Cost of indexing

(courtesy : The University of Sydney)

Comparing File Organizations

- Heap files (random order; insert at eof)
- Sorted files, sorted on *<age, sal>*
- Clustered B+ tree file, Alternative (1), search key *<age, sal>*
- Heap file with unclustered B + tree index on search key *<age, sal>*
- Heap file with unclustered hash index on search key *<age, sal>*

Operations to Compare

- Scan: Fetch all records from disk
- Equality search
- Range selection
- Insert a record
- Delete a record

Cost Model for Our Analysis

We ignore CPU costs, for simplicity:

- **B:** The number of data pages
- **R:** Number of records per page
- **D:** (Average) time to read or write disk page
- Measuring number of page I/O's ignores gains of pre-fetching a sequence of pages; thus, even I/O cost is only approximated.
- Average-case analysis; based on several simplistic assumptions.

Assumptions in Our Analysis

- Heap Files:
  - Equality selection on key; exactly one match.
- Sorted Files:
  - Files compacted after deletions.
- Indexes:
  - Alt (2), (3): data entry size = 10% size of record
  - Hash: No overflow buckets.
    - 80% page occupancy => File size = 1.25 data size
  - Tree: 67% occupancy (this is typical).
    - Implies file size = 1.5 data size

Cost of Operations

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog $_2$B	Dlog $_2$ B + # matches	Search + BD	Search +BD
(3) Clustered	1.5BD	Dlog $_F$ 1.5B	Dlog $_F$ 1.5B + # matches	Search + D	Search +D
(4) Unclustered Tree index	BD(R+0.15)	D(1 + log $_F$ 0.15B)	Dlog $_F$ 0.15B + # matches	D(3 + log $_F$ 0.15B)	Search + 2D
(5) Unclustered Hash index	BD(R+0.125)	2D	BD	4D	Search + 2D

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.

Choice of Indexes

- 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?

Choice of Indexes (Contd.)

- 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.
  - Obviously, this implies that we must understand how a DBMS evaluates queries and creates query evaluation plans!
  - For now, we discuss simple 1-table queries.
- 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.

Index Selection Guidelines

- Attributes in WHERE clause are candidates for index keys.
  - Exact match condition suggests hash index.
  - Range query suggests tree index.
    - Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
  - Order of attributes is important for range queries.
  - Such indexes can sometimes enable index-only strategies for important queries.
    - For index-only strategies, clustering is not important!
- 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.

Choosing an Index

- An index should support a query of the application that has a significant impact on performance
  - Choice based on frequency of invocation, execution time, acquired locks, table size

Example 1:

SELECT E.*Id*
FROM Employee E
WHERE E.*Salary* < :upper AND E.*Salary* > :lower

– This is a range search on *Salary*.
– Since the primary key is *Id*, it is likely that there is a clustered, main index on that attribute that is of no use for this query.
– Choose a secondary, B^+ tree index with search key *Salary*

Choosing An Index (cont'd)

Example 2:

```
SELECT  E.sid
 FROM  Enrolled E
WHERE  E.grade = :grade
```

- This is an equality search on *grade*.
  - Since the primary key is (*sid, CourseId*) it is likely that there is a main, clustered index on these attributes
  - that is of no use for this query.
- Choose a secondary, B+ tree or hash index with search key *grade*

Choosing an Index (cont'd)

Example 3:

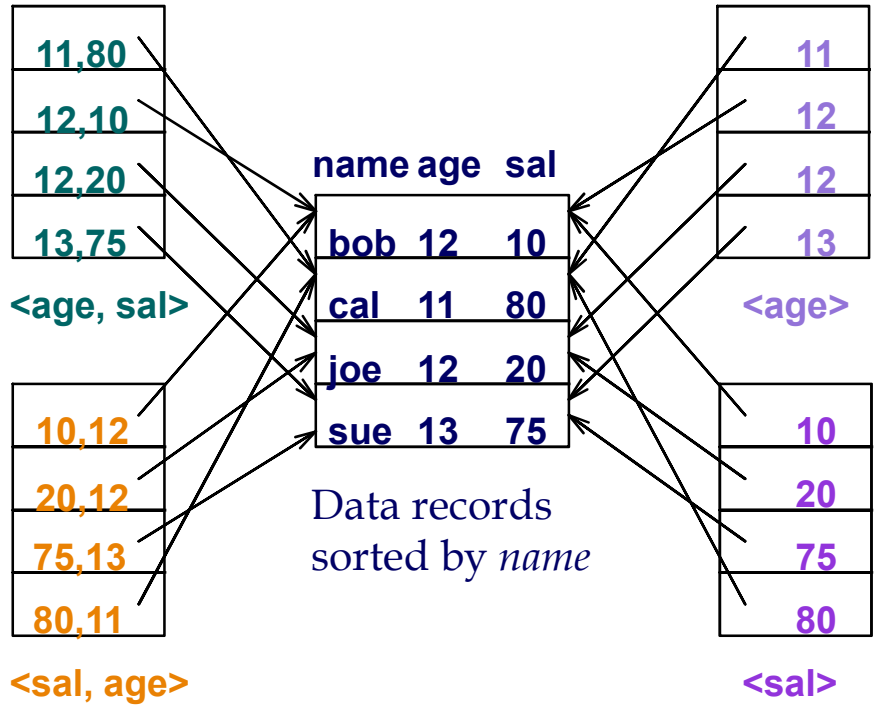
```
SELECT  E.CourseCode, E.grade
  FROM  Enrolled E
 WHERE  E.StudId = :sid AND E.grade = 'D'
```

- Equality search on *StudId* and *grade*.
- If the primary key is (*StudId, CourseId*) it is likely that there is a main, clustered index on this sequence of attributes.
  - If the main index is a B+ tree it can be used for this search.
  - If the main index is a hash it cannot be used for this search. Choose B+ tree or hash with search key *StudId* (since *grade* is not as selective as *StudId*) or (*StudId*, *grade*)

Indexes with Composite Search Keys

- *Composite Search Keys*: Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
    - age=20 and sal =75
  - Range query: Some field value is not a constant. E.g.:
    - age =20; or age=20 and sal > 10
- Data entries in index sorted by search key to support range queries.
  - Lexicographic order, or
  - Spatial order.

Examples of composite key indexes using lexicographic order.



Data entries in index sorted by <*sal,age*>

Data entries sorted by <*sal*>

Composite Search 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*.
  - 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
```

Index-Only Plans (Contd.)

- Index-only plans are possible if the key is <dno,age> or we have a tree index with key <age,dno>
  - Which is better?
  - What if we consider the second query?

```
SELECT  E.dno,  COUNT (*)
FROM  Emp E
WHERE  E.age=30
GROUP BY E.dno
```

```
SELECT  E.dno,  COUNT (*)
FROM  Emp E
WHERE  E.age>30
GROUP BY E.dno
```

Summary

- Many alternative file organizations exist, each appropriate in some situation.
- If selection queries are frequent, sorting the file or building an *index* is important.
  - Hash-based indexes only good for equality search.
  - Sorted files and tree-based indexes best for range search; also good for equality search. (Files rarely kept sorted in practice; B+ tree index is better.)
- Index is a collection of data entries plus a way to quickly find entries with given key values.

Summary (Contd.)

- Data entries can be actual data records, <key, rid> pairs, or <key, rid-list> pairs.
  - Choice orthogonal to *indexing technique* used to locate data entries with a given key value.
- Can have several indexes on a given file of data records, each with a different search key.
- Indexes can be classified as clustered vs. unclustered, primary vs. secondary, and dense vs. sparse. Differences have important consequences for utility/performance.

Summary (Contd.)

- 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?
- 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.