Query compilation I

March 13, 2003

Based on GUW 16.2-16.3

Advanced Database Technology, Spring 2003
Implementation of relational operations

Stages in query processing:

1. Query compilation – this lecture and next week (Chapter 16)
   - Parse (SQL) query to a query expression tree. (⊂ curriculum)
   - Select a logical query plan, expressing the query in relational algebra.
   - Select a physical query plan, i.e., particular algorithms and an ordering for the relational operations. (Next week)

2. Query execution (Chapter 15) – earlier today
   - Several possible algorithms for relational algebra operations.
   - The best algorithm depends on the particular relations involved, and on the internal memory available.
Parsing

- **Parsing** is the process of transforming a string into its derivation by a grammar.

- The result is usually represented as a parse tree.

- In the book, parse trees for SQL are called query expression trees.

Grammars and parsing is a major part of a course on compilers, and out of scope for this course. We will assume that:

  - We have a parse tree of the query (such as the ones on GUW page 792-793).
  
  - All operations are **semantically valid** (for example, do not use nonexisting relations or attributes).
Basic ideas:

- A transformation rule for each syntactical construct.
- Rules generally involve the logical query plans of subexpressions (recursive processing).

Main example: SELECT-FROM-WHERE

- Suppose we have the parse tree for the expression
  \[ E = \text{SELECT } A_1, \ldots, A_n \text{ FROM } E_1, \ldots, E_m \text{ WHERE } C. \]
- The corresponding algebraic expression \( \mathcal{A}(E) \) is
  \[ \pi_{A_1, \ldots, A_n} (\sigma_C (\mathcal{A}(E_1) \times \ldots \mathcal{A}(E_m))). \]
- At all times we work with trees (parse trees and algebraic expression trees) rather than the textual representation.
Subqueries in conditions

Missing detail:

- In SQL, conditions might involve subqueries, e.g., the computation of some value that an attribute must be compared to.
- The subquery may be **correlated**, meaning that its result depends on the tuple being looked at (e.g., “does the value in attribute $A$ exist in relation $R$.”)
- Correlated subqueries must in general be evaluated for every tuple (though it is often possible to do better).
- **Uncorrelated** queries just need to be evaluated once, before performing the `SELECT-FROM-WHERE`.
- We need to extend basic relational algebra to express subqueries in conditions – however, in most cases it is possible to rewrite to basic relational algebra (examples in GUW 16.3.2).
Algebraic laws

Recall the first exercises on laws in relational algebra, such as:

Commutativity:

- \( R \times S = S \times R \)
- \( R \bowtie S = S \bowtie R \)
- \( R \cup S = S \cup R \)
- \( R \cap S = S \cap R \)

Associativity:

- \( (R \times S) \times T = R \times (S \times T) \)
- \( (R \bowtie S) \bowtie T = R \bowtie (S \bowtie T) \)
- \( (R \cup S) \cup T = R \cup (S \cup T) \)
- \( (R \cap S) \cap T = R \cap (S \cap T) \)
Algebraic laws for improving query plans

Selecting a good algebraic expression

- There may be many algebraic expressions that evaluate to what we want.
- Though they are all equal, some are more equal than others!
- We want an expression that is likely to result in short computation time.

Algebraic laws give us a way of rewriting the expression produced from the parse tree in order to improve it.
Problem session: Useful algebraic laws

For each of the following algebraic laws, consider whether it might be useful for rewriting an algebraic expression to have smaller computation time:

1. \( \sigma_C(E_1 \cup E_2) = \sigma_C(E_1) \cup \sigma_C(E_2) \).
2. \( \sigma_C(E_1 - E_2) = \sigma_C(E_1) - E_2 \).
3. \( \sigma_C(E_1 - E_2) = \sigma_C(E_1) - \sigma_C(E_2) \).
4. \( \sigma_C(E_1 \times E_2) = \sigma_C(E_1) \bowtie E_2 \) if \( E_1 \) has all attributes in \( C \).
5. \( \sigma_C(E_1 \cap E_2) = \sigma_C(E_1) \cap \sigma_C(E_2) \) if \( E_1 \) and \( E_2 \) have all attributes in \( C \).
6. \( \pi_L(E_1 \bowtie E_2) = \pi_L(\pi_{(L \cup A_{E_2}) \cap A_{E_1}}(E_1) \bowtie \pi_{(L \cup A_{E_1}) \cap A_{E_2}}(E_2)) \).
7. \( \pi_L(\sigma_C(E_1)) = \pi_L(\sigma_C(\pi_A(E_1))) \) where \( A = \) attributes mentioned in \( C \).
8. \( \delta(E_1 \bowtie E_2) = \delta(E_1) \bowtie \delta(E_2) \).
More laws...

Some other laws used:

- Splitting laws like $\sigma_{C_1 \text{AND} C_2}(E) = \sigma_{C_1} (\sigma_{C_2}(E))$ for simplifying the parts of the expression.

- Laws for pushing the selection operator **up** the tree, before pushing it down (in more subexpressions than otherwise).

- Laws for special cases of the aggregation operator.

- Grouping of associative operators, e.g., $(E_1 \bowtie E_2) \bowtie (E_3 \bowtie E_4)$ becomes simply $E_1 \bowtie E_2 \bowtie E_3 \bowtie E_4$. (This is to indicate that the order of operations may be chosen freely.)
Next week we will consider how to transform the algebraic expression tree into an efficient **physical query plan**, indicating what algorithms are to be used for the operations, and in which order.

According to the book, one usually **first** chooses an algebraic expression and **then** tries to find the best physical query plan based on that expression.