Database Systems
- Query Processing and Optimization 1 -

Prof. Dr. Agnès Voisard
Institute of Computer Science,
Databases and Information Systems Group
and Fraunhofer FOKUS

2025
v1

Notes

Notes

Introduction

- Find the best method of finding an answer using the existing database structures
- Worthwhile for the system to spend some time on the selection of the good strategy

Query Processing

1. Query in a high-level query language
   $\Rightarrow$ SCANNING, PARSING, VALIDATING
2. Intermediate form of a query
   $\Rightarrow$ QUERY OPTIMIZER
3. Execution plan
   $\Rightarrow$ QUERY CODE GENERATOR
4. Code to execute the query (interpreted or compiled)
   $\Rightarrow$ RUNTIME DB PROCESSOR
5. Result of query

Query Interpretation

We do not expect the user to write queries in a way that suggests the most efficient strategy.

Improving the strategy for processing a query:
query optimization.

Before query processing, system translates the query in a usable form.

SQL: suitable for human use but not for the system.

Internal representation based on the **relational algebra**.

Query Interpretation (cont'd)

Steps:
A. Query translated into the system's internal form (cf. parser or compiler) as a tree or graph data structure. **Query tree**.
check of the syntax, relation names, etc.
(If query is a view, system replaces all references to the view name with the RA expression for computing the view).

Query Interpretation (cont'd)

B. Query is translated, optimization can begin:

1. Find a more efficient (equivalent) expression to execute.
2. Select a detailed strategy for processing the query.
   (**execution strategy**)
   Choose indices, order for processing the tuples.
   Final choice based primarily on the number of disk accesses required.

Equivalence of Expressions

Relational algebra:
Each expression represents a particular sequence of operations.

Example: *Bank* database.
CUSTOMER(CName, Street, CustomerCity)
DEPOSIT(BranchName,AccountNum,CName,Balance)
BRANCH(BranchName, Assets, BranchCity)

Instances:
CUSTOMER, DEPOSIT, BRANCH

Equivalence of Expressions (cont'd)

Selection Operation

Assets and names of banks that have depositors living in "Port Chester"?

$\Pi_{BranchName, Assets}$
$(\sigma_{CustomerCity = \text{“}PortChester\text{”}}(CUSTOMER \bowtie DEPOSIT \bowtie BRANCH))$

Selection

$CUSTOMER \bowtie DEPOSIT \bowtie BRANCH$
is a large relation.

We are interested only in a few tuples of this relation.
Intermediary result: $CUSTOMER \bowtie DEPOSIT \bowtie BRANCH$
too large to be kept in main memory => stored on disk.
=> In addition to accessing the disk to read the 3 relations, read and write intermediary results.

We need not consider tuples in $CUSTOMER$ that do not have $CustomerCity = \text{``}PortChester''$.
Reduce the size of the intermediary result.

Selection (cont'd)

Equivalent relational algebra operation:

$\Pi_{BranchName, Assets}((\sigma_{CustomerCity=``PortChester''}(CUSTOMER)) \bowtie DEPOSIT \bowtie BRANCH)$

Rule:

Perform selection operation as early as possible.

Selection (cont'd)

Selection pertains only to the *CUSTOMER* relation.

Suppose the query is:

CustomerCity = "Port Chester" and balance over \$1000 ?

$\Pi_{BranchName, Assets}$
$(\sigma_{(CustomerCity = \text{"}PortChester\text{"} \wedge Balance > 1000)}$
$(CUSTOMER \bowtie DEPOSIT \bowtie BRANCH))$

We cannot apply the selection
$CustomerCity = \text{"}PortChester\text{"} \wedge Balance > 1000$
directly to the *CUSTOMER* relation
(we need both *CUSTOMER* and *DEPOSIT*).

Branch relation does not involve either *CustomerCity* or *Balance*.

Selection (cont'd)

If join processed as
$(CUSTOMER \bowtie DEPOSIT) \bowtie BRANCH$
we can rewrite the query as
$\Pi_{BranchName, Assets}$
$((\sigma_{(CustomerCity = \text{``}PortChester'' \wedge Balance > 1000)}$
$(CUSTOMER \bowtie DEPOSIT)) \bowtie BRANCH)$

Selection (cont'd)

Subquery:

$\sigma_{(CustomerCity=\text{``}PortChester\text{''} \wedge Balance>1000)}(CUSTOMER \bowtie DEPOSIT)$

Can be rewritten as

$\sigma_{CustomerCity=\text{``}PortChester\text{''}}(\sigma_{Balance>1000}(CUSTOMER \bowtie DEPOSIT))$

With the "perform selections early" rule

$\sigma_{CustomerCity=\text{``}PortChester\text{''}}(CUSTOMER) \bowtie \sigma_{Balance>1000}(DEPOSIT)$

Selection (cont'd)

=> Second transformation rule:
Replace expressions of the form

$$\sigma_{P1 \wedge P2}(e)$$
with
$$\sigma_{P1}(\sigma_{P2}(e))$$

($P1$ and $P2$ predicates, e RA expression).
Equivalence:
$\sigma_{P1}(\sigma_{P2}(e)) = \sigma_{P2}(\sigma_{P1}(e)) = \sigma_{P1 \wedge P2}(e)$.

Projection Operation

Other technique to reduce the size of temporary results: projection like selection reduces the size of relations.
Advantageous to apply immediately any projection that is possible.

Rule: **Perform projections early**.
New expression:

$$\Pi_{BranchName,Assets} ((\pi_{Branchname}((\sigma_{CustomerCity=``PortChester''}(CUSTOMER)) \bowtie DEPOSIT)) \bowtie BRANCH)$$

Natural Join Operation

Arises frequently in practice.
One of the most costly operations.

Natural join is associative:
$(r1 \bowtie r2) \bowtie r3 = r1 \bowtie (r2 \bowtie r3)$

Equivalent expressions but cost of computing them may differ.

Example:
$\Pi_{BranchName, Assets}$
$(\sigma_{CustomerCity = \text{“}PortChester''}(CUSTOMER))$
$\bowtie DEPOSIT \bowtie BRANCH)$

Natural Join Operation (cont'd)

Hypothesis:
Compute $DEPOSIT \bowtie BRANCH$ first, and join the result with $\sigma_{CustomerCity=``PortChester''}(CUSTOMER)$.
However, $DEPOSIT \bowtie BRANCH$ is likely to be a large relation (tuples for every account).

By contrast, $\sigma_{CustomerCity=``PortChester''}(CUSTOMER)$ is small. (since the bank has a large number of distributed branches, likely that only a small fraction of the customers live in Port Chester).
Natural join is commutative: $r1 \bowtie r2 = r2 \bowtie r1$

Natural Join (cont'd)

New expression:

$\Pi_{BranchName, Assets}$
$(((\sigma_{(CustomerCity = \text{``}PortChester''}(CUSTOMER)) \bowtie BRANCH) \bowtie DEPOSIT)$

=> we could join $\sigma_{CustomerCity = \text{``}PortChester''}(CUSTOMER)$ with *Branch* as the first join operation performed.

However, they have no attribute in common. It is just a Cartesian product.

If c customers in Port Chester and b branches, generates bc tuples. Will produce a large temporary relation. Bad strategy.

=> previous expression was better.

Other Operations

Some equivalences:

$\sigma_P(r1 \cup r2) = \sigma_P(r1) \cup \sigma_P(r2)$

$\sigma_P(r1 - r2) = \sigma_P(r1) - \sigma_P(r2)$

$(r1 \cup r2) \cup r3 = r1 \cup (r2 \cup r3)$

$r1 \cup r2 = r2 \cup r1$

Large number of possible efficient strategies in complex queries.

Some query processors choose in a set of strategies on the basis of heuristics.

Estimation of Query-Processing Cost

Strategy depends upon the **size of each relation** and the **distribution of values** within columns.

In previous example, number of customers in Port Chester has a major impact on the usefulness of the techniques.

DBS may store statistics for each relation r:

1. n_r: number of tuples in relation r.
2. s_r: size of a record (tuple) of relation r (in bytes).
3. $V(A, r)$: number of distinct values that appear in relation r for attribute A.

Estimation of Query-Processing Cost (cont'd)

1 and 2: estimate the size of a Cartesian product.
rXs contains $n_r n_s$ tuples.
Each tuple occupies $s_r + s_s$ bytes.

3: estimate how many tuples satisfy a selection predicate of the form:
$attribute - name = value$.

Query-Processing Cost (cont'd)

To perform such an estimation, we need to know how often each value appears in a column.
If uniform distribution of values (each value appears with the same probability), then the query

$$\sigma_{A=a}(r)$$

is estimated to have:

$\frac{n_r}{V(A,r)}$ tuples.

Estimation of Query-Processing Cost (cont'd)

Remark: this is not always realistic.
Ex. *Branchname* attribute in *DEPOSIT* relation.
One tuple in *DEPOSIT* for each account.

Large branches have more account than smaller branches
=> certain *BranchName* values appear with greater probability than other.

In the sequel: Uniform distribution
(good approximation in many cases).

Query-Processing Cost (cont'd)

Estimation of the size of a natural join more complicated than the one of a selection or a Cartesian product.

Let $r1(R1)$ and $r2(R2)$ be relations.
$\diamond$ If $R1 \cap R2 = \emptyset$ then $r1 \bowtie r2$ is the same as $r1Xr2$
(=> use Cartesian product estimation techniques).

$\diamond$ If $R1 \cap R2$ is a key for $R1$, we know that a tuple of $r2$ will join with exactly one tuple from $r1$.
Therefore number of tuples in $r1 \bowtie r2$ is no greater than the number of tuples in $r2$.

Estimation of Query-Processing Cost

$\diamond$ If $R1 \cap R2$ is a key for neither $R1$ nor $R2$:
Use the 3rd statistics.

Consider a tuple t of $r1$, and assume $R1 \cap R2 = \{A\}$.

Estimate: tuple t produces:

$$\frac{n_{r2}}{V(A, r2)}$$

tuples in $r1 \bowtie r2$.

(number of values in $r2$ with an A value of $t[A]$).

Query-Processing Cost (cont'd)

For all tuples of $r1$, we estimate that there are:

$$\frac{n_{r1} n_{r2}}{V(A,r2)}$$

(Nb of tuples in $r1 \bowtie r2$)
If we reverse the roles of r1 and r2: $\frac{n_{r1} n_{r2}}{V(A,r1)}$

If 2 estimates differ: $\frac{n_{r1} n_{r2}}{V(A,r2)} \neq \frac{n_{r1} n_{r2}}{V(A,r1)}$

Probably some tuples do not participate in the join.
=> the lower of the 2 estimates is probably the better one.

Query-Processing Cost (cont'd)

Maintaining accurate statistics:
update the statistics every time a relation is modified.
Expensive!

Most system do not do it on every modification
but during periods of light system load.

Result: statistics not completely accurate.
But ok if interval between update of statistics not too long.

Summary

- Relational algebra expression can be optimized
- Estimation of query processing cost - statistics
- Join operations

What will come next?

1. Welcome to Database Systems
2. Introduction to Database Systems
3. Entity Relationship Design Diagram (ERM)
4. Relational Model
5. Relational Algebra
6. Structured Query Language (SQL)
7. Relational Database Design - Functional Dependencies
8. Relational Database Design - Normalization
9. Online Analytical Processing + Embedded SQL
10. Data Mining
11. Physical Representation - Storage and File Structure
12. Physical Representation - Indexing and Hashing
13. Transactions
14. Concurrency Control Techniques
15. Recovery Techniques
16. Query Processing and Optimization