DATABASDESIGN FÖR INGENJÖRER - 1DL124

Sommar 2007

En introduktionskurs i databassystem

http://user.it.uu.se/~udbl/dbt-sommar07/
alt. http://www.it.uu.se/edu/course/homepage/dbdesign/st07/

Kjell Orsborn
Uppsala Database Laboratory
Department of Information Technology, Uppsala University,
Uppsala, Sweden
Introduction to Query Processing

(Elmasri/Navathe ch. 15)
(Padron-McCarthy/Risch ch 25)

Kjell Orsborn

Department of Information Technology
Uppsala University, Uppsala, Sweden
Query optimization

- **Query optimization** is the process of choosing, among several possible execution strategies, an “optimal” **execution strategy** for processing a query.

- Two basic approaches (a combination is usually used in reality):
  - **heuristic query optimization** - uses heuristic rules for reordering the operations in a query tree. E.g. a general heuristic is to apply operations that decrease the size on intermediate results (tables).
    - Make **selections** as early as possible.
      - i.e. move $\sigma$ as far down the query tree as possible.
    - Make **projections** as early as possible.
      - i.e. move $\pi$ as far down the query tree as possible.
    - Perform the most restrictive $\sigma$- and join-operations first.
  - **cost-based query optimization** (also called systematic query optimization) - uses statistics about data size and selectivities and cost measures to rank different execution strategies.
Basic steps in query processing

query → scanning, parsing & validating → relational calculus expression → optimizer

optimizer → execution plan - annotated (physical) relational algebra → code generator

code generator → executable query → evaluation engine

evaluation engine → data

data

query output

statistics about data
Basic steps in query proc. ...

1. Parsing and translation
   - Translate the query into its internal form.
   - Parser checks syntax, verifies relations
   - Tuple calculus expression

2. Optimization
   - A relational algebra expression may have many equivalent expressions
     - e.g. $\sigma_{\text{balance} < 2500} (\Pi_{\text{balance}} (\text{account})) \leftrightarrow \Pi_{\text{balance}} (\sigma_{\text{balance} < 2500} (\text{account}))$
   - Any relational-algebra expression can be evaluated in many ways. An annotated expression specifying detailed evaluation strategy is called an execution plan (or evaluation plan).
     - e.g. can use an index on balance to find accounts with balance < 2500, or can perform complete relation scan and discard accounts with balance $\geq 2500$
   - Amongst all equivalent expressions, try to choose the one with cheapest possible evaluation-plan.
   - Cost estimate of a plan based on statistical information in the DBMS catalog.

3. Evaluation
   - The query-execution engine takes a query-execution plan, executes that plan, and returns the answers to the query.
Query Processing Steps

• Query
• PARSER (parsing and semantic checking as in any compiler)
• Parse tree (~ tuple calculus expression)
• OPTIMIZER (very advanced)
• Execution plan (annotated relation algebra expression)
• EXECUTOR (execution plan interpreter)
• Query result
Query Optimizer and Execution Steps

- Tuple calculus

- **VIEW EXPANSION**
  - Tuple calculus

- **REWRITES**
  - Tuple calculus

- **COST-BASED QUERY OPTIMIZATION**
  - Execution plan using *annotated (physical) relational algebra* (evaluation *primitive* in book)
Cost-based query optimization

- Cost-based query optimization:
  1. Generate all possible execution plans (heuristics to avoid some unlikely ones)
  2. Estimate the cost of executing each of the generated plans
  3. Choose the cheapest one
- Optimization criteria
  - # of disk blocks read (dominates)
  - CPU usage
  - disk access time (increasingly dominates)
  - disk transfer rate
  - normally weighted average of different criteria.
The query processing problem

- Transform:
  - High-Level Declarative Query --> Low-Level *Execution Plan*
  - Normally: Relational Calculus --> Annotated *Physical Relational Algebra*
  - *(evaluation primitives in book)*

- The execution plan is a (functional) program that is interpreted by the *evaluation engine* to produce the query result

- Problem:
  - For every query there may be very many possible execution plans:
    \( O(2^{|Q|}) \) where \(|Q|\) is number of operations in query

- The optimal plan can be millions of times faster than an unoptimized plan!
  - The complexity of optimal plan improved.
  - E.g from \( O(N^2) \) to \( O(1) \), where \( N \) is size of database!

- Query optimization may have huge payoff!
  - However: Query optimization time may be significant!
The query processing problem

- Degrees of freedom:
  - Query plan must be *efficient* and *correct*

- Choice of:
  - scan tuples
  - traverse index
  - order of joining tables
  - algorithms used for join
  - available main memory
  - materialization of intermediate results
  - pipelining intermediated results (streaming)
  - sort intermediate results
  - duplicate elimination
Basic algorithms for executing query operations

- Database operations can usually be executed using several different algorithms.
  - sorting algorithms (external sorting)
  - selection (search methods for simple selection - file/index scans):
    - tuple scan - linear search (brute force)
    - index scan
      - using primary index or hash key to retrieve a single record
      - using primary index for >, <, >=, <= conditions
      - using clustering index for non-key attribute
      - B+ tree index on key, non-key, >, <, >=, <= conditions
  - complex selection using individual, composite, or intersecting indexes
  - join implementation methods:
    - nested loop (brute force).
    - sort-merge join - two ordered relations are scanned once in order to find matching tuples.
    - hash-join - records from each relation are hashed by one hash function to the same file.
  - algorithms for project, union, intersection, difference, cartesian product
  - duplicate elimination
Query execution plan

• Query execution plan is a functional program with evaluation primitives:
  – Tuple scan operator
  – Tuple selection operator
  – Various index scan operators
  – Various join algorithms
  – Sort operator
  – Duplicate elimination operator
  
• Normally pipelined execution
  – Streams of tuples produced as intermediate results
  – Intermediate results can sometimes be materialized too
Measures of query cost

• Optimally a cost formula include weighted cost factors that should reflect the true cost for different database operations.
• In many situations it can be hard to come up with correct weights and only a single factor is considered.
• Access cost to secondary storage.
  – large databases
  – typically disk access is the predominant cost, and is also relatively easy to estimate.
  – therefore number of block transfers from disk is used as a measure of the actual cost of evaluation.
• Computation cost
  – small databases
• Communication cost
  – distributed and parallel databases
• Storage cost for intermediate results
Query cost models

• Basic costs parameters
  – Cost of accessing disk block
  – Data transfer rates
  – Clustering of data tuples on disk
  – Sort order of data tuples on disk

• Cost models of basic evaluation primitives
  – Cost of scanning disk segment containing tuples
  – Performance models for different index access methods (tree structures - hashing)
  – Performance models for different join methods
  – Cost of sorting intermediate results

• Total cost of an execution plan
  – The total cost depends on frequency of primitive operations invoked.
  – The frequency of invocation of primitive operations in execution plans depends on size of intermediate results.
  – Intermediate results estimated by statistical models.
Data statistics

• Used statistics to estimate size of intermediate results:
  – Size of tables
  – Number of different column values
  – Histogram of distributions of column values
  – Models for estimating sizes of selections
    • E.g. selectivity of PNR=xxxx, AGE>xxx, etc.
  – Models for estimating sizes of results from joins
• The models are often very rough
  – Work rather well since models used only for comparing different execution strategies - not for getting the exact execution costs.
• Cost of maintaining data statistics
  – Cheap: e.g size of relation, depth of B-tree.
  – Expensive: e.g. distribution on non-indexed columns, histograms
• Occentional statistics updates when load is low
  – Statistics not always up-to-date
  – Wrong statistics -> sub-optimal but correct plans
Catalog info for cost estimation

- number of tuples (r) in a relation R. (compare card(R))
- number of blocks (b) containing tuples of R.
- size of a tuple of R in bytes.
- the *blocking factor* (bfr) of R, i.e. the number of tuples of R that fit into one block.
- no of index levels (x) for multi-level indexes (or log_{d/r})
- no. of distinct values (d) that appear in R for attribute A.
- attribute selectivity (sl)
- *selection cardinality* (s = sl * r) of attribute A of relation R, i.e. the average number of records that satisfy equality on A.
  - for key attribute s = 1.
  - for non-key attribute s = r/d (r *1/d is an approximation)
Complexity of optimizer

- Standard ‘System R’ cost-based optimization method
  - NP hard in general (O(2^|Q|) where |Q| is size of query)
  - Dynamic programming (semi-exhaustive search) sometimes gives O(|Q|^2) in best case.
  - Works well up to ca 8 joins
- Heuristic methods O(|Q|^2)
  - Not optimal plan
- Randomized and genetic methods O(|Q|^2)
  - Converges to optimal plan
- Optimization timing
  - Static
    - Canned queries by SQL preprocessor
  - Dynamic
    - Optimize for every query
  - Hybride
    - Optimize at program startup (prepare in ODBC/JDBC)
    - Choose among several precompiled plans
Optimizing large queries

- Don’t optimize at all, i.e. order of predicates significant (old Oracle)
- Optimize partly, i.e. up to ca 8 joins, leave rest unoptimized (new Oracle)
- Heuristic methods (Ingres)
- Randomized (Monte Carlo) methods (research papers)
- Hybride methods, mix dynamic programming, heuristic, randomized
- User breaks down large queries to many small queries manually (often necessary for translating relational representations to complex object structures in application programs)
Clustered index

- The data file (lowest level) is sorted according to the indexing field.
Unclustered index

- The data file (lowest level) is not sorted according to the indexing field.
Nested-loop join $T \Join^{NLJ} U$

- Nested-loop join traverses one operand stream and for each row seeks a matching row in the second stream

```c
{
    Stream T, U, R;
    Tuple t, u;

    open(R, 'o');
    open(T, 'i');
    while (not(eof(T)))
    {
        t = next(T);
        open(U, 'i');
        while (not(eof(U))
        {
            u = next(U);
            if(t.K == u.K) emit(t + u, R);
        }
        close(U);
    }
    close(T);
    close(R);
}
Sort-merge join traverses the two operand streams simultaneously while matching a row from the first stream with another row in the second stream. (both T and U are assumed to be sorted on the join attributes.)

```plaintext
{  Stream T, U, R;
    Tuple t, u;
    open(R,'o'); // Öppna resultatström R
    open(T,'i'); // Öppna 1:a inströmmen
    open(U,'i'); // Öppna 2:a inströmmen
    t = next(T); // läs 1:a raden till t
    u = next(U); // detsamma för u
    while (not(eof(T)) and not(eof(U))) // så länge båda inströmmarna
      // har mer data
        {
          if(t.k > u.k) u = next(U); // om nyckelfältet mindre i u så
            // flytta fram U
          else if(t.k < u.k) t = next(T); // detsamma för T
          else emit(t + u, R); // skicka t och u konkaterade som
            // nästa resultatrad
        }
    close(T);
    close(U);
    close(R); // stäng alla strömmar
}```
Hash join

- Hash join the operand stream $T$ are read into the buffer and a hash table is created on its hash field (join attribute), then for each row $i$ of $U$ any matching row in $T$ is retrieved by applying the same hash function on $T$'s hash table. (assumes the complete hash table fits in memory - otherwise partitioned hash join)

```c
Stream T, U, R;
Tuple t, u;
HashTable h;

open(R, 'o');
h = createHashTable();
open(T, 'i');
while (not(eof(T)))
{
    t = next(T);
    putHash(h, t.k, t);
}
open(U, 'i');
while (not(eof(U))
{
    u = next(U);
    tm = getHash(ht, u.k);
    if(tm != NULL) emit(tm + u, R);
}
close(U);
close(T);
close(R);
```
Cost for various methods

- Examples of cost functions for SELECT:
  - Tuple scan (linear search)
    \[ C_{LS} = b \] (worst case)
  - Hash key
    \[ C_H \approx 1 \] (2 for extendible hashing)
  - Secondary B+ tree index (= condition results in s records)
    \[ C_{B+} = x + s \] (non-clustering)
    \[ C_{B+} = x + 1 \] (key indexing attribute)
  - Primary index to retrieve single record
    \[ C_{PI1} = x + 1 \]

- Examples of cost functions for JOIN:
  - See slides for different join methods including examples