Announcements (April 8)

- Reading assignments for next week:
  - Selinger et al. “Access Path Selection in a Relational Database Management System.” SIGMOD 1979
- Homework #4 (short) will be assigned next Tuesday and due the following Tuesday
- Final exam in 18 days (Monday, April 26)

A query’s trip through the DBMS

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

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

Logical plan example:

```
SELECT Course.title
FROM Student, Enroll, Course
WHERE Student.name = 'Bart'
  AND Student.SID = Enroll.SID
  AND Enroll.CID = Course.CID;
```

A possible equivalent physical plan:

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

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)

Logical plan example:

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

A possible equivalent physical plan:

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

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

Plan enumeration in relational algebra

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

More relational algebra equivalences

- Convert $\sigma_p \times$ to/from $\bowtie_p$: $\sigma_p (R \times S) = R \bowtie_p S$
- Merge/split $\sigma$: $\sigma_p (\sigma_q R) = \sigma_{p \wedge q} R$
- Merge/split $\pi$: $\pi_{L_1} (\pi_{L_2} R) = \pi_{L_1} R$, where $L_1 \subseteq L_2$
- Push down/pull up $\sigma$: $\sigma_p (R \times S) = (\sigma_p R) \bowtie S$, where
  - $p$ is a predicate involving only $R$ columns
  - $p$ is a predicate involving only $S$ columns
  - $p$ is a predicate involving both $R$ and $S$ columns
- Push down $\pi$: $\pi_{L_1} (\sigma_p R) = \pi_{L_1} (\sigma_p (\pi_{L_2} R))$, where
  - $L$ is the set of columns referenced by $p$ that are not in $L$
- Many more (seemingly trivial) equivalences…
  - Can be systematically used to transform a plan to new ones

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)

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

SQL query rewrite
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., FROM AS
    - E: existential quantifier, e.g., IN (subquery), or = ANY (subquery)
    - A: universal quantifier, e.g., > ALL (subquery)
    - S: scalar subquery, e.g., = (subquery)
  - Quantifiers are connected a hypergraph
  - Enforce DISTINCT, preserve duplicates, or permit duplicates?
    - For the output of this box, and for each quantifier

QGM example

SELECT DISTINCT `q1.partno`, `q1.descr`, `q2.suppno`
FROM inventory `q1`, quotations `q2`
WHERE `q1.partno` = `q2.partno`
AND `q1.descr` = 'engine'
AND `q2.price` = ALL (
  SELECT `q3.price`
  FROM quotations `q3`
  WHERE `q2.partno` = `q3.partno`
);

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 FROM
  - Convert E to F (e.g., IN/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

- 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;
  (SELMERGE rule)

- 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

- 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, to improve performance further

- SELECT * FROM Student s1
  WHERE GPA > ANY
    (SELECT GPA FROM Student s2
     WHERE s2.name = 'Bart');

- SELECT * FROM Student s1
  WHERE GPA >
    (SELECT MIN(GPA) FROM Student s2
     WHERE s2.name = 'Bart');

Does the same trick apply to ALL?

- SELECT * FROM Student s1
  WHERE GPA > ALL
    (SELECT GPA FROM Student s2
     WHERE s2.name = 'Bart');

- SELECT * FROM Student s1
  WHERE GPA >
    (SELECT MAX(GPA) FROM Student s2
     WHERE s2.name = 'Bart');

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

Does the same trick apply to ALL?

- SELECT * FROM Student s1
  WHERE GPA > ALL
    (SELECT GPA FROM Student s2
     WHERE s2.name = 'Bart');

- SELECT * FROM Student s1
  WHERE GPA >
    (SELECT MAX(GPA) FROM Student s2
     WHERE s2.name = 'Bart');

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

Correlated subqueries

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

- First compute the enrollment for all courses
  - (SELECT COUNT(*) AS cnt
    FROM Enroll GROUP BY CID) t
  WHERE t.CID = Course.CID AND min_enroll > t.cnt
  AND title LIKE 'CPS';

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

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
  AND title LIKE 'CPS';

- 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 naive 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

- SELECT CID FROM Course
  WHERE title LIKE 'CPS%'
  AND min_enroll > (SELECT COUNT(*) FROM Enroll
  WHERE Enroll.CID = Course.CID);
- CREATE VIEW Supp_Course AS
  SELECT * FROM Course WHERE title LIKE 'CPS%';
- SELECT DISTINCT CID FROM Supp_Course;
- CREATE VIEW DS AS
  (SELECT Enroll.CID, COUNT(*) AS cnt
   FROM Magic, Enroll WHERE Magic.CID = Enroll.CID
   GROUP BY Enroll.CID) UNION
  (SELECT Magic.CID, 0 AS cnt FROM Magic
   WHERE Magic.CID NOT IN (SELECT CID FROM Enroll));
- 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 the clean rules of relational algebra apply
- Handle with care—extremely tricky with duplicates, NULL’s, empty tables, and correlation