Database Tuning and Physical Design: Basics of Query Execution

Spring 2018

School of Computer Science
University of Waterloo

Databases CS348

Basics of Query Execution

Goal

Develop a simple *relational calculator* that answers queries.

Considerations:

1. How is data *physically represented*?
2. How to *compute answers* to complex queries?
3. How are *intermediate results* managed?

How do we Execute Queries?

1. Parsing, typechecking, etc.
2. Relational Calculus (SQL) translated

   to *Relational Algebra*
3. Optimization:

   $\Rightarrow$ generates an efficient *query plan*

   $\Rightarrow$ uses statistics collected about the stored data
4. Plan execution:

   $\Rightarrow$ *access methods* to access stored relations

   $\Rightarrow$ *physical relational operators* to combine relations

Relational Algebra

Idea

Define a *set of operations* on the universe of finite relations...
...called a *RELATIONAL ALGEBRA*.

$$(\mathcal{U}; R_0, \ldots, R_k, \times, \sigma_\varphi, \pi_V, \cup, -)$$

Constants:

R_i: one for each relational scheme

Unary operators:

σ_φ: selection (keeps only tuples satisfying φ)

π_V: projection (keeps only attributes in V)

Binary operators:

$\times$: Cartesian product

$\cup$: union

$-$: set difference

Examples

Account				
Acnt#	Type	Balance	Bank	Branch
1234	CHK	$1000	TD	1
1235	SAV	$20000	TD	2
1236	CHK	$2500	CIBC	1
1237	CHK	$2500	Royal	5
2000	BUS	$10000	Royal	5
2001	BUS	$10000	TD	3

Bank	
Name	Address
TD	TD Centre
CIBC	CIBC Tower

Projection

Definition:

$$\pi_V(R) = \{(x_{i_1}, \ldots, x_{i_k}) : (x_1, \ldots, x_n) \in R, i_j \in V\}$$

where V is a set of column *numbers*.

Example:

$\pi_{\{\#1,\#2\}}(\text{Account}) =$

1234	CHK
1235	SAV
1236	CHK
1237	CHK
2000	BUS
2001	BUS

Selection

Definition:

$$\sigma_{\varphi}(R) = \{(x_1, \ldots, x_n) : \quad (x_1, \ldots, x_n) \in R, \\ \wedge \varphi(x_1, \ldots, x_n)\}$$

where φ is a *built-in* selection condition.

Example:

$\sigma_{\#3>5000}(\text{Account}) =$

1235	SAV	$20000	TD	2
2000	BUS	$10000	Royal	5
2001	BUS	$10000	TD	3

Product

Definition:

$$R \times S = \{((x_1, \ldots, x_n, y_1, \ldots, y_m) : (x_1, \ldots, x_n) \in R, (y_1, \ldots, y_n) \in S\}$$

Example: $\text{Account} \times \text{Bank} =$

1234	CHK	$1000	TD	1	TD	TD Centre
1235	SAV	$20000	TD	2	TD	TD Centre
1236	CHK	$2500	CIBC	1	TD	TD Centre
1237	CHK	$2500	Royal	5	TD	TD Centre
2000	BUS	$10000	Royal	5	TD	TD Centre
2001	BUS	$10000	TD	3	TD	TD Centre
1234	CHK	$1000	TD	1	CIBC	CIBC Tower
1235	SAV	$20000	TD	2	CIBC	CIBC Tower
1236	CHK	$2500	CIBC	1	CIBC	CIBC Tower
1237	CHK	$2500	Royal	5	CIBC	CIBC Tower
2000	BUS	$10000	Royal	5	CIBC	CIBC Tower
2001	BUS	$10000	TD	3	CIBC	CIBC Tower

Union

Definition:

$$R \cup S = \{(x_1, \ldots, x_n) : \quad (x_1, \ldots, x_n) \in R \\ \vee (x_1, \ldots, x_n) \in S\}$$

Example:

$$\pi_{\#1}(\sigma_{\#2='\mathrm{CHK}'}(\mathrm{Account})) \cup \pi_{\#1}(\sigma_{\#2='\mathrm{SAV}'}(\mathrm{Account})) =$$

1234	CHK
1236	CHK
1237	CHK
1235	SAV

Difference

Definition:

$$\{(x_1, \ldots, x_n) : \quad (x_1, \ldots, x_n) \in R, \\ \wedge (x_1, \ldots, x_n) \notin S\}$$

Example:

Is there an account without a bank?

$\pi_{\#1,\#4}(\text{Account}) - \pi_{\#1,\#4}(\sigma_{\#4=\#6}(\text{Account} \times \text{Bank})) =$

1237	Royal
2000	Royal

Relational Calculus/SQL to Algebra

How do we know that these operators are sufficient to evacuate *all* Relational Calculus queries?

Theorem (Codd)

For every domain independent Relational Calculus query there is an equivalent Relational Algebra expression.

$$
\begin{aligned}
RCtoRA(R_i(x_1, \ldots, x_k)) &= R_i \\
RCtoRA(Q \wedge x_i = x_j) &= \sigma_{\#i=\#j}(RCtoRA(Q)) \\
RCtoRA(\exists x_i.Q) &= \pi_{FV(Q)-\{\#i\}}(RCtoRA(Q)) \\
RCtoRA(Q_1 \wedge Q_2)) &= RCtoRA(Q_1) \times RCtoRA(Q_2) \\
RCtoRA(Q_1 \vee Q_2)) &= RCtoRA(Q_1) \cup RCtoRA(Q_2) \\
RCtoRA(Q_1 \wedge \neg Q_2)) &= RCtoRA(Q_1) - RCtoRA(Q_2)
\end{aligned}
$$

. . . queries in $\wedge$ must have disjoint sets of free variables

. . . we must *invent* consistent way of referring to attributes

Iterator Model for RA

How do we avoid (mostly) storing *intermediate* results?

Idea

We use the *cursor OPEN/FETCH/CLOSE interface*.

Every *implementation* of an Relational Algebra operator:

1. implements the cursor interface to produce answers
2. uses the *same* interface to get answers from its children

. . . we make (at least) one *physical implementation* per operator.

Physical Operators (example: selection)

```
// select_{#i=#j}(Child)
  OPERATOR  child;
  int i,j;

public:

  OPERATOR  selection(OPERATOR c, int i0, int j0)
                { child = c; i = i0; j = j0; };
  void open()   { child.open(); };
  tuple fetch() { tuple t = child.fetch();
                  if (t==NULL || t.attr(i)=t.attr(j))
                       return t;
                  return this.fetch();
                };
  void close()  { child.close(); }
```

Physical Operators (cont.)

The rest of the lot:

product:
simple nested loops algorithm

projection:
eliminate *unwanted attributes* from each tuple

union:
simple concatenation

set difference:
nested loops algorithm that checks for tuples on r.h.s.

WARNING!

This doesn't quite work: projection and union may produce *duplicates*
. . . need to be followed by a *duplicate elimination operator*

How to make it FAST(*er*)?

Observation

Naive implementation for each operator will work

. . . very (very very very) slowly

What to do?

1. use (disk-based) data structures for efficient searching
   INDEXING (used, e.g., in selections)
2. use better algorithms to implement the operators
   commonly based on *SORTING* or *HASHING*
3. rewrite the RA expression to an equivalent, but more efficient one
   - remove unnecessary operations (e.g., duplicate elimination)
   - enable the use of better algorithms/data structures

Atomic Relations

We use the **Access Methods** (defined in last lecture) to gain access to the stored data:

- if an index $R_{index}(x)$ (where x is the *search attribute*) is available we replace a subquery of the form

  $$\sigma_{x=c}(R)$$

  with accessing $R_{index}(x)$ directly,
- Otherwise: check all file blocks holding tuples for R.

Even if an index is available, scanning the entire relation may be faster in certain circumstances:

- the relation is very small
- the relation is large, but we expect most of the tuples in the relation to satisfy the selection criteria

Joins

- THE most studied operation of relational algebra; There are many other ways to perform a join.
  1. The *Nested Loop* Join

     ```
     for t in R  do for u in S do
           if C(t,u) then output (tu)
     ```

     $\Rightarrow$ with the optional use of indices on `S`
  2. The *Sort-Merge* Join

     sort the tuples of `R` and of `R` on the common values, then merge the sorted relations.
  3. The *Hash* Join

     hash each tuple of `R` and of `S` to "buckets" by applying a hash function to columns involved in the join condition. Within each bucket, look for tuples with the matching values.
- the *cost* of the join depends on the chosen method

Duplicates and Aggregates

How do we eliminate duplicates in results of operations?
How do we group tuples for aggregation?

Similar solution:

1. sort the result and then eliminate duplicates/aggregate
2. hash the result and do the same

$\Rightarrow$ often an index (e.g., a B+ tree) can be used to avoid the sorting/hashing phase

The rest of the lot

- we assume a natural implementation for selection, duplicate-preserving projection, and duplicate preserving union.
- set difference can be evaluated similarly to a join.
- additional operations:
  - sorts (used for Sort-Merge Join, Aggregation, and Duplicate Elimination). Uses an *external sort algorithm* (essentially a merge-sort adopted for disk)
  - temporary store (to avoid recomputation of subqueries; can be inserted anywhere in the query plan)
  - ...

Query Optimization

- Many possible **query plans** for a single query:
  1. equivalences in *Relational Algebra*
  2. choice of *Operator Implementation*

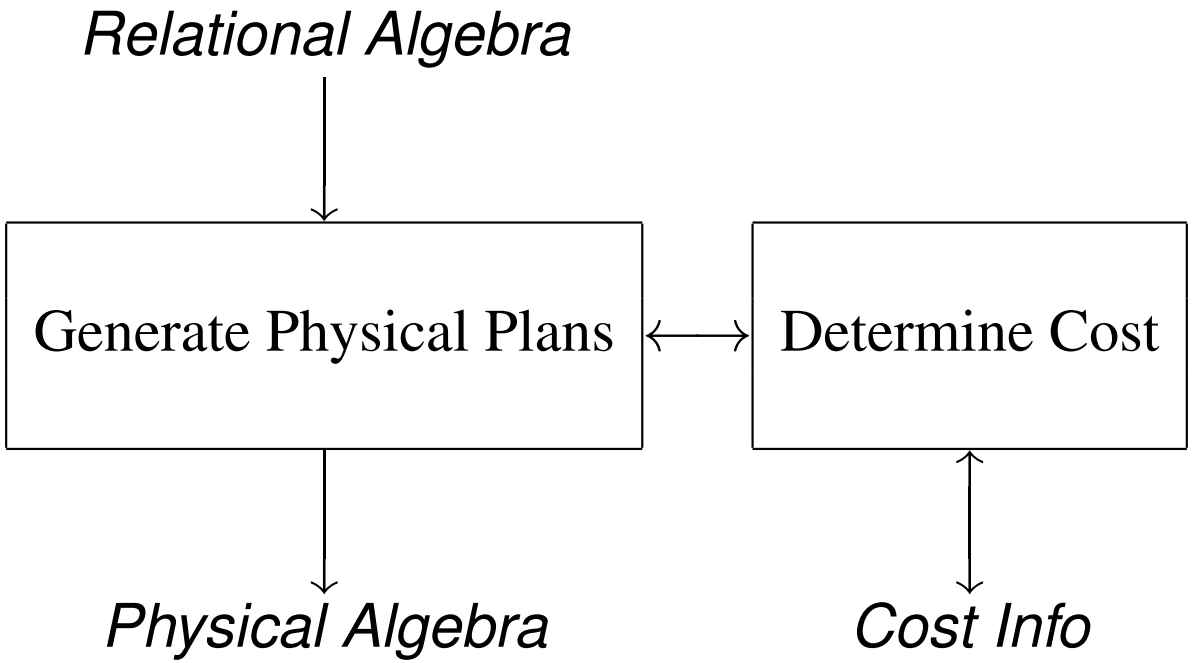
  $\Rightarrow$ performance differs greatly
- How do we choose the best plan?
  1. "always good" transformations
  2. cost-based model

  $\Rightarrow$ finding an **optimal plan** is computationally

  not feasible: we look for a *reasonable* one.

General Approach

- generate all physical plans equivalent to the query
- pick the one with the lowest cost



. . . All Equivalent Plans?!

- Cannot be done in general:

  $\Rightarrow$ it is **undecidable** if a query (un-)satisfiable

  equivalent to an *empty* plan.
- Very expensive even for **conjunctive** queries

  $\Rightarrow$ the *Join-ordering* problem
- In practice:

  $\Rightarrow$ only plans of certain form are considered

  (restrictions on the search space.)

  $\Rightarrow$ the goal is to eliminate the really bad ones.

. . . and Pick the Best one?!

- How do we determine which plan is the best one?

  $\Rightarrow$ we cannot just run the plan to find out

  $\Rightarrow$ instead we estimate the cost based on

  stats collected by the DBMS for all relations

- A **Simple Cost Model** for disk I/O; Assumptions:

  Uniformity: all possible values of an attribute are equally likely to appear in a relation.

  Independence: the likelihood that an attribute has a particular value (in a tuple) does not depend on values of other attributes.

A Simple Cost Model (cont.)

- For a stored relation R with an attribute A we keep:
  1. $|R|$: the cardinality of R (the number of tuples in R)
  2. $b(R)$: the blocking factor for R
  3. $\min(R, A)$: the minimum value for A in R
  4. $\max(R, A)$: the maximum value for A in R
  5. $\mathrm{distinct}(R, A)$: the number of distinct values of A
- Based on these values we try to estimate the **cost** of physical plans.

Cost of Retrieval

```
Mark(Studnum, Course, Assignnum, Mark)
```

```
SELECT Studnum, Mark
FROM   Mark
WHERE  Course = 'PHYS'
  AND Studnum = 100 AND Mark > 90
```

Indices:

- clustering index CourseInd on Course
- non-clustering index StudnumInd on Studnum

Assume:

- $|\text{Mark}| = 10000$
- $b(\text{Mark}) = 50$
- 500 different students
- 100 different courses
- 100 different marks

Strategy 1: Use CourseInd

Assuming *uniform distribution* of tuples over the courses, there will be about $|\mathrm{Mark}|/100 = 100$ tuples with `Course = PHYS`.

Searching the CourseInd index has a cost of 2. Retrieval of the 100 matching tuples adds a cost of $100/b(\mathrm{Mark})$ data blocks.
The total cost of 4.

Selection of N tuples from relation R using a clustered index has a cost of $2 + N/b(R)$.

Strategy 2: Use StudnumInd

Assuming *uniform distribution* of tuples over student numbers, there will be about $|\mathrm{Mark}|/500 = 20$ tuples for each student.

Searching the StudnumInd has a cost of 2. Since this is not a clustered index, we will make the pessimistic assumption that each matching record is on a separate data block, i.e., 20 blocks will need to be read. The total cost is 22.

Selection of N tuples from relation R using a clustered index has a cost of $2 + N$.

Strategy 3: Scan the Relation

The relation occupies 10,000/50 = 200 blocks, so 200 block I/O operations will be required.

Selection of N tuples from relation R by scanning the entire relation has a cost of $|R|/b(R)$.

Cost of other Relational Operations

Costs of **physical** operations (in I/O's):

- Selection: $\text{cost}(\sigma_c(E)) = (1 + \epsilon_c)\,\text{cost}(E)$.
- Nested-Loop Join (R is the **outer** relation):

$$\text{cost}(R \bowtie S) = \text{cost}(R) + (|R|/b)\,\text{cost}(S)$$

- Index Join (R is the outer relation, and S is the inner relation: B-tree with depth d_S):

$$\text{cost}(R \bowtie S) = \text{cost}(R) + d_S|R|$$

- Sort-Merge Join:

$$\text{cost}(R \bowtie S) = \text{cost}(\text{sort}(R)) + \text{cost}(\text{sort}(S))$$

  where $\text{cost}(\text{sort}(E)) = \text{cost}(E) + (|E|/b)\log(|E|/b)$.

- ...

Why don't we always use the Merge-Sort Join?

Size Estimation

In the cost estimation we need to know sizes of results of operations: we use the **selectivity**, defined, for a condition $\sigma_{\text{condition}}(R)$, as:

$$\text{sel}(\sigma_{\text{condition}}(R)) = \frac{|\sigma_{\text{condition}}(R)|}{|R|}$$

Again, the optimizer will *estimate* selectivity using simple rules based on its statistics:

$$\text{sel}(\sigma_{A=c}(R)) \approx \frac{1}{\text{distinct}(R, A)}$$

$$\text{sel}(\sigma_{A\leq c}(R)) \approx \frac{c - \min(R, A)}{\max(R, A) - \min(R, A)}$$

$$\text{sel}(\sigma_{A\geq c}(R)) \approx \frac{\max(R, A) - c}{\max(R, A) - \min(R, A)}$$

Size Estimation (cont.)

For Joins:

- General Join (on attribute A):

$$|R \bowtie S| \approx |R| \frac{|S|}{\text{distinct}(S, A)}$$

  or as

$$|R \bowtie S| \approx |S| \frac{|R|}{\text{distinct}(R, A)}$$

- Foreign key Join (Student and Enrolled joined on Sid):

$$|R \bowtie S| = |S| \frac{|R|}{|S|} = |R|$$

  May joins are foreign key joins, like this one.

More Advanced Statistics

- so far only a very primitive cost estimation approach
- in practice: more complex approaches
  - histograms to approximate non-uniform distributions
  - correlations between attributes
  - uniqueness (keys) and containment (inclusions)
  - sampling methods
  - etc, etc

Plan Generation

1. apply “always good” transformations
   $\Rightarrow$ **heuristics** that work in the majority of cases

2. cost-based join-order selection
   $\Rightarrow$ applied on **conjunctive subqueries** (the “select blocks”)
   $\Rightarrow$ still computationaly not tractable.

“Always good” transformations

- Push selections:

$$\sigma_{\varphi}(E_1 \bowtie_{\theta} E_2) = \sigma_{\varphi}(E_1) \bowtie_{\theta} E_2$$

  for φ involving columns of E_1 only (and vice versa).

- Push projections:

$$\pi_V(R \bowtie_{\theta} S) = \pi_V(\pi_{V_1}(R) \bowtie_{\theta} \pi_{V_2}(S))$$

  where V_1 is the set of all attributes of R involved in θ and V (similarly for V_2).

- Replace products by joins:

$$\sigma_{\varphi}(R \times S) = R \bowtie_{\varphi} S$$

$\Rightarrow$ also reduces the space of plans we need to search

Example

- Assume that
  - there are $|S| = 1000$ students,
  - enrolled in $|C| = 500$ classes.
  - the enrollment table is $|E| = 5000$,
  - and, on average, each student is registered for five courses.
- Then:

$$\text{cost}(\sigma_{\text{name}='\text{Smith}'}(S \bowtie (E \bowtie C))) >> \text{cost}(\sigma_{\text{name}='\text{Smith}'}(S) \bowtie (E \bowtie C))$$

Join Order Selection

- Joins are associative $R \bowtie S \bowtie T \bowtie U$ can be equivalently expressed as
  1. $((R \bowtie S) \bowtie T) \bowtie U$
  2. $(R \bowtie S) \bowtie (T \bowtie U)$
  3. $R \bowtie (S \bowtie (T \bowtie U))$

  $\Rightarrow$ try to minimize the intermediate result(s).
- Moreover, we need to decide which of the subexpressions is evaluated first

  $\Rightarrow$ e.g., Nested Loop join's cost is **not** symmetric!

Example

We have the following two join orders to pick from:

1. $\sigma_{\text{name}='\text{Smith}'}(S) \bowtie (E \bowtie C)$
   we produce $E \bowtie C$, which has one tuple for each course registration (by any student) $\sim$ 5000 tuples.
2. $(\sigma_{\text{name}='\text{Smith}'}(S) \bowtie E) \bowtie C$
   we produce an intermediate relation which has one tuple for each course registration by a student named Smith. If there are only a few Smith's among the 1,000 students (say there are 10), this relation will contain about 50 tuples.

Pipelined Plans

- all operators (except sorting) operate without storing intermediate results

  $\Rightarrow$ iterator protocols in constant storage

  $\Rightarrow$ no recomputation for **left-deep** plans

Temporary Store

- General pipelined plans lead to *recomputation*
- We introduce an additional **store** operator

  $\Rightarrow$ allows us to store intermediate results in a relation
  $\Rightarrow$ we can also built a (hash) index on top of the result
- Semantically, the operator represents the **identity**
- The costs of plans:
  1. cumulative cost—to compute the value of the expression and store then in a relation (once):

     $$\text{cost}_c(\text{store}(E)) = \text{cost}_c(E) + \text{cost}_s(E) + |E|/b$$
  2. scanning cost—to "read" all the tuples in the stored result of the expression:

     $$\text{cost}_s(\text{store}(E)) = |E|/b$$

Parallelism in Query Execution

Another approach to improving performance:

take advantage of parallelism in hardware

- mass storage usually reads/writes data in *blocks*
- multiple mass storage units can be *accessed in parallel*
- relational operators amenable to parallel execution

Summary

Relational Algebra is the basis for efficient implementation of SQL

- provides a connection between conceptual and physical level
- breaks query execution to (easily) manageable pieces
- allows the use of efficient algorithms/data structures
- provides mechanism for *query optimization* based on logical transformations (including simplifications based on integrity constraints, etc.)

Performance of database operations depends on the way queries (and updates) are executed against a particular *physical schema/design*.

. . . understanding *basics* of query processing is necessary to making *physical design decisions*

. . . performance also depends on *transaction management* (later)