and access methods layer needs to process a page, it asks the buffer manager to fetch the page and put it into memory if it is not all ready in memory. When the files and access methods layer needs additional space to hold new records in a file, it asks the disk space manager to allocate an additional disk page.

Exercise 8.2 Answer the following questions about files and indexes:

1. What operations are supported by the file of records abstraction?
2. What is an index on a file of records? What is a search key for an index? Why do we need indexes?
3. What alternatives are available for the data entries in an index?
4. What is the difference between a primary index and a secondary index? What is a duplicate data entry in an index? Can a primary index contain duplicates?
5. What is the difference between a clustered index and an unclustered index? If an index contains data records as 'data entries,' can it be unclustered?
6. How many clustered indexes can you create on a file? Would you always create at least one clustered index for a file?
7. Consider Alternatives (1), (2) and (3) for 'data entries' in an index, as discussed in Section 8.2. Are all of them suitable for secondary indexes? Explain.

Answer 8.2 The answer to each question is given below.

1. The file of records abstraction supports file creation and deletion, record creation and deletion, and scans of of individual records in a file one at a time.
2. An index is a data structure that organizes data records on disk to optimize certain kinds of retrieval operations. A search key for an index is the fields stored in the index that we can search on to efficiently retrieve all records satisfy the search conditions. Without indexes, every search would to a DBMS would require a scan of all records and be extremely costly.
3. The three main alternatives for what to store as a data entry in an index are as follows:
   (a) A data entry k^* is an actual data record (with search key value k).
   (b) A data entry is a $\langle$ *k, rid* $\rangle$ pair, where *rid* is the record id of a data record with search key value k.
   (c) A data entry is a $\langle$ *k, rid-list* $\rangle$ pair, where *rid-list* is a list of record ids of data records with search key value k.

4. A primary index is an index on a set of fields that includes the unique primary key for the field and is guaranteed not to contain duplicates. A secondary index is an index that is not a primary index and may have duplicates. Two entries are said to be duplicates if they have the same value for the search key field associated with the index.

5. A clustered index is one in which the ordering of data entries is the same as the ordering of data records. We can have at most one clustered index on a data file. An unclustered index is an index that is not clustered. We can have several unclustered indexes on a data file. If the index contains data records as 'data entries', it means the index uses Alternative (1). By definition of clustered indexes, the index is clustered.

6. At most one, because we want to avoid replicating data records. Sometimes, we may not create any clustered indexes because no query requires a clustered index for adequate performance, and clustered indexes are more expensive to maintain than unclustered indexes.

7. No. An index using alternative (1) has actual data records as data entries. It must be a primary index and has no duplicates. It is not suitable for a secondary index because we do not want to replicate data records.

Exercise 8.3 Consider a relation stored as a randomly ordered file for which the only index is an unclustered index on a field called *sal*. If you want to retrieve all records with $sal > 20$, is using the index always the best alternative? Explain.

Answer 8.3 No. In this case, the index is unclustered, each qualifying data entry could contain an rid that points to a distinct data page, leading to as many data page I/Os as the number of data entries that match the range query. In this situation, using index is actually worse than file scan.

Exercise 8.4 Consider the instance of the Students relation shown in Figure 8.1, sorted by *age*: For the purposes of this question, assume that these tuples are stored in a sorted file in the order shown; the first tuple is on page 1 the second tuple is also on page 1; and so on. Each page can store up to three data records; so the fourth tuple is on page 2.

Explain what the data entries in each of the following indexes contain. If the order of entries is significant, say so and explain why. If such an index cannot be constructed, say so and explain why.

1. An unclustered index on *age* using Alternative (1).

2. An unclustered index on *age* using Alternative (2).

Emp(*eid:* `integer`, *sal:* `integer`, *age:* `real`, *did:* `integer`)

There is a clustered index on *eid* and an unclustered index on *age*.

1. How would you use the indexes to enforce the constraint that *eid* is a key?
2. Give an example of an update that is *definitely speeded up* because of the available indexes. (English description is sufficient.)
3. Give an example of an update that is *definitely slowed down* because of the indexes. (English description is sufficient.)
4. Can you give an example of an update that is neither speeded up nor slowed down by the indexes?

Answer 8.10 The answer to each question is given below.

1. To enforce the constraint that *eid* is a key, all we need to do is make the clustered index on *eid unique* and *dense*. That is, there is at least one data entry for each *eid* value that appears in an Emp record (because the index is dense). Further, there should be exactly one data entry for each such *eid* value (because the index is unique), and this can be enforced on inserts and updates.
2. If we want to change the salaries of employees whose *eid*'s are in a particular range, it would be sped up by the index on *eid*. Since we could access the records that we want much quicker and we wouldn't have to change any of the indexes.
3. If we were to add 1 to the ages of all employees then we would be slowed down, since we would have to update the index on age.
4. If we were to change the *sal* of those employees with a particular *did* then no advantage would result from the given indexes.

Exercise 8.11 Consider the following relations:

Emp(*eid:* `integer`, *ename:* `varchar`, *sal:* `integer`, *age:* `integer`, *did:* `integer`)
Dept(*did:* `integer`, *budget:* `integer`, *floor:* `integer`, *mgr_eid:* `integer`)

Salaries range from $10,000 to $100,000, ages vary from 20 to 80, each department has about five employees on average, there are 10 floors, and budgets vary from $10,000 to $1 million. You can assume uniform distributions of values.

For each of the following queries, which of the listed index choices would you choose to speed up the query? If your database system does not consider index-only plans (i.e., data records are always retrieved even if enough information is available in the index entry), how would your answer change? Explain briefly.

1. Query: *Print ename, age, and sal for all employees.*
   (a) Clustered hash index on $\langle ename, age, sal \rangle$ fields of Emp.
   (b) Unclustered hash index on $\langle ename, age, sal \rangle$ fields of Emp.
   (c) Clustered B+ tree index on $\langle ename, age, sal \rangle$ fields of Emp.
   (d) Unclustered hash index on $\langle eid, did \rangle$ fields of Emp.
   (e) No index.
2. Query: *Find the dids of departments that are on the 10th floor and have a budget of less than $15,000.*
   (a) Clustered hash index on the $floor$ field of Dept.
   (b) Unclustered hash index on the $floor$ field of Dept.
   (c) Clustered B+ tree index on $\langle floor, budget \rangle$ fields of Dept.
   (d) Clustered B+ tree index on the *budget* field of Dept.
   (e) No index.

Answer 8.11 The answer to each question is given below.

1. We should create an unclustered hash index on $\langle ename, age, sal \rangle$ fields of Emp (b) since then we could do an index only scan. If our system does not include index only plans then we shouldn't create an index for this query (e). Since this query requires us to access all the Emp records, an index won't help us any, and so should we access the records using a filescan.
2. We should create a clustered dense B+ tree index (c) on $\langle floor, budget \rangle$ fields of Dept, since the records would be ordered on these fields then. So when executing this query, the first record with $floor = 10$ must be retrieved, and then the other records with $floor = 10$ can be read in order of budget. Note that this plan, which is the best for this query, is not an index-only plan (must look up dids).

2. Change enrollment for a student identified by her *snum* from one class to another class.
3. Assign a new faculty member identified by his *fid* to the class with the least number of students.
4. For each class, show the number of students enrolled in the class.

Answer 16.7 The answer to each question is given below.

1. Because we are inserting a new row in the table *Enrolled*, we do not need any lock on the existing rows. So we would use READ UNCOMMITTED.
2. Because we are updating one existing row in the table *Enrolled*, we need an exclusive lock on the row which we are updating. So we would use READ COMMITTED.
3. To prevent other transactions from inserting or updating the table *Enrolled* while we are reading from it (known as the phantom problem), we would need to use SERIALIZABLE.
4. same as above.

Exercise 16.8 Consider the following schema:

Suppliers(*sid:* `integer`, *sname:* `string`, *address:* `string`)
Parts(*pid:* `integer`, *pname:* `string`, *color:* `string`)
Catalog(*sid:* `integer`, *pid:* `integer`, *cost:* `real`)

The Catalog relation lists the prices charged for parts by Suppliers.

For each of the following transactions, state the SQL isolation level that you would use and explain why you chose it.

1. A transaction that adds a new part to a supplier's catalog.
2. A transaction that increases the price that a supplier charges for a part.
3. A transaction that determines the total number of items for a given supplier.
4. A transaction that shows, for each part, the supplier that supplies the part at the lowest price.

Answer 16.8 The answer to each question is given below.

1. Because we are inserting a new row in the table *Catalog*, we do not need any lock on the existing rows. So we would use READ UNCOMMITTED.

2. Because we are updating one existing row in the table *Catalog*, we need an exclusive lock on the row which we are updating. So we would use READ COMMITTED.

3. To prevent other transactions from inserting or updating the table *Catalog* while we are reading from it (known as the phantom problem), we would need to use SERIALIZABLE.

4. same as above.

Exercise 16.9 Consider a database with the following schema:

Suppliers(*sid:* `integer`, *sname:* `string`, *address:* `string`)
Parts(*pid:* `integer`, *pname:* `string`, *color:* `string`)
Catalog(*sid:* `integer`, *pid:* `integer`, *cost:* `real`)

The Catalog relation lists the prices charged for parts by Suppliers.

Consider the transactions $T1$ and $T2$. $T1$ always has SQL isolation level `SERIALIZABLE`. We first run $T1$ concurrently with $T2$ and then we run $T1$ concurrently with $T2$ but we change the isolation level of $T2$ as specified below. Give a database instance and SQL statements for $T1$ and $T2$ such that result of running $T2$ with the first SQL isolation level is different from running $T2$ with the second SQL isolation level. Also specify the common schedule of $T1$ and $T2$ and explain why the results are different.

1. `SERIALIZABLE` versus `REPEATABLE READ`.

2. `REPEATABLE READ` versus `READ COMMITTED`.

3. `READ COMMITTED` versus `READ UNCOMMITTED`.

Answer 16.9 The answer to each question is given below.

1. Suppose a database instance of table Catalog and SQL statements shown below:

sid	pid	cost
18	45	$7.05
22	98	$89.35
31	52	$357.65
31	53	$26.22
58	15	$37.50
58	94	$26.22