Datenbanksysteme II: Query Execution

Ulf Leser
Content of this Lecture

• Relational operations
• Physical query plan operators
• Implementing (some) relational operators
• Execution models
5 Layer Architecture

Data Model

Logical Access

Data Structures

Buffer Management

Operating System

We are here
Query Execution

• 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 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 operations, IO cost, …
  - Compile a declarative query in an “optimal” executable program
Steps (Sketch)

- Translate query in a relational algebra expression
- Logical rewriting: Each expression can be rewritten in many other, semantically equivalent expressions
- Physical optimization: For each relational operation, 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
Exemplary Workflow

SQL query → parse → parse tree → convert → logical query plan → estimate result sizes → "improved" l.q.p → consider physical plans → l.q.p. +sizes → estimate costs → pick best → execute → answer

{P1,P2,...} → compute statistics / adaptation → {P1,C1),(P2,C2)...} → Pi → answer
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   <SelList>    FROM    <FromList>     WHERE     <Condition>
        <Attribute>              <RelName>                 <Tuple>  IN  <Query>
        title                       StarsIn               <Attribute>      (  <Query>  )

SELECT      <SelList>    FROM     <FromList>     WHERE     <Condition>
        <Attribute>           <RelName>         <Attribute>  LIKE  <Pattern>
        name                 MovieStar              birthDate            '%1960'
Logical Query Plan

\[ \Pi_{\text{title}} \sigma_{\text{starName}=\text{name}} \times \Pi_{\text{name}} \sigma_{\text{birthdate LIKE '1960'}} \times \Pi_{\text{MovieStar}} \]
Improved Logical Query Plan

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

Question: Push projection to StarsIn?
Physical Plan

Hash join

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

sequential scan

index scan

Parameters: Selectivity, fragmentation of data file, size of tuples, ...

StarsIn

MovieStar
Content of this Lecture

• Relational operations
• Physical query plan operators
• Implementing (some) relational operators
• Execution models
Relational Operations: One Table

• In the following: Table means table or intermediate result

• Selection \( \sigma \)
  - Read table and filter tuples based on condition
  - Possibility: Use index to access only the qualifying tuples
  - Specified in WHERE clause of SQL query

• Projection \( \pi \)
  - Read table and manipulate columns (usually remove columns)
  - In SET semantic, also duplicates must be filtered
  - Possibility: Push projections down to base tables
  - Projection usually decreases size of table
    • When not?
  - Specified in SELECT clause of SQL query
Relational Operations: One Table cont’d

• Grouping
  - Read table and put all tuples with equal values in all grouping attributes into bag; output one tuple per bag by aggregating values
  - Possibility: Reuse order in table
  - SQL GROUP-BY clause

• Duplicate elimination
  - Read table and remove all duplicate tuples
  - Possibility: Reuse order in table
  - Specified in SELECT clause

• Sorting
  - Not an operation in relational algebra, but important in practice
  - Possibility: Reuse order in table
  - SQL ORDER-BY clause
Relational Operations: Two Tables

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

- **Derived operation: Join ⋈**
  - Read two tables and combine matching tuples
  - Natural join, theta join, equi join, semi join, outer join
  - Possibility: Join-order; nested-loop join, sort-merge join, hash join, index join, …
  - Specified in WHERE clause or in FROM clause
Relational Operations: Two Queries

• **Union** \( \cup \)
  - Read two tables and build union of all tuples
  - Proper SQL command for two queries

• **Intersection** \( \cap \)
  - Read two tables and build intersection of tuples
  - Same as join over all attributes
  - SQL command for two queries

• **Minus** \( \setminus \)
  - Subtract tuples of one table from tuples from the other
Content of this Lecture

- Relational operations
- Physical query plan operators
- Implementing (some) relational operators
- Execution models
Select versus Update

- We do not discuss update, delete, insert
- Update and delete have queries – “normal” optimization
  - But: data tuples must be loaded (and locked and changed and written)
  - Some tricks don’t work any more (e.g. “oversized” index)
- Insert may have query
Implementing Operations

- Most single table operations are rather straightforward
  - 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 $\log_k(n)$
      - Accessing z tuples from T costs (worst-case) z IO
    - Especially effective if A is a key: Only one tuple selected
  - Projection: Integrate into table scan
    - Read tuples, but only pass-on attributes that are needed
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
  - With indexes: Compute TID lists for each conjunct and intersect
    - Alternatives?
      - Without indexes: Scan table and evaluate condition for each tuple
  
- For complex conditions and small tables, linear scanning might be faster
  
  - Depends on expected result size
  - Cost-based optimization required
Duplicate Elimination

• Option 1: **Sorting**
• Sort input table (or intermediate result) on DISTINCT columns
  – Can be skipped if table is already sorted
• Scan sorted table and output only unique tuples
• Generates output in sorted order
• **Pipeline breaker** (see later)
Duplicate Elimination

• Option 2: Use internal sorting/hasing
  • Scan input table and build internal data structure holding each unique tuple once
    – Binary tree – some cost for balancing, robust
    – Hash table – might be faster, needs good hash function
  • 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)
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 $2b \cdot \text{ceiling}(\log_m(b)) - b$ IO
    • How?

• Using internal sorting
  – 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 all 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
- But we also need to **compute the aggregated columns**
  - No problem: SUM, COUNT, MIN, MAX, ANY
  - What to do for AVG? Top-5?
  - What to do for Median?
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

- Relational operations
- Physical query plan operators
- Implementing (some) relational operators
- Execution models
Query Execution

• Operator implementations need to call each other to pass tuples up the tree
  • Iterator concept: Open, next, close
    – Each operator implementation needs these three methods

• Two modes
  – Blocked
  – Pipelined

• Work usually done in open (if blocking) or in next (if pipelined)
Example – Blocked (Sketch)

\[ \Pi \text{title} \]
\[ \text{starName}=\text{name} \]
\[ \sigma \text{birthdate LIKE '1960'} \]
\[ \Pi \text{name} \]
\[ \text{StarsIn} \]
\[ \text{MovieStar} \]

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

class projection {
   open() {
      j = join.open();
      while j.next(t)
         tmp[i++]=t;
      j.close();
      cnt:=0;
   }
   next(t) {
      if (cnt<max)
         t = tmp[cnt++];
      return true;
   }
   close() {
      discard(tmp);
   }
}

class join {
   open() {
      l = table.open();
      while l.next(tl)
         r = projection.open()
         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<max)
         t = tmp[cnt++];
      return true;
   }
   close() {
      discard(tmp);
   }
}
```

Example – Pipelined (Sketch)

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

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

class projection {
    open() {
        j = join.open();
    }
    next(t) {
        return j.next(t);
    }
    close() {
        j.close();
    }
}

class join {
    open() {
        l = table.open();
        r = projection.open()
        l.next(tl);
    }
    next(t) {
        if r.next(tr)
            if tl.starname=tr.name
                t=tl⋈tr;
                return true;
            else
                if l.next(tl)
                    r.reset();
                    return next(t);
                else
                    return false;
        }
    close() {
        l.close();
        r.close();
    }
```
Pipelined versus Blocked

- Pipelining is in general advantageous
  - Very little demand for buffering
    - When intermediate results are large, buffers need to be stored on disk
  - Operations can be distributed to different threads or CPUs
  - 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
  - What else?