Database Tuning and Physical Design: Basics of Query Execution

Fall 2017

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:
   ⇒ generates an efficient query plan
   ⇒ uses statistics collected about the stored data
4. Plan execution:
   ⇒ access methods to access stored relations
   ⇒ physical relational operators to combine relations
Relational Algebra

Idea

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

...called a *RELATIONAL ALGEBRA*.

\[(\mathcal{U}; R_0, \ldots, R_k, \times, \sigma_\varphi, \pi_V, \cup, \neg)\]

Constants:

\(R_i\): one for each relational scheme

Unary operators:

\(\sigma_\varphi\): selection (keeps only tuples satisfying \(\varphi\))

\(\pi_V\): projection (keeps only attributes in \(V\))

Binary operators:

\(\times\): Cartesian product

\(\cup\): union

\(\neg\): set difference
### Examples

<table>
<thead>
<tr>
<th>Account</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Acnt#</td>
<td>Type</td>
<td>Balance</td>
<td>Bank</td>
<td>Branch</td>
<td></td>
</tr>
<tr>
<td>1234</td>
<td>CHK</td>
<td>$1000</td>
<td>TD</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1235</td>
<td>SAV</td>
<td>$20000</td>
<td>TD</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1236</td>
<td>CHK</td>
<td>$2500</td>
<td>CIBC</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1237</td>
<td>CHK</td>
<td>$2500</td>
<td>Royal</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>BUS</td>
<td>$10000</td>
<td>Royal</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td>BUS</td>
<td>$10000</td>
<td>TD</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

### Bank

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD</td>
<td>TD Centre</td>
</tr>
<tr>
<td>CIBC</td>
<td>CIBC Tower</td>
</tr>
</tbody>
</table>
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 \textit{numbers}.

Example:

\[ \pi\{\#1,\#2\}(\text{Account}) = \]

\[
\begin{array}{|c|c|}
\hline
1234 & CHK \\
1235 & SAV \\
1236 & CHK \\
1237 & CHK \\
2000 & BUS \\
2001 & BUS \\
\hline
\end{array}
\]
Selection

Definition:

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

where $\varphi$ is a *built-in* selection condition.

Example:

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

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1235</td>
<td>SAV</td>
<td>$20000</td>
<td>TD</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>BUS</td>
<td>$10000</td>
<td>Royal</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td>BUS</td>
<td>$10000</td>
<td>TD</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Product

Definition:

\[ R \times S = \left\{ (x_1, \ldots, x_n, y_1, \ldots, y_m) : \\
(x_1, \ldots, x_n) \in R, \\
(y_1, \ldots, y_n) \in S \right\} \]

Example: Account \times Bank =

<table>
<thead>
<tr>
<th>Account</th>
<th>Bank</th>
<th>Amount</th>
<th>Type</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>CHK</td>
<td>$1000</td>
<td>TD</td>
<td>1</td>
</tr>
<tr>
<td>1235</td>
<td>SAV</td>
<td>$20000</td>
<td>TD</td>
<td>2</td>
</tr>
<tr>
<td>1236</td>
<td>CHK</td>
<td>$2500</td>
<td>CIBC</td>
<td>1</td>
</tr>
<tr>
<td>1237</td>
<td>CHK</td>
<td>$2500</td>
<td>Royal</td>
<td>5</td>
</tr>
<tr>
<td>2000</td>
<td>BUS</td>
<td>$10000</td>
<td>Royal</td>
<td>5</td>
</tr>
<tr>
<td>2001</td>
<td>BUS</td>
<td>$10000</td>
<td>TD</td>
<td>3</td>
</tr>
<tr>
<td>1234</td>
<td>CHK</td>
<td>$1000</td>
<td>TD</td>
<td>1</td>
</tr>
<tr>
<td>1235</td>
<td>SAV</td>
<td>$20000</td>
<td>TD</td>
<td>2</td>
</tr>
<tr>
<td>1236</td>
<td>CHK</td>
<td>$2500</td>
<td>CIBC</td>
<td>1</td>
</tr>
<tr>
<td>1237</td>
<td>CHK</td>
<td>$2500</td>
<td>Royal</td>
<td>5</td>
</tr>
<tr>
<td>2000</td>
<td>BUS</td>
<td>$10000</td>
<td>Royal</td>
<td>5</td>
</tr>
<tr>
<td>2001</td>
<td>BUS</td>
<td>$10000</td>
<td>TD</td>
<td>3</td>
</tr>
</tbody>
</table>
Union

Definition:

\[ R \cup S = \{ (x_1, \ldots, x_n) : (x_1, \ldots, x_n) \in R \lor (x_1, \ldots, x_n) \in S \} \]

Example:

\[ \pi_1(\sigma_{#2='\text{CHK}'}(\text{Account})) \cup \pi_1(\sigma_{#2='\text{SAV}'}(\text{Account})) = \]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>CHK</td>
</tr>
<tr>
<td>1236</td>
<td>CHK</td>
</tr>
<tr>
<td>1237</td>
<td>CHK</td>
</tr>
<tr>
<td>1235</td>
<td>SAV</td>
</tr>
</tbody>
</table>
Difference

Definition:
\[ \{(x_1, \ldots, x_n) : (x_1, \ldots, x_n) \in R, \]
\[ \land (x_1, \ldots, x_n) \not\in S \} \]

Example:

*Is there an account without a bank?*

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>Royal</td>
</tr>
<tr>
<td>2000</td>
<td>Royal</td>
</tr>
</tbody>
</table>
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.

\[
\begin{align*}
RCtoRA(R_i(x_1, \ldots, x_k)) &= R_i \\
RCtoRA(Q \land x_i = x_j) &= \sigma_{#i=#j}(RCtoRA(Q)) \\
RCtoRA(\exists x_i. Q) &= \pi_{FV(Q) - \{#i\}}(RCtoRA(Q)) \\
RCtoRA(Q_1 \land Q_2) &= RCtoRA(Q_1) \times RCtoRA(Q_2) \\
RCtoRA(Q_1 \lor Q_2) &= RCtoRA(Q_1) \cup RCtoRA(Q_2) \\
RCtoRA(Q_1 \land \neg Q_2) &= RCtoRA(Q_1) - RCtoRA(Q_2)
\end{align*}
\]

...queries in \( \land \) 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 a 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 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
   
   \[
   \text{for } t \text{ in } R \text{ do for } u \text{ in } S \text{ do } \\
   \text{if } C(t,u) \text{ then output } (tu) \\
   \Rightarrow \text{ with the optional use of indices on } S
   \]

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

⇒ often an index (e.g., a B+ tree) can be used to avoid the sorting/hashing phase
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)
- ...
Many possible **query plans** for a single query:

1. equivalences in *Relational Algebra*
2. choice of *Operator Implementation*

⇒ performance differs greatly

How do we choose the best plan?

1. “always good” transformations
2. cost-based model

⇒ 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

```
Relational Algebra
                                              ↓
Generate Physical Plans ←→ Determine Cost
                                              ↓
Physical Algebra
```

Cost Info
... All Equivalent Plans?!

- Cannot be done in general:
  - ⇒ it is **undecidable** if a query (un-)satisfiable equivalent to an *empty* plan.
- Very expensive even for **conjunctive** queries
  - ⇒ the *Join-ordering* problem
- In practice:
  - ⇒ only plans of certain form are considered
  - (restrictions on the search space.)
  - ⇒ the goal is to eliminate the really bad ones.
... and Pick the Best one?!

- How do we determine which plan is the best one?
  - we cannot just run the plan to find out
  - 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.
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. $\text{distinct}(R, A)$: 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 $|\text{Mark}|/100 = 100$ tuples with $\text{Course} = \text{PHYS}$.

Searching the CourseInd index has a cost of 2. Retrieval of the 100 matching tuples adds a cost of $100/b(\text{Mark})$ 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 $|\text{Mark}|/500 = 20$ 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
   \[ \Rightarrow \textbf{heuristics} \] that work in the majority of cases

2. cost-based join-order selection
   \[ \Rightarrow \text{applied on} \ \textbf{conjunctive subqueries} \ (\text{the “select blocks”}) \]
   \[ \Rightarrow \text{still computationaly not tractable.} \]
“Always good” transformations

- Push selections:
  \[\sigma_\varphi(E_1 \Join_\theta E_2) = \sigma_\varphi(E_1) \Join_\theta E_2\]
  for \(\varphi\) involving columns of \(E_1\) only (and vice versa).

- Push projections:
  \[\pi_V(R \Join_\theta S) = \pi_V(\pi_{V_1}(R) \Join_\theta \pi_{V_2}(S))\]
  where \(V_1\) is the set of all attributes of \(R\) involved in \(\theta\) and \(V\) (similarly for \(V_2\)).

- Replace products by joins:
  \[\sigma_\varphi(R \times S) = R \Join_\varphi S\]
  \(\Rightarrow\) 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:

$$\text{cost}(\sigma_{\text{name}='Smith'}(S \bowtie (E \bowtie C))) >> \text{cost}(\sigma_{\text{name}='Smith'}(S) \bowtie (E \bowtie C))$$
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. \( \sigma_{\text{name}='Smith'}(S) \Join (E \Join C) \)
   
   we produce \( E \Join C \), which has one tuple for each course registration (by any student) \( \sim 5000 \) tuples.

2. \( (\sigma_{\text{name}='Smith'}(S) \Join E) \Join C \)
   
   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.
all operators (except sorting) operate without storing intermediate results
⇒ iterator protocols in constant storage
⇒ no recomputation for **left-deep** plans
Temporary Store

- General pipelined plans lead to *recomputation*
- We introduce an additional **store** operator
  ⇒ allows us to store intermediate results in a relation
  ⇒ 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:
     \[
     \text{cost}_s(\text{store}(E)) = |E|/b
     \]
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)