CompSci 516

Data Intensive Computing Systems

Lecture 5

Storage and Indexing

Instructor: Sudeepa Roy

Announcement

- Homework 1
  - Due on Feb 9, 11:59 pm
  - Please post any question you have on Piazza

- Lecture slides
  - Book chapters will be posted on the webpage two days before the lecture
    - you don't have to look at the chapters before the class!
  - Slides will be posted after the class

Reading Material

- [RG]
  - Chapters 8.1, 8.2, 8.3, 8.5

Additional reading

- [GUW]
  - Chapters 8.3, 14.1

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.

Where are we now?

We learnt

- Relational Model and Query Languages
  - RA, RC, SQL, Postgres (DBMS)
- Schema refinement
  - Normalization
  - (for Schema Design with E/R diagram – see the book or CompSci 316)

Next

- Database Internals
  - Architecture, Storage, Indexing

What will we learn?

- How does a DBMS organize files?
- What is an index?
- What are different types of indexes?
- How do we use index to optimize performance?

Data on External Storage

- Data must persist on disk across program executions in a DBMS
  - But has to be fetched into main memory when DBMS processes the data
- The unit of information for reading data from or writing data to disk is a page
- Disks: Can retrieve random page at fixed cost
  - But reading several consecutive pages is much cheaper than reading them in random order

File Organization

- File organization: Method of arranging a file of records on external storage.
  - Record id (rid) is sufficient to physically locate the page containing the record on disk
  - Indexes are data structures that allow us to find the record ids of records with given values in index search key fields

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 the fields of a relation can be the search key for an index on the relation.
  - "Search key" is not the same as "key"

    key = minimal set of fields that uniquely identify a tuple
- An index contains a collection of data entries, and supports efficient retrieval of all data entries **k*** with a given key value **k**

Alternatives for Data Entry k* in Index k

- In a data entry k* we can store:
  1. The actual data record with key value **k,** or
  2. <**k**, rid>
     - rid = record of data record with search key value **k**, or
  3. <**k**, rid-list>
     - list of record ids 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**

Alternatives for Data Entries: Alternative 1

- In a data entry k* we can store:
  1. The actual data record with key value **k,** or
  2. <**k**, rid>
     - rid = record of data record with search key value **k**, or
  3. <**k**, rid-list>
     - list of record ids of data records with search key **k**>

Advantages/
Disadvantages?

- Index structure is a file organization for data records
  - instead of a Heap file or sorted file
- How many different indexes can use Alternative 1?
- At most one index can use Alternative 1
  - Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency
- If data records are very large, #pages with data entries is high
  - Implies size of auxiliary information in the index is also large

Alternatives for Data Entries: Alternative 2, 3

- In a data entry k* we can store:
  1. The actual data record with key value **k**, or
  2. <**k**, rid>
     - rid = record of data record with search key value **k**, or
  3. <**k**, rid-list>
     - list of record ids of data records with search key **k**>

Advantages/
Disadvantages?

- Data entries typically much smaller than data records
  - So, better than Alternative 1 with large data records
  - Especially if search keys are small.

- Alternative 3 more compact than Alternative 2
  - but leads to variable-size data entries even if search keys have fixed length.

Index Classification

- Primary vs. secondary
- Clustered vs. unclustered

Primary vs. Secondary Index

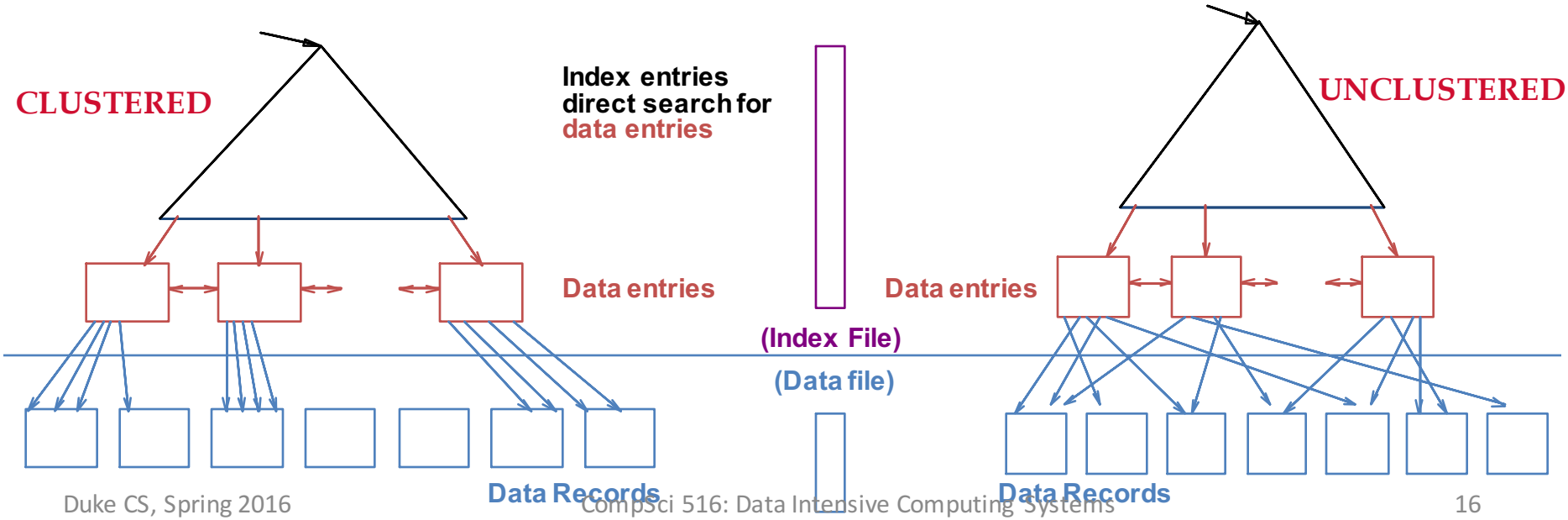
- If search key contains primary key, then called primary index, otherwise secondary
  - Unique index: Search key contains a candidate key
- Duplicate data entries:
  - if they have the same value of search key field k
  - Primary/unique index never has a duplicate
  - Other secondary index can have duplicates

Clustered vs. Unclustered Index

- If order of data records in a file is the same as, or `close to', order of data entries in an index, then clustered, otherwise unclustered
  - Alternative 1 implies clustered – 2, 3 are typically unclustered
  - 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 (range queries) 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, data records are `close to', but not identical to, sorted

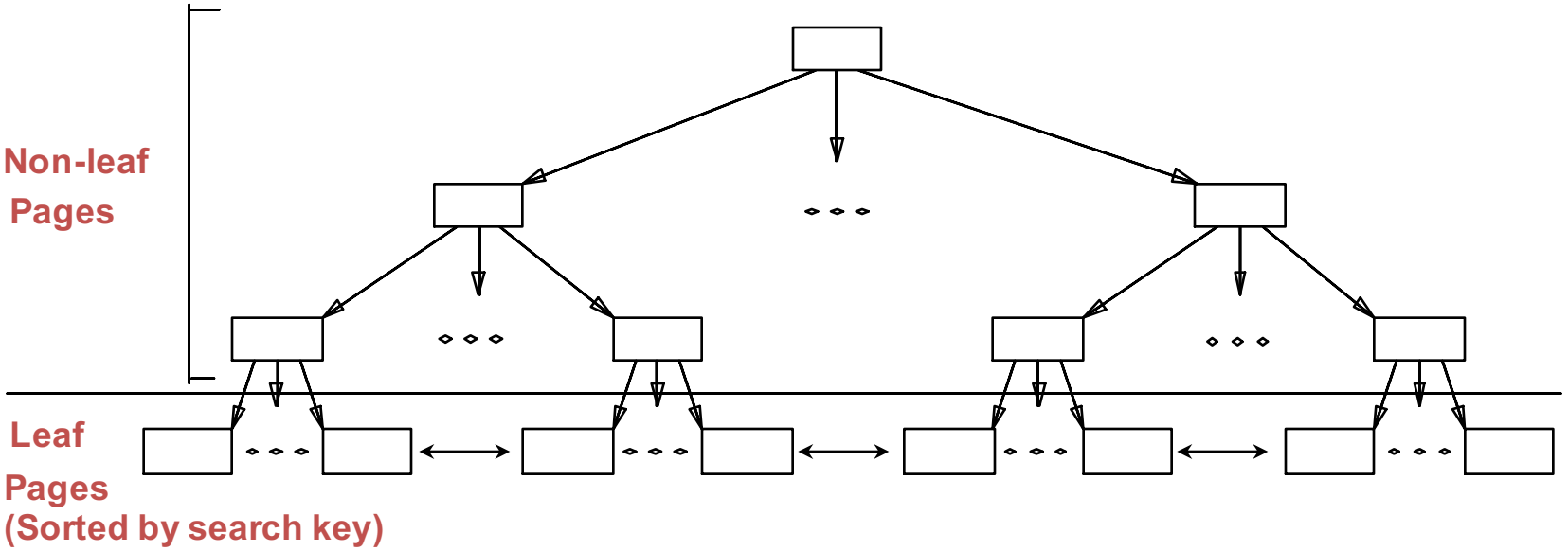


Methods for indexing

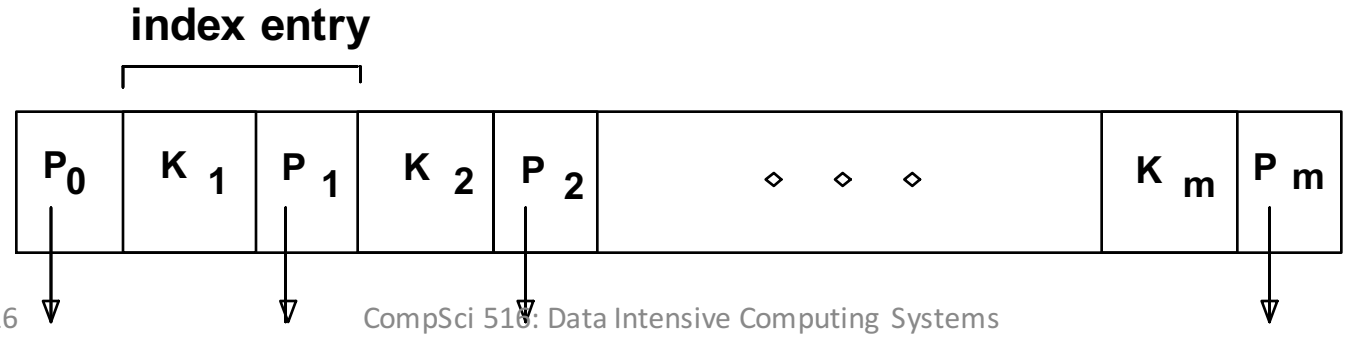
- Tree-based
- Hash-based

- In detail in the next lecture
  - Only two examples today

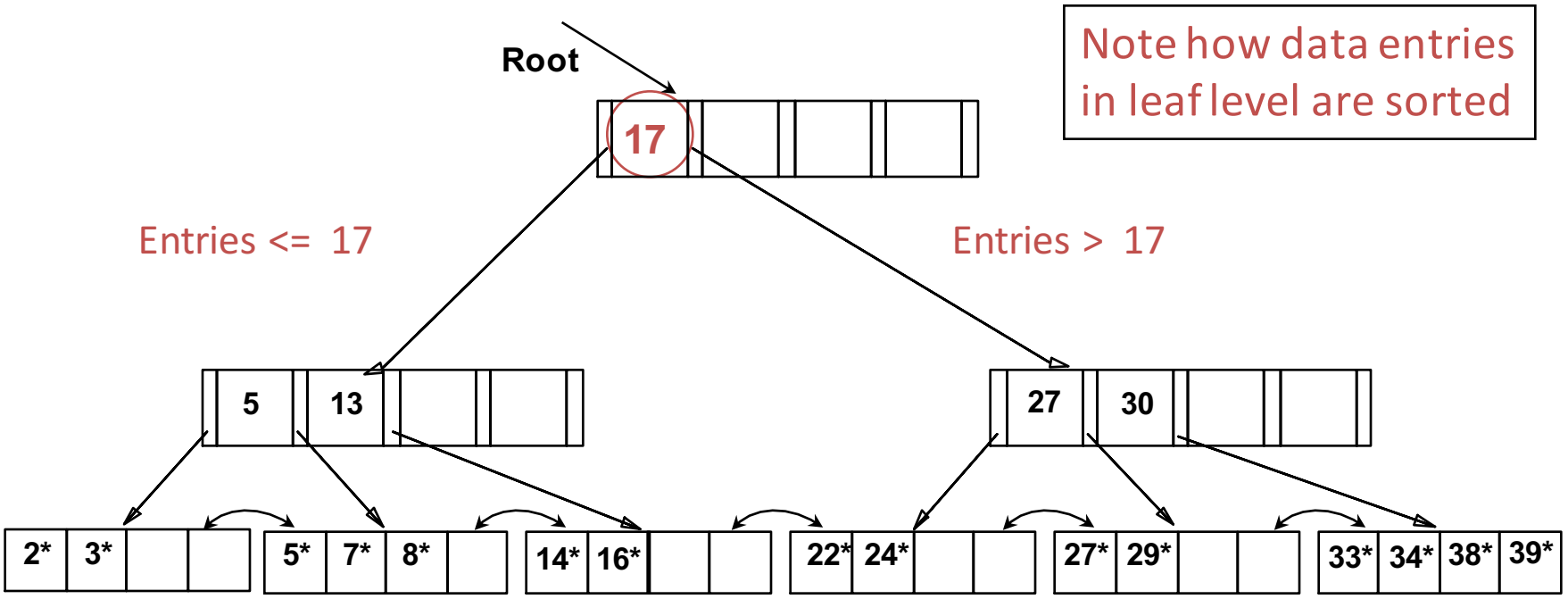
B+ Tree Indexes



- 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

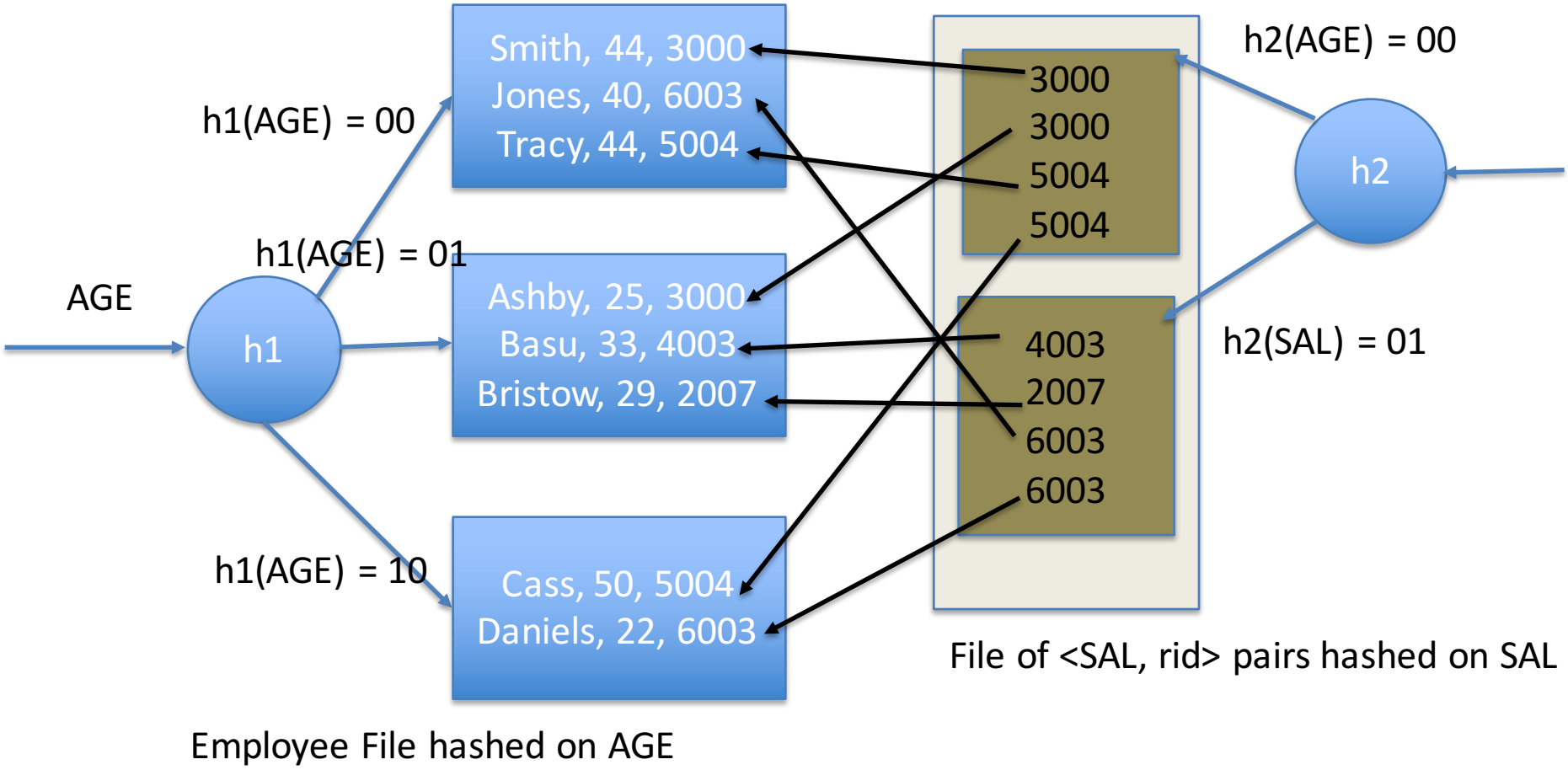
- Find
  - 28*?
  - 29*?
  - All > 15* and < 30*

Hash-Based Indexes

- Records are grouped into buckets
  - Bucket = primary page plus zero or more overflow pages
- 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
- Good for equality selections
  - find all records with name = "Joe"

Example: Hash-based index

Index organized file hashed on AGE, with Auxiliary index on SAL



Different File Organizations

We need to understand the importance of appropriate file organization and index

Search key = <age, sal>

- Heap files
  - random order; insert at end-of-file
- Sorted files
  - sorted on *<age, sal>*
- Clustered B+ tree file
  - 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>*

Possible Operations

- Scan
  - Fetch all records from disk to buffer pool
- Equality search
  - Find all employees with age = 23 and sal = 50
  - Fetch page from disk, then locate qualifying record in page
- Range selection
  - Find all employees with age > 35
- Insert a record
  - identify the page, fetch that page from disk, inset record, write back to disk (possibly other pages as well)
- Delete a record
  - similar to insert

Understanding the Workload

- A workload is a mix of queries and updates

- 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?
- For each index, what kind of an index should it be?
  - Clustered? Hash/tree?

More on Choice of Indexes

- One approach:
  - Consider the most important queries
  - Consider the best plan using the current indexes
  - 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
  - We will learn query execution and optimization later - 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 – 1/3

- 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.

Index Selection Guidelines – 2/3

- 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

Index Selection Guidelines – 3/3

- 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

- Note: clustered index should be used judiciously
  - expensive updates, although cheaper than sorted files

Examples of Clustered Indexes

- B+ tree index on E.age can be used to get qualifying tuples

What is a good indexing strategy?

- How selective is the condition?
  - everyone > 40, index not of much help, scan is as good
  - Suppose 10% > 40. Then?

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

- Depends on if the index is clustered
  - otherwise can be more expensive than a linear scan
  - if clustered, 10% I/O (+ index pages)

Examples of Clustered Indexes

Group-By query

What is a good indexing strategy?

- Use E.age as search key?
  - Bad If many tuples have *E.age* > 10 or if not clustered….
  - …using *E.age* index and sorting the retrieved tuples by E.dno may be costly
- Clustered *E.dno* index may be better
  - First group by, then count tuples with age > 10
  - good when age > 10 is not too selective
- Note: the first option is good when the WHERE condition is highly selective (few tuples have age > 10), the second is good when not highly selective

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

Examples of Clustered Indexes

What is a good indexing strategy?

Equality queries and duplicates

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

- Clustering on *E.hobby* helps
  - hobby not a candidate key, several tuples possible

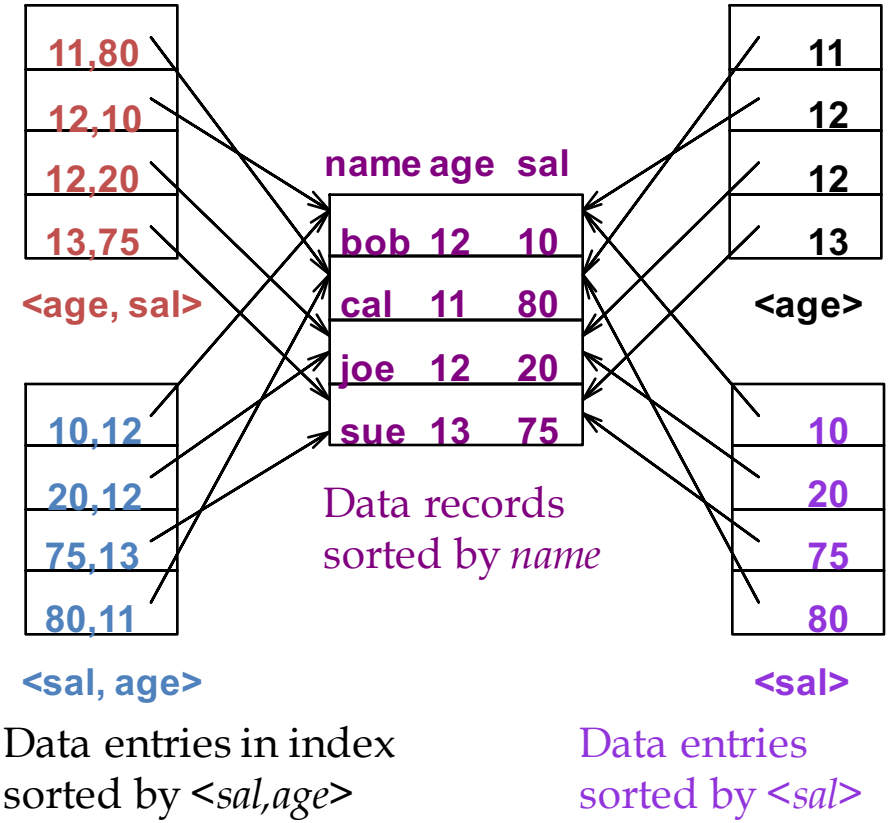
```
SELECT  E.dno
FROM  Emp E
WHERE  E.eid=50
```

- Does clustering help now?
  - Not much
  - at most one tuple satisfies the condition

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.:
  - sal > 10
  - <age, sal> does not help
  - has to be a prefix

Examples of composite key indexes using lexicographic order.



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
  - first find age = 30, among them search sal = 4000
- 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
  - more index entries are retrieved for the latter
- 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.

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

<*E.dno*>

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

<*E.dno,E.sal*>

Tree index!

<*E. age,E.sal*>

Tree index!

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