Physical Database Design

Chapter 16, Part A

Overview

- After ER design, schema refinement, and the definition of views, we have the *conceptual* and *external* schemas for our database.
- The next step is to choose indexes, make clustering decisions, and to refine the conceptual and external schemas (if necessary) to meet performance goals.
- We must begin by understanding the *workload*:
  - The most important queries and how often they arise.
  - The most important updates and how often they arise.
  - The desired performance for these queries and updates.

Understanding the Workload

- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
- For each update in the workload:
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

Decisions to Make

- What indexes should we create?
  - Which relations should have indexes? What field(s) should be the search key? Should we build several indexes?
- For each index, what kind of an index should it be?
  - Clustered? Hash/tree? Dynamic/static? Dense/sparse?
- Should we make changes to the conceptual schema?
  - Consider alternative normalized schemas? (Remember, there are many choices in decomposing into BCNF, etc.)
  - Should we ``undo'' some decomposition steps and settle for a lower normal form? (*Denormalization.*)
  - Horizontal partitioning, replication, views ...

Choice of Indexes

- One approach: consider the most important queries in turn. Consider the best plan using the current indexes, and see if a better plan is possible with an additional index. If so, create it.
- Before creating an index, must also consider the impact on updates in the workload!
  - Trade-off: indexes can make queries go faster, updates slower. Require disk space, too.

Issues to Consider in Index Selection

- Attributes mentioned in a WHERE clause are candidates for index search keys.
  - Exact match condition suggests hash index.
  - Range query suggests tree index.
    - Clustering is especially useful for range queries, although it can help on equality queries as well in the presence of duplicates.
- Try to choose indexes that benefit as many queries as possible. Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering.

Issues in Index Selection (Contd.)

- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
  - If range selections are involved, order of attributes should be carefully chosen to match the range ordering.
  - Such indexes can sometimes enable index-only strategies for important queries.
    - For index-only strategies, clustering is not important!
- When considering a join condition:
  - Hash index on inner is very good for Index Nested Loops.
    - Should be clustered if join column is not key for inner, and inner tuples need to be retrieved.
  - *Clustered* B+ tree on join column(s) good for Sort-Merge.

Example 1

```
SELECT E.ename, D.mgr
FROM Emp E, Dept D
WHERE D.dname='Toy' AND E.dno=D.dno
```

- Hash index on *D.dname* supports 'Toy' selection.
  - Given this, index on D.dno is not needed.
- Hash index on *E.dno* allows us to get matching (inner) Emp tuples for each selected (outer) Dept tuple.
- What if WHERE included: '' ... AND E.age=25'' ?
  - Could retrieve Emp tuples using index on *E.age*, then join with Dept tuples satisfying *dname* selection. Comparable to strategy that used *E.dno* index.
  - So, if *E.age* index is already created, this query provides much less motivation for adding an *E.dno* index.

Example 2

```
SELECT E.ename, D.mgr
FROM Emp E, Dept D
WHERE E.sal BETWEEN 10000 AND 20000
  AND E.hobby='Stamps' AND E.dno=D.dno
```

- Clearly, Emp should be the outer relation.
  - Suggests that we build a hash index on *D.dno.*
- What index should we build on Emp?
  - B+ tree on *E.sal* could be used, OR an index on *E.hobby* could be used. Only one of these is needed, and which is better depends upon the selectivity of the conditions.
    - As a rule of thumb, equality selections more selective than range selections.
- As both examples indicate, our choice of indexes is guided by the plan(s) that we expect an optimizer to consider for a query. *Have to understand optimizers!*

Examples of Clustering

```
SELECT E.dno
FROM Emp E
WHERE E.age>40
```

- B+ tree index on E.age can be used to get qualifying tuples.
  - How selective is the condition?
  - Is the index clustered?

```
SELECT E.dno, COUNT (*)
FROM Emp E
WHERE E.age>10
GROUP BY E.dno
```

- Consider the GROUP BY query.
  - If many tuples have *E.age* > 10, using *E.age* index and sorting the retrieved tuples may be costly.
  - Clustered *E.dno* index may be better!
- Equality queries and duplicates:
  - Clustering on *E.hobby* helps!

```
SELECT E.dno
FROM Emp E
WHERE E.hobby=Stamps
```


Clustering and Joins

```
SELECT E.ename, D.mgr
FROM Emp E, Dept D
WHERE D.dname='Toy' AND E.dno=D.dno
```

- Clustering is especially important when accessing inner tuples in INL.
  - Should make index on *E.dno* clustered.
- Suppose that the WHERE clause is instead:

  WHERE E.hobby='Stamps AND E.dno=D.dno
  - If many employees collect stamps, Sort-Merge join may be worth considering. A *clustered* index on D.dno would help.
- *Summary*: Clustering is useful whenever many tuples are to be retrieved.

Multi-Attribute Index Keys

- To retrieve Emp records with *age*=30 AND *sal*=4000, an index on <*age,sal*> would be better than an index on *age* or an index on *sal*.
  - Such indexes also called *composite* or *concatenated* indexes.
  - Choice of index key orthogonal to clustering etc.
- If condition is: 20<*age*<30 AND 3000<*sal*<5000:
  - Clustered tree index on <*age,sal*> or <*sal,age*> is best.
- If condition is: *age*=30 AND 3000<*sal*<5000:
  - Clustered <*age,sal*> index much better than <*sal,age*> index!
- Composite indexes are larger, updated more often.

Index-Only Plans

- A number of queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available.

<E.dno>

```
SELECT D.mgr
FROM Dept D, Emp E
WHERE D.dno=E.dno
```

<E.dno,E.eid> Tree index!

```
SELECT D.mgr, E.eid
FROM Dept D, Emp E
WHERE D.dno=E.dno
```

<E.dno>

```
SELECT E.dno, COUNT(*)
FROM Emp E
GROUP BY E.dno
```

<E.dno,E.sal> Tree index!

```
SELECT E.dno, MIN(E.sal)
FROM Emp E
GROUP BY E.dno
```

<E. age,E.sal> or *<E.sal, E.age> Tree!*

```
SELECT AVG(E.sal)
FROM Emp E
WHERE E.age=25 AND
  E.sal BETWEEN 3000 AND 5000
```


Summary

- Database design consists of several tasks: *requirements analysis, conceptual design, schema refinement, physical design* and *tuning.*
  - In general, have to go back and forth between these tasks to refine a database design, and decisions in one task can influence the choices in another task.
- Understanding the nature of the *workload* for the application, and the performance goals, is essential to developing a good design.
  - What are the important queries and updates? What attributes/relations are involved?

Summary (Contd.)

- Indexes must be chosen to speed up important queries (and perhaps some updates!).
  - Index maintenance overhead on updates to key fields.
  - Choose indexes that can help many queries, if possible.
  - Build indexes to support index-only strategies.
  - Clustering is an important decision; only one index on a given relation can be clustered!
  - Order of fields in composite index key can be important.
- Static indexes may have to be periodically re-built.
- Statistics have to be periodically updated.