8

OVERVIEW OF STORAGE AND INDEXING

Exercise 8.1 Answer the following questions about data on external storage in a DBMS:

1. Why does a DBMS store data on external storage?
2. Why are I/O costs important in a DBMS?
3. What is a record id? Given a record's id, how many I/Os are needed to fetch it into main memory?
4. What is the role of the buffer manager in a DBMS? What is the role of the disk space manager? How do these layers interact with the file and access methods layer?

Answer 8.1 The answer to each question is given below.

1. A DBMS stores data on external storage because the quantity of data is vast, and must persist across program executions.
2. I/O costs are of primary important to a DBMS because these costs typically dominate the time it takes to run most database operations. Optimizing the amount of I/O's for an operation can result in a substantial increase in speed in the time it takes to run that operation.
3. A record id, or rid for short, is a unique identifier for a particular record in a set of records. An rid has the property that we can identify the disk address of the page containing the record by using the rid. The number of I/O's required to read a record, given a rid, is therefore 1 I/O.
4. In a DBMS, the buffer manager reads data from persistent storage into memory as well as writes data from memory into persistent storage. The disk space manager manages the available physical storage space of data for the DBMS. When the file

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).

sid	*name*	*login*	*age*	*gpa*
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	19	3.2
53650	Smith	smith@math	19	3.8

Figure 8.1 An Instance of the Students Relation, Sorted by *age*

3. An unclustered index on *age* using Alternative (3).
4. A clustered index on *age* using Alternative (1).
5. A clustered index on *age* using Alternative (2).
6. A clustered index on *age* using Alternative (3).
7. An unclustered index on *gpa* using Alternative (1).
8. An unclustered index on *gpa* using Alternative (2).
9. An unclustered index on *gpa* using Alternative (3).
10. A clustered index on *gpa* using Alternative (1).
11. A clustered index on *gpa* using Alternative (2).
12. A clustered index on *gpa* using Alternative (3).

Answer 8.4 The answer to each question is given below. For Alternative (2), the notation $\langle\ A, (B,C)\ \rangle$ is used where A is the search key for the entry and (B,C) is the *rid* for data entry, with B being the page number of the entry and C being the location on page B of the entry. For Alternative (3), the notation is the same, but with the possibility of additional *rid*'s listed.

1. Contradiction. Cannot build unclustered index using Alternative (1) since method is inherently clustered.
2. $\langle\ 11, (1,1)\ \rangle$, $\langle\ 12, (1,2)\ \rangle$, $\langle\ 18, (1,3)\ \rangle$, $\langle\ 19, (2,1)\ \rangle$, $\langle\ 19, (2,2)\ \rangle$. The order of entries is not significant.
3. $\langle\ 11, (1,1)\ \rangle$, $\langle\ 12, (1,2)\ \rangle$, $\langle\ 18, (1,3)\ \rangle$, $\langle\ 19, (2,1), (2,2)\ \rangle$, The order of entries is not significant.
4. 11, 19. The order of entries is significant since the order of the entries is the same as the order of data record.

5. ⟨ *11*, (*1,1*) ⟩, ⟨ *19*, (*2,1*) ⟩. The order of entries is significant since the order of the entries is the same as the order of data record.

6. ⟨ *11*, (*1,1*) ⟩, ⟨ *19*, (*2,1*), (*2,2*) ⟩. The order of entries is significant since the order of the entries is the same as the order of data record.

7. Contradiction. Cannot build unclustered index using Alternative (1) since method is inherently clustered.

8. ⟨ *1.8*, (*1,1*) ⟩, ⟨ *2.0*, (*1,2*) ⟩, ⟨ *3.2*, (*2,1*) ⟩, ⟨ *3.4*, (*1,3*) ⟩, ⟨ *3.8*, (*2,2*) ⟩. The order of entries is not significant.

9. ⟨ *1.8*, (*1,1*) ⟩, ⟨ *2.0*, (*1,2*) ⟩, ⟨ *3.2*, (*2,1*) ⟩, ⟨ *3.4*, (*1,3*) ⟩, ⟨ *3.8*, (*2,2*) ⟩. The order of entries is not significant.

10. Alternative (1) cannot be used to build a clustered index on *gpa* because the records in the file are not sorted in order of *gpa*. Only if the entries in (*1,3*) and (*2,1*) were switched would this possible, but then the data would no longer be sorted on *age* as previously defined.

11. Alternative (2) cannot be used to build a clustered index on *gpa* because the records in the file are not sorted in order of *gpa*. Only if the entries in (*1,3*) and (*2,1*) were switched would this possible, but then the data would no longer be sorted on *age* as previously defined.

12. Alternative (3) cannot be used to build a clustered index on *gpa* because the records in the file are not sorted in order of *gpa*. Only if the entries in (*1,3*) and (*2,1*) were switched would this possible, but then the data would no longer be sorted on *age* previously defined.

Exercise 8.5 Explain the difference between Hash indexes and B+-tree indexes. In particular, discuss how equality and range searches work, using an example.

Answer 8.5 A Hash index is constructed by using a hashing function that quickly maps an search key value to a specific location in an array-like list of elements called buckets. The buckets are often constructed such that there are more bucket locations than there are possible search key values, and the hashing function is chosen so that it is not often that two search key values hash to the same bucket. A B+-tree index is constructed by sorting the data on the search key and maintaining a hierarchical search data structure that directs searches to the correct page of data entries.

Insertions and deletions in a hash based index are relatively simple. If two search values hash to the same bucket, called a collision, a linked list is formed connecting multiple records in a single bucket. In the case that too many of these collisions occur, the number of buckets is increased. Alternatively, maintaining a B+-tree's hierarchical search data structure is considered more costly since it must be updated whenever there

are insertions and deletions in the data set. In general, most insertions and deletions will not modify the data structure severely, but every once in awhile large portions of the tree may need to be rewritten when they become over-filled or under-filled with data entries.

Hash indexes are especially good at equality searches because they allow a record look up very quickly with an average cost of 1.2 I/Os. B+-tree indexes, on the other hand, have a cost of 3-4 I/Os per individual record lookup. Assume we have the employee relation with primary key *eid* and 10,000 records total. Looking up all the records individually would cost 12,000 I/Os for Hash indexes, but 30,000-40,000 I/Os for B+-tree indexes.

For range queries, hash indexes perform terribly since they could conceivably read as many pages as there are records since the data is not sorted in any clear grouping or set. On the other hand, B+-tree indexes have a cost of 3-4 I/Os plus the number of qualifying pages or tuples, for clustered or unclustered B+-trees respectively. Assume we have the employees example again with 10,000 records and 10 records per page. Also assume that there is an index on *sal* and query of *age* ¿ 20,000, such that there are 5,000 qualifying tuples. The hash index could cost as much as 100,000 I/Os since every page could be read for every record. It is not clear with a hash index how we even go about searching for every possible number greater than 20,000 since decimals could be used. An unclustered B+-tree index would have a cost of 5,004 I/Os, while a clustered B+-tree index would have a cost of 504 I/Os. It helps to have the index clustered whenever possible.

Exercise 8.6 Fill in the I/O costs in Figure 8.2.

File Type	*Scan*	*Equality Search*	*Range Search*	*Insert*	*Delete*
Heap file					
Sorted file					
Clustered file					
Unclustered tree index					
Unclustered hash index					

Figure 8.2 I/O Cost Comparison

Answer 8.6 The answer to the question is given in Figure 8.3. We use B to denote the number of data pages total, R to denote the number of records per page, and D to denote the average time to read or write a page.

File Type	*Scan*	*Equality Search*	*Range Search*	*Insert*	*Delete*
Heap file	*BD*	*0.5BD*	*BD*	*2D*	*Search+* D
Sorted file	*BD*	$Dlog_2B$	$Dlog_2B$+# *matching pages*	*Search+* BD	*Search+* BD
Clustered file	*1.5BD*	$Dlog_F 1.5B$	$Dlog_F B$+# *matching pages*	*Search+ D*	*Search+ D*
Unclustered tree index	*BD(R+ 0.15)*	$D(1+ log_F 0.15B)$	$D(log_F 0.15B$+# *matching records)*	$D(3+ log_F 0.15B)$	*Search+ 2D*
Unclustered hash index	*BD(R+ 0.125)*	*2D*	*BD*	*4D*	*Search+ 2D*

Figure 8.3 I/O Cost Comparison

Exercise 8.7 If you were about to create an index on a relation, what considerations would guide your choice? Discuss:

1. The choice of primary index.
2. Clustered versus unclustered indexes.
3. Hash versus tree indexes.
4. The use of a sorted file rather than a tree-based index.
5. Choice of search key for the index. What is a composite search key, and what considerations are made in choosing composite search keys? What are index-only plans, and what is the influence of potential index-only evaluation plans on the choice of search key for an index?

Answer 8.7 The answer to each question is given below.

1. The choice of the primary key is made based on the semantics of the data. If we need to retrieve records based on the value of the primary key, as is likely, we should build an index using this as the search key. If we need to retrieve records based on the values of fields that do not constitute the primary key, we build (by definition) a secondary index using (the combination of) these fields as the search key.
2. A clustered index offers much better range query performance, but essentially the same equality search performance (modulo duplicates) as an unclustered index.

Further, a clustered index is typically more expensive to maintain than an unclustered index. Therefore, we should make an index be clustered only if range queries are important on its search key. At most one of the indexes on a relation can be clustered, and if range queries are anticipated on more than one combination of fields, we have to choose the combination that is most important and make that be the search key of the clustered index.

3. If it is likely that ranged queries are going to be performed often, then we should use a B+-tree on the index for the relation since hash indexes cannot perform range queries. If it is more likely that we are only going to perform equality queries, for example the case of social security numbers, than hash indexes are the best choice since they allow for the faster retrieval than B+-trees by 2-3 I/Os per request.

4. First of all, both sorted files and tree-based indexes offer fast searches. Insertions and deletions, though, are much faster for tree-based indexes than sorted files. On the other hand scans and range searches with many matches are much faster for sorted files than tree-based indexes. Therefore, if we have read-only data that is not going to be modified often, it is better to go with a sorted file, whereas if we have data that we intend to modify often, then we should go with a tree-based index.

5. A composite search key is a key that contains several fields. A composite search key can support a broader range as well as increase the possibility for an index-only plan, but are more costly to maintain and store. An index-only plan is query evaluation plan where we only need to access the indexes for the data records, and not the data records themselves, in order to answer the query. Obviously, index-only plans are much faster than regular plans since it does not require reading of the data records. If it is likely that we are going to performing certain operations repeatly that only require accessing one field, for example the average value of a field, it would be an advantage to create a search key on this field since we could then accomplish it with an index-only plan.

Exercise 8.8 Consider a delete specified using an equality condition. For each of the five file organizations, what is the cost if no record qualifies? What is the cost if the condition is not on a key?

Answer 8.8 If the search key is not a candidate key, there may be several qualifying records. In a heap file, this means we have to search the entire file to be sure that we've found all qualifying records; the cost is $B(D + RC)$. In a sorted file, we find the first record (cost is that of equality search; $Dlog_2B + Clog_2R$) and then retrieve and delete successive records until the key value changes. The cost of the deletions is C per deleted record, and D per page containing such a record. In a hashed file, we hash to find the appropriate bucket (cost H), then retrieve the page (cost D; let's assume

no overflow pages), then write the page back if we find a qualifying record and delete it (cost D).

If no record qualifies, in a heap file, we have to search the entire file. So the cost is $B(D + RC)$. In a sorted file, even if no record qualifies, we have to do equality search to verify that no qualifying record exists. So the cost is the same as equality search, $Dlog_2B + Clog_2R$. In a hashed file, if no record qualifies, assuming no overflow page, we compute the hash value to find the bucket that would contain such a record (cost is H), bring that page in (cost is D), and search the entire page to verify that the record is not there (cost is RC). So the total cost is $H + D + RC$.

In all three file organizations, if the condition is not on the search key we have to search the entire file. There is an additional cost of C for each record that is deleted, and an additional D for each page containing such a record.

Exercise 8.9 What main conclusions can you draw from the discussion of the five basic file organizations discussed in Section 8.4? Which of the five organizations would you choose for a file where the most frequent operations are as follows?

1. Search for records based on a range of field values.
2. Perform inserts and scans, where the order of records does not matter.
3. Search for a record based on a particular field value.

Answer 8.9 The main conclusion about the five file organizations is that all five have their own advantages and disadvantages. No one file organization is uniformly superior in all situations. The choice of appropriate structures for a given data set can have a significant impact upon performance. An unordered file is best if only full file scans are desired. A hash indexed file is best if the most common operation is an equality selection. A sorted file is best if range selections are desired and the data is static; a clustered B+ tree is best if range selections are important and the data is dynamic. An unclustered B+ tree index is useful for selections over small ranges, especially if we need to cluster on another search key to support some common query.

1. Using these fields as the search key, we would choose a sorted file organization or a clustered B+ tree depending on whether the data is static or not.
2. Heap file would be the best fit in this situation.
3. Using this particular field as the search key, choosing a hash indexed file would be the best.

Exercise 8.10 Consider the following relation:

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).