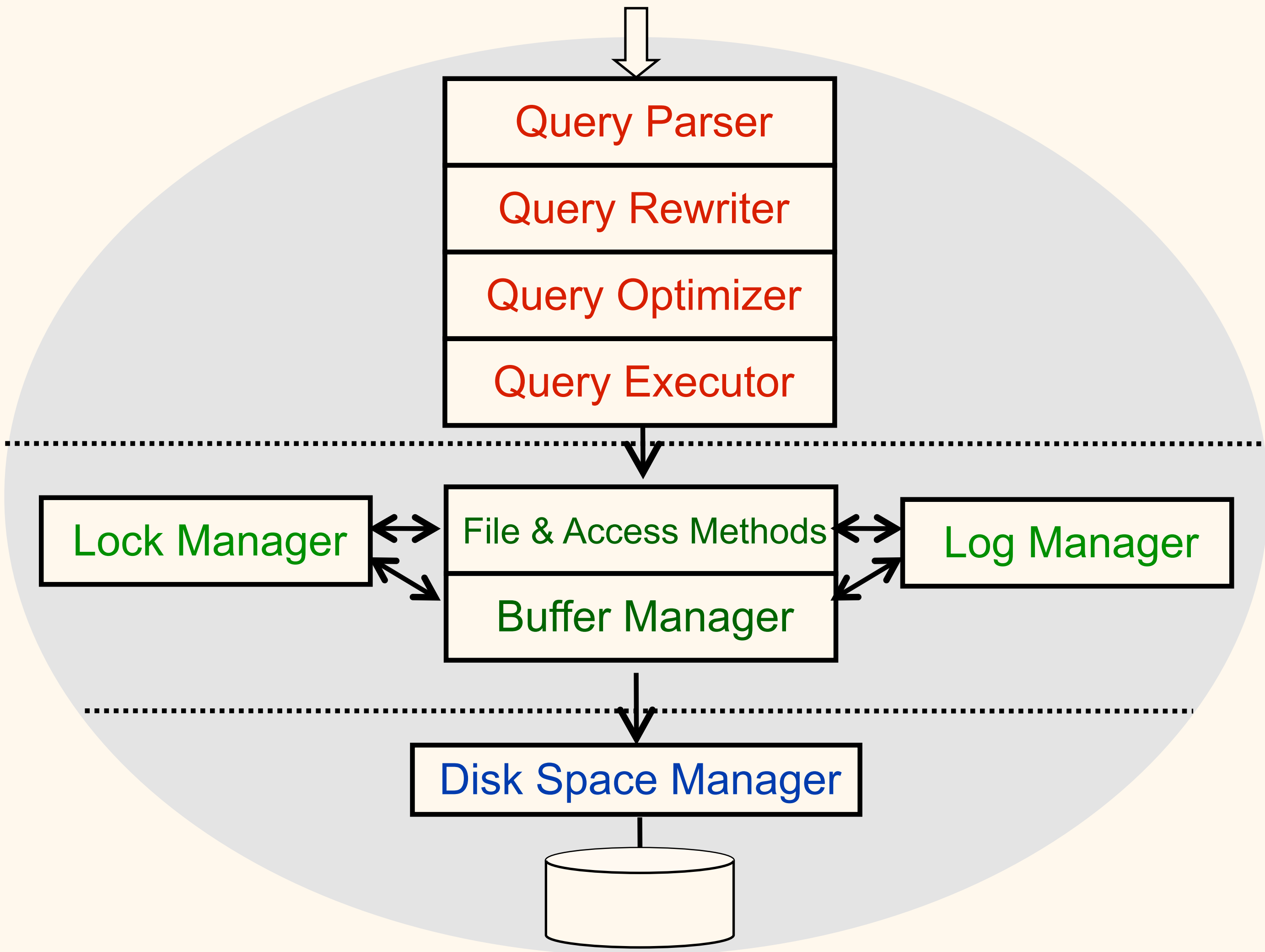
Overview of Storage and Indexing

Fall 2008

DBMS Architecture



Data on External Storage

- Disks: Can retrieve random page at fixed cost
  - But reading several consecutive pages is much cheaper than reading them in random order
- Tapes: Can only read pages in sequence
  - Cheaper than disks; used for archival storage
- Page: Unit of information read from or written to disk
  - Size of page: DBMS parameter, 4KB, 8KB
- Disk space manager:
  - Abstraction: a collection of pages.
  - Allocate/de-allocate a page.
  - Read/write a page.
- Page I/O:
  - Pages read from disk and pages written to disk
  - Dominant cost of database operations

Buffer Management

- Architecture:
  - Data is read into memory for processing
  - Data is written to disk for persistent storage
- Buffer manager stages pages between external storage and main memory buffer pool.
- Access method layer makes calls to the buffer manager.

Access Methods

- Access methods: routines to manage various disk-based data structures.
  - Files of records
  - Various kinds of indexes
- File of records:
  - Important abstraction of external storage in a DBMS!
  - Record id (rid) is sufficient to physically locate a record
- Indexes:
  - Auxiliary data structures
  - Given values in index search key fields, find the record ids of records with those values

File organizations & access methods

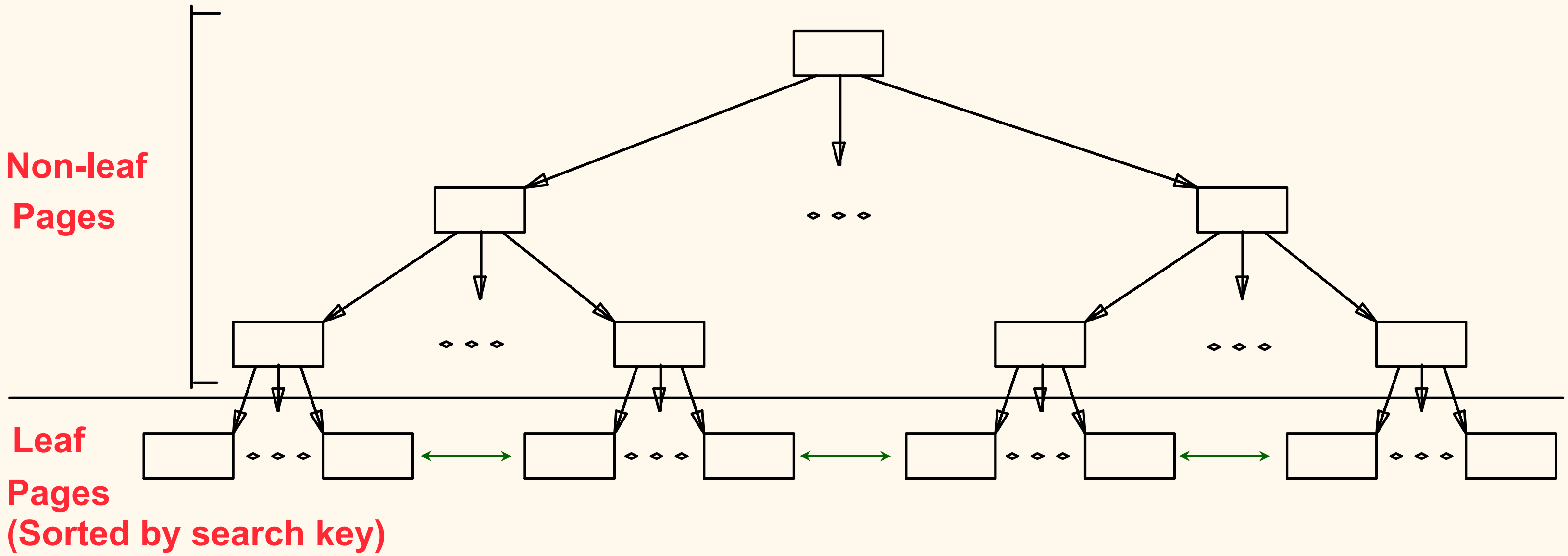
Many alternatives exist, *each ideal for some situations, and not so good in others:*

- Heap (unordered) files: Suitable when typical access is a file scan retrieving all records.
- Sorted Files: Best if records must be retrieved in some order, or only a `range' of records is needed.
- Indexes: Data structures to organize records via trees or hashing.
  - Like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields
  - Updates are much faster than in sorted files.

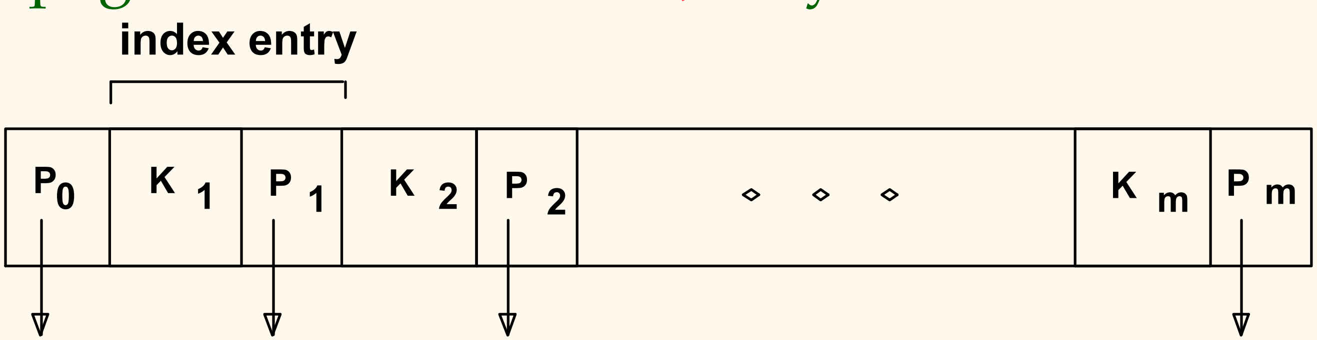
Indexes

- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).
- An index contains a collection of *data entries*, and supports efficient retrieval of all data entries **k*** with a given key value **k**.
  - Data entry versus data record.
  - Given data entry k*, we can find record with key k in at most one disk I/O. (Details soon …)

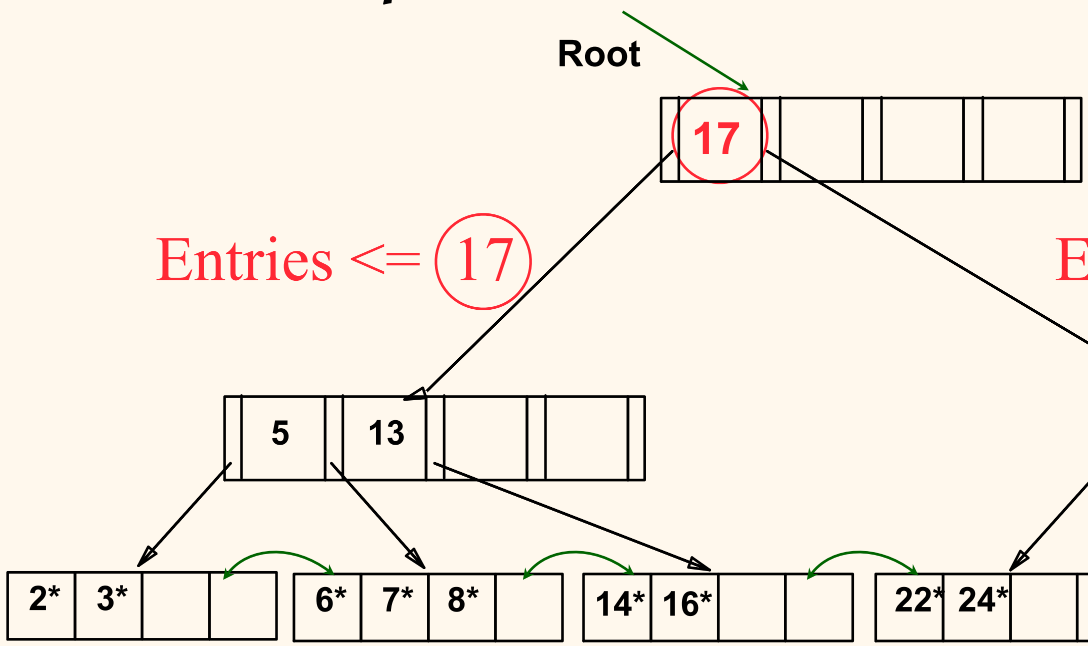
B+ Tree Indexes



- Leaf pages contain *data entries*, and are chained (prev & next)
- Non-leaf pages have *index entries;* only used to direct searches:



Example B+ Tree

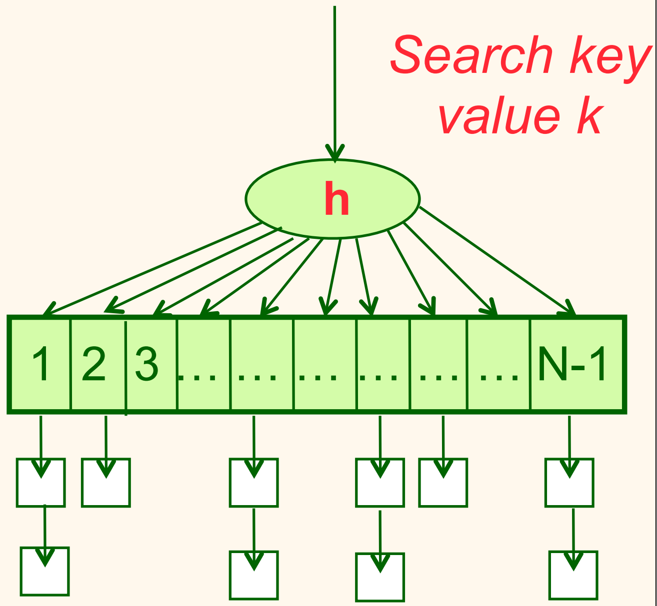


Note how data entries in leaf level are sorted

- Equality selection: find 28*? 29*?
- Range selection: find all > 15* and < 30*
- Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.
  - And change sometimes bubbles up the tree

Hash-Based Indexes

- Good for equality selections.
- Index is a collection of *buckets.*
  - Bucket = *primary* page plus zero or more *overflow* pages.
  - Buckets contain data entries.
- *Hashing function* **h**: **h**(*k*) = bucket of data entries of the search key value k.
  - *No need for "index entries" in this scheme.*



Alternatives for Data Entry k in Index*

- In a data entry k* we can store:
  - Alt1: Data record with key value **k**
  - Alt2: <**k**, rid of data record with search key value **k**>
  - Alt3: <**k**, list of rids of data records with search key **k**>
- Choice of <u>alternative for data entries</u> is orthogonal to <u>indexing technique used to locate data entries</u> with a key value **k**.
  - Indexing techniques: B+ tree index, hash index
  - Typically, indexes contain auxiliary information that directs searches to the desired data entries

Alternatives for Data Entries (Contd.)

- Alternative 1:
  - Index structure is a file organization for data records (instead of a Heap file or sorted file).
  - At most one index on a given collection of data records can use Alternative 1.
    - Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.
  - If data records are very large, # of pages containing data entries is high.
    - Implies size of auxiliary information in the index is also large, typically (e.g., B+ tree).

Alternatives for Data Entries (Contd.)

- Alternatives 2 and 3:
  - Data entries, with search keys and rid(s), typically much smaller than data records.
    - Index structure used to direct search, which depends on size of data entries, is much smaller than with Alternative 1.
    - So, better than Alternative 1 with large data records, especially if search keys are small.
  - Alternative 3 more compact than Alternative 2
  - Alternative 3 leads to variable sized data entries, even if search keys are of fixed length.
    - Variable sizes records/data entries are hard to manage.

Index Classification

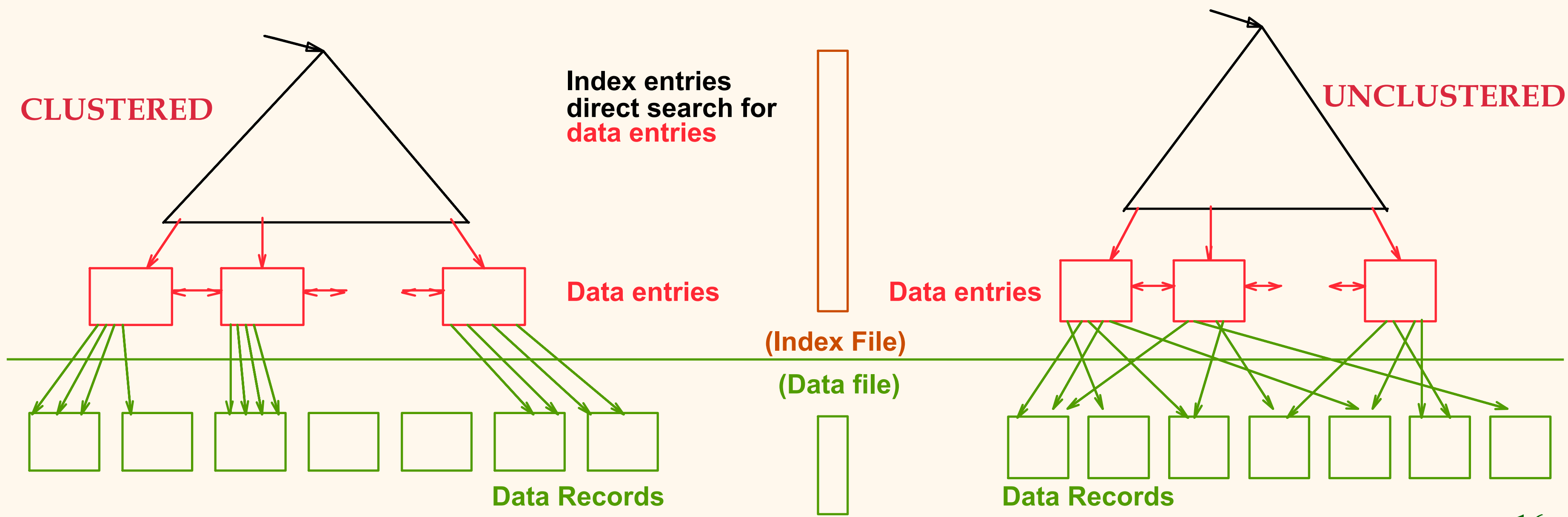
- Key? Primary key? Candidate key?
- *Primary* index vs. *secondary* index:
  - If search key contains primary key, then called primary index.
  - Other indexes are called secondary indexes.
- *Unique* index: Search key contains a candidate key.
  - No data entries can have the same value.

Index Classification (Contd.)

- *Clustered* vs. *unclustered*: If order of data records is the same as (or `close to'), order of data entries, then it's a clustered index.
  - Alternative 1 implies clustered
  - Alternatives 2 and 3 are clustered only if data records are sorted on the search key field.
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!

Clustered vs. Unclustered Index

- Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.
  - To build clustered index, first sort the Heap file (with some free space on each page for future inserts).
  - Overflow pages may be needed for inserts. (Thus, order of data recs is `close to', but not identical to, the sort order.)



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.

☛ *Good enough to show the overall trends!*

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

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 chains.
    - 80% page occupancy => File size = 1.25 data size
  - B+Tree:
    - 67% occupancy (typical): implies file size = 1.5 data size
    - Balanced with fanout F (133 typical) at each non-level

Assumptions (contd.)

- Scans:
  - Leaf levels of a tree-index are chained.
  - Index data-entries plus actual file scanned for unclustered indexes.
- Range searches:
  - We use tree indexes to restrict the set of data records fetched, but ignore hash indexes.

Cost of Operations

	Scan	Equality	Range	Insert	Delete
Heap File	BD	.5BD	BD	2D	Search + D
Sorted File	BD	$D\log_2 B$	$D(\log_2 B$ + #matching pages)	Search + BD	Search + BD
Clustered Tree Index	1.5BD	$D\log_F 1.5B$	$D(\log_F 1.5B$ + #matching pages)	Search + D	Search + D
Unclustered Tree Index	BD(R+.15)	$D(1+\log_F .15B)$	$D(\log_F .15B$ + #matching **recs**)	Search + 3D	Search + 3D
Unclustered Hash Index	BD(R+.125)	2D	BD	4D	4D

☛ *Several assumptions underlie these (rough) estimates!*

Index selection

- 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:
  - 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?
  - Hash/tree?
  - Clustered?

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.
  - 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 - possible index search keys.
  - Exact match condition suggests <u>hash index</u>.
  - Range query suggests <u>tree index</u>.
    - 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!
- 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.

Examples of Clustered Indexes

```
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?

Examples of Clustered Indexes

Compare index on <*age*>, index on <*dno*>.

```
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 by *E.dno* may be costly.
  - Clustered *E.dno* index may be better!

Examples of Clustered Indexes

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

- Equality queries and duplicates:
  - Clustering on *E.hobby* helps!

Index-Only Plans

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

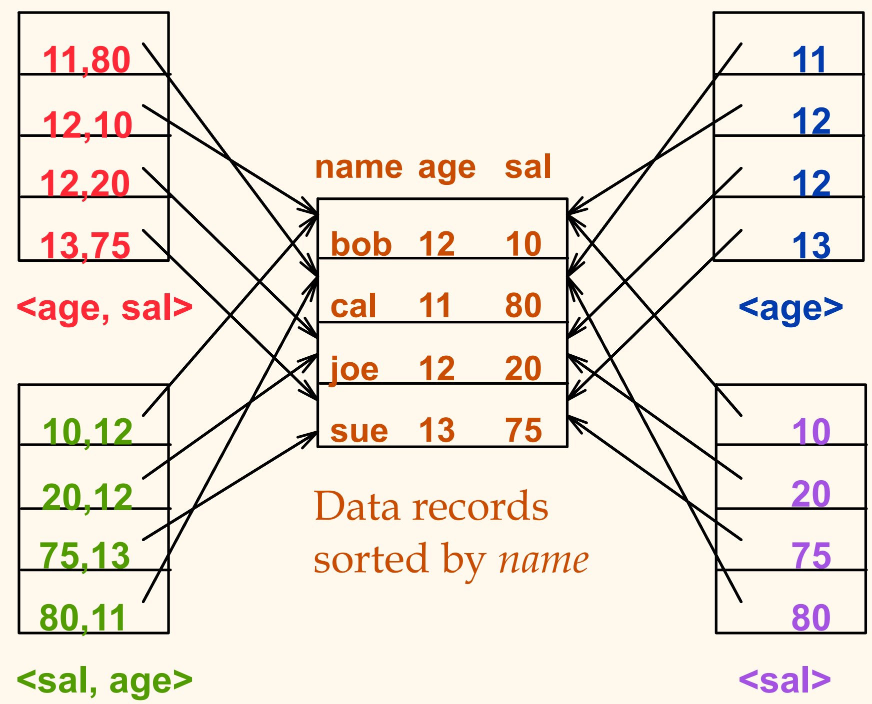
<*E.dno*>

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

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

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*.
- 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

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

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

What about <*E.sal,E.dno*> ?

Index-Only Plans

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

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

Creating indexes in SQL

- SQL:1999 standard does not include any statement for creating or dropping index structures!

```
CREATE INDEX age_index
USING BTREE ON Emp(age)
```

Summary

- 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.