Physical Database Design

Tamer Özsu

David R. Cheriton School of Computer Science
University of Waterloo

CS 640
Principles of Database Management and Use
Winter 2013

Notes

Outline

Notes

Physical Database Design and Tuning

Physical Design The process of selecting a physical schema (collection of data structures) to implement the conceptual schema

Tuning Periodically adjusting the physical and/or conceptual schema of a working system to adapt to changing requirements and/or performance characteristics

Good design and tuning requires understanding the database **workload**.

Notes

Workload Modeling

Definition (Workload Description)

A *workload description* contains

- the most important queries and their frequency
- the most important updates and their frequency
- the desired performance goal for each query or update

- For each query:
  - Which relations are accessed?
  - Which attributes are retrieved?
  - Which attributes occur in selection/join conditions? How *selective* is each condition?
- For each update:
  - Type of update and relations/attributes affected.
  - Which attributes occur in selection/join conditions? How *selective* is each condition?

Notes

The Physical Schema

- A storage strategy is chosen for each relation
  - Possible storage options:
    - Unsorted (heap) file
    - Sorted file
    - Hash file
- Indexes are then added
  - Speed up queries
  - Extra update overhead
  - Possible index types:
    - B-trees (actually, B+-trees)
    - R trees
    - Hash tables
    - ISAM, VSAM
    - ...

Notes

A Table Scan

```
select *
from Employee
where Lastname = 'Smith'
```

- To answer this query, the DBMS must search the blocks of the database file to check for matching tuples.
- If no indexes exist for Lastname (and the file is unsorted with respect to Lastname), all blocks of the file must be scanned.

Notes

Creating Indexes

```
create index LastnameIndex
on Employee(Lastname) [CLUSTER]

drop index LastnameIndex
```

Primary effects of `LastnameIndex`:

- Substantially reduce execution time for selections that specify conditions involving `Lastname`
- Increase execution time for insertions
- Increase or decrease execution time for updates or deletions of tuples from `Employee`
- Increase the amount of space required to represent `Employee`

Notes

Clustering vs. Non-Clustering Indexes

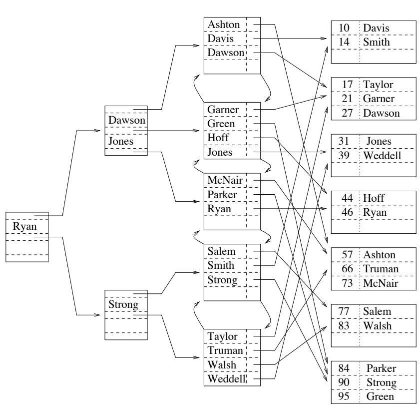
- An index on attribute A of a relation is a **clustering** index if tuples in the relation with similar values for A are stored together in the same block.
- Other indices are **non-clustering** (or secondary) indices.

Note

A relation may have at most one clustering index, and any number of non-clustering indices.

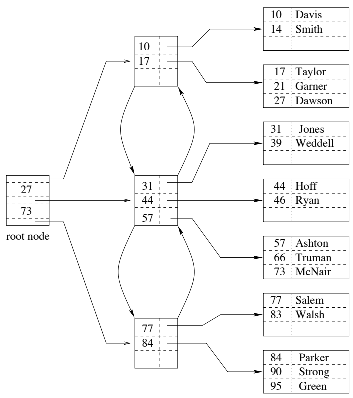
Notes

Non-Clustering Index Example



Notes

Clustering Index Example



Notes

Co-Clustering Relations

Definition (Co-Clustering)

Two relations are **co-clustered** if their tuples are interleaved within the same file

- Co-clustering is useful for storing hierarchical data (1:N relationships)
- Effects on performance:
  - Can speed up joins, particularly foreign-key joins
  - Sequential scans of either relation become slower

Notes

Range Queries

- B-trees can also help for **range queries**:

  select *
  from R
  where A $\geq$ c

- If a B-tree is defined on A, we can use it to find the tuples for which $A = c$. Using the forward pointers in the leaf blocks, we can then find tuples for which $A > c$.

Notes

Multi-Attribute Indices

- It is possible to create an index on several attributes of the same relation. For example:

```
create index NameIndex
on Employee(Lastname,Firstname)
```

- The order in which the attributes appear is important. In this index, tuples (or tuple pointers) are organized first by `Lastname`. Tuples with a common surname are then organized by `Firstname`.

Notes

Using Multi-Attribute Indices

- The `NameIndex` index would be useful for these queries:

```
select *
from Employee
where Lastname = 'Smith'
```

```
select *
from Employee
where Lastname = 'Smith'
and Firstname = 'John'
```

- It would be *very* useful for these queries:

```
select Firstname
from Employee
where Lastname = 'Smith'
```

```
select Firstname, Lastname
from Employee
```

- It would not be useful at all for this query:

```
select *
from Employee
where Firstname = 'John'
```


Notes

Physical Design Guidelines

1. Don't index unless the performance increase outweighs the update overhead
2. Attributes mentioned in WHERE clauses are candidates for index search keys
3. Multi-attribute search keys should be considered when
   - a WHERE clause contains several conditions; or
   - it enables index-only plans
4. Choose indexes that benefit as many queries as possible
5. Each relation can have at most one clustering scheme; therefore choose it wisely
   - Target important queries that would benefit the most
     - Range queries benefit the most from clustering
     - Join queries benefit the most from co-clustering
   - A multi-attribute index that enables an index-only plan does not benefit from being clustered

Notes