INFO20003 Tutorial – Week 6 Solutions

(Tutorial: Storage and Indexing)

Objectives:

This tutorial will cover:

I. Storage and Indexing review – key concepts with examples – 30 mins
II. Exercises – 30 mins

Key Concepts:

NOTE for students: *This is a brief summary of some of the concepts taught in lecture 10. The lectures contain detailed content related to these and many more concepts. These notes should be considered quick revision instead of a sole resource for the course material.*

Before we continue, it's important to know the terminology used for storage and indexing. Database management systems store information on disks (normally hard disks), which involves many READ and WRITE operations when data is accessed. The READ operation refers to the transfer of data from the disk to main memory (RAM) whereas WRITE will transfer data from RAM to the disk. Both operations are high cost but required for storing data on secondary storage.

The following table shows alternative terms used in this tutorial with respect to disk storage:

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

  A **record** refers to an individual row of a table and has a unique *rid*. The *rid* has the property that we can identify the disk address of the page containing the record by using the *rid*. The *rid* consists of the page ID and the offset within that page, for example, an *rid* of (3, 7) refers to the seventh record from the beginning of the third page.

  A **page** is an allocation of space on disk or in memory containing a collection of records. Typically, every page is the same size.

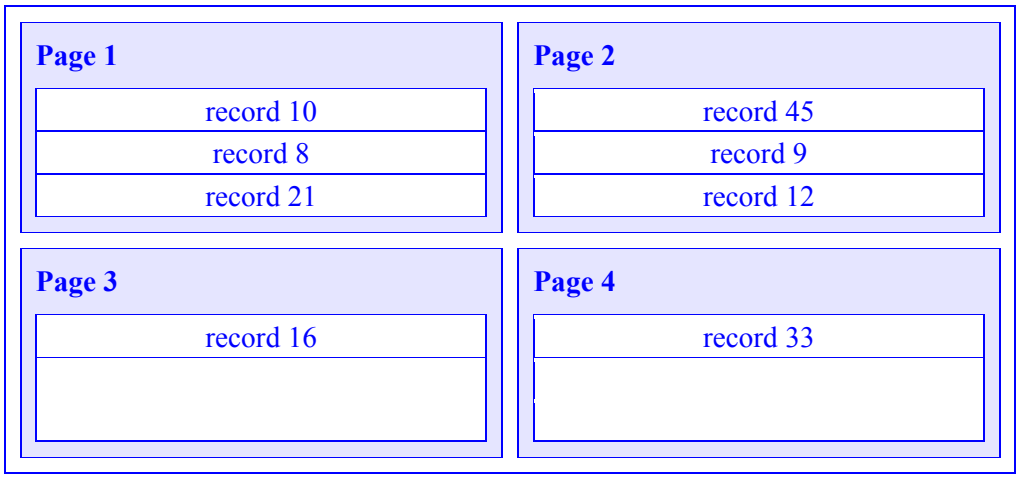
  A **file** consists of a collection of pages containing records. In simple database scenarios, a file corresponds to a single table.

- File organisations

  File organisation defines how file records are mapped onto pages (stored on disk). Typically, the file is organised in one of three ways:

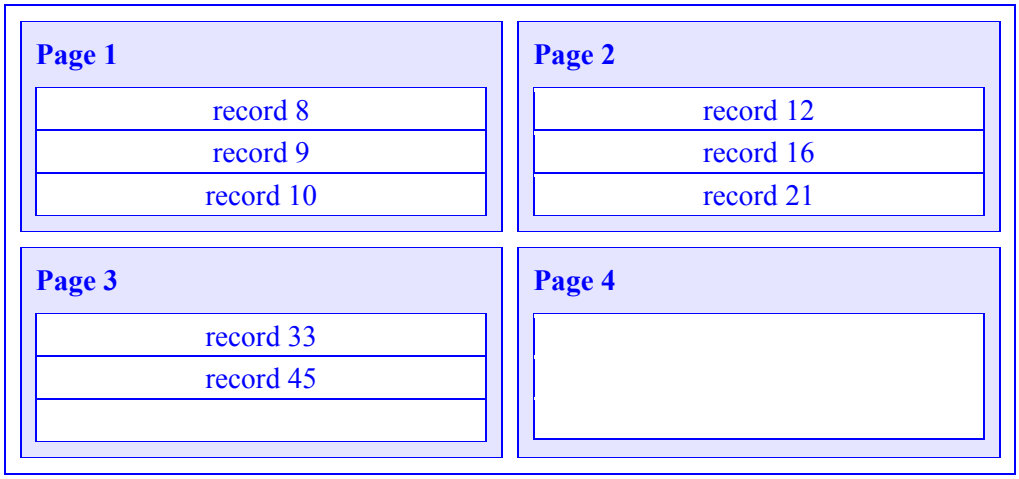
- Heap file organisation

  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 is the allocated memory area.



- 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* (note, this is a different use of the word "key" from the meaning we saw in the modelling topic).



- Index file organisation

  In an **index file organization**, a data structure on top of data files is built for efficient searching. Any subset of the fields of a table is indexed based on queries that are frequently run against the database.

- What is an index?

  An **index** on a relation speeds up selection on the search key fields. The *search key* is the subset of the attributes of a relation on which the index is built – it should not be confused with a relation's key as used in data modelling.

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*) where *k* is the search key and *rid* is the record ID in the relation.

The indexes are stored in an *index file*, in contrast to the *data file* which contains the actual records themselves.

Indexes can either be *clustered* (data records in the data file have the same order as data entries of the index) or *unclustered* (data records in the data file are not sorted by search key/data entries). Moreover, indexes can either be *primary* (on the primary key of the relation) or *secondary* (on any other set of attributes).

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

- 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?

Later on (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. It should be kept in mind that indexes in general make SELECT queries faster but slow down the updates. Indexes also require additional disk space. Therefore, these points should be carefully analysed before constructing an index.

There are several indexing techniques such as hash-based, tree-based, and so on.

- Hash-based indexing

In **hash-based indexing**, a hash function $h(r)$ is applied, where r is the field value. The output of the hash function will point to a bucket which refers to the primary page and other overflow pages if there is any. These buckets contain a representation (*k*, *rid*) for data entries.

Hash indexes are best suited to support *equality* selections (queries where the WHERE clause has an equality condition).

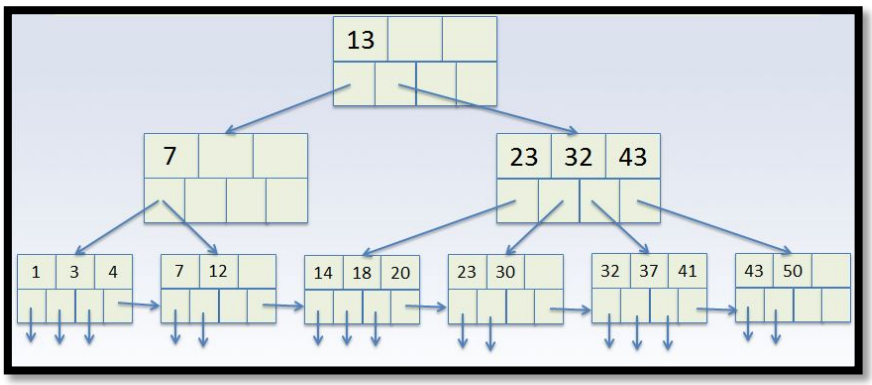
There are many ways to build a hash function – we will not discuss them in depth in this subject. Here is an example of a very simple hash function:

> Suppose you are given 5 buckets and $h(k) = k \% 5$ where % is the modulus (remainder) operator. Insert 200, 22, 119, 8, and 33 into a hash table.
>
> **Solution:**

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

- B-tree indexes

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



  To find a key *K* in a B-tree:

  a. Start at the root.
  b. For internal nodes, search the keys to find the range *K* belongs in and follow that pointer.
  c. 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.

  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.

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

  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. Generally, a table should always have a primary index (in fact, 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 cases where a query consists of a condition to check for a range, a clustered index is preferred as compared to unclustered. However, for equality conditions it does not

have any advantage over the unclustered index if the search key does not have duplicate values.

Clustered indexes are 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.

- Hash vs. tree indexes

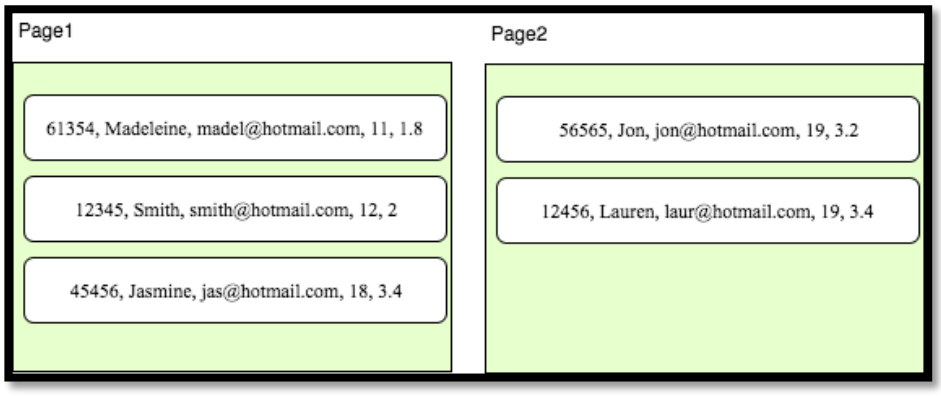
  Similar to clustered and unclustered indexes, if we want to perform many range queries, creating a B-tree index for the relation is the best choice. On the other hand, if there are equality queries such as driving licence numbers, then hash indexes are the best choice as they allow faster retrieval than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

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

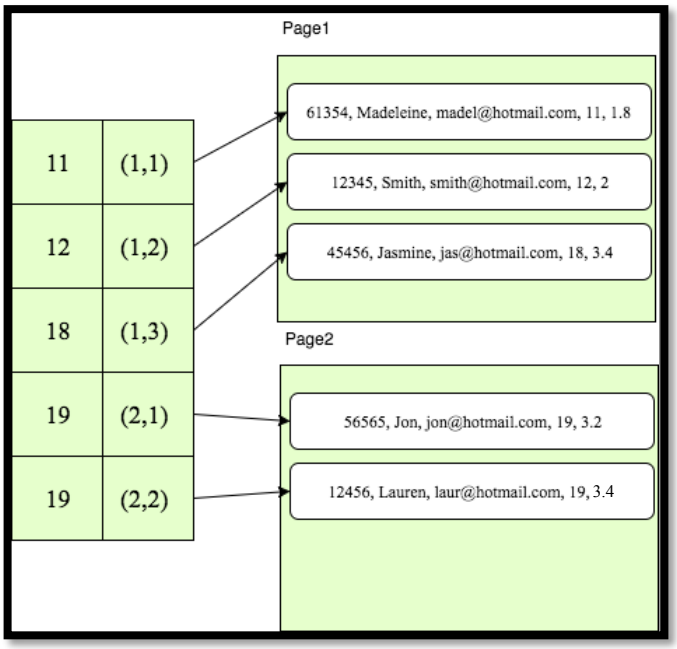
As you can see the tuples are sorted by age and we are assuming that the order of tuple is the same when stored on disk. The first record is on page 1 and each page can contain only 3 records. The arrangement of the records is shown below:



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

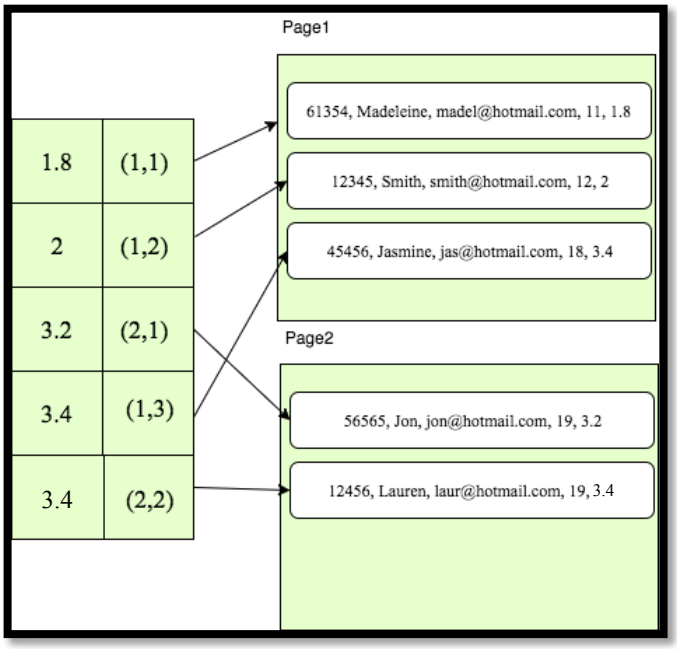
a. An index on Age

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.



b. An 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:



3. Consider the following relations:

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?

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

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.

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

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

An unclustered hash index on EmployeeName, Age and Salary fields will help in this case or a Clustered B+ tree index, 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!) Note that we consider that 'clustered hash indexes' are not feasible/used.