Duplicate elimination
  - Read table and remove all duplicate tuples
  - May also be injected to speed-up EXIST clauses
  - Implementation by sorting or hashing

- **Order-by**: Sorting
  - Always last clause in query, but injected often by optimizer
  - Pipeline breaker
Relational Operations: Two Tables

- **Cartesian product \( \times \)**
  - Read two tables and build *all pairs* of tuples
  - Usually avoided - combine product and selection to join
  - Products in a plan are hints to *wrong queries*
  - Specified *implicitly* by FROM clause

- **Join \( \bowtie \)**
  - All pairs of tuple matching the join condition
  - Natural join, theta join, *equi join*, semi join, outer join
  - Expensive - favorite target of optimizers
  - Possibility: *Join-order* and join implementation
  - Specified *implicitly or explicitly* in WHERE clause
Relational Operations: Two Queries

- **Union** \( \cup \)
  - Read two tables and build union of all tuples
  - Duplicates are removed (alternative: \textit{UNION-ALL})
  - Requires tables to have same schema

- **Intersection** \( \cap \)
  - Read two tables and build intersection of tuples
  - Requires tables to have same schema
  - Same as join over all attributes

- **Minus** \( \setminus \)
  - Subtract tuples of one table from tuples from the other
  - Requires tables to have same schema
Content of this Lecture

- Overview: Query optimization
- Relational operators
- Implementing (some) relational operators
- Query execution models
Select versus Update

• We do not discuss update, delete, insert
• Update and delete usually have embedded queries – “normal” optimization
  – But: data tuples must be loaded (and locked and changed and persistently written if TX not rolled-back)
  – Some tricks don’t work any more
• Insert may have query
Implementing Operations

• Most single table operations are straight-forward
  - See book by Garcia-Molina, Ullmann, Widom for detailed discussion
• Joins are more complicated – later
• Sorting, especially for large tables, is important
  - External sorting – we have seen Merge-Sort
• We sketch three single table operations
  - Scanning a table
  - Duplicate elimination
  - Group By
Scanning a Table

- At the bottom of each operator tree are relations
- Accessing them implies a table scan
  - If table T has b blocks, this costs b IO
- Often better: Combine with next operation in plan
  - `SELECT t.A, t.B FROM t WHERE A=5`
  - Selection: If index on T.A available, perform index scan
    - Assume |T|=n, |A|=a different values, z=n/a tuples with T.A=a
      - Index has height \( \log_k(n) \)
      - Scan B+ index and find all matching TIDs
      - Accessing z tuples from T costs 1-z IO (sequential or random)
    - Especially effective if A is a key: Only one tuple selected
  - Projection: Integrate into table scan
    - Read complete tuples, but only pass-on attributes that are needed
      - Why not read partial tuples?
Scanning a Table 2

- Conditions can be complex
  
  ```sql
  SELECT t.A, t.B FROM t
  WHERE A=5 AND (B<4 OR B>9) AND C='müller' ...
  ```

- Approach
  - Compute conjunctive normal form
  - Independent indexes: Find TID lists for each conjunct, then intersect
  - With MDIS: Directly find matching TIDs
  - Without indexes: Scan table and evaluate condition for each tuple

- For complex conditions and small tables, linear scanning usually is faster
  - Depends on expected result size
  - Cost-based optimization required
Duplicate Elimination

• **Option 1: Sorting**
• Sort table on **DISTINCT** columns
  - Can be skipped if table is already sorted
• Scan sorted table and output only unique tuples
• Generates output in sorted order (for later reuse)
• **Pipeline breaker** (see later)
• Memory: Use external sorting, then pipeline
Duplicate Elimination

• Option 2: Use **hashing**
  • Scan table and build hash table on all **unique values**
    - Needs good hash function, avoid conflicts
  • When reading a tuple, check if it has already been seen
    - If not: insert tuple and copy it to the output; else: skip tuple
    - No pipeline breaker
    - Does not sort result (but existing sorting would remain)
• No pipeline breaker
• Memory: Problem; assumes **S to fit in memory**
Performance

- Assumptions
  - Main memory: m blocks
  - Table: b blocks
- Using external sorting
  - If table is sorted, we need b IO
  - If table not sorted, we need $2 \cdot b \cdot \text{ceiling}(\log_m(b)) - b$ IO
- Using internal data structure
  - If all distinct values fit into m, we need b IO
    - Estimate from statistics
  - Otherwise … use two pass algorithms (e.g. hash-join like; later)
Grouping and Aggregation

SELECT T.day_id, sum(amount*price)  
FROM sales S  
GROUP BY T.day_id

• SELECT must contain only GROUP BY attributes and aggregate functions
• Partition result of inner query by GROUP BY attributes
• For each partition, compute one result tuple: GROUP BY attributes and aggregate function applied on values of other attributes in this partition
  – Note: Depending on the aggregate function, we might need to buffer more than one value per partition – examples?
Implementing GROUP BY

• Proceed like duplicate elimination
• Also keep to-be-aggregated attributes
• Eventually, **compute the aggregated columns**
  - Simple: SUM, COUNT, MIN, MAX, ANY
  - More memory required: AVG, Top-5, median
• Pipelining? Same properties as for duplicate elimination
Computing Median

- **Option 1:** Partition table *into k partitions*
  - Scan table
  - Build (hash) table for first k different GROUP BY values
  - When reading one of first k, add value to (sorted) list
  - When reading other GROUP value, discard
  - When scan finished, output median of k groups
  - Iterate - next k groups

- **Option 2:** Sort table on GROUP BY and Median attribute
  - Then scan sorted data
  - Buffer all values per group
  - When next group is reached, output middle value

- **What if we cannot buffer all values of a group?**
Content of this Lecture

• Overview: Query optimization
• Relational operators
• Implementing (some) relational operators
• Query execution models
Query Execution

• Typical model: Operator implementations call each other to pass tuples up the tree
  • Iterator concept: Open, next, close
    – Each operator implementation needs these three methods
  • Produces deep stacks and many push/pops
  • Plan generation is simple: Composition of independent blocks

• Two modes: Blocked, Pipelined
• Work mostly done in open (if blocking) or in next (if pipelined)

• Modern alternative: Compile into function-free program
Example – Blocked (Sketch)

```
p = projection.open();
while p.next(t)
    output t;
p.close();
```

```java
class projection {
    open() {
        j = join.open();
        while j.next(t)
            tmp[i++]=t.title;
        j.close();
        cnt:=0;
    }
    next( t) {
        if (cnt<tmp.max)
            t = tmp[cnt++];
        return true;
        else return false;
    }
    close() {
        discard( tmp);
    }
}
class join {
    open() {
        l = table.open(starsIn);
        while l.next(tl)
            r = table.open(movieStar)
                while r.next(tr)
                    if tl.starname=tr.name
                        tmp[i++]=tl⋈tr;
                    r.close();
        end while;
    l.close();
    cnt:=0;
    }
    next( t) {
        if (cnt<tmp.max)
            t = tmp[cnt++];
        return true;
        else return false;
    }
    close() {
        discard( tmp);
    }
}
```
Example – Pipelined (Sketch)

\[ \pi_{\text{title}} \Join \starName=\text{name} \]

\[
\begin{align*}
p &= \text{projection}.\text{open}(); \\
\text{while } p.\text{next}(t) &\quad \text{output } t; \\
p.\text{close}(); \\
\end{align*}
\]

\[
\begin{align*}
\text{class projection} \{ \\
\text{open}() \{ \\
\quad j &= \text{join}.\text{open}(); \\
\} \\
\text{next}(t) \{ \\
\quad \text{if } j.\text{next}(t) &\quad \text{return } t.\text{title} \\
\quad \text{else} &\quad \text{return } \text{false}; \\
\} \\
\text{close}() \{ \\
\quad j.\text{close}(); \\
\} \\
\end{align*}
\]

\[
\begin{align*}
\text{class join} \{ \\
\text{open}() \{ \\
\quad l &= \text{table}.\text{open}(\text{starsIn}); \\
\quad r &= \text{table}.\text{open}(\text{movieStar}); \\
\quad l.\text{next}(t); \\
\} \\
\text{next}(t) \{ \\
\quad \text{if } r.\text{next}(tr) &\quad \text{if } tl.\starName=tr.\text{name} \\
\quad t &= tl \Join tr; &\quad \text{return } \text{true}; \\
\quad \text{else} &\quad \text{if } l.\text{next}(tl) \\
\quad r.\text{close}(); &\quad r = \text{table}.\text{open}(\text{movieStar}); \\
\quad \text{return } \text{false}; &\quad \text{return } \text{next}(t); \\
\quad \text{else} &\quad \text{return } \text{false}; \\
\} \\
\text{close}() \{ \\
\quad l.\text{close}(); \\
\quad r.\text{close}(); \\
\} \\
\end{align*}
\]
Example – Compiled (Sketch)

\[ \Pi_{\text{title}} \left( \star_{\text{starName}=\text{name}} \right) \]

```
l = table.open(starsIn);
r = table.open(movieStar);
go = l.next( tl);
while go do
  if r.next(tr)
    if tl.starname=tr.name
      t=tl⋈tr;
      output t.title;
    else
      if l.next( tl)
        r.close();
        r = table.open(movieStar);
      else
        l.close();
        r.close();
        go = false;
  else
    if l.next( tl)
      r.close();
      r = table.open(movieStar);
    else
      l.close();
      r.close();
      go = false;
```
Pipelined versus Blocked

• Pipelining is in general advantageous
  - Very little demand for buffer space
    • When intermediate results are large, buffers need to be stored on disk
  - Operations can be assigned to different threads or CPUs
    • Overlapping execution
  - Results come early and continuously

• Pipeline breaker
  - Some operations cannot be pipelined
  - Sorting: next() can be executed only after entire table was read
    • Exception: When input is sorted
  - Grouping and aggregation
    • Depending on implementation
  - Minus, intersection
Pipelined versus Blocked

- Projection with duplicate elimination
  - Need not be a pipeline breaker
  - Recall implementation without sorting
  - next() can return early
  - But we need to keep track of all values already returned - requires large buffer
Bag and Set Semantic

• Relational algebra has **SET semantic**
  - All relations are duplicate-free
  - Result of each query is duplicate-free
  - Result of each intermediate result is duplicate-free

• SQL databases use **BAG semantic**
  - More practical in applications
  - PKs are used to prevent existence of “real” duplicates

• But: Duplicate elimination remains an important task
  - Explicit **DISTINCT clause**
  - **EXIST**
  - ..