Database Indexing

8

Single Record and Range Search

- Single record retrieval:
  - "Find student name whose Age = 20"
- Range queries:
  - "Find all students with Grade > 8.50"
- Sequentially scanning of file is costly
- If data is in sorted file:
  - Binary search to find first such student
  - Scan to find others.
  - Cost of binary search can still be quite high.

Indexes

- Indexes are file structures that enable us to answer value-based queries efficiently.
- 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 with a given search key value **k***".

Indexing properties

- *Propagation of changes*
  - When records are inserted/deleted, all indexes must be updated.
  - Any change in the search key implies updates of corresponding index file(s)
- *Index size* - since the index must be moved to main memory to be searched, it must remain small enough to fit within a reasonable memory area.
  - What if the index size is too large?
    - The partial-index structure
    - An index to the index (layered or tree structure).
- Two issues:
  - What is stored as a data entry in an index *(index content)?*
  - How are date entries organized (*indexing technique*)?

Alternatives of Data Entry in Index

- Three alternatives:
  - (1) Data record with key value **k**
  - (2) <**k**, rid of data record with search key value **k**>
  - (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

Alternatives of Data Entry in Index

- Alternative (1):
  - Index and data records stored together
    - 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, no. of pages containing data entries is high. Implies size of auxiliary information in the index is also large, typically.

Alternatives of Data Entry in Index

- Alternatives (2) and (3):
  - Data entries point to data records
    - Data entries typically are much smaller than data records (so, better than Alternative (1) especially if search keys are small).
    - Portion of index structure used to direct search, which depends on size of data entries, is much smaller than with Alternative (1).
  - If more than one index is required on a given file, at most one index can use Alternative (1); rest must use Alternatives (2) or (3).
  - Alternative (3) more compact than Alternative (2), but leads to variable sized data entries even if search keys are of fixed length.

Creating an Index

	Name	*Age*	*Grade*
rid_1	John	22	8.50
rid_2	Jack	21	9.00
		…	
rid_n	Peter	22	10.00

search key

rid_1: 22

rid_2 : 21

…

rid_n : 22

22 : rid_1, rid_n …

21 : rid_2 …

…

…

21 : rid_2 …

22 : rid_1, rid_n …

…

data entry

index file (reversed)

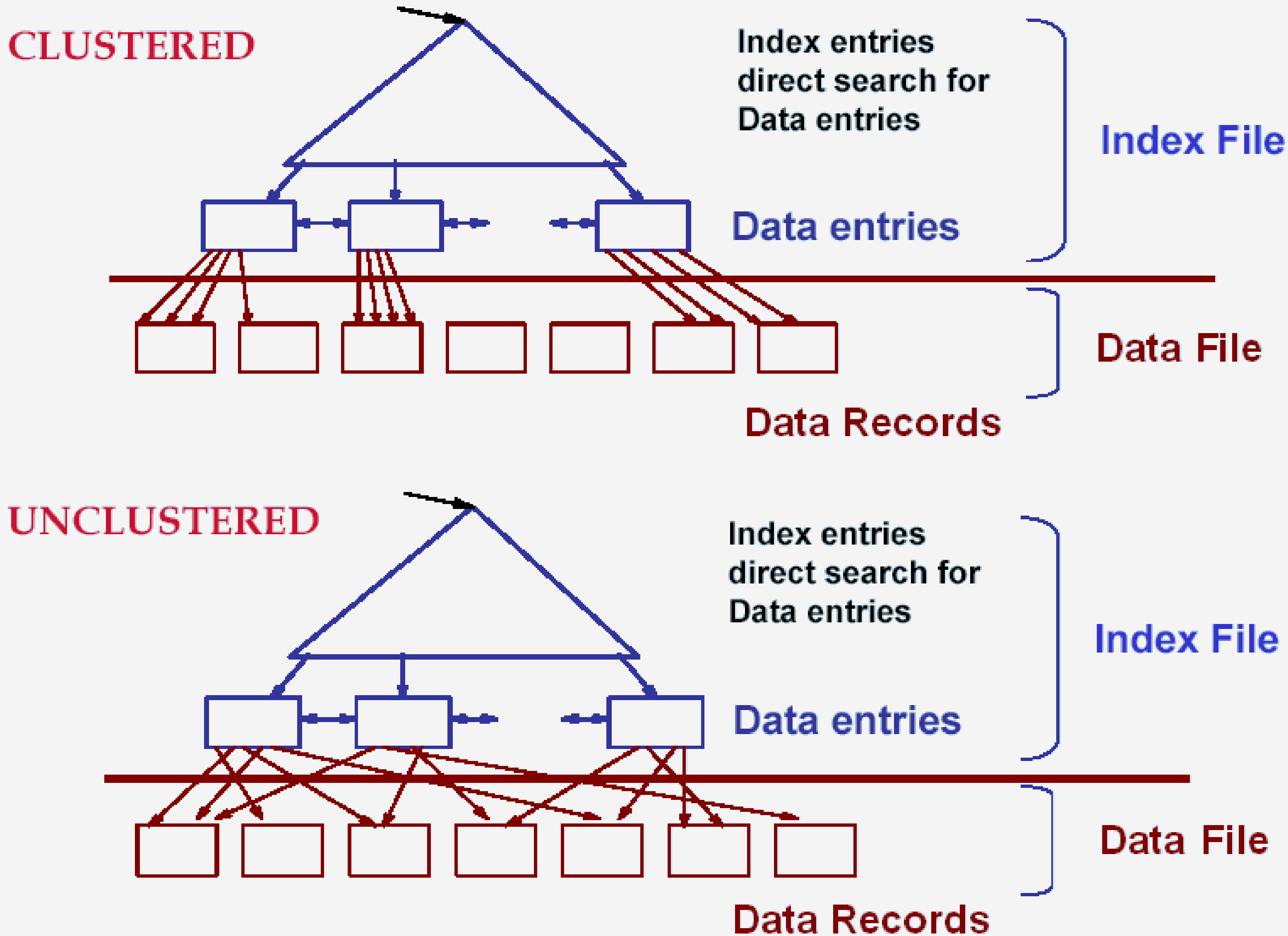
Index Classification

- *Primary* vs. *secondary* indexes:
  - A *primary* index is an index on a set of fields that includes the primary key.
  - An index is called a ***unique*** index if the search key contains a candidate key.
    - No duplicates in the data entries
  - In general, *secondary* index contains duplicates

Index Classification

- *Clustered* vs. *un-clustered* indexes:
  - An index is *clustered* if order of data records in a file is the same as, or `close to', the order of data entries in the 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. Un-clustered Index



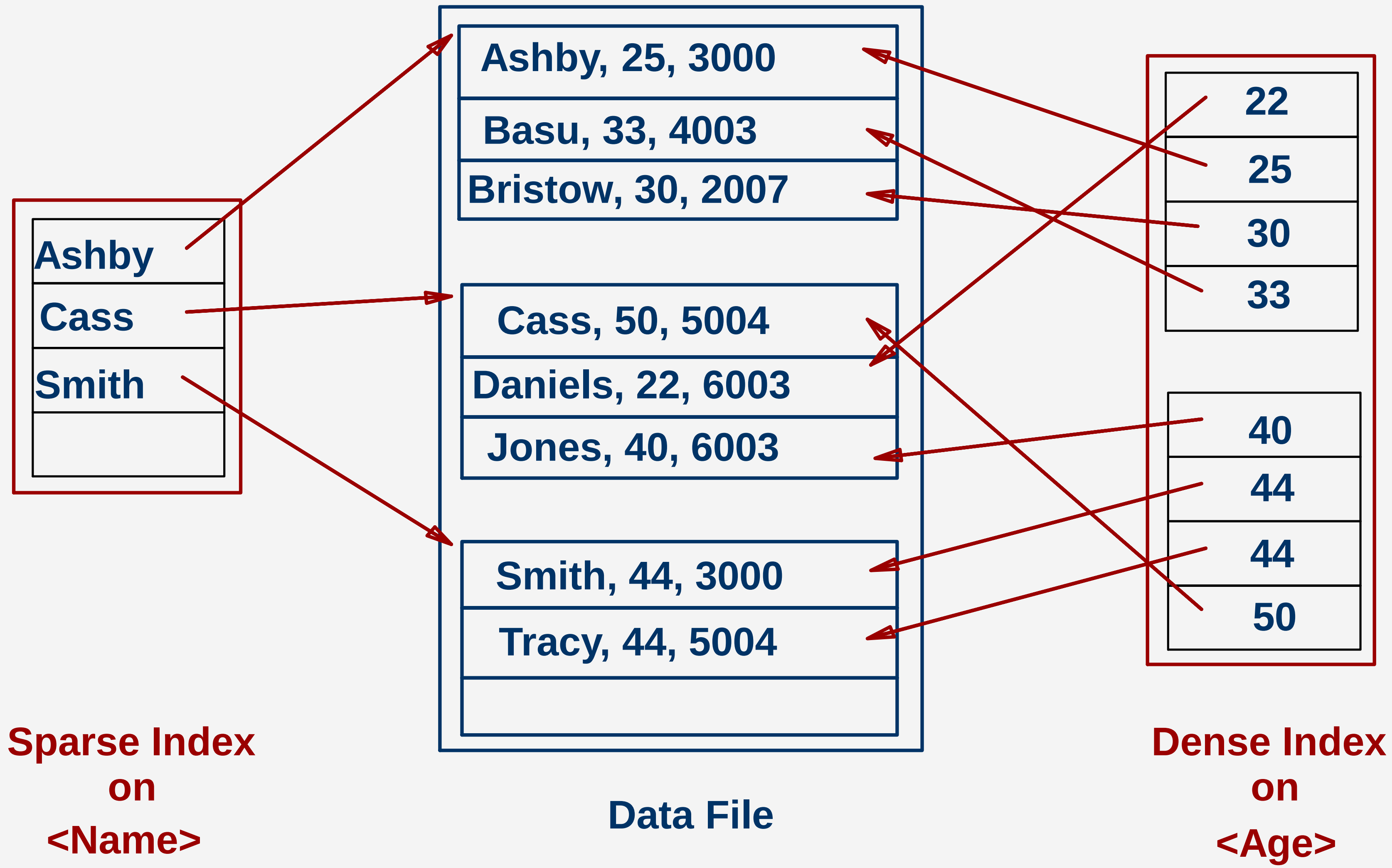
Clustered vs. Un-clustered Index

- To build clustered indexes:
  - Sort the records in heap file
  - Leave some free space in each page to absorb future insertions
    - If free space is used up subsequently, further insertions to the page is handled using a linked list of overflow pages
    - Need to reorganize file periodically to ensure good performance
- Clustered index is expensive to maintain

Index Classification

- *Dense* vs. *sparse* indexes:
- An index is *dense* if there is at least one data entry per search key value (in some data record)
  - Several data entries can have the same search key value if there are duplicates and Alternative (2) is used
  - Alternative (1) always leads to dense index.
- An index is *sparse* if it contains one data entry for each page of records in the data file
  - Every sparse index is clustered!
  - Sparse indexes are smaller; however, some useful optimizations are based on dense indexes.

Index Classification



Index Classification

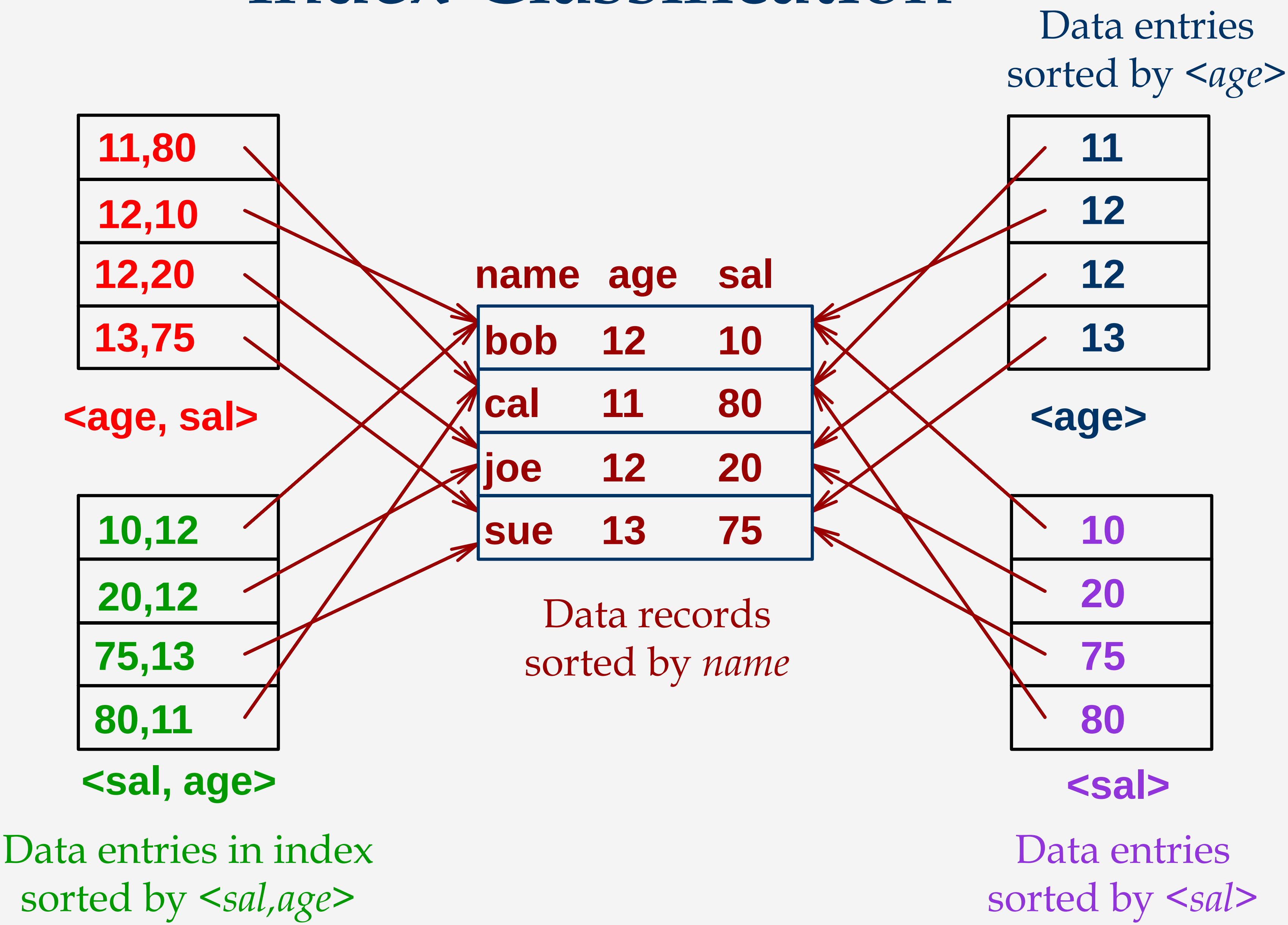
- *Composite Search Keys*:
- Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value:

    age=20 and sal =75
  - Range query: Some field value is not a constant:

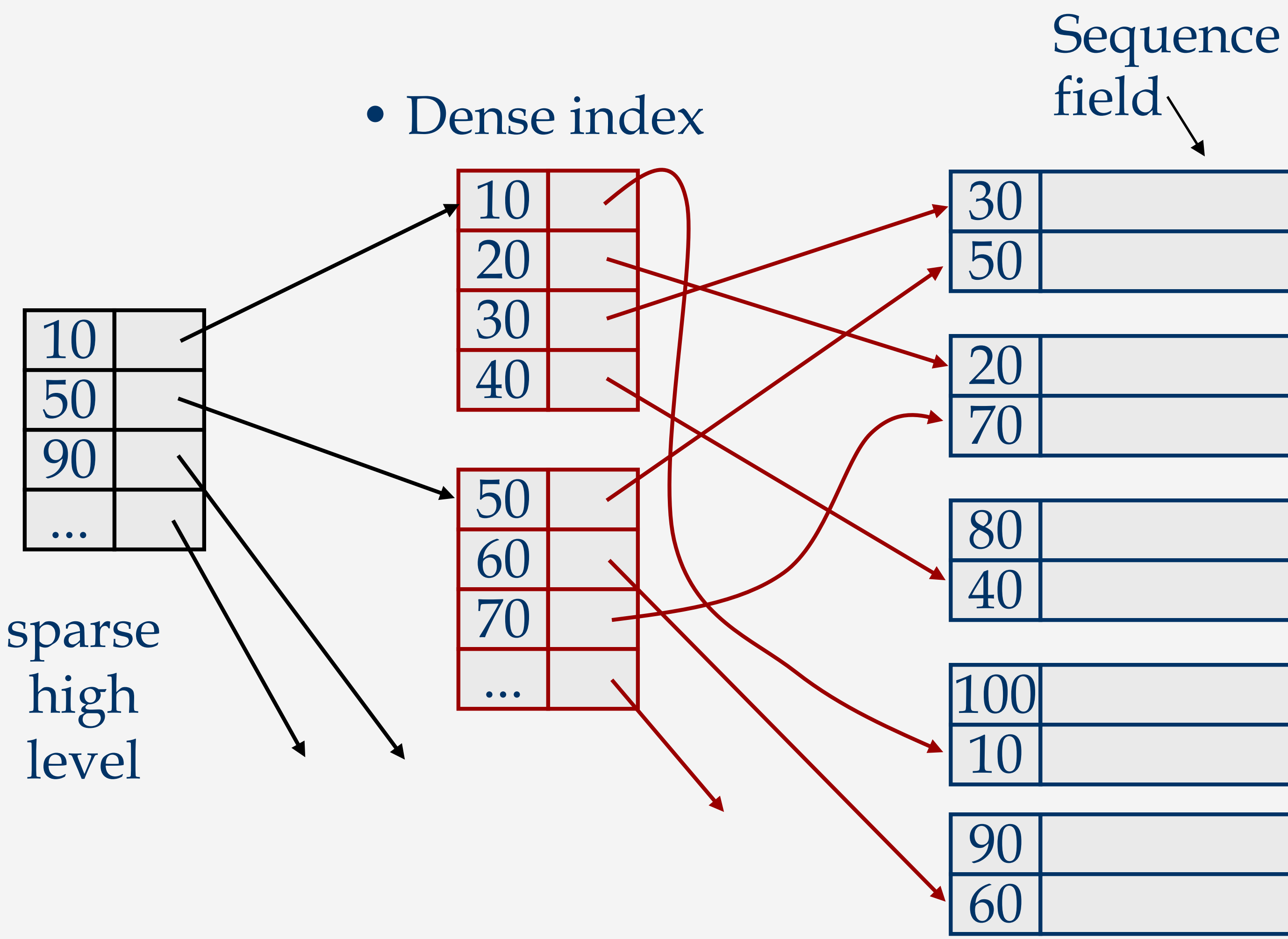
    age=20 and sal > 10

- Data entries in index sorted by search key to support range queries.
  - Lexicographic order, or
  - Spatial order.

Index Classification



Secondary Indexes



Example

- Suppose that the records of Students table are stored in a sorted (by *Age* field) file. Each page can store up to 3 records.

ID	*Name*	*Age*	*Grade*	*Course*
53831	Melanie	11	7.8	Music
53832	George	12	8.0	Music
53666	John	18	9.4	ComputerS
53688	Sam	19	9.2	Music
53650	Sam	19	9.8	ComputerS

- List the data entries in each of the following indexes (you can use <*page_id, slot no*> to identify a tuple).

Example

ID	*Name*	*Age*	*Grade*	*Course*
53831	Melanie	11	7.8	Music
53832	George	12	8.0	Music
53666	John	18	9.4	ComputerS
53688	Sam	19	9.2	Music
53650	Sam	19	9.8	ComputerS

1. *Age* – Dense, alternative (1)
   -the file itself

2. *Age* – Dense, alternative (2)
   (11,<1,1>) (12,<1,2>)(18,<1,3>)(19,<2,1>)(19,<2,2>)

3. *Age* – Dense, alternative (3)
   (11,<1,1>) (12,<1,2>)(18,<1,3>)(19,<2,1>,<2,2>)

4. *Age* – Sparse, alternative (1)
   - cannot build such index (by definition)

Example

ID	*Name*	*Age*	*Grade*	*Course*
53831	**Melanie**	**11**	**7.8**	**Music**
53832	**George**	**12**	**8.0**	**Music**
53666	**John**	**18**	**9.4**	**ComputerS**
53688	**Sam**	**19**	**9.2**	**Music**
53650	**Sam**	**19**	**9.8**	**ComputerS**

5. *Age* – Sparse, alternative (2)

   (11,<1,1>) (19,<2,1>) - the order of entries is significant

6. *Age* – Sparse, alternative (3)

   (11,<1,1>) (19,<2,1>,<2,2>) - the order of entries is significant

7. *Grade* – Dense, alternative (1)

   7.8, 8.0, 9.2, 9.4, 9.8

8. *Grade* – Dense, alternative (2)

(7.8,<1,1>)(8.0,<1,2>)(9.2,<2,1>)(9.4,<1,3>)(9.8,<2,2>)

Example

ID	*Name*	*Age*	*Grade*	*Course*
53831	Melanie	11	7.8	Music
53832	George	12	8.0	Music
53666	John	18	9.4	ComputerS
53688	Sam	19	9.2	Music
53650	Sam	19	9.8	ComputerS

9. *Grade* – Dense, altern. (3)

(7.8,<1,1>)(8.0,<1,2>)(9.2,<2,1>)(9.4,<1,3>)(9.8,<2,2>)

10. *Grade* – Sparse, alternative (1)

- cannot build such index (by definition)

11. *Grade* – Sparse, alternative (2)

- search key values are not ordered

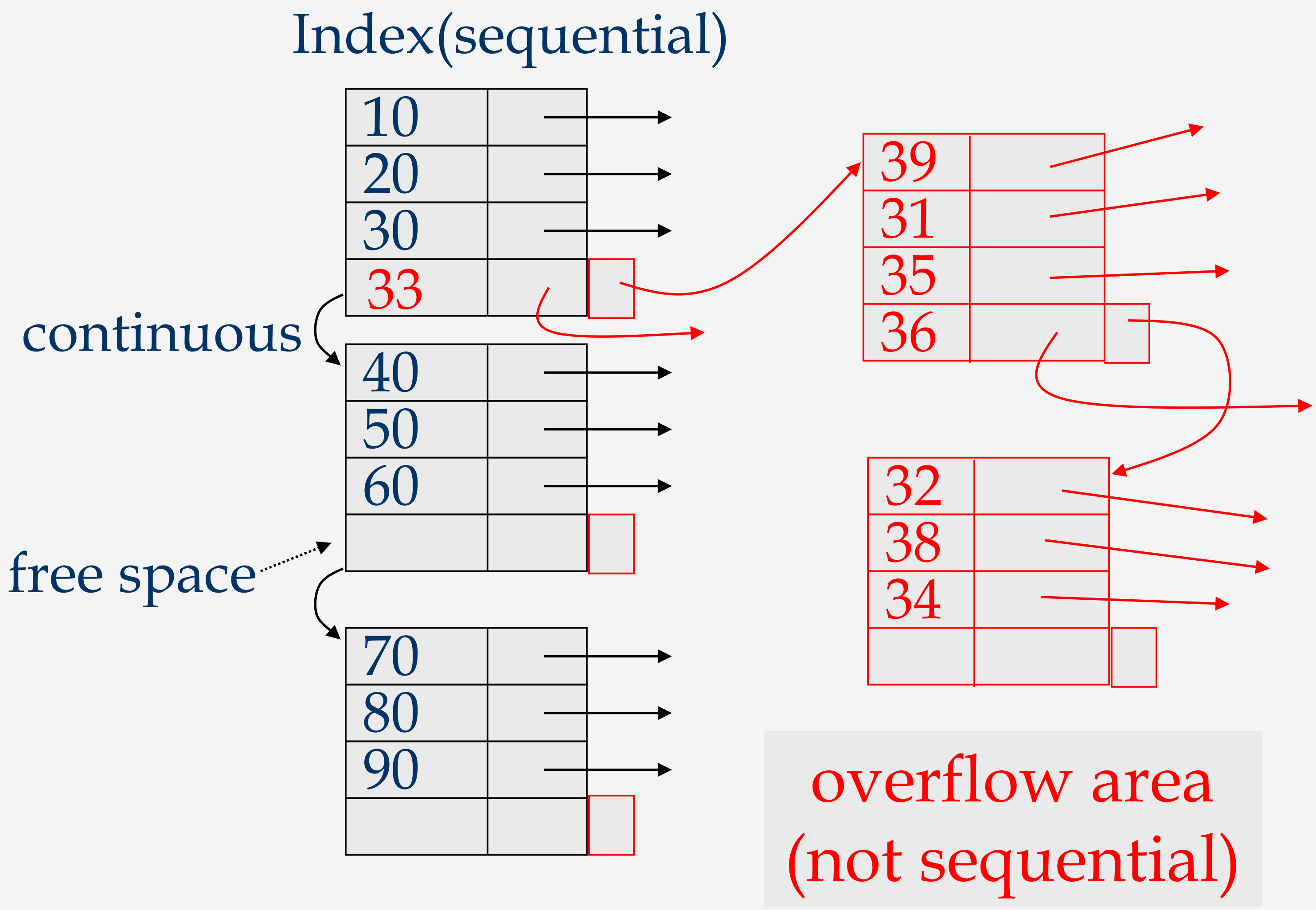
12. *Grade* – Sparse, alternative (3)

- search key values are not ordered

Conventional Indexes

- Advantages:
  - Simple
  - Index is sequential file $\rightarrow$ good for scans
- Disadvantages:
  - Inserts expensive, and/or
  - Lose sequentiality & balance

Example



Index(sequential)
10
20
30
33
continuous
40
50
60
free space
70
80
90
39
31
35
36
32
38
34
overflow area
(not sequential)

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?
- For each index, which storing method will use?
- 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. 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.
- 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 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* > 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  Employees E
WHERE  E.age>40
```

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

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

Indexing today

- Single dimensional Indexing
  - B+ Trees, Hash Files
- Multi-dimensional Indexing
  - for Spatial Databases (GIS)
  - R - Trees
- High-dimensional Indexing
  - Similarity range queries
- Indexing for advanced applications
  - Main memory indexing

Composite Search Keys

- To retrieve Employees 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.

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.

<*E.dno*>

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

<*E.dno,E.eid*> *Tree index!*

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

<*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
```