Query Optimization I

CPS 216
Advanced Database Systems

A query’s trip through the DBMS

Parsing & validation

- Parser: SQL → parse tree
  - Good old lex & yacc
  - Detect and reject syntax errors
- Validator: parse tree → logical plan
  - Detect and reject semantic errors
    - Nonexistent tables/views/columns
    - Type mismatches (e.g., AVG(name), name + GPA, Student UNION Enroll)
    - Wildcard (SELECT *) and view expansion
    - Use information stored in system catalog tables (contains all metadata/schema information)

Logical plan

- A tree whose nodes are logical operators
  - Often a tree of relational algebra operators
  - DB2 uses QGM (Query Graph Model)
- There are many equivalent logical plans

Query optimization and execution

- Recall that a physical plan tells the DBMS query execution engine how to execute the query
  - One logical plan can have many possible physical plans (with equivalent results, but different costs and assumptions)
- Query optimizer: one logical plan → “best” physical plan
- Query execution engine: physical plan → results

Query optimization

- Conceptually
  - Consider a space of possible plans (next)
  - Estimate costs of plans in the search space (next Tuesday)
  - Search through the space for the “best” plan (next Thursday)
- Often the goal is not picking the absolute optimum, but instead avoiding the horrible ones

Any of these will do
Plan enumeration in relational algebra

- Apply relational algebra equivalences
  - Join reordering: \( \times \) and \( \triangleright \) are associative and commutative (except when column ordering is considered, but that is unimportant)

\[
\begin{align*}
R \times S &= T \\
S \times R &= T \\
R \times R &= T \\
S \times S &= T \\
R \times \cdots &= \
\end{align*}
\]

More relational algebra equivalences

- Convert \( \sigma_p \times \) to/from \( \times \)
  \[
  \sigma_p (R \times S) = R \times \sigma_S S
  \]

- Merge/split \( \sigma \)’s:
  \[
  \sigma_p_1 (\sigma_p_2 R) = \sigma_{p_1 \land p_2} R
  \]

- Merge/split \( \pi \)’s:
  \[
  \pi_{L_1} (\pi_{L_2} R) = \pi_{L_1 \cap L_2} R
  \]

- Push down/pull up \( \sigma \):
  \[
  \sigma_{p \land \sigma \land \sigma} (R \times S) = (\sigma_p R) \times (\sigma_S S)
  \]
  where
  - \( \sigma \) is a predicate involving only \( R \) columns
  - \( \sigma \) is a predicate involving only \( S \) columns
  - \( \sigma \) is a predicate involving both \( R \) and \( S \) columns

- Push down \( \pi \):
  \[
  \pi_{L_1} (\sigma_p R) = \pi_{L_1} (\sigma_p (\pi_{L_1} L'))
  \]
  where
  - \( L' \) is the set of columns referenced by \( \sigma \) that are not in \( L \)

- Many more (seemingly trivial) equivalences...
  - Can be systematically used to transform a plan to new ones

Relational query rewrite example

- More complicated—subqueries and views 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
  - Then we just deal with select-project-join queries
  - Where the clean rules of relational algebra apply

Heuristics-based query optimization

- Start with a logical plan
  - Push selections/projections down as much as possible
    - Why? Reduce the size of intermediate results
    - Why not? May be expensive; maybe joins filter better
  - Join smaller relations first, and avoid cross product
    - Why? Reduce the size of intermediate results
    - Why not? Size depends on join selectivity too
  - Convert the transformed logical plan to a physical plan (by choosing appropriate physical operators)

SQL query rewrite

- More complicated—subqueries and views 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
  - Then we just deal with select-project-join queries
  - Where the clean rules of relational algebra apply

DB2’s QGM

Leung et al. “Query Rewrite Optimization Rules in IBM DB2 Universal Database.”

- 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., \texttt{FROM R AS r}
    - E: existential quantifier, e.g., \texttt{IN (subquery)} or \texttt{ANY (subquery)}
    - A: universal quantifier, e.g., \texttt{> ALL (subquery)}
  - S: scalar subquery, e.g., \texttt{= (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
  - Merge subqueries in \texttt{FROM}
  - Convert E to F (e.g., \texttt{IN}/\texttt{ANY} subqueries to joins)
  - Convert intersection 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

- \texttt{SELECT DISTINCT name}
  \texttt{FROM Student}
  \texttt{WHERE SID = ANY (SELECT SID FROM Enroll)};
- \texttt{SELECT DISTINCT name}
  \texttt{FROM Student, (SELECT SID FROM Enroll) t}
  \texttt{WHERE Student.SID = t.SID};
  (EtoF rule)
- \texttt{SELECT DISTINCT name}
  \texttt{FROM Student, Enroll}
  \texttt{WHERE Student.SID = Enroll.SID};
  (SELMERGE rule)

Problem with duplicates

Same query, without \texttt{DISTINCT}
- \texttt{SELECT name}
  \texttt{FROM Student}
  \texttt{WHERE SID = ANY (SELECT SID FROM Enroll)};
- \texttt{SELECT name}
  \texttt{FROM Student, Enroll}
  \texttt{WHERE Student.SID = Enroll.SID};
- Suppose some student takes multiple classes
  - The first query returns name once; the second multiple times
  - Adding DISTINCT to the second query does not help
  - Suppose two students have the same name

A way of preserving duplicates

- \texttt{SELECT name}
  \texttt{FROM Student}
  \texttt{WHERE SID = ANY (SELECT SID FROM Enroll)};
  Suppose that \texttt{SID} is a key of \texttt{Student}
- \texttt{SELECT DISTINCT Student.SID, name}
  \texttt{FROM Student, Enroll}
  \texttt{WHERE Student.SID = Enroll.SID};
  (ADDKEYS rule)
- Then simply project out \texttt{Student.SID}
Another E to F trick

- Sometimes an \textit{ANY} subquery can be turned into an aggregate subquery without \textit{ANY}, to improve performance further

\begin{verbatim}
\begin{itemize}
  \item SELECT * FROM Student s1
  \hspace{1em} WHERE GPA > \texttt{ANY}
  \hspace{1em} (SELECT GPA FROM Student s2
  \hspace{2em} WHERE \texttt{s2.name} = 'Bart');
  \item SELECT * FROM Student s1
  \hspace{1em} WHERE GPA >
  \hspace{2em} (SELECT \texttt{MIN}(GPA) FROM Student s2
  \hspace{3em} WHERE \texttt{s2.name} = 'Bart');
\end{itemize}
\end{verbatim}

Does the same trick apply to \textit{ALL}?

- \begin{verbatim}
  \begin{itemize}
    \item SELECT * FROM Student s1
    \hspace{1em} WHERE GPA > \texttt{ALL}
    \hspace{2em} (SELECT GPA FROM Student s2
    \hspace{3em} WHERE \texttt{s2.name} = 'Bart');
    \item SELECT * FROM Student s1
    \hspace{1em} WHERE GPA >
    \hspace{2em} (SELECT \texttt{MAX}(GPA) FROM Student s2
    \hspace{3em} WHERE \texttt{s2.name} = 'Bart');
  \end{itemize}
\end{verbatim}

- Suppose there is no student named Bart
  - The first query returns all students; the second returns none

Correlated subqueries

- \begin{verbatim}
  \begin{itemize}
    \item SELECT CID FROM Course
    \hspace{1em} WHERE title LIKE 'CPS%'
    \hspace{2em} AND \texttt{min_enroll} >
    \hspace{3em} (SELECT \texttt{COUNT(*)} FROM Enroll
    \hspace{4em} WHERE \texttt{Enroll.CID} = Course.CID);
  \end{itemize}
\end{verbatim}

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

  - Decorrelate!

COUNT bug

- \begin{verbatim}
  \begin{itemize}
    \item SELECT CID FROM Course
    \hspace{1em} WHERE title LIKE 'CPS%'
    \hspace{2em} AND \texttt{min_enroll} >
    \hspace{3em} (SELECT \texttt{COUNT(*)} FROM Enroll
    \hspace{4em} WHERE \texttt{Enroll.CID} = Course.CID);
  \end{itemize}
\end{verbatim}

- Suppose a CPS class is empty
  - The first query returns this course; the second does not

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 decorrelation example

- \begin{verbatim}
  \begin{itemize}
    \item SELECT CID FROM Course
    \hspace{1em} WHERE title LIKE 'CPS%'
    \hspace{2em} AND \texttt{min_enroll} >
    \hspace{3em} (SELECT \texttt{COUNT(*)} FROM Enroll
    \hspace{4em} WHERE \texttt{Enroll.CID} = Course.CID);
    \item CREATE VIEW Supp_Course AS
    \hspace{1em} SELECT * FROM Course WHERE title LIKE 'CPS%';
    \item CREATE VIEW Magic AS
    \hspace{1em} SELECT DISTINCT CID FROM Supp_Course;
    \item CREATE VIEW DS AS
    \hspace{1em} (SELECT Enroll.CID, \texttt{COUNT(*)} AS \texttt{cnt}
    \hspace{2em} FROM Magic, Enroll
    \hspace{3em} WHERE Magic.CID = Enroll.CID
    \hspace{4em} GROUP BY Enroll.CID)
    \hspace{2em} UNION
    \hspace{3em} (SELECT Magic.CID, 0 AS \texttt{cnt}
    \hspace{4em} FROM Magic
    \hspace{5em} WHERE Magic.CID NOT IN (SELECT CID FROM Enroll));
    \item SELECT Supp_Course.CID FROM Supp_Course, DS
    \hspace{1em} WHERE Supp_Course.CID = DS.CID
    \hspace{2em} AND \texttt{min_enroll} > DS.cnt;
  \end{itemize}
\end{verbatim}

- Process the outer query without the subquery
- Collect bindings
- Evaluate the subquery with bindings
- Finally, refine the outer query
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 the clean rules of relational algebra apply
- Handle with care—extremely tricky with duplicates, NULL's, empty tables, and correlation