Database Systems

October 28, 2009

Lecture #6

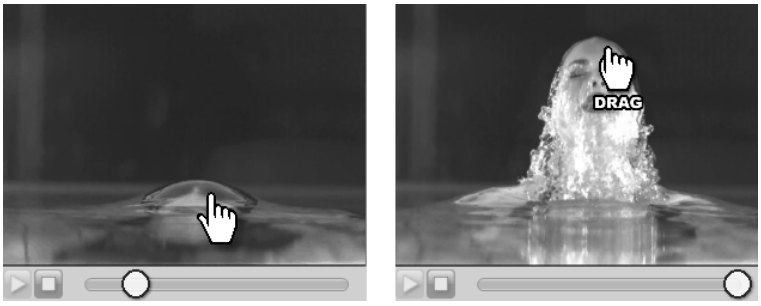
1

Course Administration

- Assignment #2: due next week

2

Video Browsing by Dragging Content (video, Toronto & INRIA; Aachen & Apple)



Reflection

- How to design a database?
  - Conceptual design: ER Model
  - Logical design: Relational Model
- How to ask questions about a database?
  - Relational Algebra & SQLs
- What's next?
- How to build a system that answers questions efficiently?
  - How to get fast access to records?
    - File organizations & indexes

4

Overview of Storage & Indexing

Chapter 8

5

Introduction

- Consider a worker database: name, age, salary
- How to store it in a file on a disk?
  - File organization
- What makes a good file organization?
  - What defines “good”?
  - Often need to retrieve data based on [name] alphabetical order
- What can be a potential problem with frequent updates?
  - How to solve this problem?

6

Outline

- Types of external storage devices
- 3 main types of file organizations
  - Heap file
  - Sorted file
  - Indexing data structures
    - Tree-based indexing, Hash-based indexing
- Comparison on file organizations
  - Which one is better/worse in performance?
- Indexes and Performance
  - How to use indexing for better performance?

7

Data on External Storage

- External Storage: offer persistent data storage
  - Unlike physical memory, data saved on a persistent storage is not lost when the system shutdowns or crashes.

8

Types of External Storage Devices

- Magnetic Disks: Can retrieve random page at fixed cost
  - NTD 4 per Gigabyte (2 T ~ NTD 8,000)
  - But reading several consecutive pages is much cheaper than reading them in random order
- Tapes: Can only write/read pages in sequence
  - NTD 10 per Gigabyte (old)
  - Cheaper than disks; used for archival storage
- Flash memory:
  - NTD 75 per Gigabyte (16 G ~ NTD 1,200)
- Other types of persistent storage devices:
  - Optical storage (CD-R, CD-RW, DVD-R, DVD-RW)

9

Definition

- A *record* is a tuple or a row in a relation table.
  - Fixed-size records or variable-size records
- A *file* is a collection of records.
  - Store one table per file, or multiple tables in the same file
- A *page* is a fixed length block of data for disk I/O.
  - A file is consisted of pages.
  - A data page also contains a collection of records.
  - Typical page sizes are 4 and 8 KB.

10

File Organization

- Method of arranging a file of records on external storage.
  - *Record id (rid)* is used to locate a record on a disk
    - [page id, slot number]
  - *Indexes* are data structures to efficiently search rids of given values

11

DB Storage and Indexing

- *Layered Architecture*
  - *Disk Space Manager* allocates/de-allocates spaces on disks.
  - *Buffer manager* moves pages between disks and main memory.
  - *File and index layers* organize records on files, and manage the indexing data structure.

12

Alternative File Organizations

- Many alternatives exist, *each ideal for some situations, and not so good in others:*
  - *Heap files*: scan retrie
    - Fast up
  - *Sorted Files*: Records are sorted. Best if records must be retrieved in some order, or only a `range' of records is needed.
    - Examples: employees are sorted by age.
    - Slow update in comparison to heap file.
  - *Indexes*: Data structures to organize records via trees or hashing.
    - For example, create an index on employee age.
    - Like sorted files, speed up searches for a subset of records that match values in certain ("search key") fields
    - Updates are much faster than in sorted files.

13

Alternative File Organizations

- Many alternatives exist, *each ideal for some situations, and not so good in others:*
  - *Heap files*: Records are unsorted. Suitable when typical access is a file scan retrieving all records without any order.
    - Fast update (insertions / deletions)
  - *Sorted Files*: Records are sorted. Best if records must be retrieved in some order, or only a `range' of records is needed.
    - Examples: employees are sorted by age.
    - Slow update in comparison to heap file.
  - *Indexes*: Data structures to organize records via trees or hashing.
    - For example, create an index on employee age.
    - Like sorted files, speed up searches for a subset of records that match values in certain ("search key") fields
    - Updates are much faster than in sorted files.

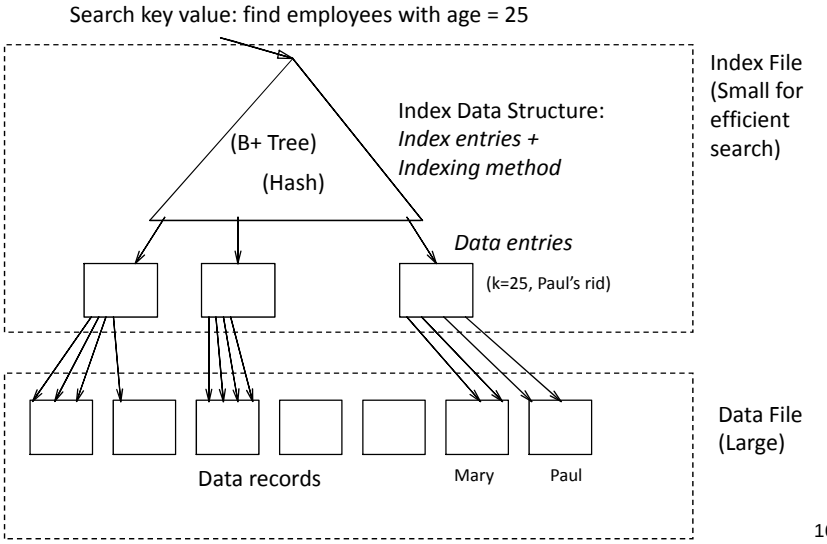
14

Indexes

- An index speeds up search (on key fields)
  - Any subset of the attributes can be the search key for an index
  - Search key does not have to be candidate key
    - Example: employee age is not a candidate key.
- An index file contains a collection of data entries (called **k*).**
  - Quickly search an index to locate a data entry with a key value k.
    - Example of a data entry: <age, rid>
  - Can use the data entry to find the data record.
    - Example of a data record: <name, age, salary>
  - Can create multiple indexes on the same data records.
    - Example indexes: age, salary, name

15

Indexing Example



16

Alternatives for Data Entry **k***

- Three alternatives for a data entry:
  - (Alternative 1): Data record with key value **k**
    - Example data record = data entry: <**age**, name, salary>
  - (Alternative 2): <**k**, rid of data record with search key value **k**>
    - Example data entry: <**age**, rid>
  - (Alternative 3): <**k**, list of rids of data records with search key **k**>
    - Example data entry: <**age**, rid_1, rid_2, …>
- Data entry != indexing method
  - Indexing methods: B+ tree, hashing, etc.
  - Indexing method takes a search key and finds the data entries matching the search key.

17

Alternatives for Data Entries (Contd.)

- Alternative 1: data record with key value **k**
  - Data entries are also the data records.
  - At most one index on a given collection of data records can use Alternative 1.
  - If data records are very large, # of pages containing data entries is high.
    - *May* lead to less efficient search.

18

Alternatives for Data Entries (Contd.)

- Alternative 2: <**k**, rid with key value **k>**
- Alternative 3: <**k**, rid-list of data record(s)>
  - Data entries typically much smaller than data records.
    - May lead to more efficient search than Alternative 1. Why?
  - Alternative 3 more compact than Alternative 2,
    - Lead to variable sized data entries (size of rid-list is not fixed)

19

Index Classification

- Clustered vs. unclustered: If order of data records is the same as, or close to the order of data entries, then it is called clustered index.
  - Alternative 1 implies clustered; in practice, clustered also implies Alternative 1.
  - One clustered index and multiple unclustered indexes
  - Why is this important?
    - Consider the cost of range search query: find all records 30<age<39 [next slide]

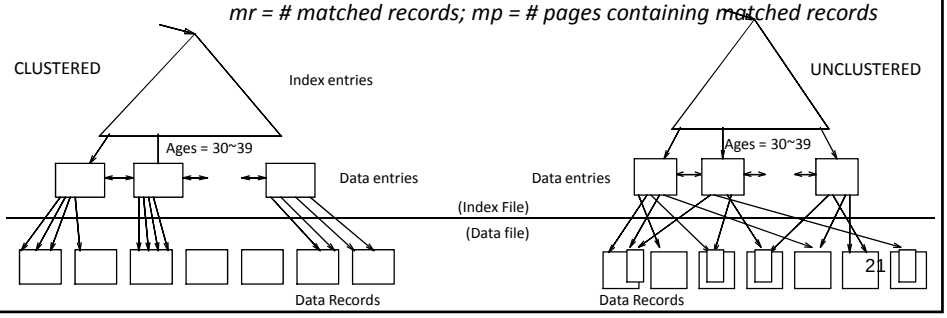
20

Clustered vs. Unclustered Index

- Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!
- Examples: retrieve all the employees of ages 30~39.
  - What is the worst-case cost (# disk page I/Os) of clustered index?
  - What is the worst-case cost of unclustered index?

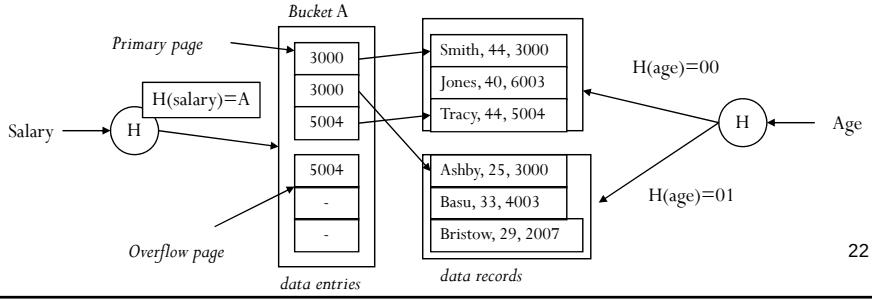
Cost = number of pages retrieved

mr = # matched records; mp = # pages containing matched records



Hash-Based Indexes

- Good for equality selections.
  - Data entries (key, rid) are grouped into buckets.
  - Bucket = *primary* page plus zero or more *overflow* pages.
  - *Hashing function* **h**: **h**(*r*) = bucket in which record *r* belongs. **h** looks at the *search key* fields of *r.*
  - If Alternative (1) is used, the buckets contain the data records.

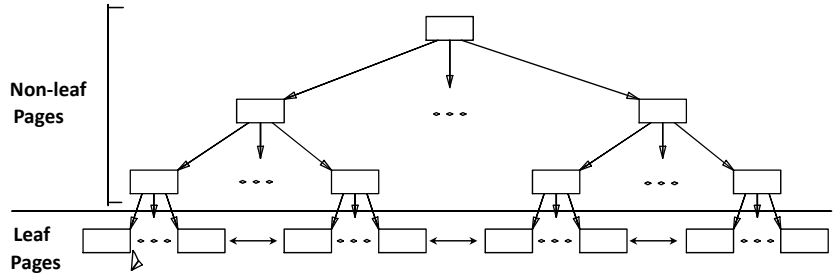


Hash-based Indexes (Cont)

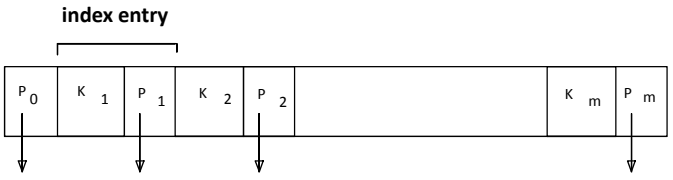
- Search on key value:
  - Apply key value to the hash function → bucket number
  - Retrieve the primary page of the bucket. Search records in the primary page. If not found, search the overflow pages.
  - Cost of locating rids: # pages in bucket (small)
- Insert a record:
  - Apply key value to the hash function → bucket number
  - If all (primary & overflow) pages in that bucket are full, allocate a new overflow page.
  - Cost: similar to search.
- Delete a record
  - Cost: similar to search.

23

B+ Tree Indexes

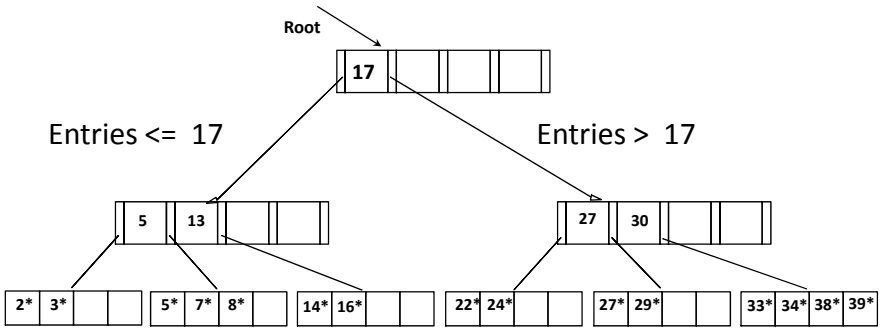


Leaf pages contain data entries, and are chained (prev & next)
Non-leaf pages contain index entries and direct searches:



24

Example B+ Tree



- Find 7*, 29*? 15* < age < 30*
- Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.
  - And change sometimes bubbles up the tree (keep the tree balance)
- More details about tree-based index in Chapter 10.

25

Cost Model for Our Analysis

- Ignore CPU costs, for simplicity.
  - Per instruction time: < 1 nanosecond (10^{-9})
  - Per disk access time: 10 millisecond (10^{-2})
- Measure disk I/O costs: number of page I/O's
  - Ignores gains of pre-fetching a sequence of pages
- Cost analysis
  - ***B*:** The number of data pages
  - ***R*:** Number of records per page

** Good enough to show the overall trends!*

26

Comparing File Organizations

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

27

Operations to Compare

- Scan: Fetch all records from disk
- Equality search
  - Example: find all employees with age = 23 and salary = 5000.
- Range selection
  - Example: find all employees with age > 35.
- Insert a record
  - Identify the page for inserting the record, fetch it, modify it, and write it back.
- Delete a record
  - Similar to insert.

28

Assumptions in Our Analysis

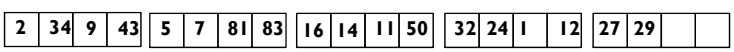
- Heap Files:
  - Equality selection on key; exactly one match.
- Sorted Files:
  - Files compacted after deletions.
- Indexes:
  - Alt (2), (3): data entry size = 10% size of data record

29

Heap Files

B: *The number of data pages*
R: *Number of records per page*

- Scan:
- Search with equality selection:
- Search with range selection:
- Insert:
  - New record is inserted at the end of the file. Read/write out last page.
- Delete a record: (no compacting)
- Delete a record (with rid):
  - search cost = *1*

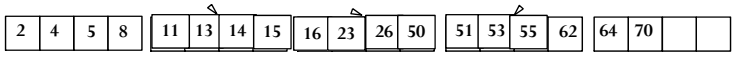


30

Sorted Files

B: The number of data pages
R: Number of records per page

- Scan:
- Search with equality selection:)
  - Binary search
- Search with range selection: *records*
  - *14<x<60*: Search for first matching record / page, then find all the qualifying records / pages in sequential order.
- Insert:
  - Find the right position/page (search), make space for the inserting record by shifting all subsequent records by one slot, then insert.
- Delete:
  - Search the record, delete it, shift all subsequent records by one slot.

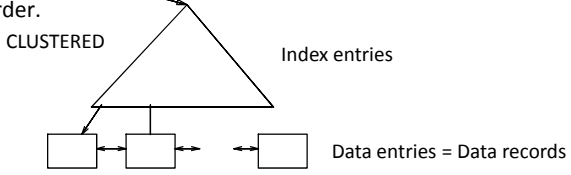


Clustered B+ Tree File

Un-occupancy in clustered file ~ 0.5 B

B: The number of data pages
R: Number of records per page

- Scan:
- Search with equality selection:
  - *F* is number of children per B+ tree node
- Search with range selection: *records*
  - Search for first matching record (page), then find all the qualifying records / pages in sequential order.

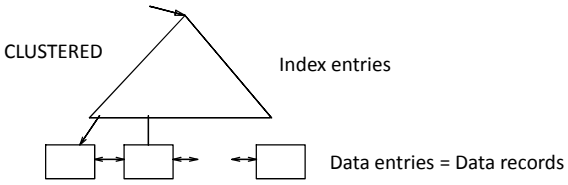


Clustered B+ Tree File

B: The number of data pages

R: Number of records per page

- Insert:
  - Find the right position (page), insert + write out the modified page. No need to shift records -> empty data entries in modified page.
- Delete:
  - Search record, delete record, and write back modified page. No need to shift.



33

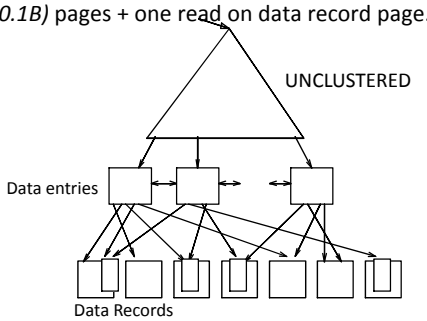
Heap File with Unclustered Tree Index

Size of Data/index entry ~ 10% of size of data record

B: The number of data pages

R: Number of records per page

- Scan (in-order):
  - Unorder scan: *B*
- Search with equality selection:
  - Search for data entry takes $log_F(0.1B)$ pages + one read on data record page.



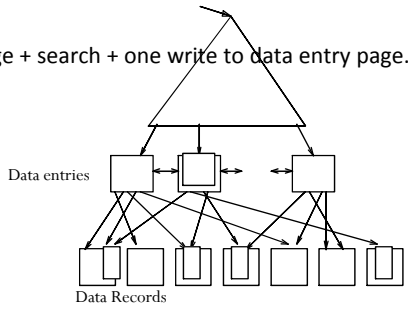
34

Heap File with Unclustered Tree Index

B: The number of data pages

R: Number of records per page

- Search with range selection:
  - Search for first matching data entry, then find all the qualifying entries in sequential order. But each data entry may point to a data record on a different data page.
- Insert:
  - One read/write to heap file page + search + one write to data entry page.
- Delete:



35

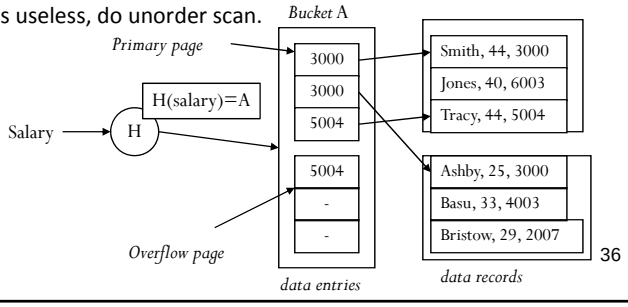
Heap File with Unclustered Hash Index

Hash: No overflow buckets. 80% page occupancy => # data entry pages = 0.125B

B: The number of data pages

R: Number of records per page

- Scan (in-order):
  - Unorder scan: *B*
- Search with equality selection:
  - Hash function + read data entry page + read data record page
- Search with range selection:
  - Hash function is useless, do unorder scan.



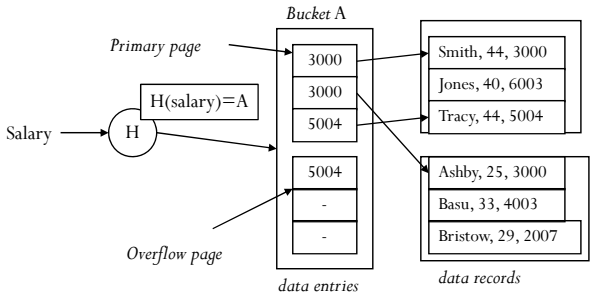
36

Heap File with Unclustered Hash Index

Hash: No overflow buckets. 80% page occupancy => # data entry pages = 0.125B

B: The number of data pages
R: Number of records per page

- Insert:
  - Read & Write Heap file page + Read & Write data entry page
- Delete:



37

Cost of Operations

	(a) Scan	*(b) Equality*	*(c) Range*	*(d) Insert*	*(e) delete*
(1) Heap	*B*	*0.5B*	*B*	*2*	*Search + 1*
(2) Sorted	*B*	*$log_2(B)$*	*$log_2(B)$ + #matches*	*Search + B*	*Search + B*
(3) Clustered Tree Index	*1.5 B*	*$log_F(1.5B)$*	*$log_F(1.5B)$ + #matches*	*Search + 1*	*Search + 1*
(4) Unclustered Tree index	*B(R+0.1)*	*$1+log_F(0.1B)$*	*$log_F(0.1B)$ + #matches*	*Search + 3*	*Search + 3*
(5) Unclustered Hash Index	*B(R+0.125)*	*2*	*B*	*4*	*4*

**No one file organizations is uniformly superior*

38

General Guidelines

- An index supports efficient retrieval of data entries satisfying a selection condition:
  - Two types of selections: *equality* and *range*
- What is Hash-based indexing optimized for?
  - equality selection, useless for range selection.
- What is Tree-based indexing good/best for?
  - Better for both.
  - Tree-based clustering index is best for range selection.

39

General Guidelines (Cont)

- Clustered index can be more expensive than unclustered index:
  - When inserting a new record into a full page, shift existing records into other pages → change data entries for these records → expensive.
  - Tradeoff for more efficient range selection.

40

Understanding the Workload

- How to decide the best indexing for a table?
  - Need to understand the workload
- For each query in the workload:
  - Which tables does it access?
  - Which fields are retrieved?
  - Which fields are involved in selection/join conditions?
  - How selective are these conditions likely to be?
- For each update in the workload:
  - Which fields are involved in selection/join conditions?
  - How selective are these conditions likely to be?
  - The type of update *(INSERT/DELETE/UPDATE*), and the fields that are affected.

41

Choice of Indexes

- What indexes should we create?
  - Which tables 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 or unclustered?
  - Hash index or Tree index?

42

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 (because also have to update the indexes). Indexes also require disk space, too.

43

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.

44

Examples of Clustered Indexes

- What index would you create for what fields?
- B+ tree index on `E.age` can be used to get qualifying records.
  - How selective is the condition? (selective means % of qualified records)
  - Is this index useful?
- Consider the `GROUP BY` query.
  - Is `E.age` index good? Why not?
  - Clustered `E.dno` index may be better.
- Equality queries and duplicates:
  - Unclustering is bad in case of many qualified records.
  - Clustering on *E.hobby* helps!

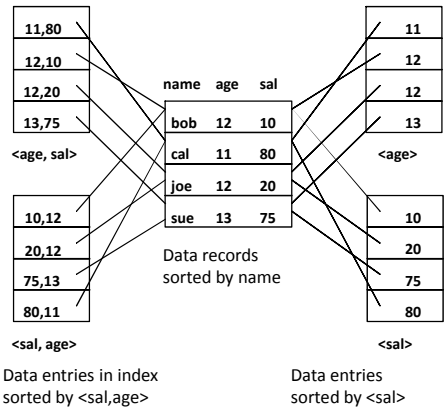
```
SELECT  E.dno
FROM  Emp E
WHERE  E.age>40

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
```

45

Composite Search Keys

- Search on a combination of fields.
- Which index can be applied?
  - Equality query: Every field value is equal to a constant value.
    - `age=12 and sal =10`
  - Range query: Some field value is not a constant.
    - `age =13`; or `sal=10 and age > 5`
  - The order of fields in composite key is important!
    - <sal, age>: data entries are sorted by sal first, then age.

Examples of composite key indexes using lexicographic order.



46

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.
- If condition is: `20<age<30 AND 3000<sal<5000`:
  - Clustered tree index on <*age,sal*> or <*sal,age*>.
- If condition is: `age=30 AND 3000<sal<5000`:
  - Clustered <*age,sal*> index much better than <*sal,age*> index. Why?
    -
  - The order of fields in composite key is important!
- Composite indexes are larger, updated more often.

47

Index-Only Plans

- A number of queries can be answered without retrieving any records from one or more of the relations involved if a suitable index is available.
- What index is needed for index-only evaluation?

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

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

48

Index-Only Evaluation (Contd.)

- What index is needed for index-only evaluation?
  - *<dno,age>* or *<age,dno>*
- Consider selective-ness of condition vs. cost of sorting
  - Selective: *<age, dno>*
  - Not selective: *<dno, age>*

```
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
```

49

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.
- Index is a collection of data entries plus a way to quickly find entries with given key values.

50

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. Differences have important consequences for utility/performance.

51

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 fields/relations are involved?
- Indexes must be chosen to speed up important queries (and perhaps some updates!).
  - Index maintenance overhead on updates to key fields.
  - 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.

52