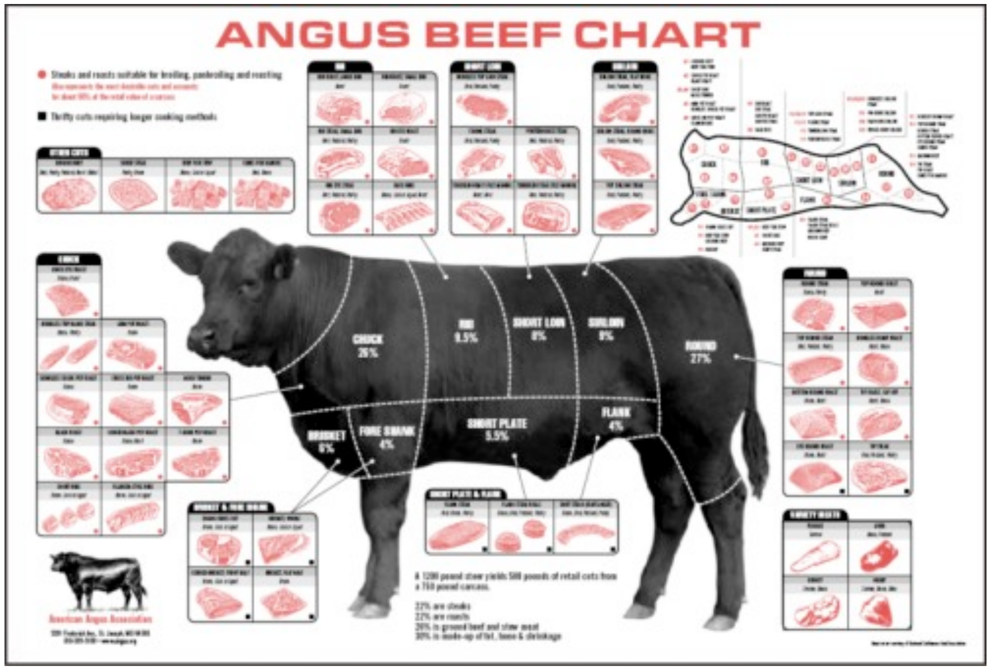
Overview of Storage and Indexing

Chapter 8



Data on External Storage

- Disks (and SSDs): Can retrieve random *page* at fixed cost
  - But *reading consecutive pages is much cheaper* than reading them in random order
- Tapes: Can only read pages sequentially
  - Cheaper than disks; used for archival storage
- File organization: Method of arranging a file of records on external storage.
  - Record id (rid) is sufficient to physically locate record
  - Indexes are data structures that allow us to find the record ids of records with given values in index search key fields
- Architecture: Buffer manager stages pages from external storage to main memory buffer pool. File and index layers make calls to the buffer manager.

Alternative File Organizations

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

- Heap (random order) 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.

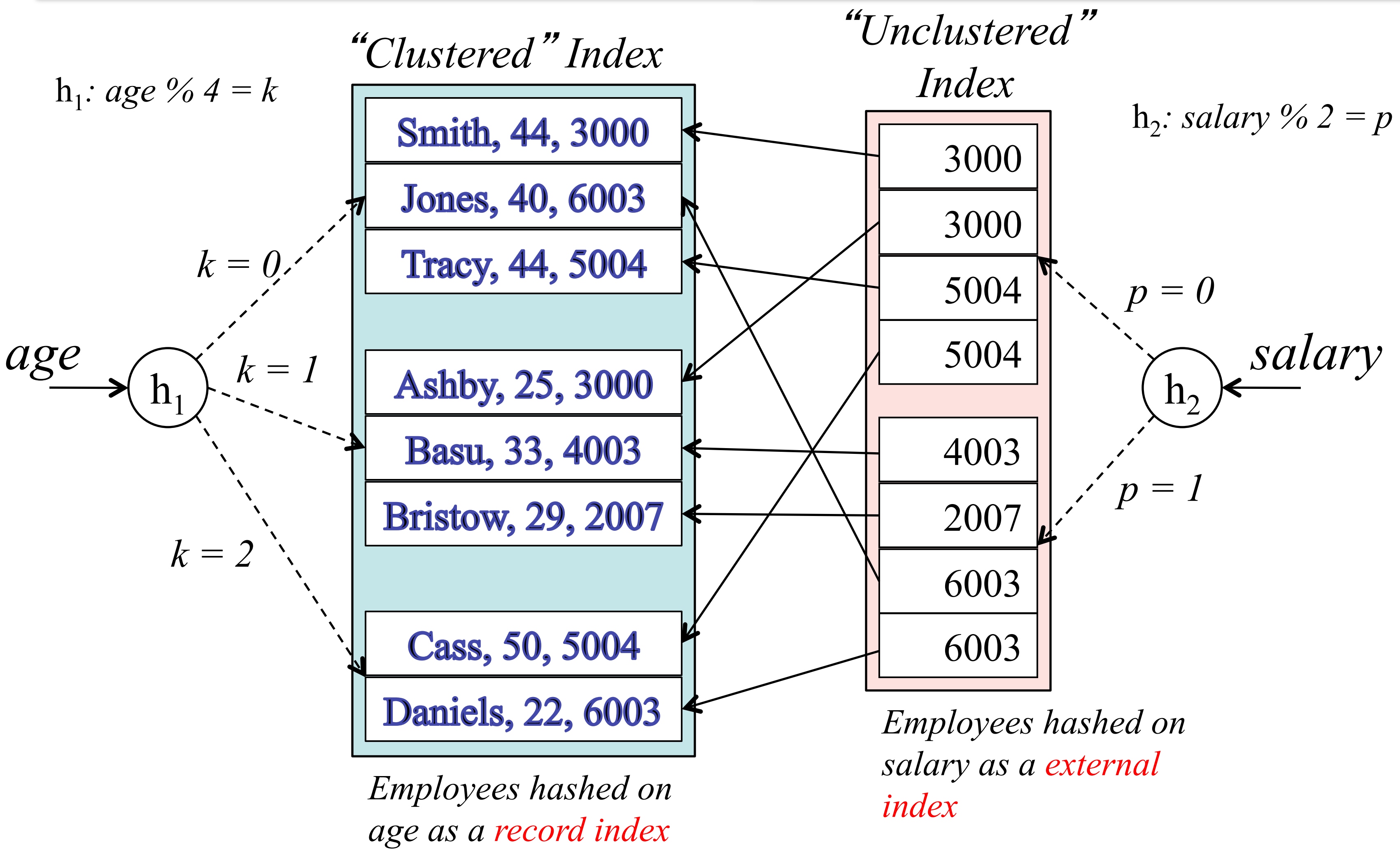
Indexes

- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Any subset of fields from a relation can be a search key.
  - *Search key* is not necessarily the same as the relation's *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**.
  - Given data entry k*, we can find record with key k in at most one disk I/O. (Details soon …)

Hash-Based Index

- Place all records with a common attribute together.
- 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(r)**:
  Mapping from the index' s *search key* to a bucket in which the (data entry for) record *r* belongs.

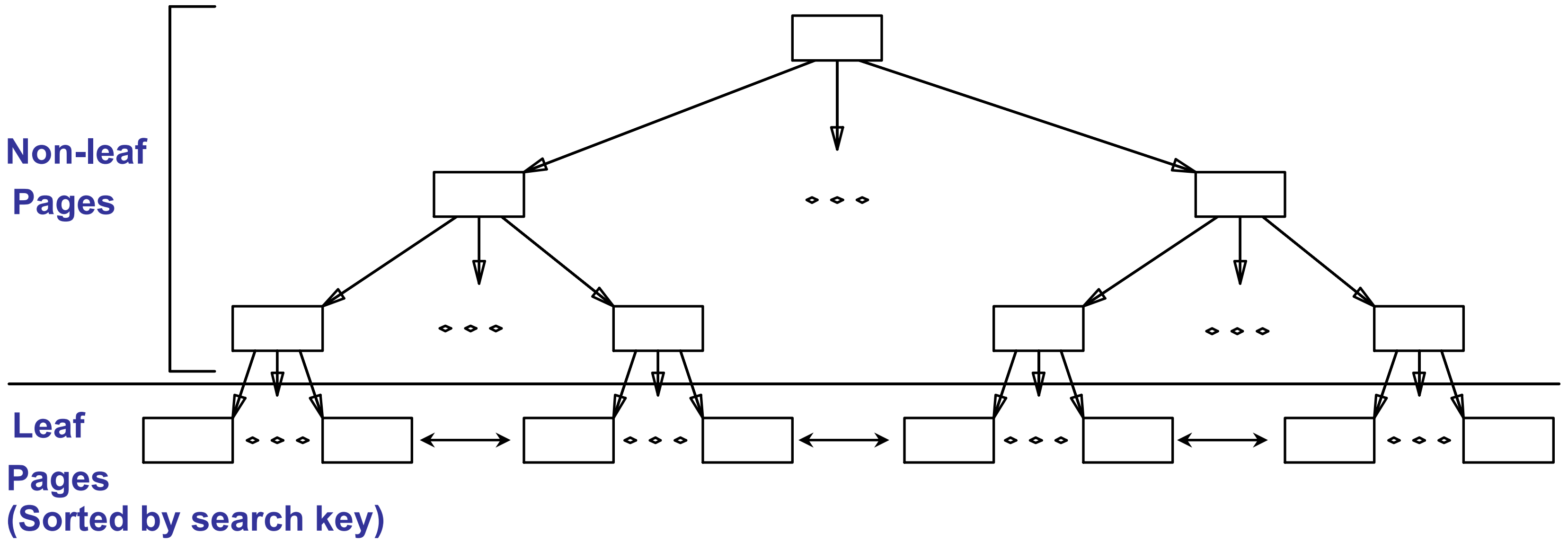
Hashing Index Example



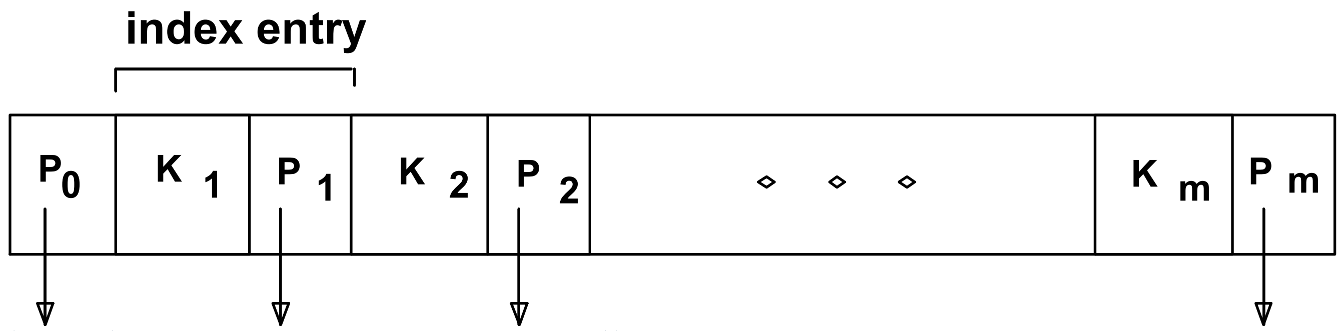
Employees hashed on age as a record index

Employees hashed on salary as a external index

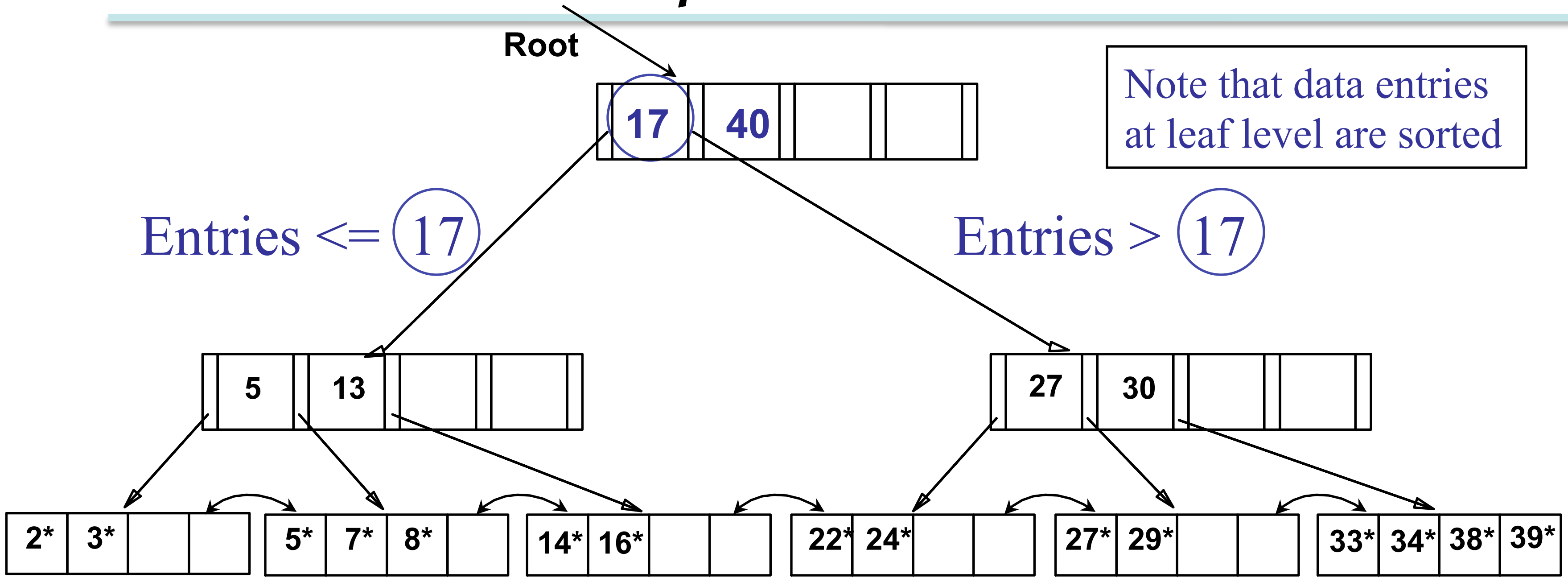
Tree-based Index (B+ Tree)



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



B+ Tree Example



Note that data entries at leaf level are sorted

- Find 28*? 29*? 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

Alternative Data/Index Organizations

- In data entry k* we store one of the following:
  - a actual data record with key value **k** (clustered)
  - <**k**, rid of data record with search key value **k**>
  - <**k**, list of rids of data records with search key **k**>
- Data organization choice is independent of the indexing method.
  - Clustered indices save on accesses, but you can only have 1 per relation
  - Unclustered alternatives tradeoff uniformity of index entries versus size considerations
  - Often, indices contains auxiliary information

Alternatives (Cont)

- Alternative 1:
  "an actual data record with key value **k** (clustered)"
  - If this is used, the index structure determines the file organization of the data records (instead of a Heap file or sorted file).
  - Can have at most *one clustered index* on a given relation. Else, data records would be duplicated, leading to redundant storage and potential inconsistency.
  - Space savings: No auxiliary information
  - Reduces records per leaf node
  - More involved to maintain index on inserts and deletes

Alternatives (Cont Again)

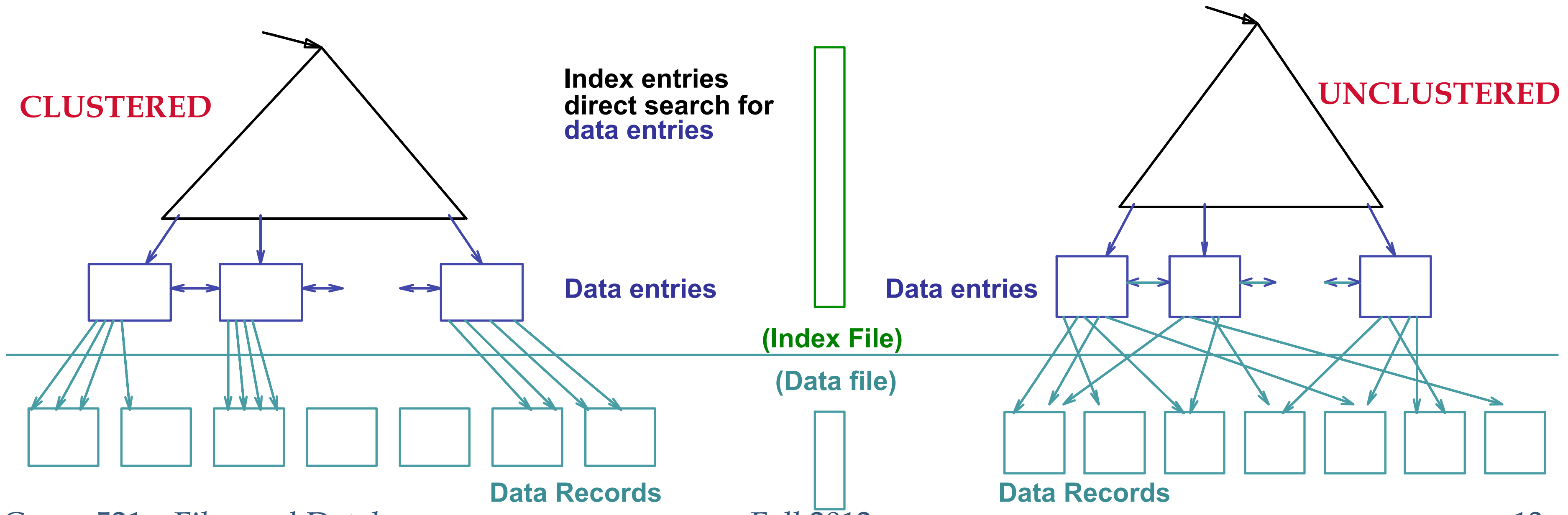
- Alternatives 2 and 3:
  "<**k**, one or more rids of records with search key **k**>"
  - Index entries contain *rids* which are typically much smaller than data records. So, better than Alternative 1 with large data records, especially if search keys are small. (More records per leaf node)
  - Alternative 3 more compact than Alternative 2, but leads to variable-sized data entries even if search keys are of fixed length. More complicated.

Index Classification

- *Primary* vs. *Secondary*: If search key contains primary key, then it is called a *primary index*.
  - *Unique* index: Search key contains a candidate key.
- *Clustered* vs. *unclustered*: If order of data records is the same as, or "close to", order of data entries, then called clustered index.
  - Alternative 1 implies clustered; in practice, clustered also implies Alternative 1 (since sorted files are rare).
  - 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 as a Heap.
  - To build clustered index, we must first sort the records (perhaps allowing for some free space on each page for future inserts).
  - Later inserts might create overflow pages. Thus, eventual order of data records is "close to", but not identical to, the sort order.



Costs / Benefits of Indexing

- Adding an index incurs some
  - Storage overhead
  - Maintenance overhead
- Without indexing, searching the records of a database for a particular record would require on average

  Number of Records * Cost to read a Record * 0.5

  (assumes records are in random 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 file (random record order; insert at eof)
- Sorted files, sorted on <*age, sal*>
- Clustered B+ tree file, clustered on 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

```
SELECT *
FROM Emp
```

```
SELECT *
FROM Emp
WHERE Age = 25
```

```
SELECT *
FROM Emp
WHERE Age > 30
```

```
INSERT
INTO Emp(Name, Age, Salary)
VALUES( 'Jordan' , 49, 3000000)
```

```
DELETE
FROM Emp
WHERE Name = 'Bristow'
```

Assumptions in Our Analysis

- Heap Files:
  - Equality selection is 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
    - Tree Fan-out = F

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

File Type	Scan	Equality Search	Range Search	Insert	Delete
Heap	BD	0.5BD	BD	2D	Search + D
Sorted	BD	$D\log_2 B$	$D\log_2 B$ + #matches	Search + BD	Search + BD
Clustered	1.5BD	$D\log_F 1.5B$	$D\log_F 1.5B$ + #matches	Search + D	Search + D
Unclustered tree index	BD(R +0.15)	$D(1+\log_F 0.15B)$	$D(1+\log_F 0.15B$ + #matches)	$D(\log_F 0.15B)$	Search + 2D
Unclustered hash index	BD(R +0.125)	2D	BD	4D	Search + 2D

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

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 the conditions applied 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, and updates go slower. They require disk space, too.

Index Selection Guidelines

- Attributes in WHERE clauses are candidate 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.

Examples of 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* > 40, 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
```

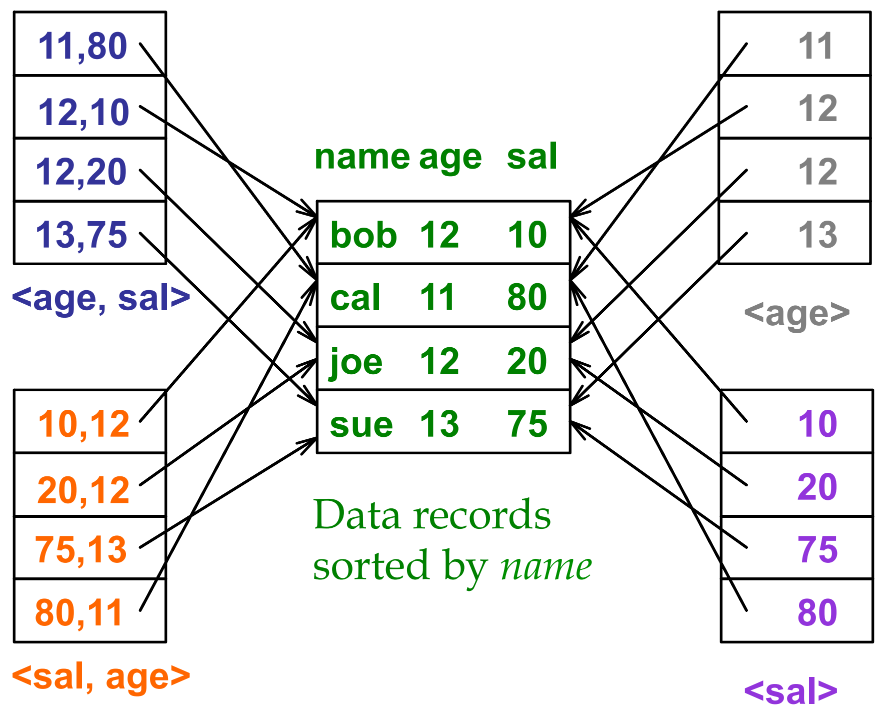
```
SELECT  E.dno,  COUNT (*)
FROM  Emp E
WHERE  E.age>40
GROUP BY E.dno
```

```
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 records sorted by *name*

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, and require updating more often.

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
```

<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 BETWEEN 3000 AND 5000
```

Index-Only Plans (Contd.)

- Index-only plans can also be found for queries involving more than one table; more on this later.

<E.dno>

```
SELECT  D.mgr
FROM  Dept D, Emp E
WHERE  D.dno=E.dno
```

<E.dno,E.eid>

```
SELECT  D.mgr, E.eid
FROM  Dept D, Emp E
WHERE  D.dno=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.