Improving Query Plans

CS157B
Chris Pollett
Mar. 21, 2005.

Outline

- Parse Trees and Grammars
- Algebraic Laws for Improving Query Plans
- From Parse Trees To Logical Query Plans

Syntax Analysis and Parse Trees

- The job of the parser is to take SQL and convert it to a parse tree.
- Nodes in this tree can be of the following types:
  - Atoms -- keywords (like SELECT), names of attributes, constants, parentheses, operators. These are leaves in the tree
  - Syntactic Categories - names for families of query subpart. For example, <Condition> might represent a possible condition part in the query.

A Grammar for a Simple Subset of SQL

- Here are some example rules a parser might use for SQL:

<Query> ::= <SFW>

<Query> ::= (<Query>) // '::=' should be read as 'can be expressed as'

<SWF> ::= SELECT <SelList> FROM <FromList> WHERE <Condition>

<SelList> ::= <Attribute>, <SelList>

<SelList> ::= <Attribute>

<FromList> ::= <Relation>, <FromList>

<FromList> ::= <Relation>

<Condition> ::= <Condition> AND <Condition>

<Condition> ::= <Tuple> IN <Query>

<Condition> ::= <Attribute> = <Attribute>

<Condition> ::= <Attribute> LIKE <Pattern> //etc….

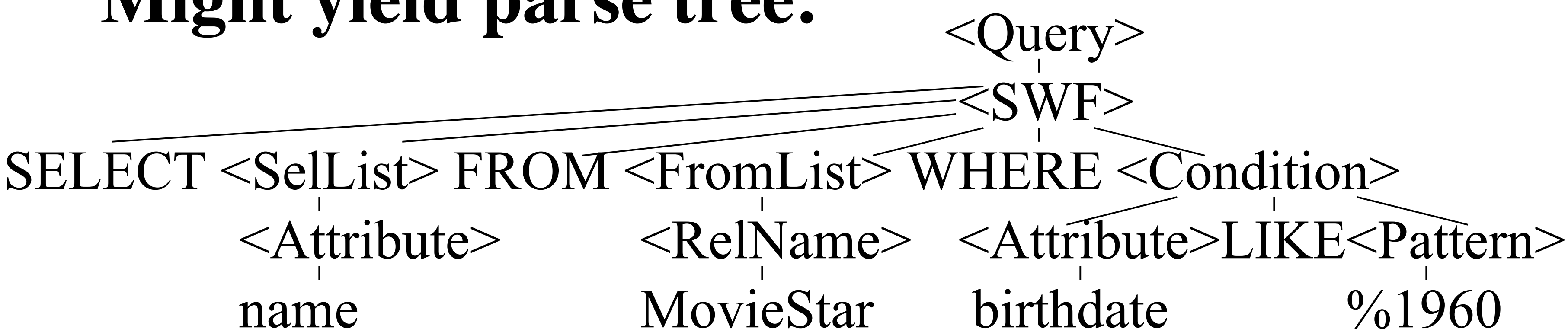
Example

SELECT name

from MovieStar

WHERE birthdate LIKE ‘%1960’

Might yield parse tree:



The Preprocessor

- The preprocessor does semantic checking:
  - It expands views to their corresponding query
  - It checks that the items in the FROM clause are actual tables in the DB
  - It checks that the attributes in the SELECT clause belong to some table and matches them to the correct table.
  - It checks that attributes in the WHERE clause are being put in conditions that are appropriate for their type.

Algebraic Laws For Improving Query Plans

- We are now going to consider different ways we can write the same query. The hope being some ways of writing are faster to process than others.
- For instance, it is probably faster to compute selects before taking a join, then vice versa.
- The optimizer's job will be to consider several plans and usually heuristically choose the best one.

Commutative and Associative Laws

- R x S = S x R; (R x S) x T = R x (S x T)
- R join S = S join R; (R join S) join T =R join (S join T)
- $R \cup S = S \cup T$; $(R \cup S) \cup T = R \cup (S \cup T)$
- $R \cap S = R \cap S$; $(R \cap S0 \cap T = R \cap (S \cap T)$
- Notice the intermediate tables in computing a sequence of join can be considerable smaller depending on the order of execution.

Laws Involving Selection

- $\sigma_{C1 \text{ AND } C2}(R) = \sigma_{C1}(\sigma_{C2}(R)) = \sigma_{C2}(\sigma_{C1}(R))$
- $\sigma_{C1 \text{ OR } C2}(R) = \sigma_{C1}(R) \cup \sigma_{C2}(R)$ provided R is a set.
- $\sigma_{C}(R \cup S) = \sigma_{C}(R) \cup \sigma_{C}(S)$
- $\sigma_{C}(R - S) = \sigma_{C}(R) - S$
- Similar rules hold for joins and cartesian products.

Pushing Selections

- To reduce the cost of computing joins and products as much as possible, a good heuristic is to try to push selects as far down all branches in the query tree as possible
- For instance, $\sigma_{year=1996}$(Movie join StarsIn) should be rewritten $\sigma_{year=1996}$(Movie) join $\sigma_{year=1996}$(StarsIn).

Laws Involving Projection

- π_L(R join S) = $\pi_L(\pi_M$(R) join π_N(S)) where M is those attributes in L that are from R; N is those attributes in L that are from S.
- This also works for joins with conditions on them.
- As with selections, it is often useful to try to push projections as far down the tree as possible.

Laws for Joins and Products

- We have already seen that it is possible to rewrite a join using:

  R $\text{join}_{\text{Condition}}$S = $\sigma_{\text{Condition}}$(R x S)
- A natural join can be written as

  R $\text{join}_{\text{Condition}}$S = $\pi_L(\sigma_C$ (R x S)) where L is the attributes from the natural join and C is the condition which matches like named attributed.

From Parse Trees to Logical Query Plans

- After calculating the parse tree for a query…
- We convert the appropriate groups in the tree by relational operators.
  - For example, we replace the <FromList> node in a <Query> tree with the cartesian product of the tables listed under it.
- Then try to optimize this relational algebra tree to create a physical query plan.
  - Might apply heuristics like pushing selection and projections which we mentioned earlier.