**COMP 418 TME 1**

**Jason Bell 3078931**

**March 17, 2016**

**Part 1**

1. **Briefly explain the three main alternatives for storing information in a data entry of an index.**

Alternative 1: the data entry in the index is the actual data. That is, the index does not point to data in a separate file: the data is stored in the index itself.

Alternative 2: the data entry in the index is a key/value pair (the data records are in a separate file). The key is the search value for the entry, and the value points to the location of the data in the other file.

Alternative 3: similar to alternative 2, except that the key/value pair points to a list of indexes matching the search key, rather than just one index.

1. **Define clustered index, and discuss the relation between the three alternatives and clustered/unclustered indexes.**

A clustered index stores its indexes in the same order as the data records that they point to. The result is that if one knows the position of the entry in the index, one can jump to the corresponding record in the data file.

Alternative 1: This is a clustered index, as the data exists in the same order as the indexes, by virtue of each index holding its own data. For the same reason, alternative 1 can not be unclustered.

Alternative 2: In theory an alternative 2 index could be clustered, but this would require sorting the data records to match their orders in the index, each time a data entry is added or removed. Due to the extra cost of maintaining a sorted data file, alternative 2 records are typically unclustered.

Alternative 3: Alternative 3 indexes are typically unclustered, for the same reason given for alternative 2.

1. **Consider the following file organizations: sorted files, heap files with an unclustered tree index on the search key, and heap files with an unclustered hash index. Briefly discuss the suitability of each of these file organizations to perform the following operations: file scans, range selections, inserts, and deletes.**

File scan:

Sorted files: no better than unsorted files, as in both cases the entire set of records in the file must be read. However, because we don’t care about order of scanning (we just want to scan the whole file), when scanning an unindexed file, whether sorted or unsorted, the records can be loaded in batches.

Heap file with an unclustered tree index: because the index is not clustered, scanning the file through the index is very expensive. As the indexes are iterated, it’s corresponding data record(s) must be read from a different location. Thus, batched loading of records is not possible.

Heap file with an unclustered hash index: as with the unclustered tree index, scanning is very expensive. For each index, it’s corresponding data records(s) must be loaded from a different location, and batching is impossible.

Clearly, in the cases given, it is best to not use the index to scan: it is preferable to do a file scan of the data file, whether it is sorted or unsorted.

Range selections:

Sorted files: if the record sizes are fixed, sorted files are well suited to range selections, as they allow use of a search algorithm such as binary search to find the start and end of the scan range. Once these are known, scanning all records between the start and end will result in all values within that range. Otherwise, one must scan from the start of the file until the first match is found, then keep scanning and read data until the records no longer match.

Heap files with unclustered tree index: poorly suited to file scans. While finding the start and end indexes corresponding to the scan range is a simple matter of navigating through a few layers of the tree to the appropriate leaf nodes, the data being unclustered results in many file reads to get the data corresponding to each index.

Heap files with unclustered hash index: completely unsuitable to file scans. The indexes do not exist in the order of their search key, instead they are in the order of the value that the search key hashes to. Therefore, a hash index is of no use: although we can find the start and end indexes, it is unlikely that the desired selection exists at the indexes between them. Thus, the data file must be scanned directly.

Sorted files provide the most efficient method of performing range selections, at least for large selections. For the unclustered indexes, in the case of large selections it will likely be faster to simply perform a file scan of the data file, sort the records, and return the records that fall in the range.

Inserts:

Sorted files: very unsuited to insertion. When a new record is inserted, one of two things must happen: we either find the appropriate location and all records at or below the insertion point must be read and rewritten below the new record, or the new record is appended and the file resorted. Both are costly procedures.

Heap files with unclustered tree index: the cost is appending the record to the data file, finding the index for the new data record, and adding an entry to the index corresponding to the new data record. Clearly this is cheaper than inserting into a sorted file.

Heap files with unclustered hash index: as with unclustered tree index.

Both forms of indexes are superior to sorted files for inserts.

Deletes:

Sorted files: very unsuited to deletion: we must find where the record is located, delete the record, and move up all following records to eliminate the free space.

Heap files with unclustered tree index: the cost is searching for the index, then deleting the index and corresponding data record.

Heap files with unclustered hash index: as with unclustered tree index.

Both forms of indexes are superior to sorted files for deletes.

1. **Briefly describe the two internal organizations for heap files (using lists versus directory of pages).**

A hash index that uses lists of pages in the bucket to which, given a search key, a hash function points to, is simple to implement but becomes inefficient as the lists get longer. The search key may point to the bucket, but searching through the bucket’s list of pages may take some time if the list is long. This is the case with static hashing methods, where the number of buckets is fixed when the index is created: the lists of entries in each bucket will continuously get longer and degrade performance. A static hash can be rebuilt with a new hash function to accommodate the larger set of data, but this is an expensive procedure.

Hash indexes that use directories solve this problem by providing a dynamic solution (similar to ISAM vs B+ trees). The hash function does not point to a bucket of data records, but to a directory pointing to the location of the bucket containing the data record. Because the directory is much smaller than the contents of all the data buckets, reformatting the directory is much cheaper than rehashing the entire index. When a full bucket is to be written to, it is split and the two new buckets are identified by a new bit in the hash value: for example, if the original bucket was pointed to by 01, the new ones may be 001 and 101. This of course is only the case if the current directory can accommodate new buckets: this will not be the case if the directory only uses two bits to identify buckets. If there is no room in the directory for new buckets, the directory is doubled: it now identifies buckets by using three bits from the hash value. This is a dynamic process that is much more efficient than recreating the entire index. (See my answer to question 6 for more details).

1. **Explain which organization you would choose if records are variable in length.**

Using variable length records with a list of pages would be slower then with a directory: one would have to locate a bucket with the space to accommodate it, which could take some time if there are many overflow pages. Conversely, if the records are fixed length, one will know how far to jump from record to record while scanning for space or searching for a record, which will be a faster search for an empty space. I understand that this is not an issue if the lists are linked for easy iteration, in which case handling variable length records will be no different than fixed length records (ie, I’m operating under the assumption that this question applies to non-linked lists).

A directory handles variable length records well because once pointed to a bucket, one only has to inspect it’s one page to determine whether there is space for the record, rather than a potentially large number of overflow pages as well. If the page has room, the record can be added. If it does not, the bucket can be split and the new target bucket will have room.

1. **Compare ISAM and B+ Tree indexes. Explain briefly their differences in handling Search, Insert and Delete, and discuss when you would use ISAM and when you would use B+ Tree index.**

ISAM index trees are created ahead of time and are static: new entries are added to the appropriate pre-defined index leaf node or to attached overflow pages. Creating a tree structure ahead of time allows optimizations such as tailoring its indexes to the type of data to be stored, but can have degraded performance as overflow pages get longer: these overflow pages are themselves essentially unindexed heap files that are costly to search and manipulate.

B+ index trees are dynamic: their structure is adjusted to accommodate inserts and deletes. This ensures that unlike ISAM trees, B+ trees do not have overflow pages. Therefore, although one can not do the same ahead-of-time domain specific optimizations that one can do with an ISAM tree, B+ trees do not have the performance degrading overflow pages. The absence of overflow pages also makes it simpler to link adjacent leaf nodes as a linked list for easier iteration through the indexes.

Searches:

ISAM index trees will be slower for searches, especially if there are many overflow pages: these overflow pages must be searched like heap files.

Insert:

ISAM index trees will be faster for inserts, as no restructuring of the tree is required: the new entry is simply appended to an existing leaf node or overflow page. An insertion into a B+ tree may result in expensive restructuring if the leaf node is full.

Delete:

ISAM index trees will be slower than B+ trees because of the search that is required to find the entry before deleting it.

ISAM trees may be preferable when the data is known ahead of time and is unlikely to change: the ISAM tree can be tailored to ensure that it will have minimal overflow pages. With this being the case, the fact that ISAM allows concurrent access to non-leaf nodes during searches sets it ahead of B+ trees. The non-leaf nodes are static, so transactions do not need to gain exclusive access to them, and no queues of transactions appear. Therefore, ISAM trees provide faster concurrent access to data stored in an ISAM tree tailored to that data.

In all other cases, B+ trees are preferable due to their flexibility in the form of adjusting dynamically to inserts and deletes: these do not result in overflow pages that degrade the performance of searches.

1. **Does the final structure of a B+ tree depend on the order in which the terms are added to it? Explain your answer using an illustration example.**

Order of insertion does matter when a number of insertions are done. The different order results in different nodes being split in different ways and order. For example, given one order of insertion, node y may get split at the 5th insertion, while in another order that same node may be split at the 7th insertion, or even not at all.

Follows is an example using the values 3\*, 2\*, 4\*, 5\*, 1\* inserted into a B+ tree of order d = 2:

3

3

3

3

2\*

2\*

3\*

4\*

2\*

4\*

5\*

3\*

3\*

3\*

3

1\*

2\*

5\*

4\*

3\*

Now consider a B+ tree of order d = 2 inserting the values of 2\*, 5\*, 3 \*, 4\*, 1\*:

2

2

2

2

3\*

5\*

2\*

5\*

2\*

2\*

3\*

5\*

4\*

1\*

2

1\*

5\*

3\*

4\*

2\*

Clearly, adding the same entries in a different order does result in a different B+ tree.

1. **Explain how extendible hashing uses a directory of buckets, and discuss the global depth of the index and local depth of a bucket.**

Extensible hashing uses a directory of buckets because dynamic modification of a directory is cheaper than rehashing the entire index. Rather than pointing to a bucket, a hash value points to an entry in the directory, which in turn points to the bucket (so the directory is essentially an index within an index: a simpler structure that is easy to modify while facilitating locating entries).

Depth in this context is the number of bits that are inspected in the hash value in order to identify the bucket through the directory. The global depth is the number of bits that the directory may potentially need to inspect in order to identify the bucket, and is equal to the highest local depth found among the buckets.

Suppose, for the sake of explanation, that a newly created hash index has a global depth of 1, which will allow it to have two buckets. These buckets are identified by the Boolean values 0 and 1, and also have a local depth of 1, as only one bit is necessary to identify them. If something needs to be added to bucket 0, but the bucket is full, the bucket must be split. This is done by increasing the global depth of the index and the local depth of bucket 0 (bucket 1’s local depth does not change, as it is not split).

Now the capacity of the directory has doubled to 4, with the entries identified as 00, 01, 10, and 11. 00 and 10 point to the split buckets that were formerly bucket 0, and are now buckets 00 and 10. Entries 01 and 11 both point to the unsplit bucket 1.

If at this point bucket 1 need so be split, it can be split without first increasing the global depth and doubling the directory, as there are already two entries pointing to it. It is just split into buckets 01 and 11, and the corresponding directory entries are updated to point to the new buckets.

As can be seen, when a bucket is to be split, it may or may not result in doubling of the directory. It will require doubling the directory if the buckets local depth is the same as the global depth, while it will not require doubling the directory if its local depth is less than the global depth.

Finally, to briefly take it a step further, if bucket 00 needs to be split, the global depth increases to 3, allowing eight entries in the directory. Bucket 00 is split into buckets 000 and 100 with a local depth of 3. The other buckets remain local depth 2, and each is pointed to by two global depth 3 entries in the directory.

**Part 2**

1. **Consider the following relations:**

**Professor (profid: integer, name: varchar, salary: integer, age: integer, depid: integer)  
Department (did: integer, budget: integer, location: varchar, mgr eid: integer)**

**Salaries range from $30,000 to $100,000, ages vary from 20 to 80, each department has about 20 employees on average, there are 10 locations, and budgets vary from $100,000 to $1 million. You can assume uniform distributions of values.**

**For each of the following queries, what index 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. **Query1: Print name, age, and salary for all professors.**

This query would benefit from an index scan of a hash index with the composite search key (name, age, salary): the cost to look up entries in a hash table is insignificant, and this index would return all needed data without reading the data file. Clustering is unnecessary: since we are not accessing the data file, it does not matter if it’s contents are ordered the same as in the index.

A B+ tree would be almost as effective for carrying out the query quickly (or close enough as to be unnoticeable), but would have more overhead when adding or deleting indexes. This is a separate consideration from simply increasing the speed of the query, but is still worth considering.

If the DBMS does not use index-only plans, a file scan will be required since all values are to be returned.

1. **Query2: Find the dids of departments that are located in Edmonton and have a budget of more than $150,000.**

This query would benefit from a B+ tree index with the composite search key (location, budget). This index is first sorted by location, then by budget. The query would require searching for the first index entry with location Edmonton and budget $150,000, then retrieving data records for subsequent index entries until the location is no longer Edmonton. This is not an index only scan, nor can it be one: “did” is being retrieved, rather than being a criterion for the search.

My answer would not change if index only scans were disallowed, as this is already not an index-only scan.

1. **The CVT Company is a leader in the manufacture of work clothes. You are hired as database administrator for the company and your IT supervisor asked you to solve a retrieval speed problem they used to have with a large file for item records. Your supervisor mentioned that they have sorted the file but the problem didn’t improve, so they need to create a B+ tree index to solve the problem. Your supervisor outlined the way to do it: “The best way to accomplish this task is to scan the file, record by record, inserting each one using the B+ tree insertion procedure.” Being a fresh graduate, you noticed that since the file is already sorted there is a better way to do it.**
2. **What performance and storage utilization problems are there with your supervisor’s approach?**

Feeding a sequence of sorted entries individually results in an increasingly deep tree: this is an issue I encountered when drawing B+ trees for question 5 in part 1. Given the values 1\*, 2\*, 3\*, 4\*, and 5\* you end up with a tree with d = 1 that looks like:

2

1

4

3

2\*

1\*

5

4\*

3\*

5\*

The result is a chain of nodes each with a single leaf node containing a single value, regardless of the capacity of the leaf nodes (I drew this with d = 1 in mind, so a capacity of 2, but this is irrelevant as each leaf node only has one entry). Clearly this structure will not provide any benefit to queries: one must still search through a long chain of nodes, which is no better than simply scanning the data file. This could be overcome by then performing redistributions, but that does not appear to be done here. In any case, the fact that the data is already sorted begs for it to be bulk loaded.

1. **Explain how the bulk-loading algorithm provides a better alternative than the proposed scheme.**

Bulk loading requires that the data be sorted before bulk loading, so it is ideal for this situation. The result is a shallow, balanced B+ tree that will be much faster to search. In fact, the sequence 1\*, 2\*, 3\*, 4\* and 5 \* results in the following B+ tree with d = 1 after bulk loading:

2\*

3

5

1\*

5\*

3\*

4\*

This tree is shallow with evenly distributed entries, so will be much faster to search.

1. **Your team in charge of database administration was discussing different alternatives for indexing your organization’s databases. Some tables in one database have very few insertions but they are used intensively by different services to check for information about items using the item\_ID number. While many of your colleagues proposed using a tree index, you argued for a Hash index for these tables because it provides an average-case search cost of only slightly more than one disk I/O. The team leader agrees to adopt your solution but has asked you to write a short explanation for two questions:**
2. **How does Linear Hashing provide an average-case search cost of only slightly more than one disk I/O, given that overflow buckets are part of its data structure?**

Linear hashing is on average only slightly faster than one disk I/O because the cost of generating hash values is insignificant in comparison to disk I/O. Overflow pages do not, on average, contribute to the cost because they are temporary. When a bucket is full, the entry is added to an overflow page, but the bucket will eventually be split once it’s turn arrives in round-robin fashion. Therefore, overflow pages only exist as long as it takes for the buckets turn to be split to arrive, after which the data will be in a more reasonable size rather than in a potentially long overflow page.

1. **If a Linear Hashing index using Alternative (1) for data entries contains 10,000 records, with 10 records per page and an average storage utilization of 80 percent, what is the worst-case cost for an equality search? Under what conditions would this cost be the actual search cost?**

The worst case would be:

The bucket that the hash function points to is full, with 10 records: regardless of the 80% average utilization, the worst case is still a full bucket.

The fact that alternative (1) is used and that there are 10,000 records does not matter as the search value will result in a hash that points to the appropriate bucket. However, a hash only suggests that the search key *may* match a record in the bucket: hashing functions can result in different values resulting in the same hash. The result is that if two hash values match, they *may* be derived from the same value. Conversely, if two hash values do not match, they definitely *are not* derived from the same value. This means that while you can reject all other buckets because they do not match the bucket, you must still inspect each record in the matching bucket to ensure that they satisfy the equality.

Since the worst case is that we must inspect 10 entries in a full bucket to see which satisfy the equality, the worst case would be slightly more than 10 file I/O operations: the insignificant cost of the hash function(s) necessary to locate the bucket plus the 10 file reads necessary to inspect all 10 records.

**Part 3**

This problem was completed using mySQL and Java 1.8 in Eclipse. One interesting issue encountered is that the first query done often takes far longer than subsequent queries. For this reason I execute a query once before doing the timing. Timing is done by using System.nanotime(), Java’s high precision timer, and taking the average time of 1000 runs: single run values were extremely variable. Another interesting problem is that the time to create new connections affected the measurements: entirely different results were obtained after maintaining the same connection between tests. Finally, strange results were obtained when not manually closing the Statement objects instead of letting it happen automatically: the test program would run out of memory and performance would degrade, giving erroneous results. Clearly memory was being consumed more rapidly than the JVMs garbage collector was releasing it.

The tables are created with mySQL’s default engine: InnoDB. The developers of mySQL clearly consider B+ trees to be preferable, as it is not possible to create hash indexes through mySQL Workbench: they must be created manually with SQL statements.

I have included the Java source code for the testing in the workspace folder, though it’s not really “assignment quality” code: we aren’t asked to submit code, and I just included it in case the marker is curious about how the timing is done, and how the data is generated (found in Database.java).

I created the table in mySQL workbench, but I have provided a SQL dump of the script to generate the tables: this is comp418tme1.sql .

The generated data consists of 100 courses, 1000 students, and 50 registrations per course for 5000 registrations.

I apologize for the length of this section, but I honestly don’t see how this section can be properly answered in 1-2 pages. To be fair, a lot of space is taken up by tables.

1. **“List all courses in the database.”**

SELECT \* FROM course

Since we are presumably reading all columns of the course table, which would involve a file scan, I would not expect indexing to help this query. However, at least in the case of this table, there is some benefit from an index on all columns. Presumably, MySQL is internally setup in some way that reading all columns from an index is faster than reading them from the data file: perhaps in this case, the index is being cached in memory for faster access?

|  |  |  |
| --- | --- | --- |
| No index | B+ index on all columns | Hash index on all columns |
| 0.27 ms | 0.2 ms | 0.21 ms |

We can see that in this instance, there is some benefit to using either kind of index on all columns. I am skeptical as to whether this benefit would hold for very large sets of data.

1. **Update all the course fees by adding 6 dollars to each course.**

The first thought is that this query can be helped by indexing on cousefees, but this is not so… to be seen shortly.

UPDATE course SET coursefees = coursefees + 6

This was timed by taking the average times of 1000 runs with no index, a B+ index on the coursefees column, and a hash index on the coursefees column:

|  |  |  |
| --- | --- | --- |
| No index | B+ index on coursefees column | Hash index on coursefees column |
| 29.21 ms | 31.73 ms | 31.75 ms |

We see no real difference in performance in these scenarios. When considering the process involved, the reason becomes self-evident: indexes only facilitate quick retrieval of entries, and we are doing more than that here. The updated course fee must be written to the data file in addition to being updated in the index. The cost of jumping around a likely unclustered data file and updating the course fee values cancels out any benefit that the index provides.

1. **List the course numbers and titles of courses whose course fees are between 400 and 600 dollars.**

SELECT courseno, title, coursefees FROM course

WHERE coursefees >= 400 AND coursefees <= 600

As opposed to the previous section, here we can see that an index will benefit us: we are reading three columns from the course table and not writing data. Therefore, it is reasonable to expect that an index on the coursefees, courseno, and title columns will be beneficial (with coursefees first so that it is sorted by that column first):

|  |  |  |
| --- | --- | --- |
| No index | B+ index on the courseno, title, and coursefees columns | Hash index on the courseno, title, and coursefees columns |
| 0.2 ms | 0.11 ms | 0.12 ms |

Thus by properly sorting the columns in the index search key, the time to find a range has been halved. It is interesting that the hash index is about the same speed as the B+ index: presumably 100 data entries is still small enough that the difference between them is marginal (I further confirmed this by getting similar results with an equivalent query on the registration table, which has 5000 entries… even this isn’t enough to widen the gap between the index types).

1. **List the course numbers and titles of courses that have more than 10 students getting a grade lower than 50. [(Use group by courseNo and count(SID)].**

SELECT course.courseno, title

FROM course, registration

WHERE registration.grade < 50 AND course.courseno = registration.courseno

GROUP BY registration.courseno

HAVING COUNT(registration.sid) > 10

The obvious first step in optimizing this query is an index on the courseno and title columns of the course table, since these are the only columns of interest from that table:

|  |  |  |
| --- | --- | --- |
| No index | B+ index on the courseno and title columns of course table | Hash index on the courseno and title columns of course table |
| 17.74 ms | 5.56 ms | 5.65 ms |

This index has already reduced the time of the query to a third of the unindexed time. As we’ve seen before, the difference between the B+ and hash indexes are marginal, at least for this data set.

How to optimize the registration table is a more complex question, and requires considering the order in which columns are considered in the query. The first column in an index on this table should be grade, so that the entries are sorted by grade first to facilitate selecting a range of grades. The second column should be courseno, so that within the sorted grades, registrations are also already grouped by courseno. Finally, the third column should be sid so that each grouped set of courseno also as sids present in the index for faster counting: no need to go to the data file for the sid.

|  |  |  |
| --- | --- | --- |
| No index | B+ index on the courseno and title columns of course table,  B+ index on the grade, courseno, and sid columns of the registration table | Hash index on the courseno and title columns of course table,  Hash index on the grade, courseno, and sid columns of the registration table |
| 17.74 ms | 4.61 ms | 4.66 ms |

The two indexes in combination result in a query time one quarter of that of the unindexed version. And again, the difference between B+ trees and hash indexes are marginal.

1. **List the student numbers and names of students who received a grade greater or equal to 70% in the course “COMP418,” sorted by age ascending.**

SELECT student.sid, name

FROM student, registration, course

WHERE grade >= 70 AND registration.sid = student.sid AND registration.courseno = course.courseno AND title = 'COMP0'

ORDER BY age ASC

(I’m using “COMP0” because “COMP418” is not a course in the generated data).

As with the previous query, the obvious first thing to index is the age, sid and name columns of the student table for quick retrieval. We put the age first so that they are sorted in the index by age, at least in the case of the B+ index.

|  |  |  |
| --- | --- | --- |
| No index | B+ index on the age, sid and name columns of the student table | Hash index on the age, sid and name columns of the student table |
| 138.94 ms | 139.78 ms | 139.09 ms |

Surprisingly, this index does not provide any benefit to the query: presumably any potential advantage gained by pre-sorting ages is cancelled out by later operations. Inspecting the query in MySQL Workbench’s visual explain tool reveals that the data from the three tables are being fed into two nested loop, which will have an exponential impact on performance. Therefore, anything that we can do to reduce the amount of data being fed into those loops will increase performance. Finally, the data is being sorted after coming out of the loops, so pre-sorting will not help matters.

Let us try indexing registration on courseno first, rather than grade, since the explain tool says that courseno is the comparison in the first nested query. We should index the course table on courseno and title for the same reason.

|  |  |  |
| --- | --- | --- |
| No index | B+ index on the courseno, grade, and sid columns of the registration table,  B+ index on courseno and title columns of course table | Hash index on the courseno, grade, and sid columns of the registration table,  Hash index on courseno and title columns of course table |
| 124.61 ms | 1.73 ms | 1.79 ms |

By ensuring the indexes first index by courseno, we have suddenly reduced the runtime of the query by a massive amount by reducing the data fed into the nested loops.

Continuing from this, the final step is to index the student table on sid and name to further reduce the data being fed into the nested loops:

|  |  |  |
| --- | --- | --- |
| No index | B+ index on the courseno, age, and sid columns of the registration table,  B+ index on courseno and title columns of course table,  B+ index on sid and name columns of student table | Hash index on the courseno, age, and sid columns of the registration table,  Hash index on courseno and title columns of course table,  Hash index on sid and name columns of student table |
| 124.61 ms | 0.37 ms | 0.39 ms |

We’ve further reduced the query time to less than one percent of the unindexed query time!

**Conclusions**

The first conclusion is that, for this data and these queries, the difference between B+ tree and hash indexes is insignificant (either that, or MySQL is choosing to use B+ trees even when I specified a hash index). I would expect B+ tree indexes to get faster for range selections of very large data sets (in the millions).

The second conclusion is that it’s a poor idea to second guess the DBMS: it may seem to be intuitive that a particular index may or may not increase performance, but that may not be so. For example, for the course table, an index on all columns was faster than a file scan. For query 5, the first index did not reduce performance. This query required using the explain tool to investigate what the DBMS was actually doing. Once it was revealed that there were nested loops, the path of optimization became obvious.