ICS 321 Fall 2009
Storage & Indexing (iii)

Asst. Prof. Lipyeow Lim
Information & Computer Science Department
University of Hawaii at Manoa

Creating Indexes

- Most DBMS (eg. DB2) supports only B+ tree indexes:

  CREATE INDEX myIdx **ON** mytable(col1, col3)
  CREATE UNIQUE INDEX myUniqIdx **ON** mytable(col2, col5)
  CREATE INDEX myIdx **ON** mytable(col1, col3) **CLUSTER**

- If a primary key is specified in the CREATE TABLE statement, an (unclustered) index is automatically created for the PK.
- To create a clustered PK index:
  – Create table without PK constraint
  – Create index on PK with cluster option
  – Alter table to add PK constraint
- To get rid of unused indexes: **DROP INDEX** myIdx;

Indexes & Performance Tuning

- Most DBMSs have very few knobs for storage – heap files are dominant
- In contrast, index creation is user-controlled
- What indexes should we create ?
  - Which relations should have indexes ?
  - Which columns should be indexed ?
  - How many indexes do we need ?
- What kind of indexes should we use ?
  - Clustered ?
  - Hash or Tree ?

Depends on the “Workload”

- For each query in the workload:
  - How frequent does this query occur ?
  - Which relations & attributes are accessed ?
  - Which attributes are involved in selection/join conditions ?
  - How selective are these conditions?
- For each update in the workload:
  - How frequent does the update occur ?
  - Which attributes are involved in selection/join conditions ?
  - How selective are these conditions?
  - What type of update (INSERT/DELETE/UPDATE) ?
  - Which relation & attributes are affected ?

Choosing 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.
  - 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, consider also 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.

Example: Clustered Indexes

- B+ tree index on E.age can be used to get qualifying tuples.
  - How selective is the condition?
  - Is the index clustered?
- 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.age>40
```

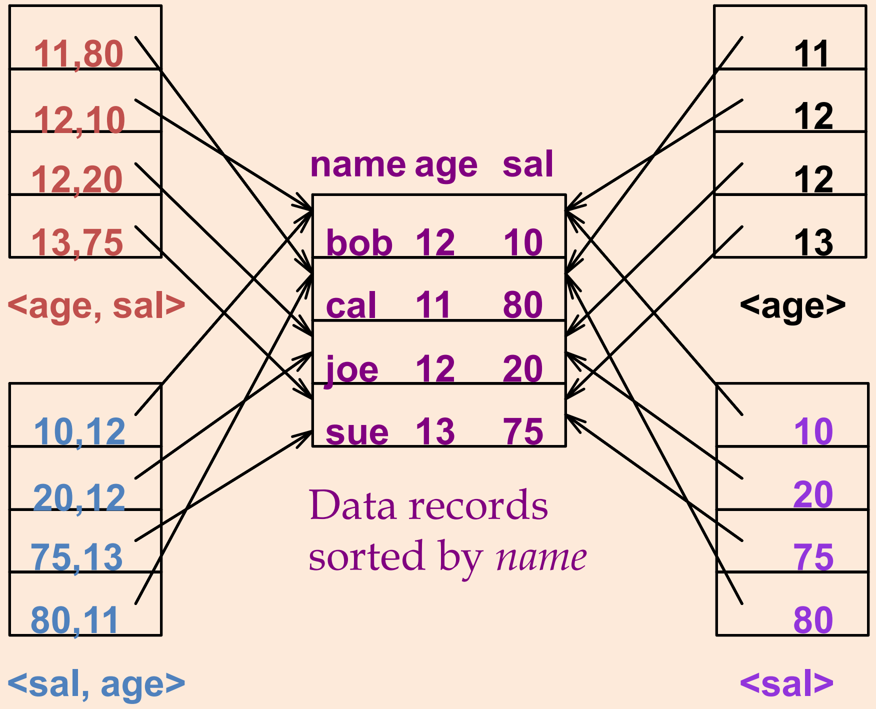
```
SELECT  E.dno,
            COUNT (*)
FROM  Emp E
WHERE  E.age>10
GROUP BY E.dno
```

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

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 records sorted by *name*

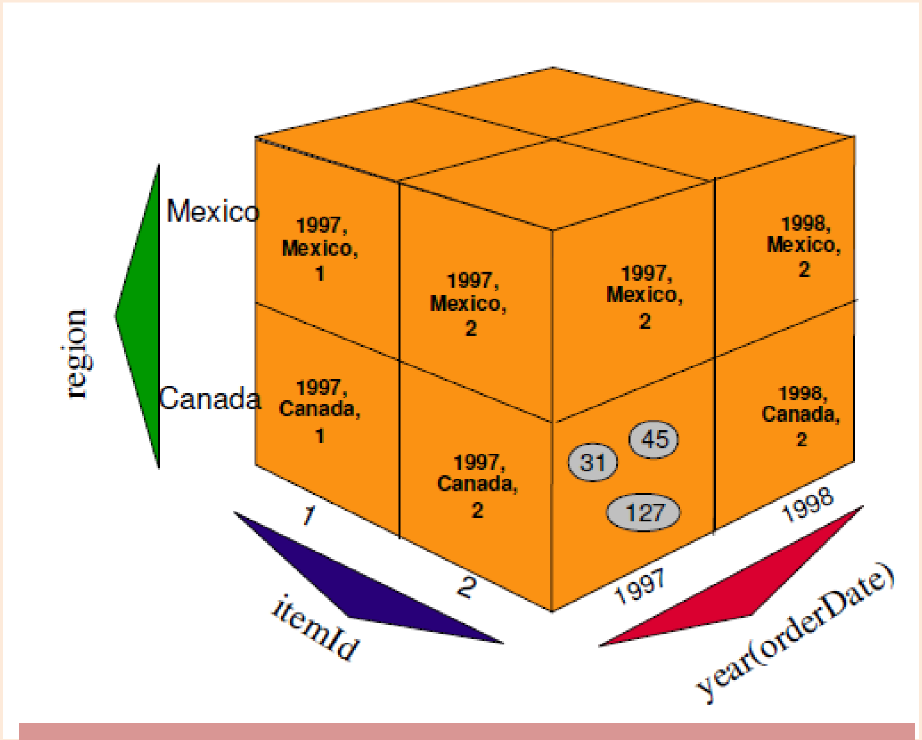
Data entries sorted by <*sal*>

Composite Search Keys

- **SELECT * FROM** Emp **WHERE** *age*=30 AND *sal*=4000,
  - an index on <*age,sal*> vs an index on *age* or an index on *sal*.
  - Choice of index key orthogonal to clustering etc.
- … **WHERE** 20<*age*<30 AND 3000<*sal*<5000:
  - Clustered tree index on <*age,sal*> or <*sal,age*> is best.
- … **WHERE** *age*=30 AND 3000<*sal*<5000:
  - Clustered <*age,sal*> index much better than <*sal,age*> index!
- Composite indexes are larger, updated more often.

Multi-Dimensional Clustering (MDC)

- DB2 v8 and above.
- Physical layout mimics a multi-dimensional cube
- Associates a physical region called blocks for each unique combination of dimension attribute values.
- These blocks are the units of addressability for our clusters.
- A block index that addresses these blocks.



```
CREATE TABLE Sales(
int storeId, date orderDate, int region,
int itemId, float price, int yearOd
generated always as year(orderDate))
ORGANIZE BY DIMENSIONS
(region, yearOd, itemId)
```

See *Multi-Dimensional Clustering: A New Data Layout Scheme* in DB2. SIGMOD 2003: 637-641

Index-Only Plans

- Query answered using index pages only
- Data pages are not retrieved at all
- In practice, such plans are rare, because of consistency issues.

<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 index!

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

REORG

- After many updates, a clustered index may become less and less clustered.
- The REORG TABLE command used to re-organized, ie, re-cluster the table and indexes.
- The REORGCHK command can be used to compute statistics relevant to making a decision on REORG.