DB Week 6 Workshop

INFO20003 | Sandy Luo

Workshop Overview

01

Storage & Indexing Review

02

Practice Questions

03

Lab: More SQL

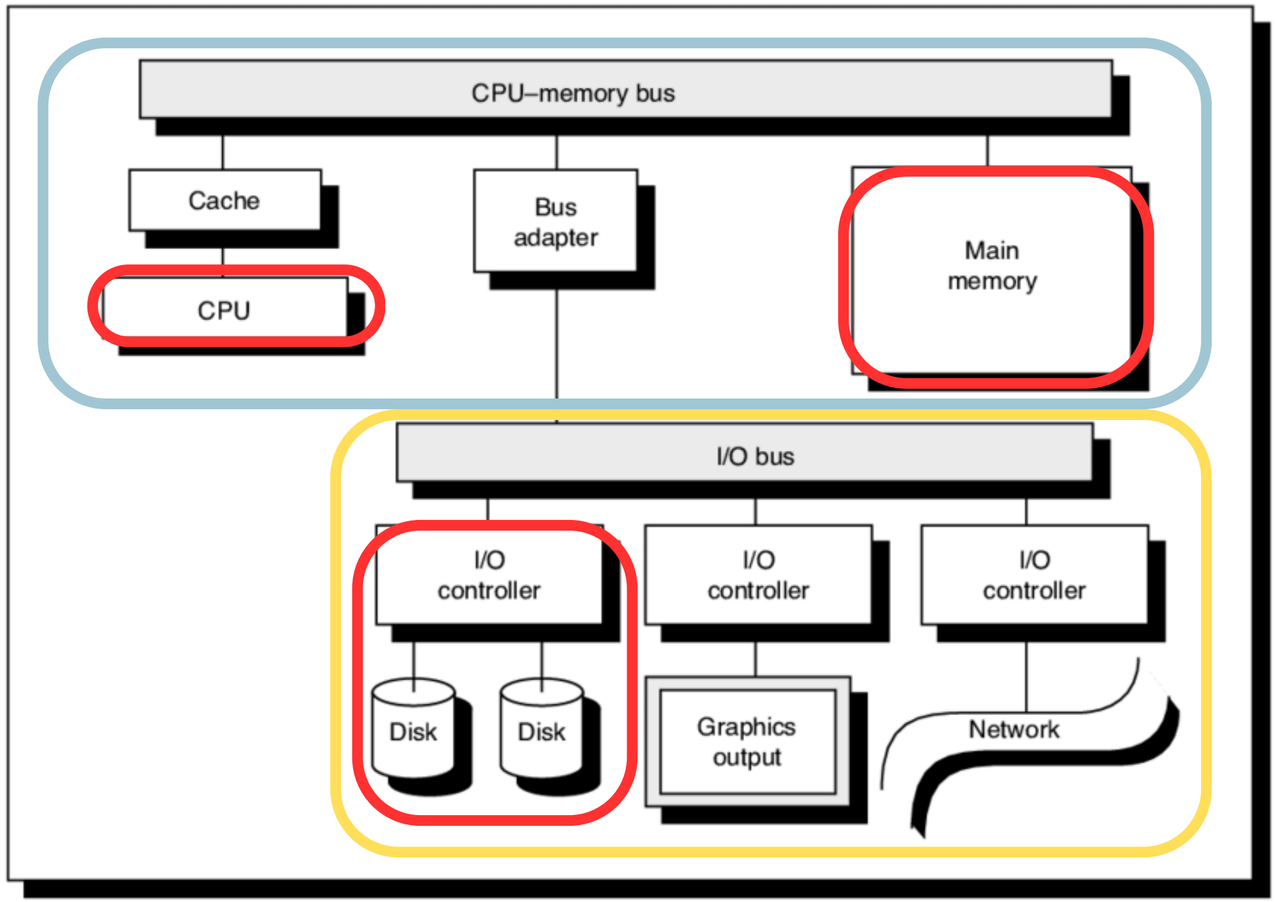
What’s DB Storage / Indexing?

- DBMS needs to support CRUD for data instances:
  - Create
  - Read
  - Update
  - Delete

- How would data be stored on disks (generally hard disks)?

- **READ**:
  - Transfer data from Disk → RAM
- **WRITE**:
  - Transfer data from RAM → Disk
- Very costly!

fast!



slow!

Terminology

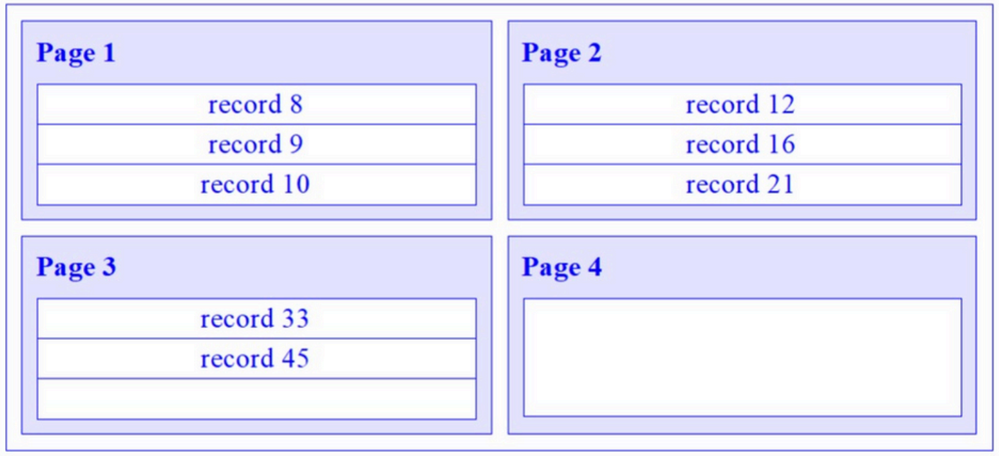
Conceptual modelling	Entity	Attribute	Instance of an entity
Logical modelling	Relation	Attribute	Tuple
Physical modelling/SQL	Table	Column/Field	Row
Disk storage	File	Field	Record

Physical Model / SQL:

	employee_first	employee_last	departmentID	boss_first	boss_last
▶	Rita	Skeeter	2	Clare	Underwood
	Gigi	Montez	3	Clare	Underwood
	Maggie	Smith	3	Clare	Underwood
	Paul	Innit	4	Andrew	Jackson
	James	Mason	4	Andrew	Jackson
	Pat	Clarkson	5	Andrew	Jackson
	Sanjay	Patel	6	Andrew	Jackson
	Mark	Zhang	7	Andrew	Jackson

table column / field row

Disk storage:



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

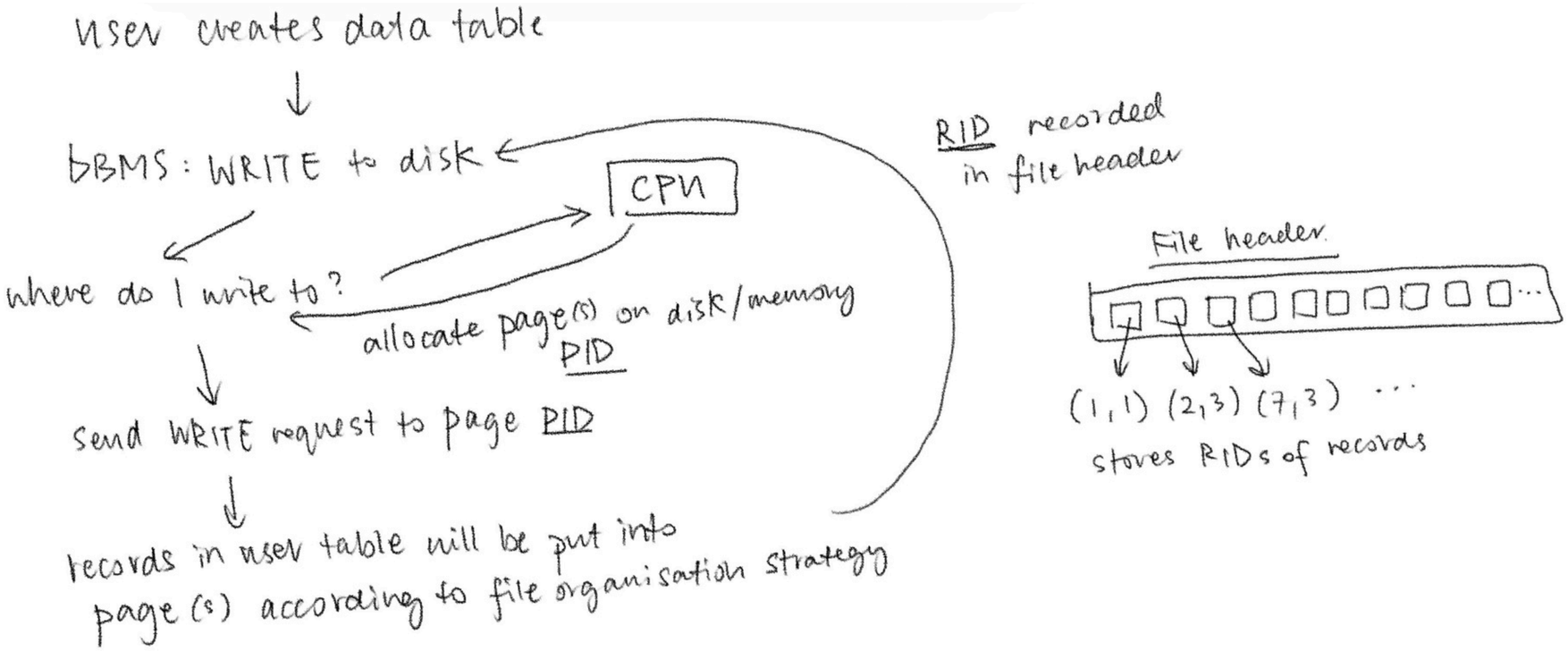
1. **Record**: A row in a table w/ unique *rid*
    - *rid*: Identify disk address of record
    - e.g. (3, 7) = 7th record on the 3rd page

2. **Page**: An allocation of space on disk / memory containing records
    - Typically, pages are the same size

3. **File**: Collection of pages
    - In simple database scenarios, file <-> table

File Organisation

- There are different ways for file records to be mapped onto pages
- Intuition:



01

Heap File Organisation

02

Sorted File Organisation

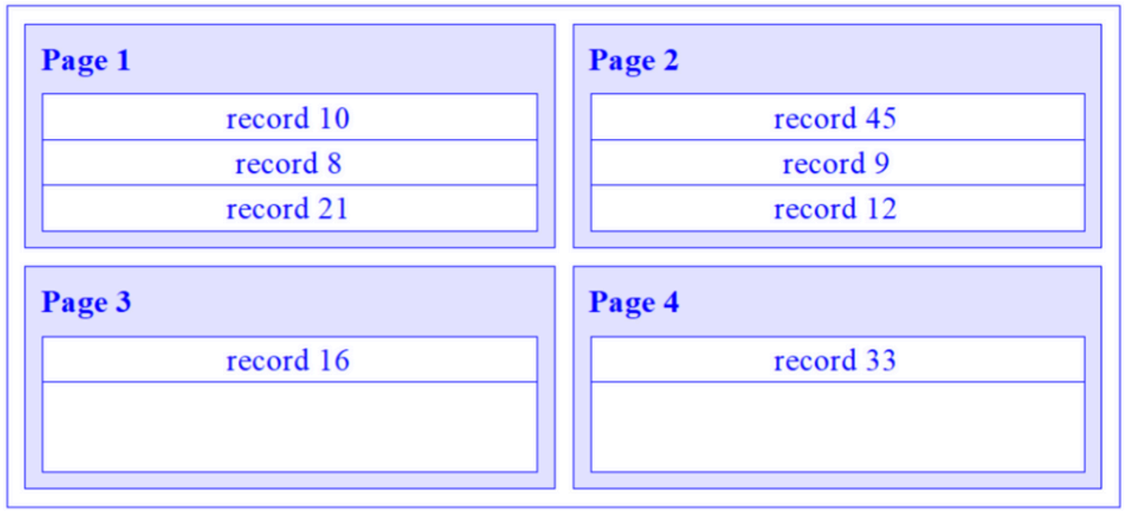
03

Index File Organisation

Storage & Indexing

1. Heap File Organisation

- No ordering, sequencing or indexing (random order)



- Advantage: Fast insertion / deletion
- Disadvantage: Slow searching; need to go through all records

01

Heap File Organisation

02

Sorted File Organisation

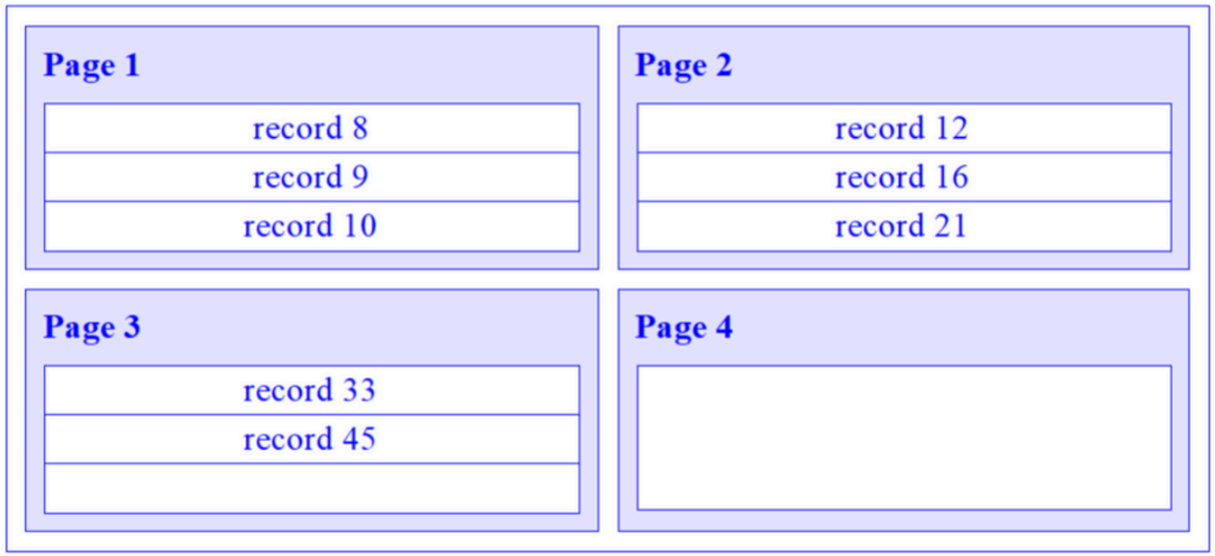
03

Index File Organisation

Storage & Indexing

2. Sorted File Organisation

- Records **ordered** based on **search key**



- Advantage: Fast range / equality query
- Disadvantage: Slow insertion / deletion (need to sort)

01

Heap File Organisation

02

Sorted File Organisation

03

Index File Organisation

Storage & Indexing

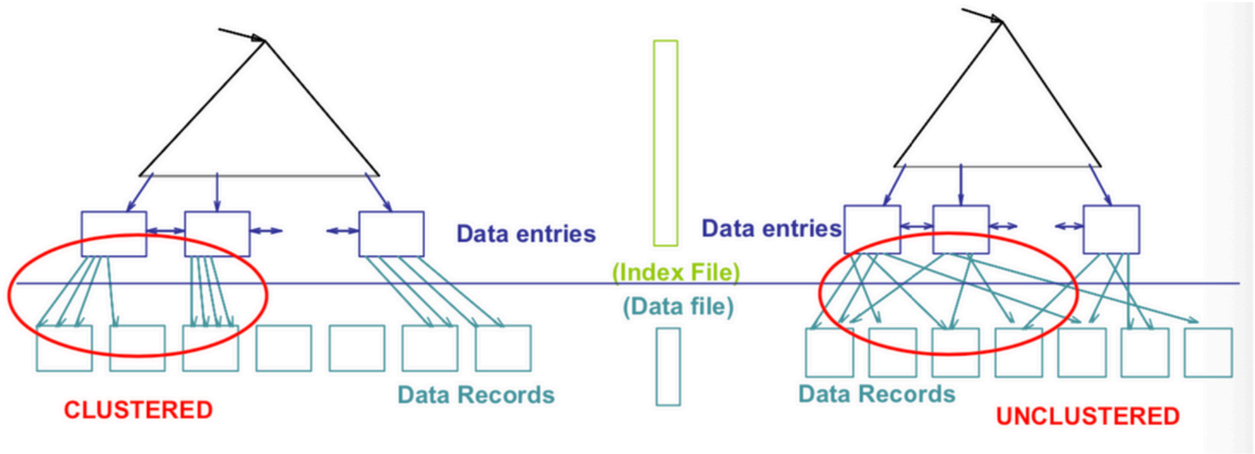
Index

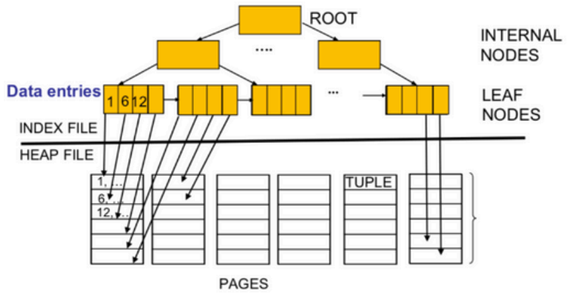
- **Data structure** built on top of data files
- Makes selection on search key fields faster
  - Any subset of the fields of a table is indexed based on queries that are frequently run against the database
  - Allow for **index-only scan**: scan w/o accessing data pages

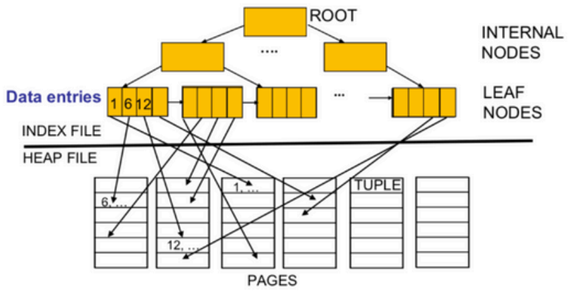
- Data entries that refer back to data in the relation:
  - Data entries: (k, *rid*)
    - k = search key
    - *rid* = record ID
- Stored in **index file**

Clustered / Unclustered Index

- Whether the **index & data** has the same **ordering**
- Useful for range search, but costly to update data
- Can have clustered index on <u>max 1 search key combination</u>
  - Impossible to have > 1 “order” at once







Primary / Secondary Index

- Whether the search key is the Primary Key of the relation

- Primary:
  - Includes the table’s primary key
  - NEVER contains duplicates

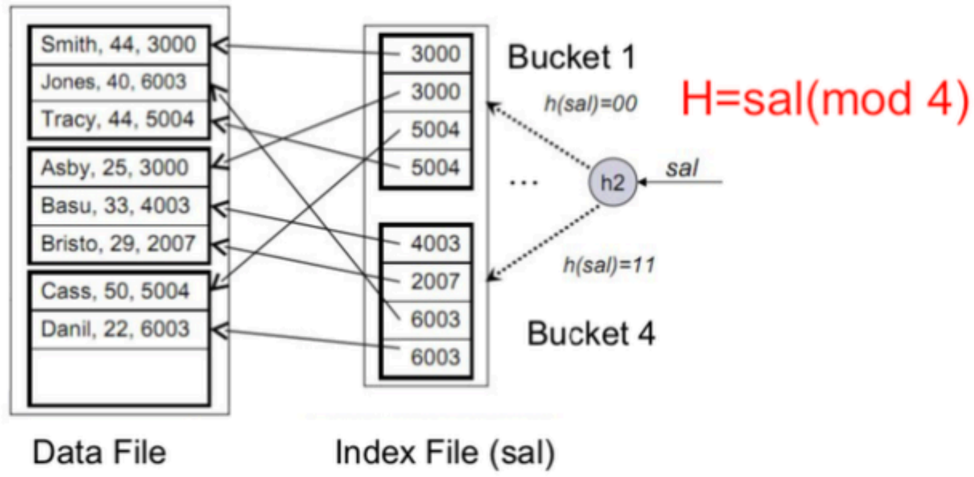
- Secondary:
  - Any other index
  - MAY contain duplicates

Hash-based Index

- Represents index as a collection of buckets
- Hash function maps search key → corresponding bucket
  - h(*r*.search_key) = bucket in which record *r* belongs

- Best for **equality selections** (doesn't help w/ range)

Find Sal = 2007
2007 mod 4 = 3 go to Buck.4



B-tree Index

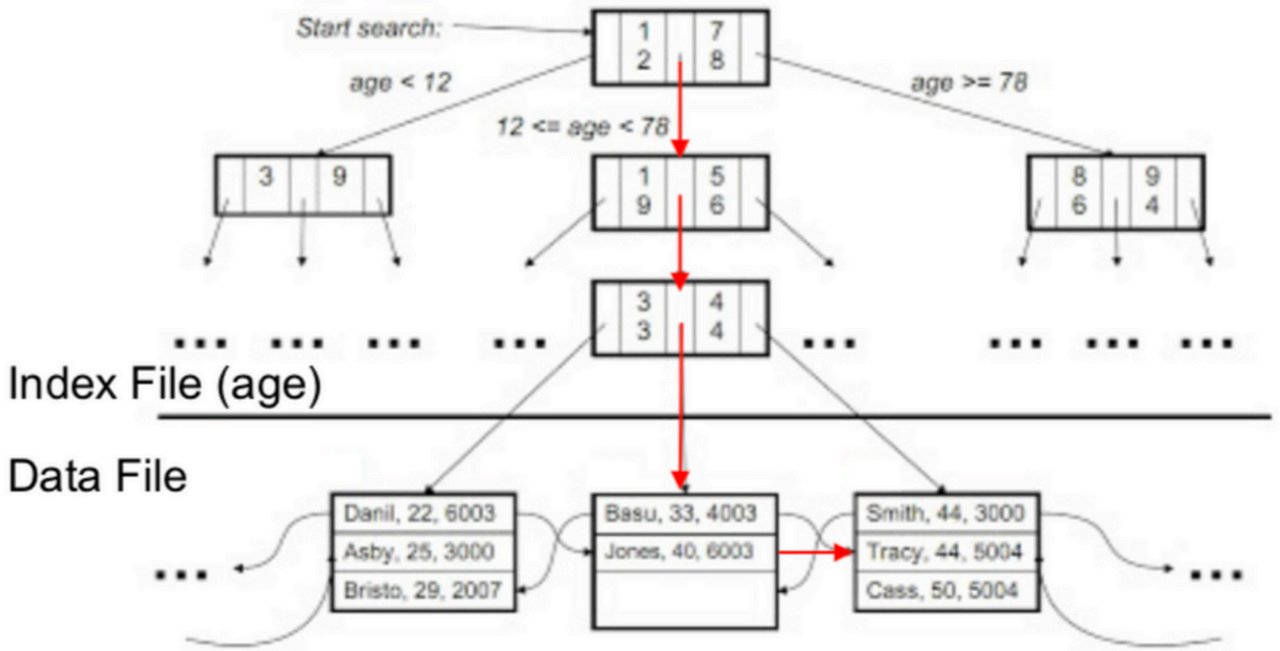
Created by:

1. Sorting data on the search key
2. Maintaining a hierarchical search data structure (B+ tree)

- Underlying data structure → Binary (B+) tree
  - Nodes contain pointers → lower levels (left lower, right higher)
  - Leaves contain data entries sorted by search key values
- Good for range selections

B-tree Index

- **Example**: Tree-based index on (age)



Find age > 39

Choosing an index

Before deciding which search key(s) should be used to build an index, frequent queries against the DB should be evaluated:

- Which relations are accessed frequently?
- Which attributes are retrieved?
- Which attributes are involved in selection, join and other conditions?
- If a query involves updating the relation, what attributes are affected

Practice Questions

Q1: Choosing Indexes

- In what situations would you use one over the other?
- **Primary vs Secondary**

Q1: Choosing Indexes

Primary	Secondary
• Used when records are retrieved based on **primary key values**	• Used when the fields (non-primary key attributes) are frequently used in the queries

- Generally, tables should always have a primary index
  - MySQL auto-creates one

Q1: Choosing Indexes

- In what situations would you use one over the other?

- **Clustered vs Unclustered**

Q1: Choosing Indexes

Clustered	Unclustered
• **Range-queries** • Choose the most frequently used combination	• **Equality query** • Especially when there is no duplicate on search key values

- NOTE: Clustered indexes are more expensive to maintain

Q1: Choosing Indexes

- In what situations would you use one over the other?
- **Hash vs Tree Index**

Q1: Choosing Indexes

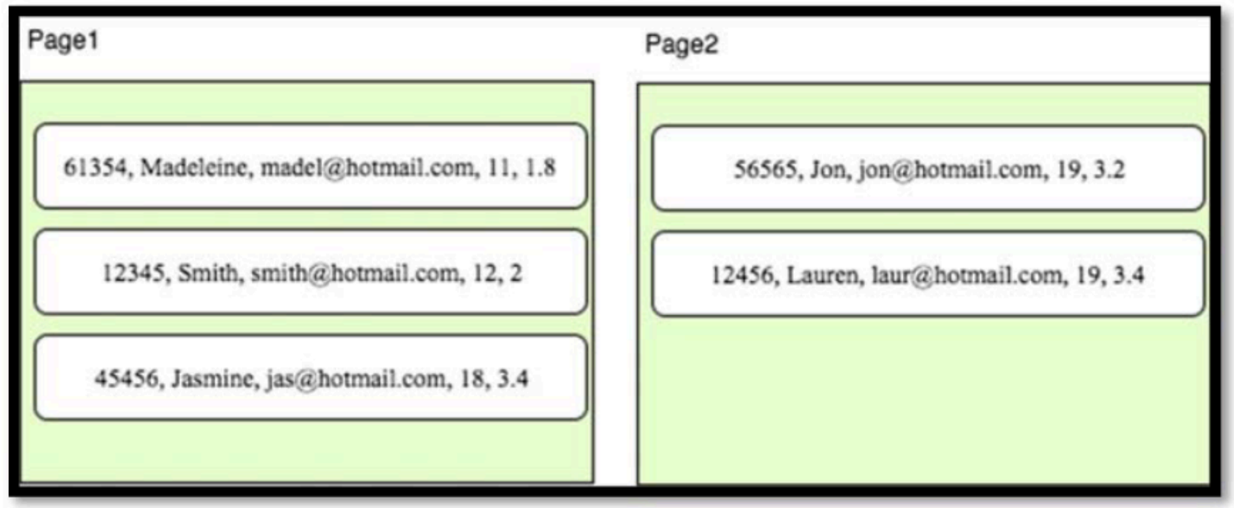
Tree	Hash
• **Range queries** • Tree nodes are sorted by search key values • Still alright for equality (compared to e.g. heap), but really good for range	• **Equality queries** • Records put into hash bins • Does NOT help range queries

Q2

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:



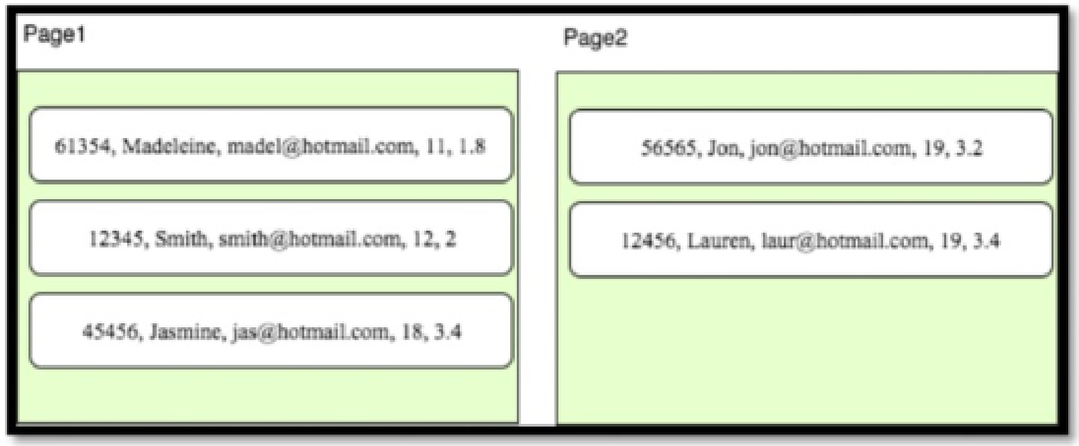
Show what the data entries of the index will look like for:

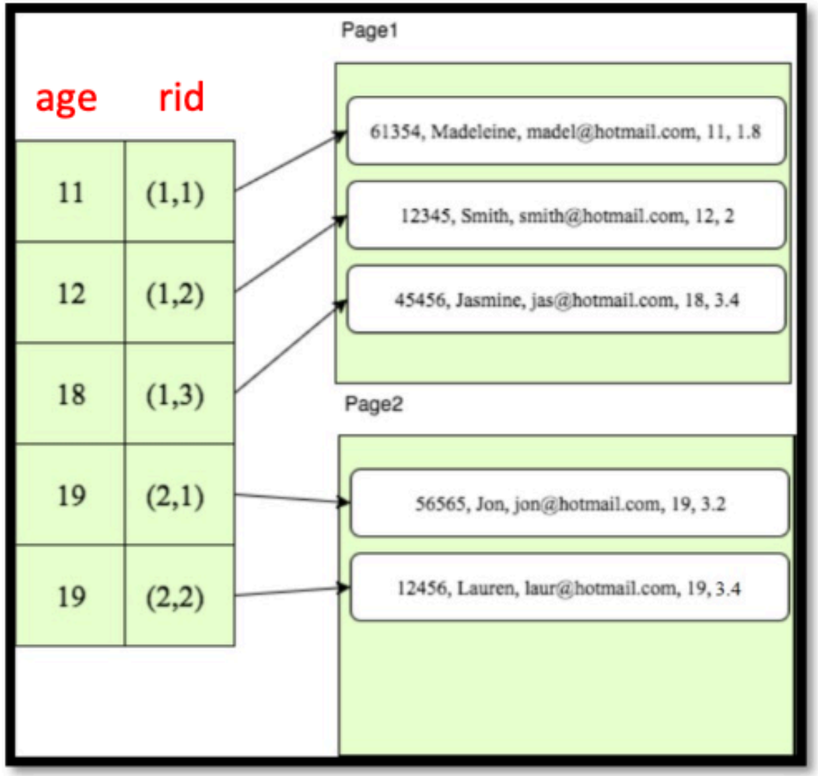
a. An index on Age

b. An index on GPA

Q2(a): Index on Age

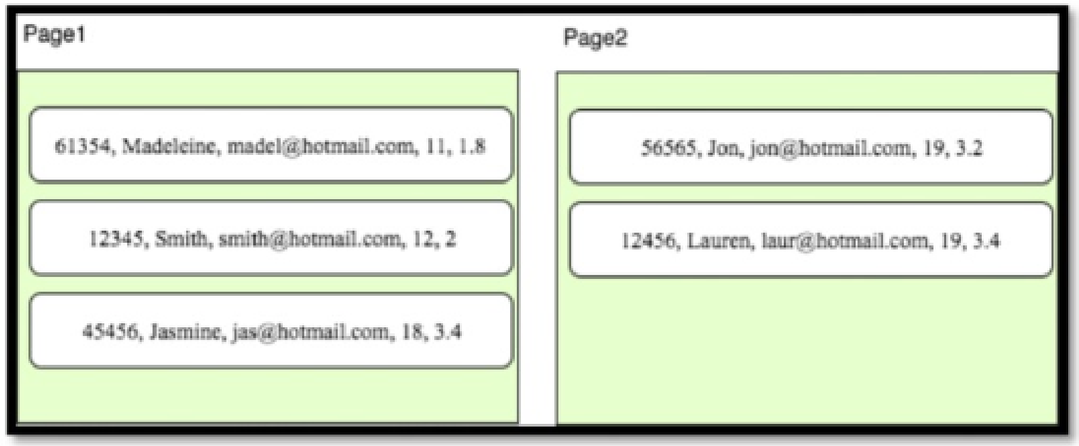
- Clustered index
- Data entry: (age, rid)

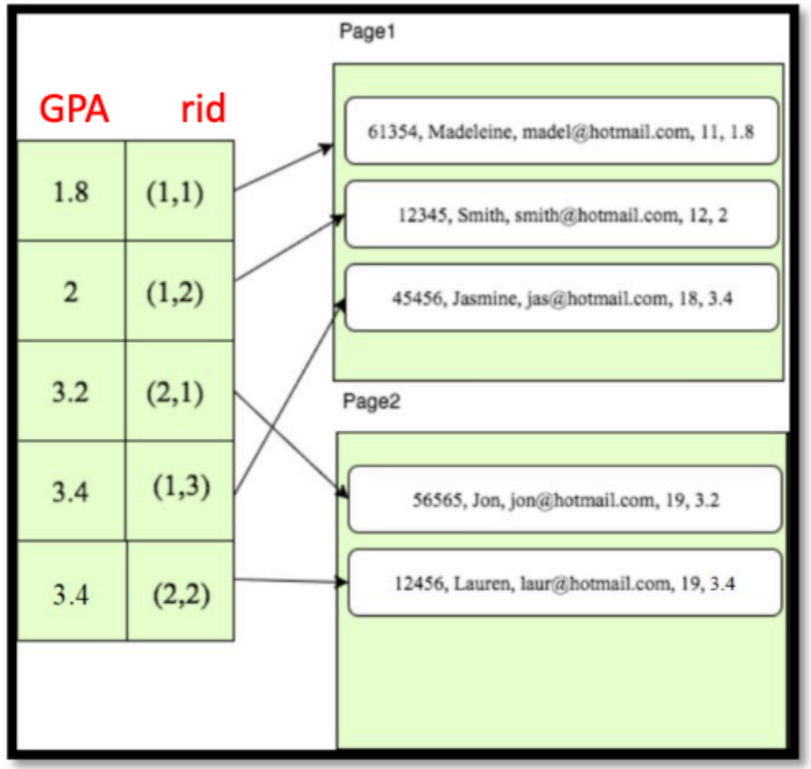




Q2(b): Index on GPA

- Unclustered index
- Data entry: (GPA, rid)





Q3

Employee (EmployeeID, EmployeeName, Salary, Age, DepartmentID) FK

Department (DepartmentID, DepartmentBudget, DepartmentFloor, ManagerID) FK

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?

Q3(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

Q3(a)

- Records will be ordered on the two fields that are used in WHERE
- The first record with DepartmentFloor = 10 will be accessed
- The following records will be read continuing from there in the order of budget

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

Q3(b)

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

Q3(b)

- Can get requested attributes with an index-only scan (and we can avoid accessing the table completely)

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