Query processing

Tore Risch
Information Technology
Uppsala University
2011-03-08

What is query processing?

- A given SQL query is translated by the *query processor* into a low level program called an *execution plan*
- An execution plan is a program in a functional language:
  - The *physical relational algebra*, specialized for internal storage representation in the DBMS.
- The physical relational algebra extends the relational algebra with:
  - Primitives to search through the internal storage structures of the DBMS

What is query optimization?

- SQL is a very high level language:
  - The users specify *what* to search for – not *how* the search is actually done
  - The algorithms are chosen automatically by the DBMS
- For a given SQL query there may be very many possible execution plans
- The cost of these execution plans very widely
  - The costs vary with orders of magnitude
- The task of the *query optimizer* is to choose the cheapest plan out of the possible ones
- The query optimizer is the most complex (and important) part of the query processor

Complexity of query optimization

- Very many possible execution plans for a given SQL query, e.g.:
  - J joins can be permutated with different costs O(J!)
  - In addition there are different join algorithms to choose from
- Query optimization combinatorical over # of operations |Q| in query. With ‘dynamic programming’ $O(|Q|^2)$ in best case

Why does query optimization pay off?

- Query optimization radically improves speed of executing query
  - The complexity of a good plan may be O(log N), while a bad one is $O(N^2)$, where N is *size of the database*
  - Query optimization enables *scalability* of declarative queries
- Since N is large the payoff is huge
  - The size of the query |Q| << N
- Classical query optimization can handle up to ca 12 joins
- Good query optimizer critical for competitive DBMS!
- Query optimization is the key to the success of SQL

Query Processing Steps

SQL Query

PARSER (parsing and semantic checking as in any compiler)

Parse tree (~ tree structure representing relational calculus expression)

OPTIMIZER (very advanced)

Execution plan (physical relation algebra expression)

EXECUTOR (execution plan interpreter)

DBMS kernel
Data structures

Query Optimizer Steps

Tuple calculus

View expansion

Tuple calculus

Query transformations

Tuple calculus

Cost-based query optimization

Physical relational algebra

View expansion

- A view is a virtual table expressed through a single SQL statement, a *named query*
- Views are textually substituted (macro expanded) by view expansion
- View expansion makes query larger (and thus optimization slower)
- View expansion allows optimizer to look inside view definitions rather than regarding views as black boxes
- View expansion required in order to detect hidden indexes

View expansion

```
CREATE TABLE SUPPLIES(
    STORE CHAR(10),
    ITEM CHAR(10),
    PRICE DECIMAL(10,2),
    PRIMARY KEY(STORE, ITEM))

CREATE VIEW ICASUPPLIES AS
    SELECT *
    FROM SUPPLIES
    WHERE STORE = ‘ICA’
```

View expansion

SELECT PRICE

FROM ICASUPPLIES S

WHERE S.ITEM = 'Tomatoes'

Translated by the view expander into:

SELECT PRICE

FROM SUPPLIES S

WHERE S.ITEM = 'Tomatoes'

AND S.STORE = 'ICA'

Query optimizer will now discover index(es) on ITEM or STORE!

These indexes would NOT have been discovered if view was black box.

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 using a *cost model* based on database statistics and properties of DBMS algorithms
  3. Choose the cheapest one
- Optimization criteria
  - # of disk blocks read (dominates), DB
  - CPU usage, CP
  - Communication costs for distributed data, CO
  - Normally weighted average of different criteria:

    $W_1 * DB + W_2 * CP + W_3 * CO$

    W_1, W_2, W_3 system configured weights
- The costs are computed based on *data statistics* and *cost model* for operators in physical relational algebra

Query execution plan (physical algebra)

- Query execution plan is functional program with evaluation primitives:
  - Table scan operator
  - Primary index access operators (index kind dependent)
  - Index scan operators (index kind dependent)
  - Various join algorithms
  - Sort operator
  - Duplicate elimination operator
  - .....
- Normally *pipelined* execution
  - Streams of tuples produced as intermediate results
  - Intermediate results can sometimes be materialized as temporary tables

Degrees of freedom for optimizer

- Query plan must be efficient and correct
- Choice of physical operators, e.g.:
  - Scan table sequentially
  - Traverse index structure (e.g. B-tree, hash table)
  - Choose order of joining tables
  - Choose algorithms used for each join
  - Adapt to available main memory
  - Materialize intermediate results if favourable
  - Eliminate duplicates in stream
  - Sort intermediate results

Query Cost Model

- Basic costs parameters
  - Cost of accessing disk block randomly
  - Data transfer rate
  - Clustering of data tuples on disk
  - Sort order of data tuples on disk
  - Cost of scanning disk segment containing tuples
  - Cost models for different index access methods (tree structures - hashing)
  - Cost models for different join methods
  - Cost of sorting intermediate results
- Total cost of an execution plan
  - The total cost depends on how often primitive operations are invoked.
  - The invocation frequency depends on size of intermediate results.
  - Intermediate results are estimated by statistics computed over data stored in database.

Selectivity

- *Selectivity* important for estimating size of (intermediate) query result
- Example join of relations T1(ssn,name), T2(ssn,income):

select t2.income from T1 t1, T2 t2
where t1.name = "Kalle" and
t2.income > 95000 and
t1.pnr = t2.pnr

Assume index on T1.pnr, T2.pnr, T1.name, and T2.income!

If T2.income is more *selective* than T1.name then join on PNR(select(T2.INCOME>95000),(select T1.PNR=t2.PNR and T1.NAME="Kalle")),

otherwise join on PNR(select(T1.name="Kalle"),select(T2.PNR=t1.PNR and T2.INCOME>95000))

- *Selectivitity(P(t))* defined as percentage of tuples *t* that are selected by predicate *P(t)*.
  - Selectivity(t2.income>95000) depends on value distributions in column T2.income.
    - DBMS maintains this, e.g. number of rows in table, number of different values, highest and lowest value, even histogram. Regular statistics refresh can be done.

    Assume: highest income=100000, lowest income=15000.

    Then selectivity(t2.income>95000) can be estimated to (100000-95000)/(100000-15000)=0.058

    Assume: 100 rows in T1, but only 80 different T1.name

    Then selectivity(t1.name="Kalle") can be estimated to 1/80=0.0125.

    => join(select(T1.name="Kalle"),T2) cheapest
  - Above calculations assume flat value distributions (classical)
  - Modern DBMSs maintain histograms
  - Some statisticts incrementally maintained (e.g. size of tables, indexed join selectivities)
  - Update statistics for table command to update some statistics

Data statistics

- Statistics used to estimate size of intermediate results:
  - Size of tables
  - Number of different values in column
  - Histogram of distributions of column values
  - Model for estimating how selective a predicate is, its *selectivity*:
    - E.g. selectivity of PNR=xxxx, AGE>xxx, etc.
  - Model for estimating sizes of intermediate 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
  - Occasional statistics updates when load is low
- Statistics not always up-to-date
  - Wrong statistics -> sub-optimal but correct plans

Optimizing large queries

- Don’t optimize at all or partly, i.e. order of predicates significant (old Oracle, old MySQL)
- Optimize partly, i.e. up to ca 10 joins, leave rest unoptimized (new Oracle)
- Heuristic methods (e.g. greedy optimization)
- Randomized (Monte Carlo) methods
- To speed up data access the user may sometimes manually break down very large queries into smaller optimizable queries
  - This is often necessary for translating relational representations to complex object structures in application programs