Physical Data Organization and Query Processing

Question: How do we organize a database physically in order to achieve efficient query processing?

Obvious points:

- Physical database organization has a profound effect upon the efficiency of query processing.
- Indices (both primary and secondary) to the most important attributes are the key to efficiency.

Assumptions:

- A relation is stored as a group of tuples.
- Each tuple of a relation is stored as a record.
- The primary key of the relation is used as the primary key of the physical storage organization.
- Other indices are possible; these are secondary design decisions.

Note further:

- Secondary indices are expensive to maintain.
- It may not be feasible to maintain a secondary index on every attribute.

We start by looking at each of the fundamental query types in isolation:

- Select
- Project
- Join

Processing of Select Queries:

The “easiest” situation occurs when the selection criterion (the "Where" part) involves a simple selection on a primary key:

```
Select   *
From     DEPARTMENT
Where    DNUMBER = 3
```

- Just use the primary key index to identify the desired tuples.

In other cases, efficiency depends upon other issues:

```
Select   *
From     EMPLOYEE
Where    SUPERSSN  = 123456789
```

- If the select is on a secondary index, things are almost as good.

- If selection is not on a secondary index, then the tuples must be processed one-by-one.

If there are multiple select conditions, those which are indexed should be processed first.

```
Select   *
From     EMPLOYEE
Where    (DNO  = 5)  AND (SEX = 'F')
```

- Assume that DNO is a secondary index. Then, it is more efficient to select the tuples satisfying (DNO = 5) first, and then the tuples satisfying (SEX = 'F').

- Alternatively, for each tuple selected with (DNO = 5), the check for (SEX = 'F') may be performed immediately.

With disjunctive queries, there is no easy solution:

```
Select   *
From     EMPLOYEE
Where    (DNO  = 5) OR (SEX = 'F')
```

- The best way to process the query is to check both conditions simultaneously on each tuple. This avoids processing each tuple twice.

With range queries, an index which allows sequential access is the best:

```
Select   *
From     EMPLOYEE
Where    SSN < 300000000
```

- If we can process tuples in order of SSN's, the operation will be far more efficient.

- In such a case, hashed-table access is not very useful.
  - B^+-tree access is superior to extendible-hashing access.

Processing of Project Queries:

- With pure projections, the only nontrivial issue is the removal of duplicate entries.

```
Select    distinct SALARY
From      EMPLOYEE
```

- There are two options:

1. Retrieve the tuples, sort the list, and remove the duplicates.

2. Sort the list on the fly, as it is built. Throw out duplicates on the fly.

- Either option effectively requires sorting the list.

Processing Join Queries:

- It is important to realize that, in the worst case, a join can consist of $n_1 \bullet n_2$ tuples, where n_1 and n_2 are the sizes of the two relations. Thus, efficiency is paramount.

- There are two general strategies:
  - Use existing index structures.
  - Build custom, temporary index structures.
- The first option is employed, whenever possible, since constructing temporary indices is expensive.

Sorted sequential processing:

- First consider the case that the matched attributes of each relation are indexed sequentially.

- Assume that MGRSSN is indexed sequentially in DEPARTMENT, and that SSN is the primary key of EMPLOYEE:

```
Select   *
From     EMPLOYEE, DEPARTMENT
Where    EMPLOYEE.SSN =
                  DEPARTMENT.MGRSSN
```

The processing method is similar to the familiar algorithm for merging sorted lists.

- Maintain a pointer to each list.
- Repeat:
  - Increment the one pointing to the smaller value until it matches or exceeds the other.
  - If there is a match, create a join tuple.

  Until one list is exhausted.

- The time complexity of this strategy is $\Theta(n_1+n_2)$, where n_1 and n_2 are the respective sizes of the two relations.

- Because adjacent records are usually blocked together in a (primary) sequential index, this strategy is particularly attractive in that the constant multiplier of the complexity will be relatively low.

Indexed Processing:

- When the join attributes are indexed, but without rapid sequential access (*e.g.,* with extendible hash indices), this approach will prove attractive.

- Only one of the relations need be indexed.

  - Process the tuples of the non-indexed relation, one-by-one.
  - For each tuple, search the index for matching tuples in the other relation.

- This strategy is $\Theta(n_1 \bullet s(n_2))$, where:
  - n_1 = size of the non-indexed relation.
  - n_2 = size of the indexed relation.
  - $s(n_2)$ = time required to retrieve an indexed element in the indexed relation.

- In extendible hashing, $s(n_2) = \Theta(1)$, so the complexity is just $\Theta(n_1)$.

- The constant multiplier will be substantial in comparison to the indexed sequential approach, however, since a separate access is needed for each element in the indexed part.

- A non-sequential index on the first relation ("non-indexed" above) will be of little use.

- If both relations are indexed, process the smaller one sequentially, and use the non-sequential index of the larger one.

Non-indexed Processing:

If indices on the join attributes are not available, there are still several choices: Assume the following parameters:

- n_1 = size of relation which is processed linearly or sorted.
- n_2 = size of the other relation.
- $s(n_2)$ = time to search for one element of the second relation.

1. Brute force processing:
   - In this approach, for each record of the first relation, a search is conducted in the second for a matching tuple.
     - Complexity: $\Theta(n_1 \bullet s(n_2))$.
     - With no special indexing, $\Theta(n_1 \bullet n_2)$.

2. Common-hash
   - One can also build a temporary hash table.
   - In this case, it is often best to hash both relations into the same table. Matching entries will then be found in the same buckets.
   - Usually an intermediate index is used, to avoid physical movement of records.
   - The big cost of this approach is building the intermediate hash table: $\Theta(n_1 + n_2)$, with a large constant multiplier.
   - The join complexity is then also $\Theta(n_1 + n_2)$.

When the join is on more than one attribute:

- Usually it is best to create a join on just one attribute first, and then pare down that result with further select-style checks.

- Whenever possible, choose the most ideal attributes for the first join.
  - The fastest operation.
  - Creation of the fewest tuples.

- In the example below, assume that no indices exist for the join attributes:
  - Join on the second condition first. Why?

- Now assume that DEPARTMENT is indexed by MGRSSN.
  - Use the index on MGRSSN, and join on the first condition first. Why?

```
Select   *
From     EMPLOYEE, DEPARTMENT
Where    (EMPLOYEE.SUPERSSN =
                DEPARTMENT.MGRSSN)
         AND
         (DEPARTMENT.DNAME =
                EMPLOYEE.LNAME)
```

General principle:

- In all of these approaches, view asymptotic complexity measures with caution.
- The size of constant multipliers often determines the complexity in practical terms.

Processing Compound Queries:

- With compound queries, there may be options to arrange things to make the processing more efficient.

- The general strategy is to try to perform operations which reduce the size of relations:
  - Selection
  - Projection
  - Intersection

  before performing operations which increase the size of things:
  - Join
  - Union.

Example:

```
Select   *
From     EMPLOYEE, DEPARTMENT
Where    (EMPLOYEE.SUPERSSN =
                DEPARTMENT.MGRSSN)
          AND
          (SALARY > 50000)
```

- This may be realized in two ways.

$X_1 \leftarrow$
EMPLOYEE $\bowtie_{(SUPERSSN=MGRSSN)}$ DEPARTMENT

$X_2 \leftarrow \sigma_{(SALARY > 50000)}(X_1)$

or

$X_1 \leftarrow \sigma_{(SALARY > 50000)}(EMPLOYEE)$

$X_2 \leftarrow X_1 \bowtie_{(SUPERSSN=MGRSSN)}$ DEPARTMENT

- Clearly, the second alternative is more efficient, in that far fewer tuples are generated.

- There is an extensive theory of such operations, known as *query optimization.*

Examining Query Plans in PostgreSQL

- PostgreSQL has a (nonstandard) command called EXPLAIN.
- Example:

```
company=> explain select * from
    employee, department
company-> where
    employee.dno=department.dnumber;]

              QUERY PLAN
------------------------------------
 Hash Join  (cost=1.04..2.24 rows=8
width=128)
   Hash Cond: ("outer".dno =
"inner".dnumber)
   ->  Seq Scan on employee
(cost=0.00..1.08 rows=8 width=92)
   ->  Hash  (cost=1.03..1.03 rows=3
             width=36)
         ->  Seq Scan on department
(cost=0.00..1.03 rows=3 width=36)
(5 rows)
```

Query Processing on Distributed Databases: Semijoins

In distributed database systems, the cost of transmitting data becomes an important concern.

The *semijoin* is a relational operator which arose in the context of efficient query processing on distributed databases.

- Suppose that we wish to compute the join of the instances of two relation schemata which are stored at distinct remote sites:
  - R[AB] stored at node 1
  - S[BC] stored at node 2.

- Suppose that the query processing may be performed at either remote site, and the result then shipped to the local site.

- Using ordinary joins, we would have to ship at least one of the relations to the other remote site. This could be expensive.

- The semijoin operation provides a way to reduce this cost.

- Suppose that B is a key for S[BC], but that it is not a key for R[AB].

- Suppose further that the size of an “A” value is much larger than that of a “B” value.

- Suppose further that it is expected that there will be a lot of "mismatches" between the two "B" columns (*i.e.*, tuples which will not find a match in the other relation).

- It would then be more economical to ship just those tuples of R[AB] which match a tuple of S[BC] to site 2, rather than to ship all tuples of R[AB].

- We can follow this plan:

1. Send the projection $\pi_B(R[AB])$ to node 2.

2. Compute the *semijoin*

$$S[BC] \ltimes R[AB] = S[BC] \bowtie \pi_B(R[AB])$$

at node 2. In words, $S[BC] \ltimes R[AB]$ consists of just those tuples of S[BC] which match some tuple of R[AB] in the join.

3. Ship this semijoin back to node 1.

4. Compute

$$R[AB] \bowtie (S[BC] \ltimes R[AB])$$

at node 1. This value is equal to $R[AB] \bowtie S[BC]$.

- Observe that communication costs may have been reduced, because the whole of S[BC] did not have to be transmitted across the network.

- This must be balanced against the cost of shipping $\pi_B(R[AB])$ to node 2.