Query Processing/Optimization

CPS 216
Advanced Database Systems

Plan for today

- Overview of query processing
- Query execution
- Query plan enumeration
- Query rewrite heuristics
- Query rewrite in DB2

A query’s trip through the DBMS

SQL query

Parser

Parse tree

Validator

Logical plan

Optimizer

Physical plan

Executor

Result

SELECT title, SID
FROM Enroll, Course
WHERE Enroll.CID = Course.CID;

π title, SID
σ Enroll.CID = Course.CID
SCAN (Enroll)
SCAN (Course)
SORT (CID)
MERGE-JOIN (CID)
PROJECT (title, SID)
Parsing

- Parser: SQL → parse tree
  - Good old lex & yacc
  - Detect and reject syntax errors
- A short review of SQL
  - `SELECT Course.title` Step 3: \( \pi \)
  - `FROM Student, Enroll, Course` Step 1: \( \times \)
  - `WHERE Student.name = 'Bart'` Step 2: \( \sigma \)
  - AND `Student.SID = Enroll.SID`
  - AND `Enroll.CID = Course.CID`;
  - Subqueries, aggregates
  - Duplicates, NULLs

Validation

- Validator: parse tree → logical plan
- Detect and reject semantic errors
  - Nonexistent tables/views/columns?
  - Insufficient access privileges?
  - Type mismatches?
    - Examples: `AVG(name)`, `name + GPA`, `Student UNION Enroll`
- Also
  - Expand `*`
  - Expand view definitions
- Where does the validator get the information required for semantic checking?
  - System catalog (contains all metadata/schema information)

Logical plan

\[
\pi_{\text{title}} \sigma_{\text{Student.name} = 'Bart' \text{ AND } \text{Student.SID} = \text{Enroll.SID} \text{ AND } \text{Enroll.CID} = \text{Course.CID}} \times_{\text{Enroll}} \text{Course} \times_{\text{Student}} \text{Enroll}
\]

Another equivalent one:

Note: Not all systems use relational algebra to represent logical plans—DB2 uses QGM.
Physical plan

Even more physical plans!

- Equivalent semantics, but not costs or assumptions!
- Optimizer: one logical plan → “best” physical plan

Physical plan execution

- Executor: physical plan → result
  - Detect and report run-time errors
    - Example: scalar subquery returns multiple tuples
- Plan is a tree of operators
- How are intermediate results passed from children to parents?
  - Temporary files
    - Compute the tree bottom-up
    - Children write intermediate results to temporary files
    - Parents read temporary files
  - Iterator interface (next)

Iterator interface

- Every operator maintains its own execution state and implements the following methods:
  - open(): Initialize state and get ready for processing
  - getNext(): Return the next tuple in the result (or a null pointer if there are no more tuples); adjust state to allow subsequent tuples to be obtained
  - close(): Clean up
An iterator for table scan

- **open()**
  - Allocate buffer space

- **getNext()**
  - If no block of R has been read yet, read the first block from the disk and return the first tuple in the block (or the null pointer if R is empty)
  - If there is no more tuple left in the current block, read the next block of R from the disk and return the first tuple in the block (or the null pointer if there are no more blocks in R)
  - Return the next tuple in the block

- **close()**
  - Deallocone buffer space

An iterator for nested-loop join

- **open()**
  - R.open(); S.open();
  - r = R.getNext();

- **getNext()**
  - Repeat until r and s join:
    - s = S.getNext();
    - if (s == null) { S.close(); S.open(); s = S.getNext();
      - if (s == null) return null;
      - r = R.getNext();
      - if (r == null) return null;}
  - return rs;

- **close()**
  - R.close(); S.close();

Execution of an iterator tree

- Call root.open(), root.getNext() (repeat until it returns a null pointer, and root.close() )

- Requests go down the tree

- Intermediate result tuples go up the tree

- No intermediate files are needed!
  - But still useful when an iterator is opened many times
    - Example:
Back to query optimization

- One logical plan → “best” physical plan
- Why bother?
  - The difference in cost can be huge

\[
\pi_{\text{title}} \sigma_{\text{Student.name} = 'Bart'} \text{AND} \text{Student.SID = Enroll.SID AND Enroll.CID = Course.CID} \times \text{Enroll} \times \text{Course} \times \text{Student} \\
\pi_{\text{title}} \sigma_{\text{Student.name} = 'Bart'} \text{Student.SID = Enroll.SID} \text{Enroll} \\
\sigma_{\text{Student.name} = 'Bart'} \text{Course}
\]

Query optimization!

- Conceptually
  - Enumerate all possible plans (coming right up)
  - Estimate costs (next week)
  - Pick the “best” one (next week)
- Often the goal is not getting the optimum plan, but instead avoiding the horrible ones

Any of these will do

1 second 1 minute 1 hour

Plan enumeration in relational algebra

Apply relation algebra equivalences

- $\times$ and $\sigma$ are associative and commutative
  - Except column ordering, but that is easy to fix
  - Join reordering

\[
\begin{align*}
R \times S &= S \times R \\
R \sigma (T) &= T \sigma (R) \\
R \sigma (S) &= S \sigma (R)
\end{align*}
\]
More relational algebra equivalences

- Convert $\sigma_{p \times} \land \sigma_{q} (R \times S)$ to/from $\sigma_{q} (R \times S)$
- Merge/split $\sigma$'s: $\sigma_{p}(\sigma_{q}(R)) = \sigma_{p \land q}(R)$
- Merge/split $\pi$'s: $\pi_{L_{1}}(\pi_{L_{2}}(R)) = \pi_{L_{1}}(R), \text{ where } L_{1} \subseteq L_{2}$
- Push down/pull up $\sigma$:
  - $\sigma_{p \land q}(R) \land \sigma_{q}(S)$, where $\sigma_{p}$ is a predicate with only $R$ attributes
  - $\sigma_{q}$ is a predicate with only $S$ attributes
  - $p$ is a predicate with $R, S$ attributes
- Push down $\pi$: $\pi_{L}(\sigma_{p}(R)) = \pi_{L}(\sigma_{p}(\pi_{L'}(R))), \text{ where } L'$ is the set of attributes referenced by $p$ that are not in $L$
- Many more (seemingly trivial) equivalences…
  - Can be systematically used to transform a plan to new one

Transformation Example

Too many plans!

- Use heuristics
  - Push selections and projections down as much as possible
    - Why?
    - Why not?
  - Join smaller relations first, and avoid cross product
    - Why?
    - Why not?
- Rigorous cost-based approach (next week)
Problem with SQL

- Not exactly relational algebra—enumerating plans is not simple
- Subqueries and views naturally divide a query into nested “blocks”
  - Processing each block separately forces particular join methods and join order
  - Even if the plan is optimal for each block, it may not be optimal for the entire query
  ➢ Unnest query: convert subqueries/views to joins
    - We know how to deal with select-project-join queries

DB2’s QGM

- Query Graph Model: DB2’s logical plan language
  - More high-level than relational algebra
- A graph of boxes
  - Leaf boxes are tables
  - The standard box is the SELECT box (actually a select-project-join query block with optional duplicate elimination)
  - Other types include GROUPBY (aggregation), UNION, INTERSECT, EXCEPT
  - Can always add new types (e.g., OUTERJOIN)

More on QGM boxes

- Head: declarative description of the output
  - Schema: list of output columns
  - Property: Are output tuples DISTINCT?
- Body: how to compute the output
  - Quantifiers: tuple variables that range over other boxes
    - F: regular tuple variable, e.g., FROM R AS r
    - E: existential quantifier, e.g., r IN (subquery), or r = ANY (subquery)
    - A: existential quantifier, e.g., r > ALL (subquery)
    - S: scalar subquery, e.g., r = (subquery)
  - Quantifiers are connected a hypergraph
    - Hyperedges are predicates
    - Enforce DISTINCT, preserve duplicates, or permit duplicates?
      - For the output of this box, and for each quantifier
Query rewrite in DB2

- Goal: make the logical plan as general as possible, i.e., merge boxes
- Rule-based transformations on QGM (Leung et al., in red book)
  - Merge subqueries in FROM
  - Convert E to F (e.g., IN/ANY subqueries to joins)
  - Intersect to join
  - Convert S to F (i.e., scalar subqueries to joins)
  - Convert outerjoin to join
  - Magic (i.e., correlated subqueries to joins)

E to F conversion

- SELECT DISTINCT name
  FROM Student
  WHERE SID = 
    ANY (SELECT SID FROM Enroll);

- SELECT DISTINCT name
  FROM Student, (SELECT SID FROM Enroll) t
  WHERE Student.SID = t.SID;
  (EtoF rule)

- SELECT DISTINCT name
  FROM Student, Enroll
  WHERE Student.SID = Enroll.SID;
  (SELMERGE rule)

Problem with duplicates

Same query, without DISTINCT

- SELECT name
  FROM Student
  WHERE SID = 
    ANY (SELECT SID FROM Enroll);

- SELECT name
  FROM Student, Enroll
  WHERE Student.SID = Enroll.SID;

- Suppose…
A way of preserving duplicates

- SELECT name
  FROM Student
  WHERE SID =
    ANY (SELECT SID FROM Enroll);

- Suppose that SID is a key of Student

- SELECT DISTINCT Student.SID, name
  FROM Student, Enroll
  WHERE Student.SID = Enroll.SID;

  (ADDKEYS rule)

- Then simply project out Student.SID

Another E to F trick

- Sometimes an ANY subquery can be turned into an aggregate subquery without ANY

- SELECT * FROM Student s1
  WHERE GPA > ANY
    (SELECT GPA FROM Student s2
     WHERE s2.age > s1.age);

- SELECT * FROM Student s1
  WHERE GPA >
    (SELECT MIN(GPA) FROM Student s2
     WHERE s2.age > s1.age);

Does the same trick apply to ALL?

- SELECT * FROM Student s1
  WHERE GPA > ALL
    (SELECT GPA FROM Student s2
     WHERE s2.age < s1.age);

- SELECT * FROM Student s1
  WHERE GPA >
    (SELECT MAX(GPA) FROM Student s2
     WHERE s2.age < s1.age);

- Suppose…
Correlated subqueries

- SELECT CID FROM Course
  WHERE title LIKE 'CPS%'
  AND min_enroll > (SELECT COUNT(*) FROM Enroll
  WHERE Enroll.CID = Course.CID);

- Executing correlated subquery is expensive
  - The subquery is evaluated once for every CPS course

  ➢ Decorrelate!

COUNT bug

- SELECT CID FROM Course
  WHERE title LIKE 'CPS%'
  AND min_enroll > (SELECT COUNT(*) FROM Enroll
  WHERE Enroll.CID = Course.CID);

- SELECT CID
  FROM Course, (SELECT CID, COUNT(*) AS cnt
  FROM Enroll GROUP BY CID) t
  WHERE t.CID = Course.CID
  AND min_enroll > t.cnt;

- Suppose…

Magic decorrelation

- Simple idea
  - Process the outer query using other predicates
    • To collect bindings for correlated variables in the subquery
  - Evaluate the subquery using the bindings collected
    • It is a join
    • Once for the entire set of bindings
      - Compared to once per binding in the naïve approach
  - Use the result of the subquery to refine the outer query
    • Another join

- Name “magic” comes from a technique in recursive processing of Datalog queries
Magic example

- Original query
  - SELECT CID FROM Course
    WHERE title LIKE 'CPS%'
    AND min_enroll > (SELECT COUNT(*) FROM Enroll
    WHERE Enroll.CID = Course.CID);

- Process the outer query without the subquery
  - CREATE VIEW Supp_Course AS
    SELECT * FROM Course WHERE title LIKE 'CPS%';

- Collect bindings
  - CREATE VIEW Magic AS
    SELECT DISTINCT CID FROM Supp_Course;

- Evaluate the subquery with bindings
  - CREATE VIEW DS AS
    SELECT Enroll.CID, COUNT(*) AS cnt
    FROM Magic, Enroll WHERE Magic.CID = Enroll.CID
    GROUP BY Enroll.CID;
    UNION
    SELECT Enroll.CID, 0 AS cnt (the COUNT patch)
    FROM Enroll
    WHERE Enroll.CID NOT IN (SELECT CID FROM Magic);

- Finally, refine the outer query
  - SELECT Supp_Course.CID FROM Supp_Course, DS
    WHERE Supp_Course.CID = DS.CID
    AND min_enroll > DS.cnt;

Summary of query rewrite

- Break the artificial boundary between queries and subqueries
- Combine as many query blocks as possible in a select-project-join block, where clean rules of relational algebra apply
- Extremely tricky stuff with duplicates, NULLs, empty tables, and correlation
- Next step
  - Cost-based (Tuesday) optimization (Thursday) on each select-project-join block