T06 Prashansa Singh

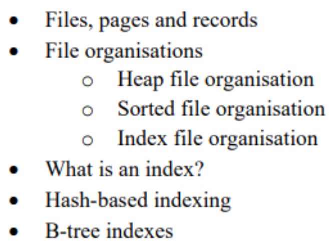
INFO20003 Tutorial 6

Starting ~2.20 pm

Today's tutorial

- Review of Storage & Indexing
- Exercises
  - group work

Github Link
https://github.com/Prashansa-Singh/INFO20003-Sem1-2022



- Files, pages and records
- File organisations
  - Heap file organisation
  - Sorted file organisation
  - Index file organisation
- What is an index?
- Hash-based indexing
- B-tree indexes

Terminology for Disk Storage

READ, WRITE Operations:

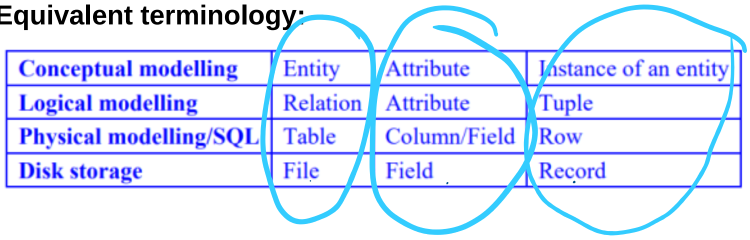
- Database management systems store information on disks
  - ○ normally hard disks
- Accessing data involves many READ and WRITE operations

- We need to bring data to main memory (RAM) to process it
  - ○ processing it in memory is much faster

- Operations
  - ○ READ = data transfered from disk to RAM
  - ○ WRITE = transfer data from RAM to disk

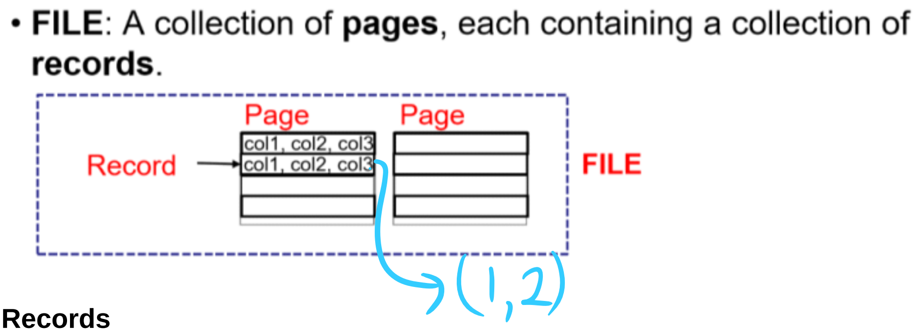
- Both operations are high cost
  - ○ But are required for storing data on secondary storage (disks)
  - ○ We need secondary storage because RAM is not large enough

Equivalent terminology:



Conceptual modelling	Entity	Attribute	Instance of an entity
Logical modelling	Relation	Attribute	Tuple
Physical modelling/SQL	Table	Column/Field	Row
Disk storage	File	Field	Record

Files, pages and records



- **FILE**: A collection of **pages**, each containing a collection of **records**.

Records

- record = individual row of table
- each record has unique rid (record id)
  - rid allows us to identify the disk address of the page containing the record with that rid
  - rid = (page ID, offset of record within that page)
  - e.g. rid of (3,7) refers to the record on page 3, in position 7
    - (the seventh record from the beginning of the third page)

Pages

- A page is an allocation of space on disk (or in memory) containing a **collection of records.**
- Usually, every page is the same size.
  - e.g. each page may be able to store 10 records of a specific relation

Files

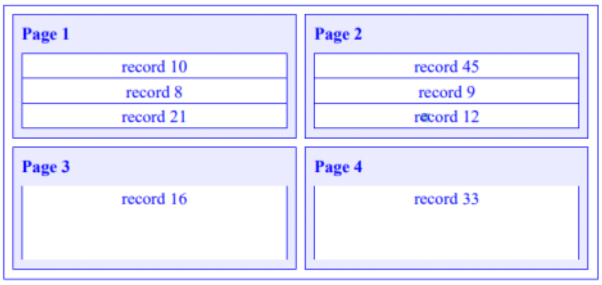
- A file = a collection of pages containing records.
- In simple database scenarios, a file corresponds to a single table.

File Organisation

- File organisation defines how file records are stored in pages on the disk
- The file is organised in one of three ways:
  - 3 ways of files organisation:
    - Heap
    - Sorted
    - Index

Heap file organisation

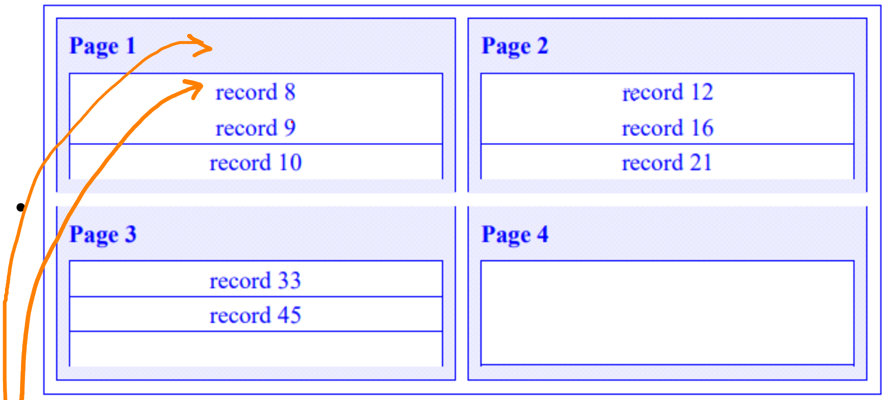
- A file organised using heap file organisation **does not support any ordering**, sequencing or indexing on its own.
- The heap file records are placed anywhere in the allocated memory area
  - Pages aren't necessarily filled up sequentially



Sorted file organisation

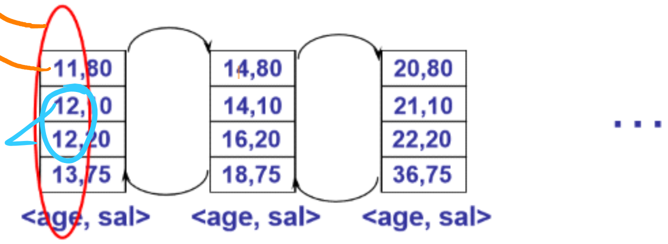
- A sorted file organisation has essentially the same structure as a heap file organisation, but in an **ordered** form.
- The sorted file records are placed in some sequential order based on the **search key(s)**
  - ○ Note: This is a different use of the word “key” which we saw in modelling topic
- Sorted file organisation is best for retrieval of a range of records in some order

- Search key does not have have unique values, e.g. age = 12



- We don't know what the search key in this example is, it could be an attribute of the relation such as age. So records from the relation are stored in order of age on the disk.

- **Example**: A sorted file ordered by *age*



Index file organisation

- In an index file organization, **a data structure on top of data files**
- This is built for efficient searching.
- Can create indexes on the fields of a table based on looking at which queries are **frequently** run against the database
- This is quite different from the above two file organisation methods because it's not about how the underlying data file is organised. We build an auxillary structure on top of the data files.

Why do we use indexes?

- An index on a relation speeds up **selection** on the search key fields.
  - WHERE ..... in SQL queries
  - An index on the fields/columns <A,B> speeds up queries which are performing selection on these columns
    - e.g. WHERE A = 5 AND B > 9

What are search key fields?

- The search key fields are the fields in which the index is built
  - Search key can be on one or more attributes
    - e.g. index on <age>
    - vs index on <age, sal>

  - Order matters!
    - An index on <age, sal> is different to an index on <sal, age>
    - One of these indexes can be more useful depending on the queries

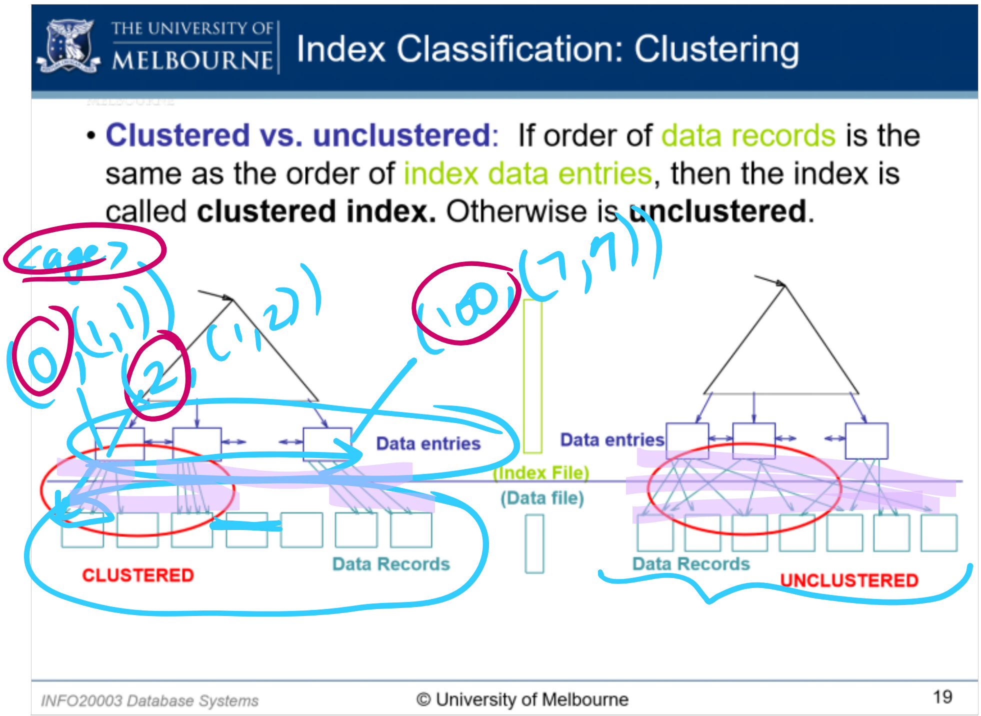
  - Search key fields are a subset of the attributes of a relation
  - Search key is not same as key (it doesn't have to be unique!)
    - It is different to keys e.g. PKs, FKs etc in modelling

What is an index?

- An index is made up of **data entries** which refer back to the data in the relation.
- Usually, data entries are represented as (k, rid)
  - k = value of search key(s)
  - rid = record ID in the relation.

- The indexes are stored in an **index file**,
- But the actual records of the relation are stored in the **data file**

- Indexes can either be
  - **clustered**
    - data records in the data file have the **same order** as data entries of the index
  - **unclustered**
    - data records in the data file are not sorted by search key/data entries
      - □ (but could be either unsorted or sorted by another attribute)



THE UNIVERSITY OF MELBOURNE

Index Classification: Clustering

- **Clustered vs. unclustered:** If order of data records is the same as the order of index data entries, then the index is called **clustered index.** Otherwise is **unclustered**.

INFO20003 Database Systems © University of Melbourne 19

- Also indexes can either be
  - **primary**
    - on the primary key of the relation
  - **secondary**
    - on any other set of attributes

- There are several indexing techniques
  - hash-based
  - tree-based (e.g. B-trees)
  - etc

How to decide which search key(s) to build an index on?

- What queries are run most frequently against the database?
- In these queries:
  - Which relations are accessed frequently?
  - Which attributes are retrieved?

In a few weeks, we will discuss how to analyse a given query plan and see if a better query plan exists with an additional index.

You should consider these points carefully before constructing an index:

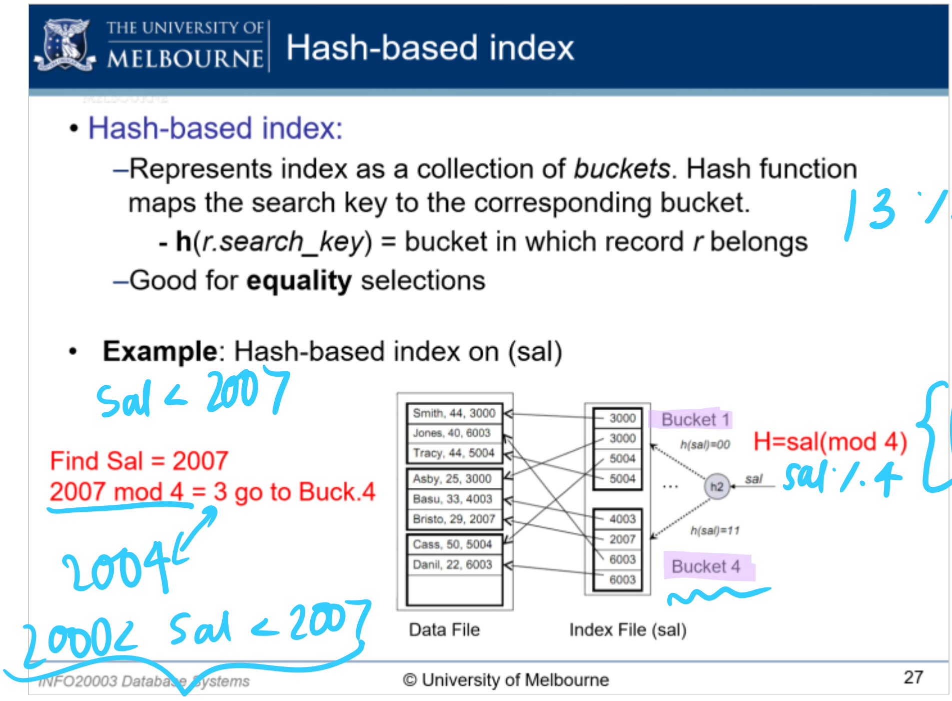
- In general, indexes make SELECT queries faster but slow down the updates
  - Because if you insert/delete/update records in data file, you may need to update the data entries in your index file
- Indexes also require additional disk space.

Hash-based Indexing

- We apply a hash function h(r)
    - h is the name of the function
    - r is the input value into the function
      - where r = field value
- Output of hash function points to a bucket
  - this is pointer to the first (primary) page of this bucket
  - and also refers to any overflow (additional) pages if the bucket has any

- **Hash indexes are the best choice for equality queries**
  - queries where the WHERE clause has an equality condition
  - e.g. WHERE ColumnA = 12;
- Hash indexes perform poorly for **range** queries (why?)

- Example of simple hash function:
  - ○ Note: we will not discuss building hash functions in depth in this subject



THE UNIVERSITY OF MELBOURNE

Hash-based index

- Hash-based index:
  - –Represents index as a collection of *buckets*. Hash function maps the search key to the corresponding bucket.
    - **h**(*r.search_key*) = bucket in which record *r* belongs
  - –Good for **equality** selections

13 % 4 = 1

- **Example**: Hash-based index on (sal)

Sal < 2007

Find Sal = 2007

2007 mod 4 = 3 go to Buck.4

20004

20002 Sal < 2007

H=sal(mod 4)

sal % 4

{0, 1, 2, 3}

INFO20003 Database Systems © University of Melbourne 27

- ○ Suppose you are given 5 buckets and h(k) = k % 5 where % is the modulus (remainder) operator.

  What are the possible output values for h(k)? {0, 1, 2, 3, 4

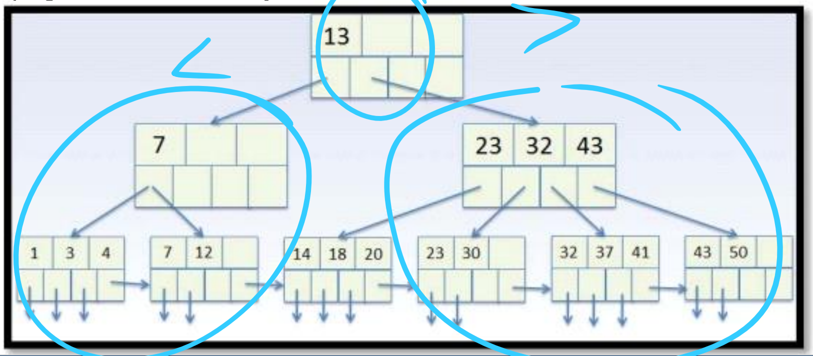
  Insert 200, 22, 119, 8, and 33 into a hash table.

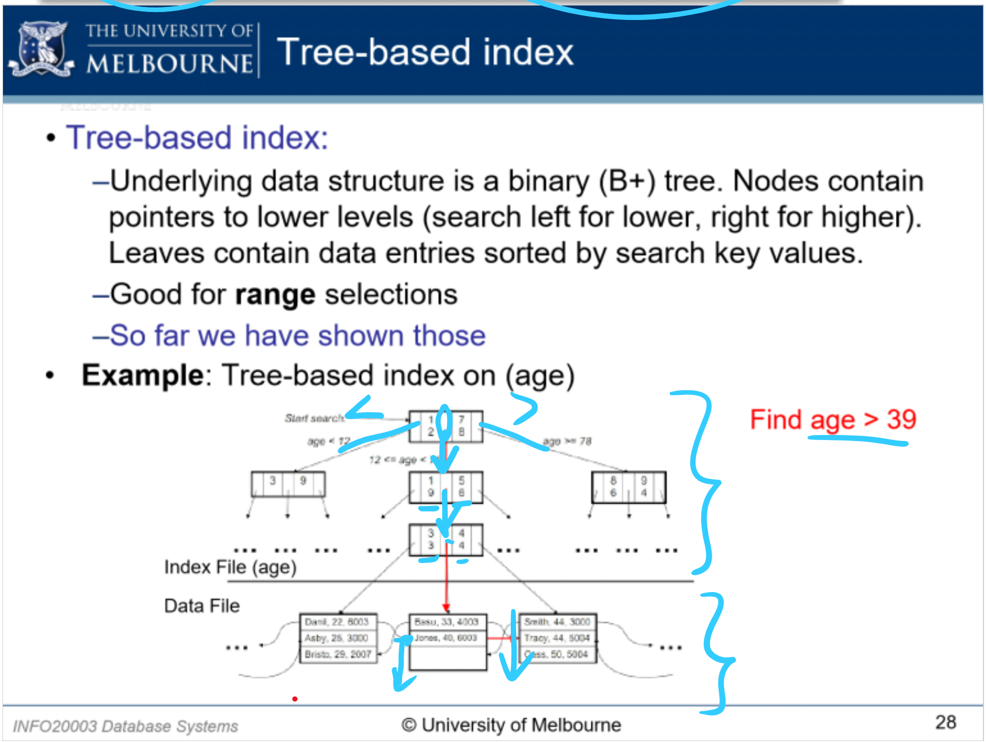
Buckets	Key
0	200
1	
2	22
3	8, 33
4	119

Solution:

Bucket	Key
0	200
1	
2	22
3	8, 33
4	119

B-tree Indexing

- A B-tree index is created by
  - ○ sorting the data on the search key
  - ○ and maintaining a hierarchical search data structure (B+ tree) that will direct the search to the respective page of the data entry.
- 

- 

THE UNIVERSITY OF MELBOURNE

Tree-based index

- Tree-based index:
  - –Underlying data structure is a binary (B+) tree. Nodes contain pointers to lower levels (search left for lower, right for higher). Leaves contain data entries sorted by search key values.
  - –Good for **range** selections
  - –So far we have shown those
- **Example**: Tree-based index on (age)

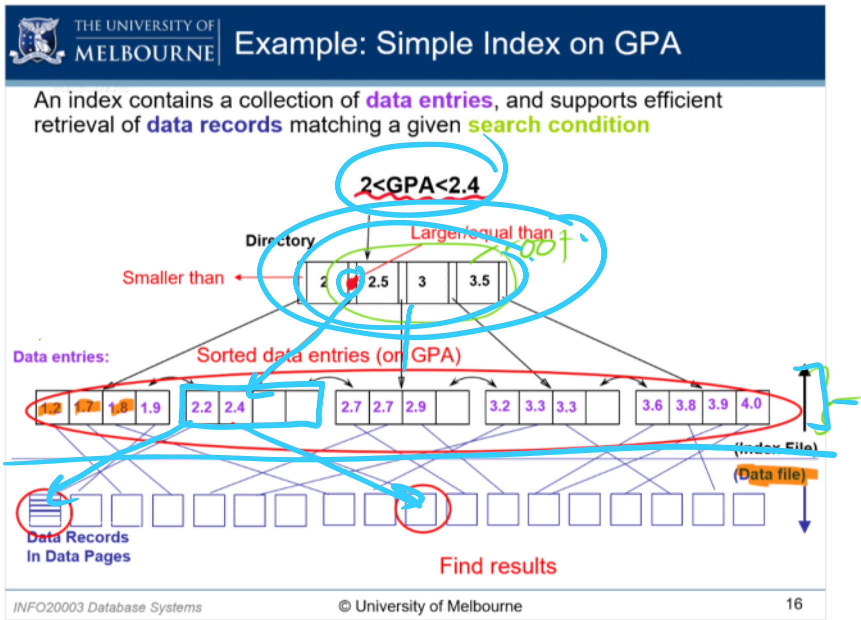
Find age > 39

INFO20003 Database Systems

© University of Melbourne

28

- Find the first record where age > 39 , then go through all following records



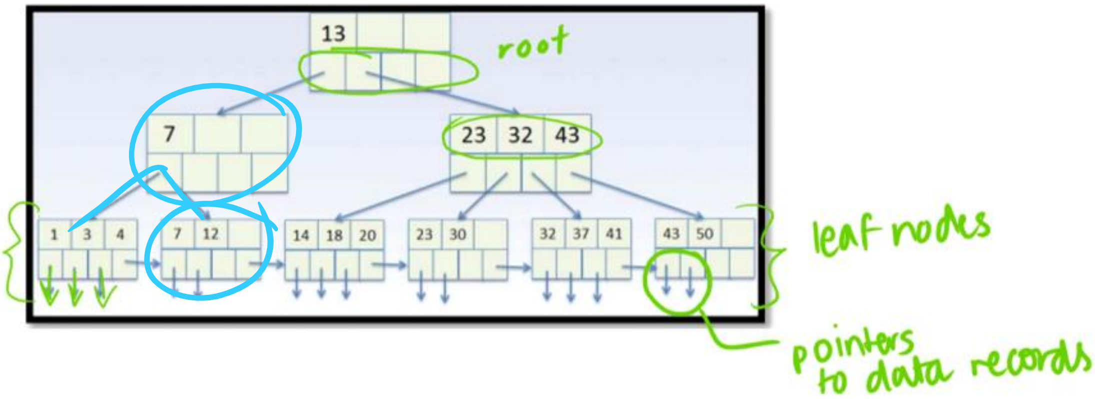
THE UNIVERSITY OF MELBOURNE

Example: Simple Index on GPA

An index contains a collection of data entries, and supports efficient retrieval of data records matching a given search condition

INFO20003 Database Systems © University of Melbourne 16

Is this clustered or unclustered?



To find a key K in a B-tree

1) Start at the root.
2) For internal nodes, search the keys to find the range K belongs in and follow that pointer.
3) For leaf nodes (nodes with no child nodes), search the keys to find K and follow the pointer to find the data record of interest.

- Insertion in such a structure is costly as the tree is updated with every insertion or deletion.
  - There will be situations in which a major part of tree will be re-written if a particular node is overfilled or under-filled.
  - A detailed understanding of the mechanisms for B-tree modification (insertion of new nodes and deletion of existing nodes) is not required for this subject.

- Ideally the tree will automatically maintain an appropriate number of levels, as well as optimal space usage in blocks.

Exercises:

1. Choosing an index

You are asked to create an index on a suitable attribute. What are the important aspects you will analyse to make this decision? To get you started, the following might help you by providing scaffolding to the discussion:

- Primary vs. secondary index = vs >
- Clustered vs. unclustered index

- 6 Primary vs. secondary index
- Clustered vs. unclustered index
- 8 Hash vs. tree indexes

= vs >

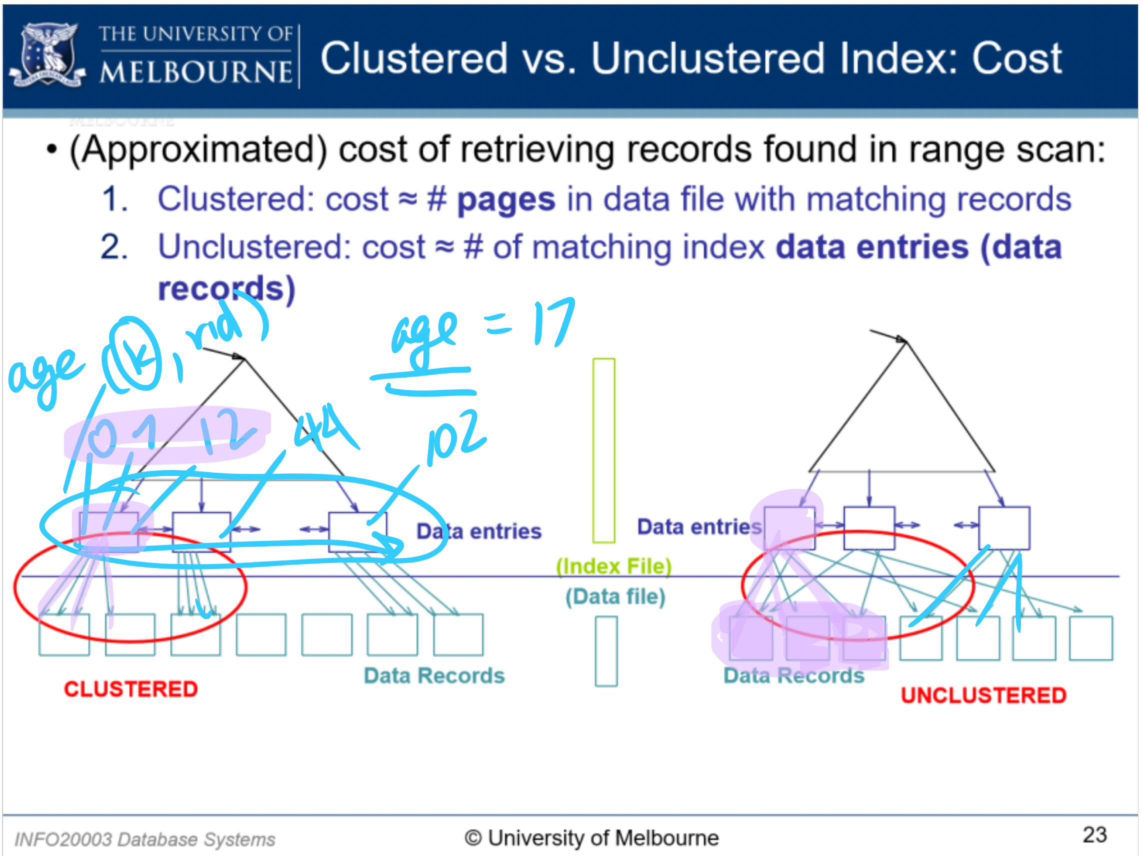
Primary vs. secondary index

- The primary key can be used as the search key in the cases where the records are retrieved based on the value of primary key.
    - Selection occurs on the primary key column

- Generally, a table should always have a primary index (MySQL creates one automatically).
- Otherwise the secondary indexes should be preferred using fields that are frequently used in the queries.

Clustered vs. unclustered index

- For **range** queries (query consists of a condition to check for a range)
    - a clustered index is preferred as compared to unclustered.
    - Why?



THE UNIVERSITY OF MELBOURNE

Clustered vs. Unclustered Index: Cost

- (Approximated) cost of retrieving records found in range scan:
    1. Clustered: cost ≈ # **pages** in data file with matching records
    2. Unclustered: cost ≈ # of matching index **data entries (data records)**

INFO20003 Database Systems © University of Melbourne 23

- However, for **equality** conditions:
    - it does not have any advantage over the unclustered index if the search key does not have duplicate values.
    - Which is better if duplicate values exist?

    - Search key does not have to be unique!
        - e.g. WHERE age = 12
            - many records may be returned, which could be stored on different pages
        - We can only say search key values are unique if search key contains the PKs of the relation

- Clustered indexes are **more expensive** to maintain as compared to unclustered indexes

- As the update/delete/insert might lead to reordering of the records.
- Therefore, the index should only be clustered if there is a **frequently-executed range query.**

- In addition, when more than one combination of columns is used in range queries, you should **choose the most frequently used combination** and make those fields search keys of the clustered index.
  - <age, sal>

Hash vs. tree indexes

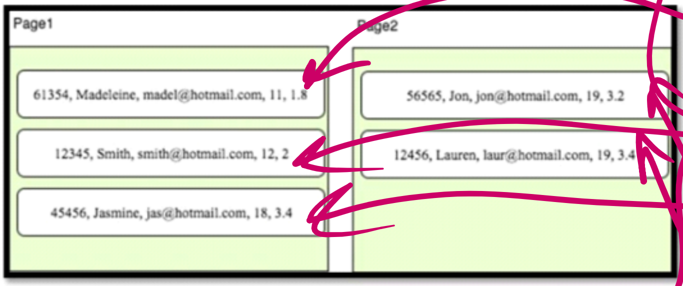
- Similar to clustered and unclustered indexes:
  - If we want to perform many **range** queries:
    - creating a **B-tree** index for the relation is the best choice.
  - On the other hand, if there are **equality** queries
    - then **hash indexes** are the best choice
    - e.g. equality queries such as driving licence numbers,
    - As hash indexes allow faster retrieval than a B-tree in such cases
      - B-trees will still increase the speed of most equality queries, but hash indexes are even faster
    - Why is hash bad for range queries?

2. Data entries of an index:

Consider the following instance of the relation Student (SID, Name, Email, Age, GPA):

SID	Name	Email	Age	GPA
61354	Madeleine	madel@hotmail.com	11	1.8
12345	Smith	smith@hotmail.com	12	2.0
45456	Jasmine	jas@hotmail.com	18	3.4
56565	Jon	jon@hotmail.com	19	3.2
12456	Lauren	laur@hotmail.com	19	3.4

As you can see the tuples are sorted by age and we are assuming that the order of tuple is the same when stored on disk. The first record is on page 1 and each page can contain only 3 records. The arrangement of the records is shown below:



Page1

61354, Madeleine, madel@hotmail.com, 11, 1.8

12345, Smith, smith@hotmail.com, 12, 2

45456, Jasmine, jas@hotmail.com, 18, 3.4

Page2

56565, Jon, jon@hotmail.com, 19, 3.2

12456, Lauren, laur@hotmail.com, 19, 3.4

k, rid

age value	rid
11	(1,1)
12	(1,2)
18	(1,3)
19	(2,1)
19	(2,2)

Show what the *data entries* of the index will look like for:

a. An index on Age
b. An index on GPA

clust

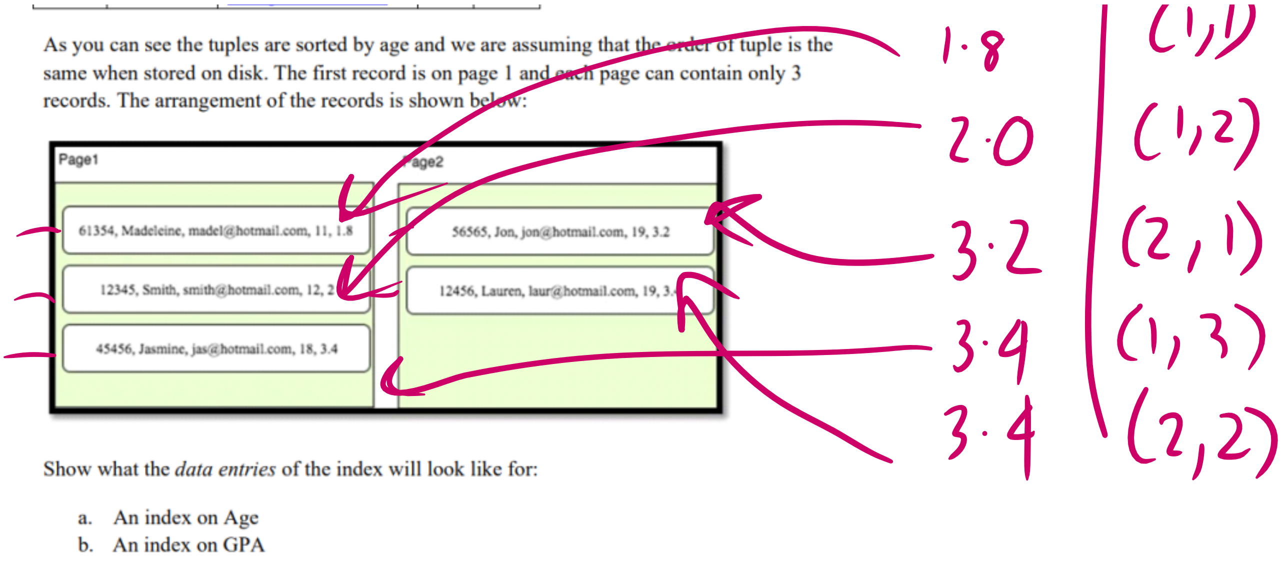
2. Data entries of an index:

Consider the following instance of the relation Student (SID, Name, Email, Age, GPA):

SID	Name	Email	Age	GPA
61354	Madeleine	madel@hotmail.com	11	1.8
12345	Smith	smith@hotmail.com	12	2.0
45456	Jasmine	jas@hotmail.com	18	3.4
56565	Jon	jon@hotmail.com	19	3.2
12456	Lauren	laur@hotmail.com	19	3.4

As you can see the tuples are sorted by age and we are assuming that the order of tuple is the same when stored on disk. The first record is on page 1 and each page can contain only 3 records. The arrangement of the records is shown below:

GPA	rid
1.8	(1,1)



As you can see the tuples are sorted by age and we are assuming that the order of tuple is the same when stored on disk. The first record is on page 1 and each page can contain only 3 records. The arrangement of the records is shown below:

Page1

61354, Madeleine, madel@hotmail.com, 11, 1.8

12345, Smith, smith@hotmail.com, 12, 2

45456, Jasmine, jas@hotmail.com, 18, 3.4

Page2

56565, Jon, jon@hotmail.com, 19, 3.2

12456, Lauren, laur@hotmail.com, 19, 3.4

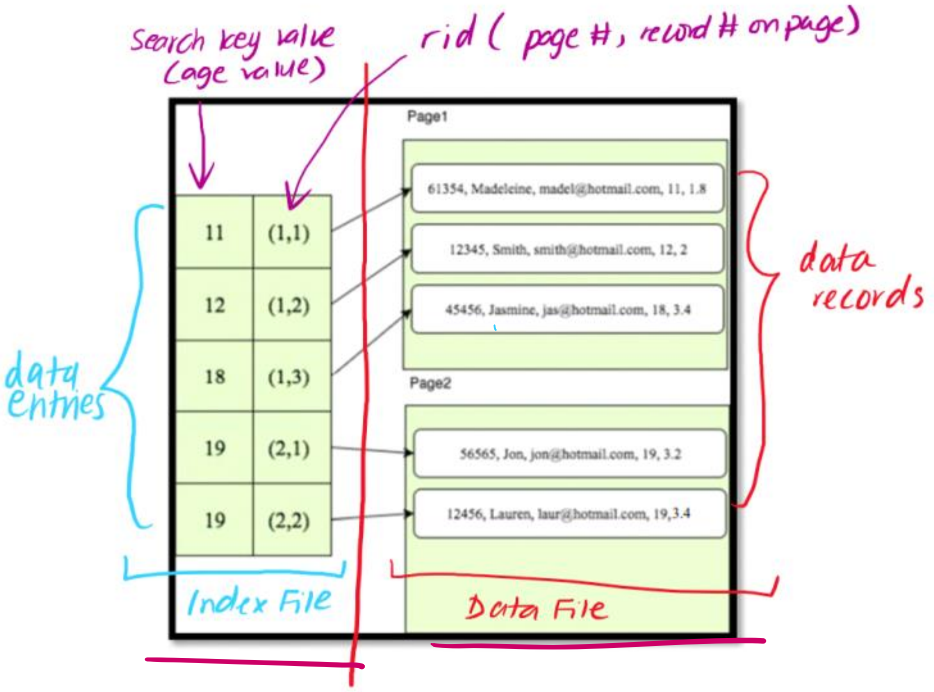
1.8	(1,1)
2.0	(1,2)
3.2	(2,1)
3.4	(1,3)
3.4	(2,2)

Show what the *data entries* of the index will look like for:

a. An index on Age
b. An index on GPA

Are these clustered or unclustered indexes?

1. Index on Age



11	(1,1)
12	(1,2)
18	(1,3)
19	(2,1)
19	(2,2)

61354, Madeleine, madel@hotmail.com, 11, 1.8

12345, Smith, smith@hotmail.com, 12, 2

45456, Jasmine, jas@hotmail.com, 18, 3.4

56565, Jon, jon@hotmail.com, 19, 3.2

12456, Lauren, laur@hotmail.com, 19,3.4

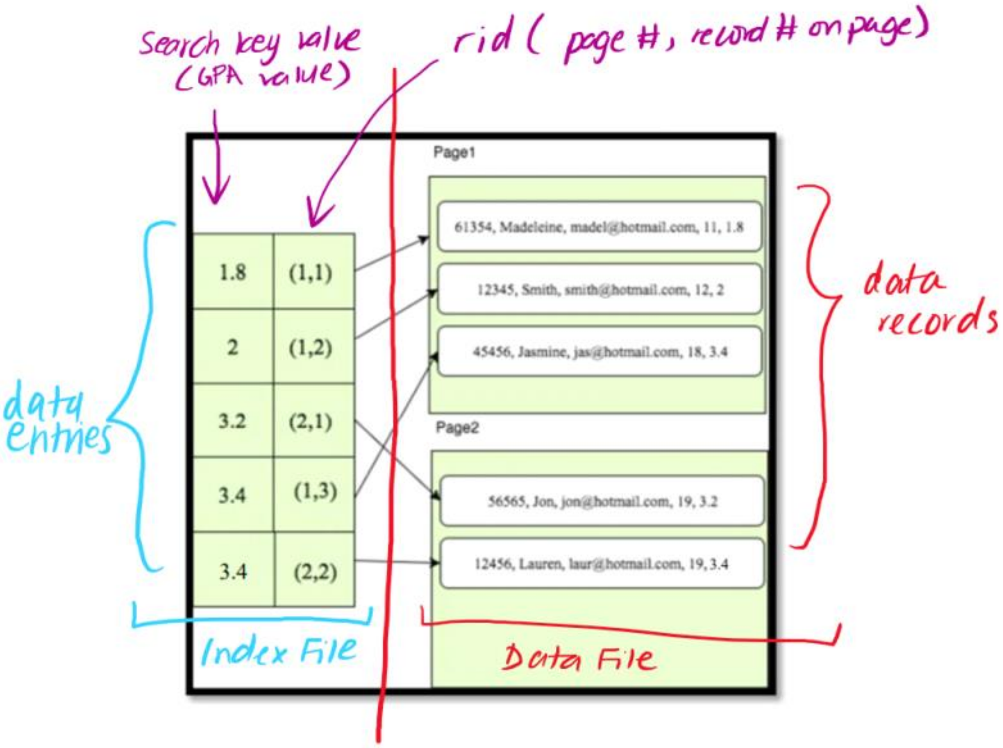
The index contains the search key and rid in the format (a, b), where a is the page number and b is the record number.

As the file is sorted on age and the data entries have the same order as data records, we have constructed a clustered index.

1. Index on GPA

The records in the file are not sorted in order of GPA.
As the records are not sorted on GPA, the index created will be unclustered.

The data entries (leaf pages) of the unclustered index would look like this:


Search key value
(GPA value)
rid (page #, record # on page)
Page1
61354, Madeleine, madel@hotmail.com, 11, 1.8
12345, Smith, smith@hotmail.com, 12, 2
45456, Jasmine, jas@hotmail.com, 18, 3.4
Page2
56565, Jon, jon@hotmail.com, 19, 3.2
12456, Lauren, laur@hotmail.com, 19, 3.4
1.8 (1,1)
2 (1,2)
3.2 (2,1)
3.4 (1,3)
3.4 (2,2)
data entries
data records
Index File
Data File

3. Consider the following relations:

Employee (<u>EmployeeID</u>, EmployeeName, Salary, Age, $\overset{\text{FK}}{\text{DepartmentID}}$)

Department (<u>DepartmentID</u>, DepartmentBudget, DepartmentFloor, $\overset{\text{FK}}{\text{ManagerID}}$)

In the database, the salary of employees ranges from AUD10,000 to AUD100,000, age varies from 20-80 years and each department has 5 employees on average. In addition, there are 10 floors, and the budgets of the departments vary from AUD10,000 to AUD 1million.

Given the following two queries frequently used by the business, which index would you prefer to speed up the query? Why?

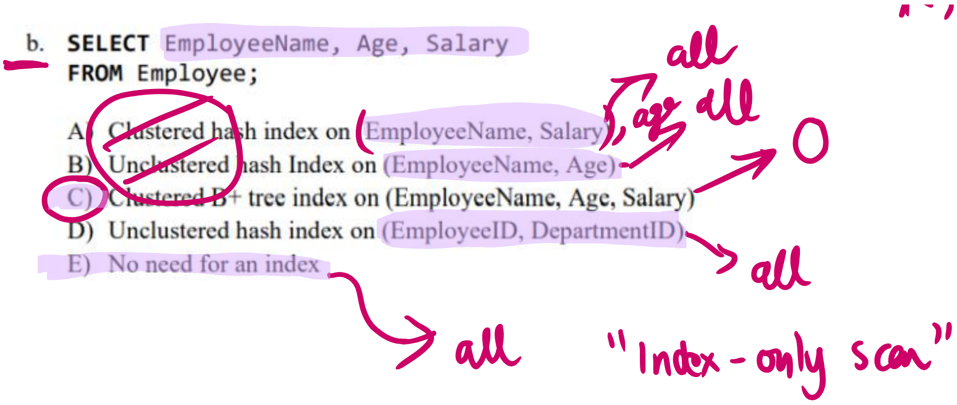
a.
```
SELECT DepartmentID
FROM Department
WHERE DepartmentFloor = 10
  AND DepartmentBudget < 15000;
```

A) Clustered Hash index on DepartmentFloor
B) Unclustered Hash Index on DepartmentFloor
[✓ C)] Clustered B+ tree index on (DepartmentFloor, DepartmentBudget) ⇐
D) Unclustered hash index on DepartmentBudget
E) No need for an index

k, rid

b.
```
SELECT EmployeeName, Age, Salary
FROM Employee;
```

all

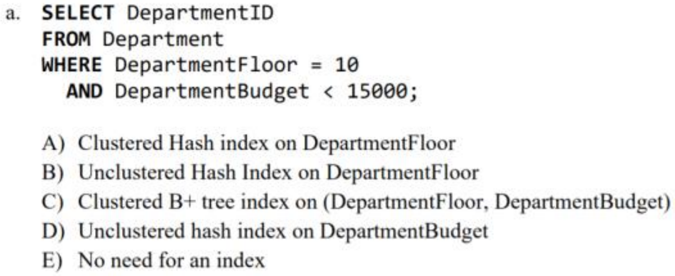


b. SELECT EmployeeName, Age, Salary
FROM Employee;

A) ~~Clustered hash index on~~ (EmployeeName, Salary), age → all
B) ~~Unclustered hash Index on~~ (EmployeeName, Age) → all
[✓ C)] ~~Clustered B+~~ tree index on (EmployeeName, Age, Salary) → 0
D) Unclustered hash index on (EmployeeID, DepartmentID) → all
E) No need for an index → all

"Index-only scan"

a



```
a. SELECT DepartmentID
   FROM Department
   WHERE DepartmentFloor = 10
     AND DepartmentBudget < 15000;
```

A) Clustered Hash index on DepartmentFloor
B) Unclustered Hash Index on DepartmentFloor
C) Clustered B+ tree index on (DepartmentFloor, DepartmentBudget)
D) Unclustered hash index on DepartmentBudget
E) No need for an index

Lets go through all the options

A, B, D) all use hash indexes which perform very poorly for range queries.

- You'd likely have to go into all the buckets and hence access all the pages of the data file, which is very high cost.

E

- If we didn't have an index, we would have to read all the pages in the data file. This is more expensive than the best option (C). We do want to make an index as this will make the SQL query more efficient.

C

- This is the best option
- **B tree is a good idea to deal with both equality and range queries.**
  - B-trees are good at equalities and the best for range.
  - Hash indexes are the best at equalities but terrible for range.

- The index is primarily on DepartmentFloor, then secondarily on Budget. So it first sorts on Floor, then whenever values of Floor are the same, it sorts on Budget (breaks tie in Floor values using the budget value) which is what we want

- As it is clustered, the data records in the data file are sorted in this order. So all the floor 10 records are next to each other, and after all the floor 9 records. Within the floor 10 records, the are sorted in increasing order of budget.

- This means we can quickly use B-tree to f**ind where the first record for floor 10 is, and then read the records onwards from that.**
  - We know the first record for floor 10 will have the lowest budget value, so start reading from there and go until we see a budget value higher than 15K, then we can stop.
  - We know all the records after that have a higher budget value and there's no need to read them.

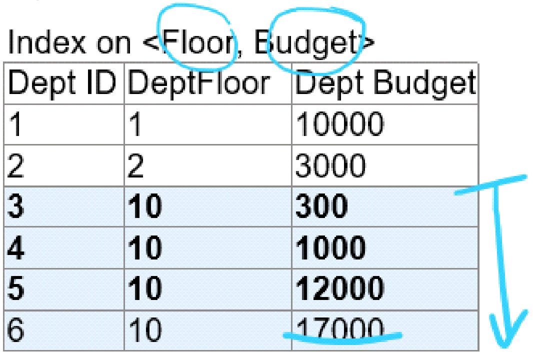
- All these records we read in are stored **sequentially**, in adjacent pages (since Clustered index). Hence, this is the **cheapest option**. We are **only ever accessing the records with Floor 10** and budget < 15K (except for the final record which tells us to stop as we've exceeded budget of 15K).
- This is better than A because in A we have to (at least) look through all the records for floor 10. In C) we only look at a subset of the records for floor 10.
- Hence C is best option

In this case, the best option will be clustered B+ tree index on floor and budget field

(option C). The records will be ordered on the two fields that are used in the WHERE clause. The query will be executed in such a way that the first record with DepartmentFloor = 10 will be accessed. After that the following records will be read continuing from there in the order of budget.

- What if we changed order of fields in the index?

Original Index



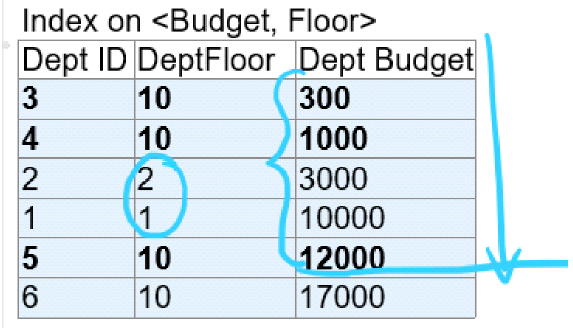
Index on <Floor, Budget>

Dept ID	DeptFloor	Dept Budget
1	1	10000
2	2	3000
3	**10**	**300**
4	**10**	**1000**
5	**10**	**12000**
6	10	17000

Blue = rows of table which were scanned

We start from the first record of floor 10, look through subsequent records as they are sorted from lowest to highest budget. We stop after we've read a budget > 15K.

Flipped order Index on <Budget, Floor>:



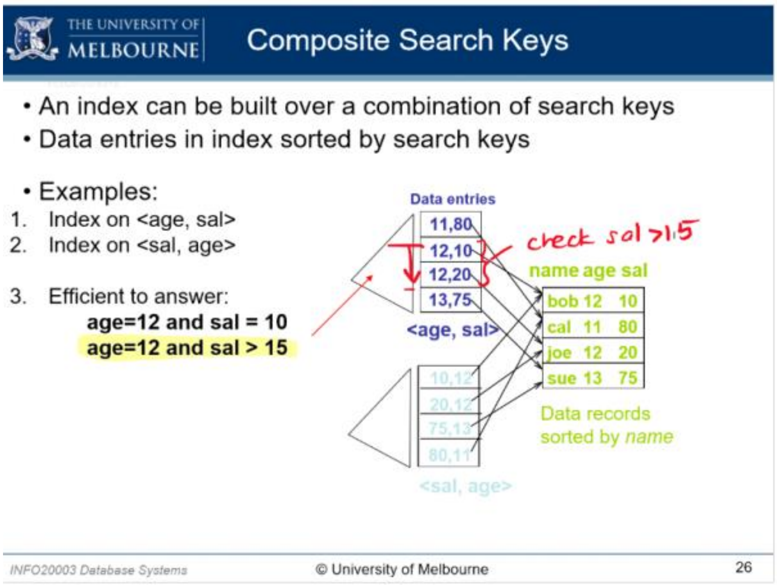
Index on <Budget, Floor>

Dept ID	DeptFloor	Dept Budget
3	**10**	**300**
4	**10**	**1000**
2	2	3000
1	1	10000
5	**10**	**12000**
6	10	17000

We look through all the records from the start until you hit budget of over 15K. But these records could be from all of the floors (not just from floor 10). So, you have to look through more records to answer this query with this flipped order index.
This index is less efficient to answer this query.
But would be efficient for a query of the form e.g. Budget = 10000, DeptFloor < 5

Notice an index on <Floor, Budget> is efficient to answer a query of the form where Floor is an equality query and Budget is a range query.

This is similar to this example in the lecture with an index on <age, sal> which is efficient to answer a query where age is an equality query and sal is a range query (age = 12 and sal > 15)



THE UNIVERSITY OF MELBOURNE

Composite Search Keys

- An index can be built over a combination of search keys
- Data entries in index sorted by search keys

- Examples:

1. Index on <age, sal>
2. Index on <sal, age>

3. Efficient to answer:

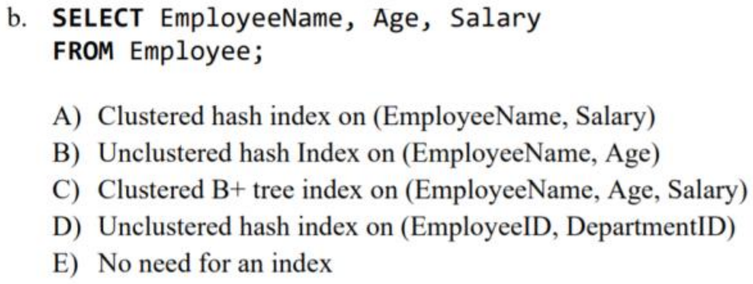
   age=12 and sal = 10

   age=12 and sal > 15

name	age	sal
bob	12	10
cal	11	80
joe	12	20
sue	13	75

INFO20003 Database Systems © University of Melbourne 26

===



b. **SELECT** EmployeeName, Age, Salary
FROM Employee;

A) Clustered hash index on (EmployeeName, Salary)
B) Unclustered hash Index on (EmployeeName, Age)
C) Clustered B+ tree index on (EmployeeName, Age, Salary)
D) Unclustered hash index on (EmployeeID, DepartmentID)
E) No need for an index

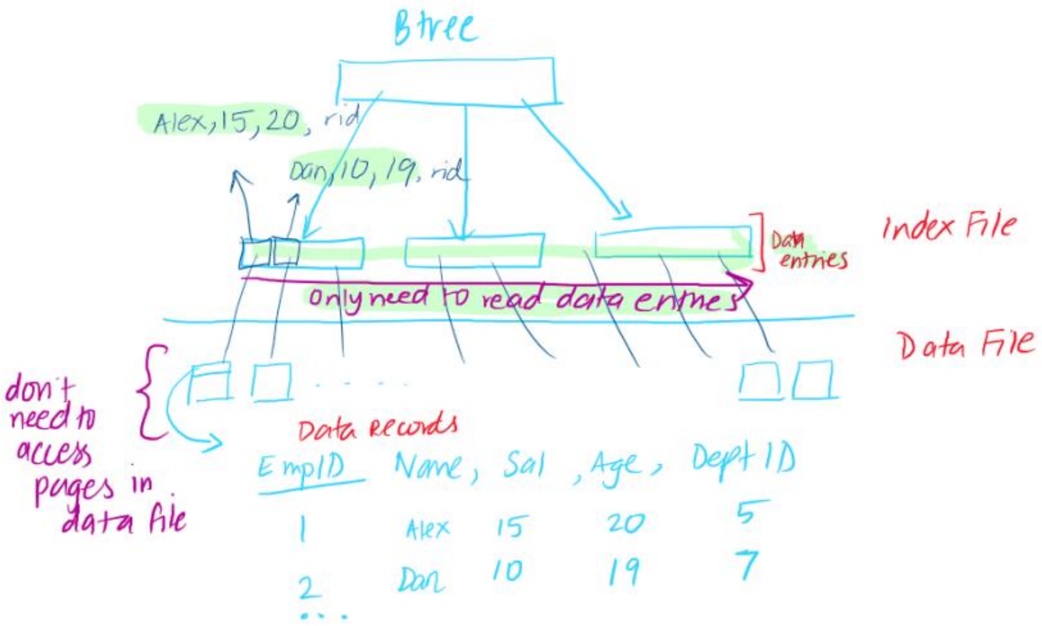
b)
Option (C) - the Clustered B+ tree index on EmployeeName, Age and Salary fields will help in this case, as we can get requested attributes with an **index-only scan** (and we can avoid accessing the table completely).

An index-only scan is an index scan without subsequently accessing the data pages – hence, accessing the index only (we only consider the data entries in the index itself to get the answer, since it has all the attributes we need… **no need to go to the actual datafile itself!)**

You never have to access any records in pages. This is the lowest cost option since accessing a page is expensive
Also note, we don't need all the columns of the table. We just want the 3 columns specified in the query

All the information we need is stored in the data entries!

- e.g. a data entry is Alex, 15, 20, rid
- Remember the data entries of an index stores the search key(s) values and the rid. The search keys values for this index in (C) are just the values of the name, age and salary we are looking for. We don't need to know any more information and hence don't need to access any pages in the data file.



Notice the attributes we want to return are the columns EmployeeName, Age, and Salary.
Lets look at the other examples
A)

- Notice this is an index only on 2 of 3 attributes we want to get in our result. **We are missing the Age attribute. So we would still have to actually go into the data file** and access the records in the pages in order to find the value for that attribute. This is more expensive than (C) since we are doing page accesses.
- We would need to **acccess all the buckets**, then for each data entry in the index, go to the corresponding record in the data file.
- All pages would get accessed.

B)

- This is only index on name and age, it doesn't have the salary attribute. S**imilar to A) we'd have to go into the data file,** access records so we could get the salary attribute. More expensive than (C) since doing page accesses

D)

- The indexes here are on attributes that are **not even relevant to the SQL query.**
- So the data entries in the index aren't sufficient and we'd have to access the records in data file to get the name, age and salary attributes.More expensive than (C) since doing page accesses

E)

- If we did E, we'd access **all the pages in the data file**. But this is more expensive than C since doing page acceses.