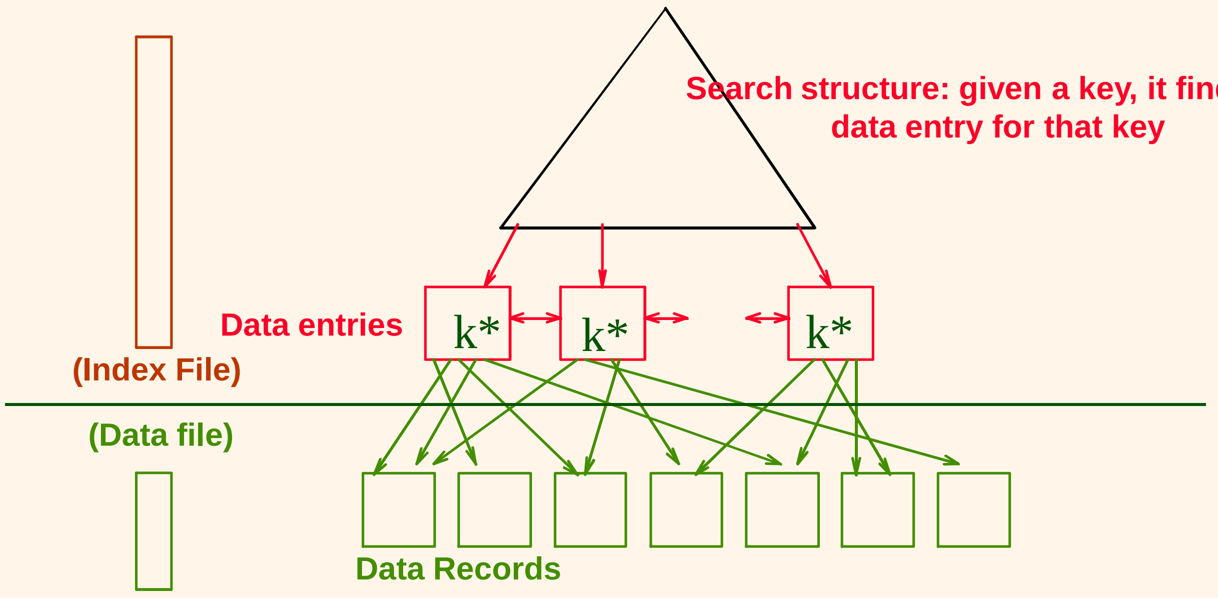
Indexes

- An ***index*** on a file speeds up selections on the ***search key fields*** for the index.
  - Any subset of the fields, or a prefix of a field, 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**.
  - Given data entry k*, we can find record with key k in at most one disk I/O. (Details soon …)

Typical index architecture



Search structure: given a key, it finds the data entry for that key
Data entries
k*
k*
k*
(Index File)
(Data file)
Data Records

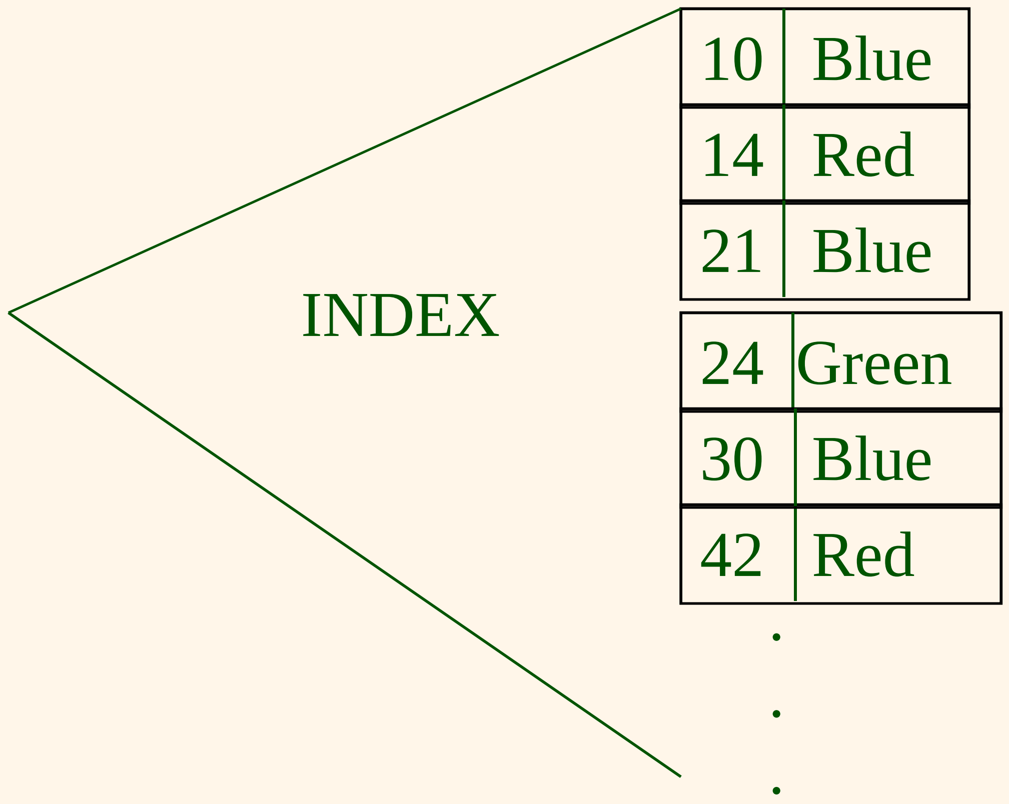
- Used by modern DBMSs
- What are typical search structures in CS?

*Alternatives for Data Entry **k*** in Index*

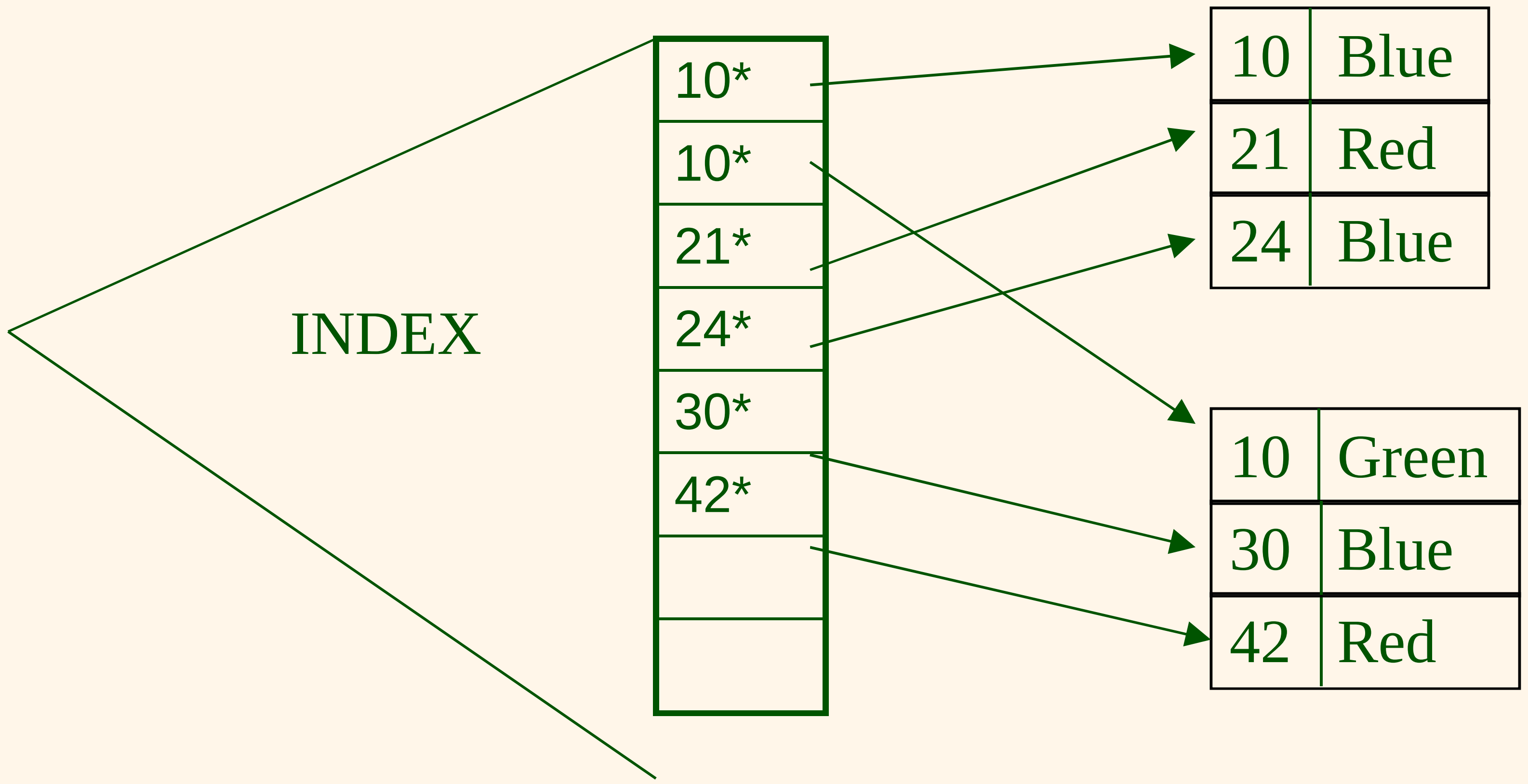
- In a data entry k* we can store:
  1. Data record with key value **k,** or
     - In this case the previous page is not accurate
  2. <**k**, rid of data record with search key value **k**>, or
  3. <**k**, list of rids of data records with search key **k**>

- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value **k**.
  - Examples of indexing techniques: B+ trees, hash-based structures
  - Typically, index contains auxiliary information that directs searches to the desired data entries

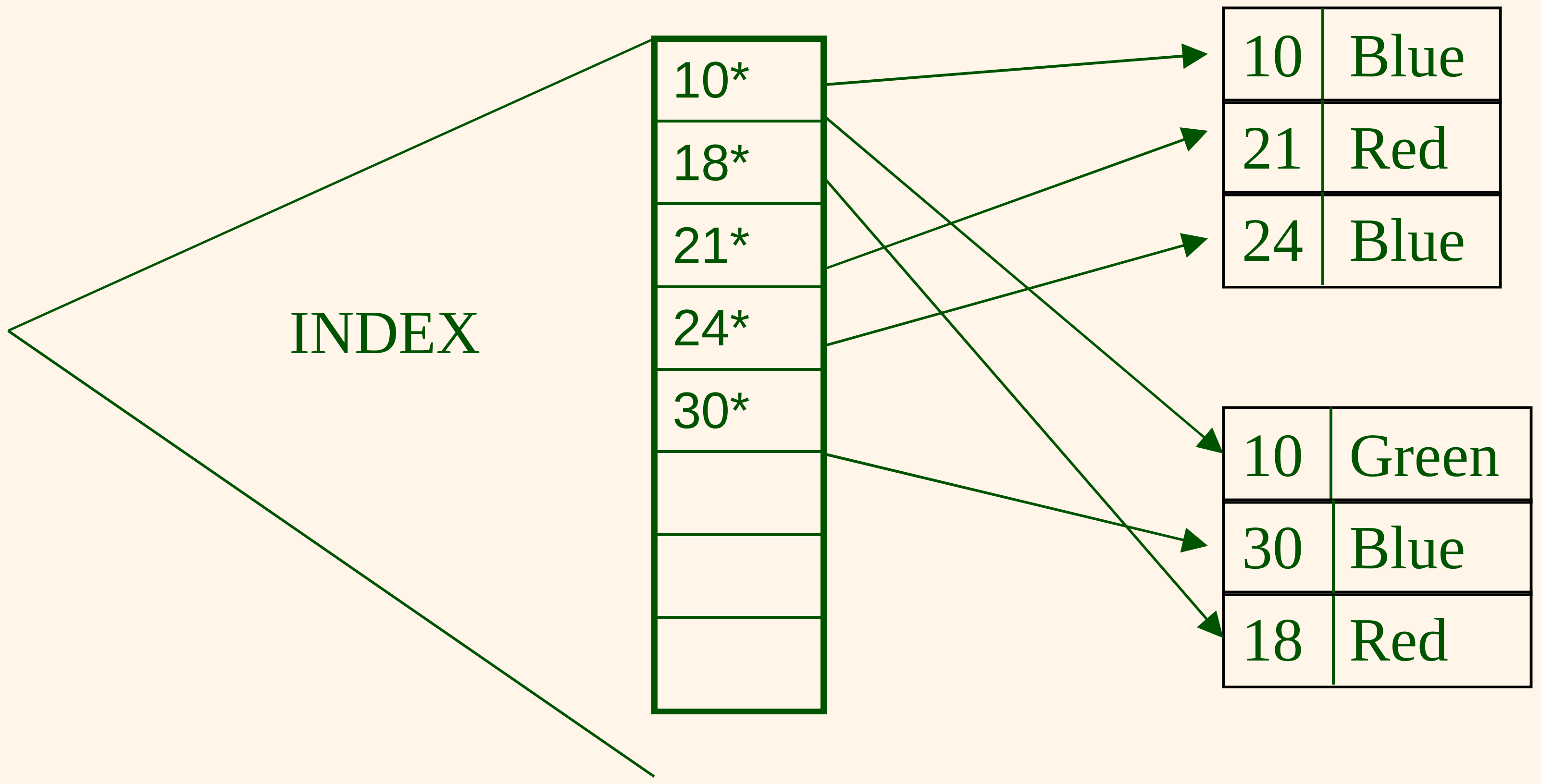
Alternative 1: k holds the data*



Alternative 2: One pointer per $k*$



Alternative 3: 1 or more pointers per k^*

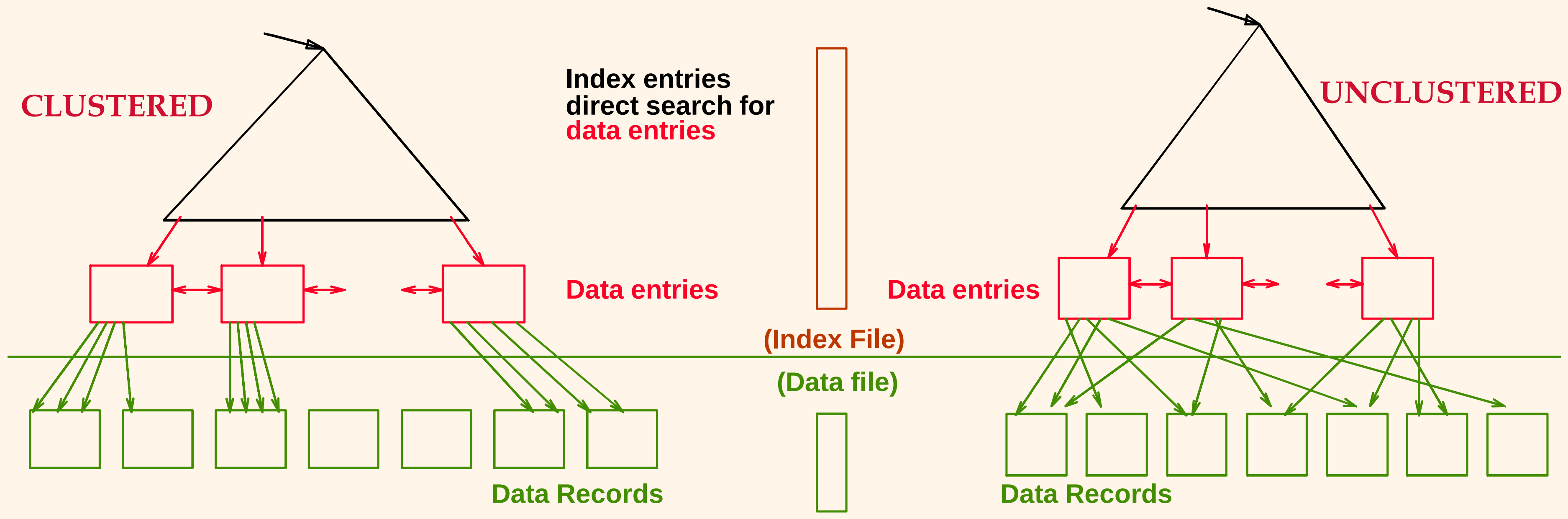


Alternatives for Data Entries (Contd.)

- Alternative 1:
  - If this is used, index structure is a file organization for data records (instead of a Heap file or sorted file).
  - At most one single-attribute index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.)
- Alternatives 2 and 3:
  - Alternative 3 more compact than Alternative 2, but leads to variable sized data entries even if search keys are of fixed length.

8.2.1 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.)



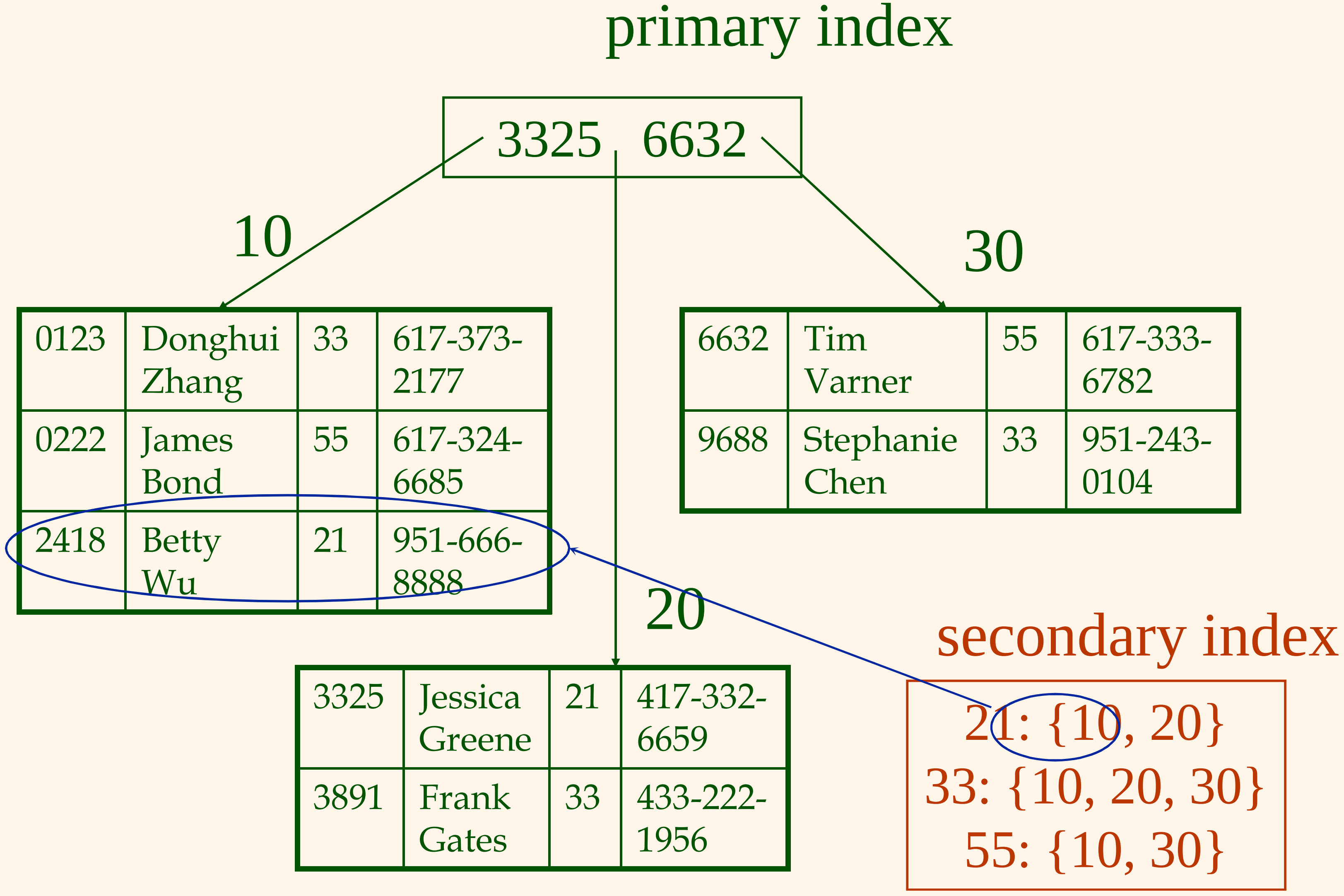
8.2.2 Index Classification

- ***Primary*** vs. ***secondary***: If search key contains primary key, then called primary index.
  - In practice, primary means alternative 1
- ***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.
  - A file can be clustered on at most one single-attributesearch key.
  - Cost of retrieving data records through index varies ***greatly*** based on whether index is clustered or not!

Index Types

- A ***primary index*** is an index which controls the actual storage of a table. Typically this index is built using the primary key of the table.
  - A data entry is one record of the table.
- A ***secondary index*** is an index which is built using some other attribute(s).
  - A data entry contains a set of RIDs.

An Example



0123	Donghui Zhang	33	617-373-2177
0222	James Bond	55	617-324-6685
2418	Betty Wu	21	951-666-8888

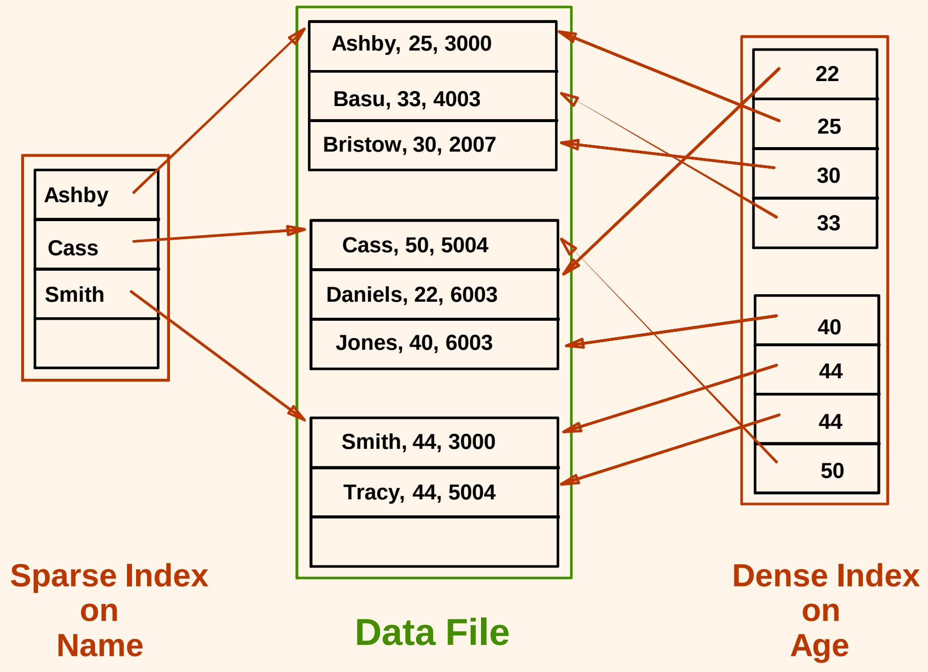
3325	Jessica Greene	21	417-332-6659
3891	Frank Gates	33	433-222-1956

6632	Tim Varner	55	617-333-6782
9688	Stephanie Chen	33	951-243-0104

21: {10, 20}
33: {10, 20, 30}
55: {10, 30}

Index Classification (Cont.)

- **Dense vs. Sparse:** If there is at least one data entry per search key value (in some data record), then it's dense.
  - Alternative 1 always leads to dense index.
  - Every sparse index is clustered!
  - Sparse indexes are smaller; however, some useful optimizations are based on dense indexes.



Index Classification

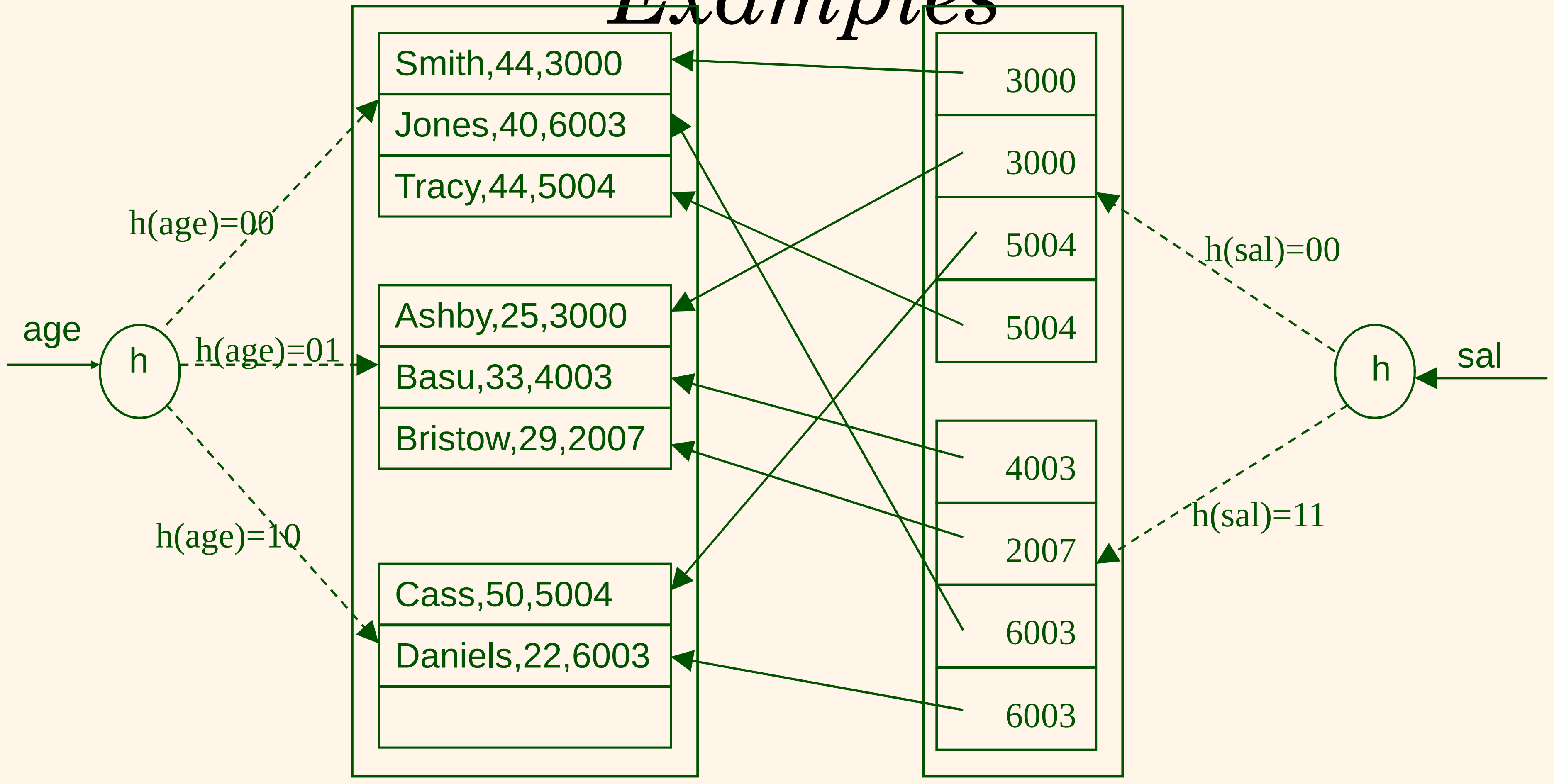
Clustered, sparse indexes are smaller; they work well for range searches and sorting.

But…data blocks are difficult to keep sorted and some useful optimizations are based on dense indexes.

Note: one file can have at most one clustered index - all of the additional indices must be unclustered.

	sparse	dense
clustered	YES	YES
unclustered	NO!	YES

8.3.1 Hash-based Index Examples



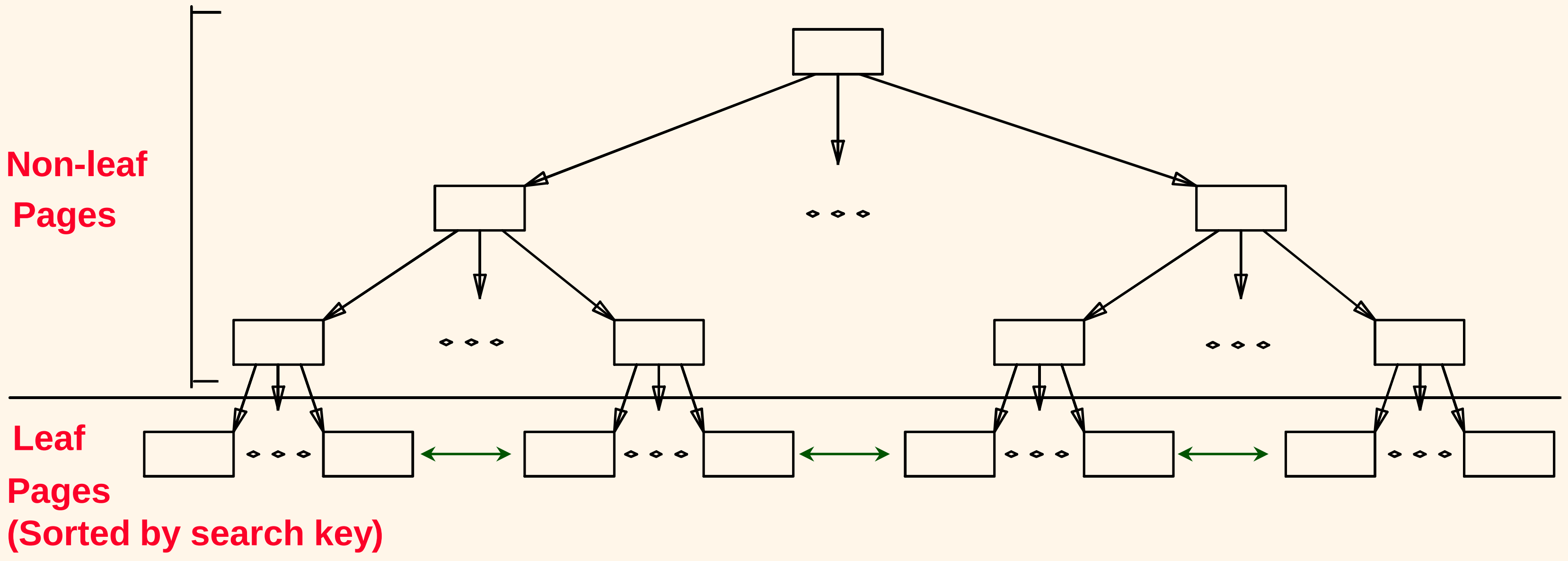
Employees file hashed on age

Index on salary

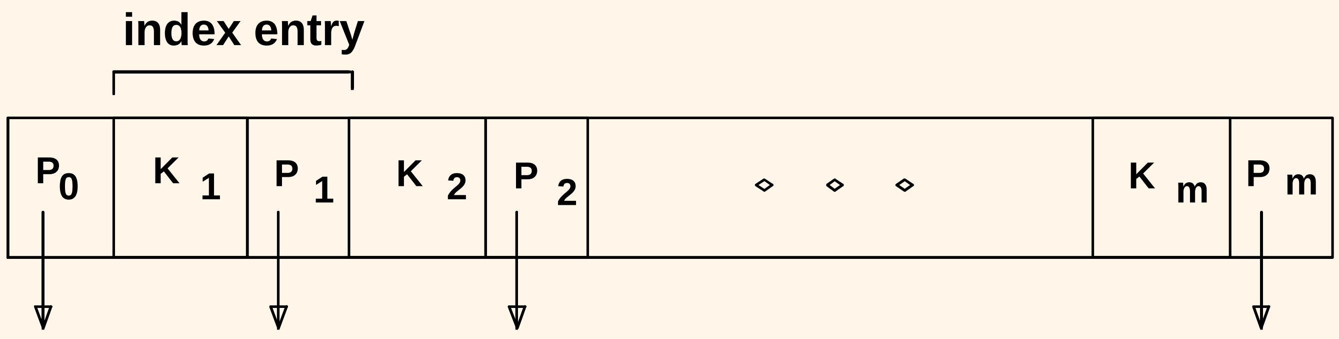
8.3.1 Hash-Based Indexes

- 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**(***r***) = bucket in which (data entry for) record ***r*** belongs. **h** looks at the ***search key*** fields of ***r.***
  - ***No need for "index entries" in this scheme.***

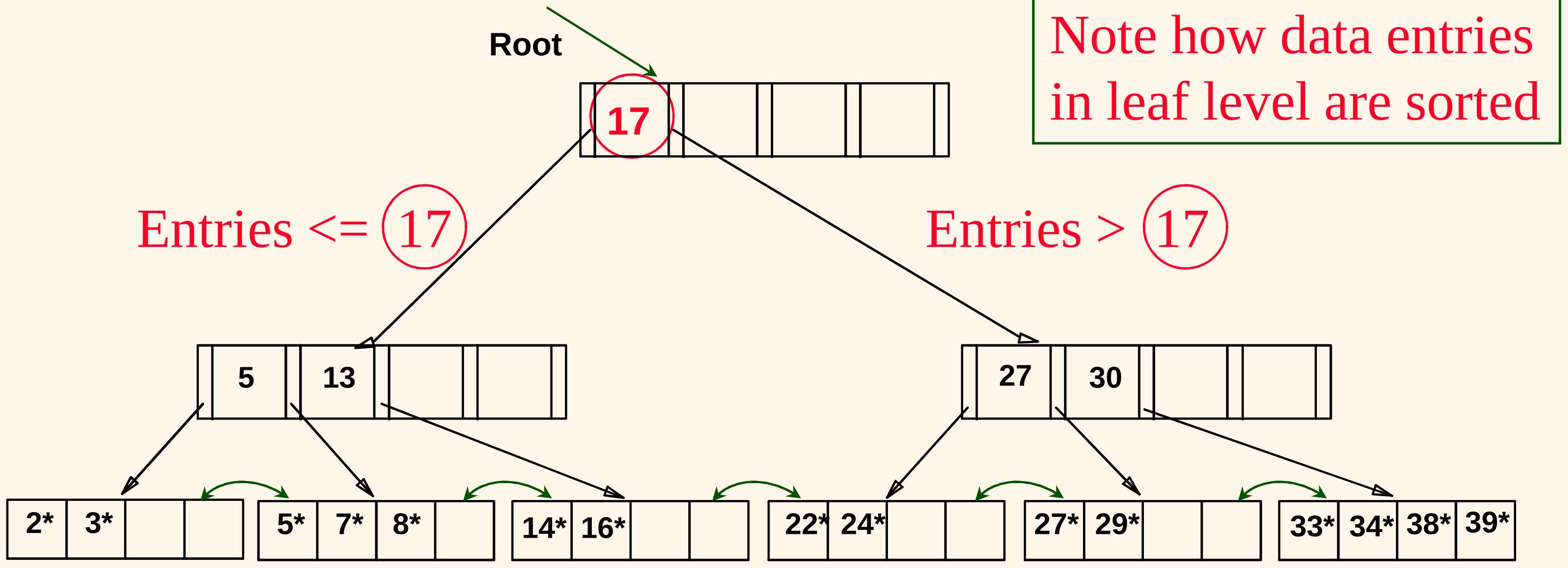
8.3.2 B+ Tree Indexes



- 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

- 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

8.4 Comparing File Organizations

- Assume an employee file is organized by the search key <age, salary>.
- We will consider various file organizations, and operations like scan, equality, range, insert, delete.
- For each file organization, we will compute the cost of each operation.

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 data records per page
- **F:** Fanout, number of index records per page
- **D:** (Average) time to read or write disk page
- **C:** (Average) time to process a record
- 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!*

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

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

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	$D\log_2 B$	$D(\log_2 B$ + # pgs with match recs)	Search + BD	Search +BD
(3) Clustered	1.5BD	$D\log_F 1.5B$	$D(\log_F 1.5B$ + # pgs w. match recs)	Search + D	Search +D
(4) Unclust. Tree index	BD(R+0.15)	$D(1 + \log_F 0.15B)$	$D(\log_F 0.15B$ + # match recs)	Search + 2D	Search + 2D
(5) Unclust. Hash index	BD(R+0.125)	2D	BD	Search + 2D	Search + 2D

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

8.5 Indexes and Performance Tuning

- One of a DBA's most important duties is to deal with performance problems
- One of the most effective methods for improving performance is to choose the best indexes for the given workload
- Why not index everything?
  - What are the two costs of an index?
- Part of a DBA's job: choose indexes so the database will "run fast".
  - What information does the DBA need, to do this?

8.5.1 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?
- What kinds of indexes should we create?
  - Clustered? Hash/tree?
- 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.
- Also consider dropping indexes

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!
- Second approach: Use an Index Tuning Wizard [163].
  - Inputs: A workload (it can also capture a workload) and an amount of disk storage for indexes
  - Output: an index configuration

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.
- 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.
- 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.

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.
- For index-only strategies, clustering is not important!

<E.dno>

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

<E.dno,E.sal>

Tree index May help

```
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
```

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?

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

- 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!

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

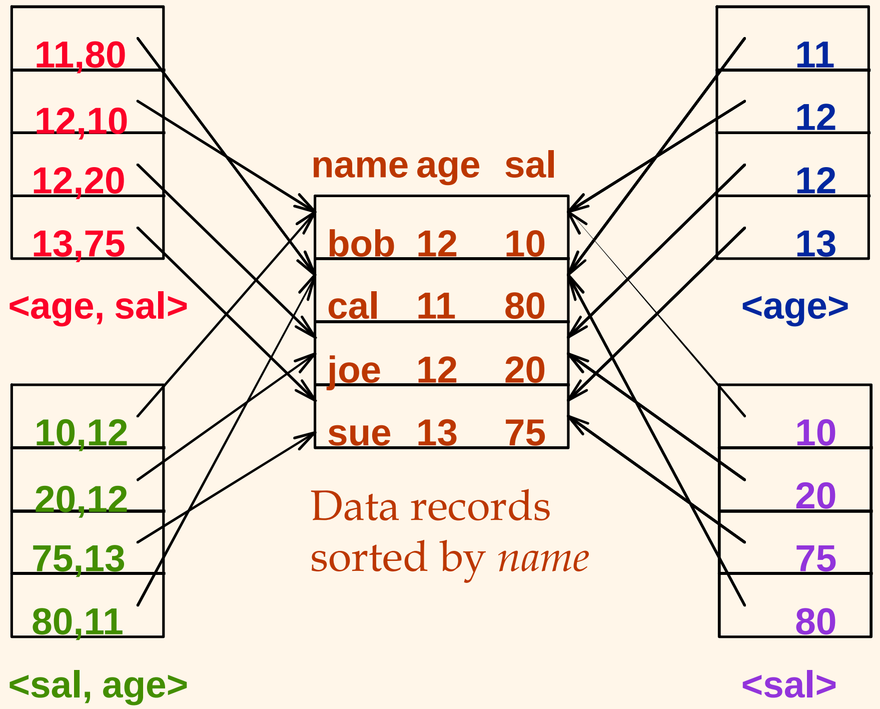
- Equality queries and duplicates:
  - Clustering on ***E.hobby*** helps!

```
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.

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, updated more often.