Problem 1.

A table $R(K, A, \ldots)$ with 100,000 rows is stored in 10,000 disk blocks. The rows are sorted by $K$, but not by $A$. There is a dense, secondary $B^+$-tree index on $R(A)$, which has 3 levels and 500 leaves.

Suppose we want to sort $R$ by $A$. We have 101 memory blocks at our disposal. Method 1 performs an external-memory merge sort using all memory available. Method 2 takes advantage of the fact that the values of $A$ are already sorted in the $B^+$-tree index on $R(A)$: It simply scans the leaves of the index to retrieve and output $R$ rows in order.

How many disk I/O's do these two methods require? Which one is the winner?

Problem 2.

Consider the join $R \bowtie_{K \cdot A = S \cdot B} S$, given the following information about the tables to be joined. The cost metric is the number of disk I/O's and the cost of writing out the result should be uniformly ignored.

- $R$ contains 10,000 rows and has 10 rows per page.
- $S$ contains 2,000 rows and also has 10 rows per page.
- $S.B$ is a key of $S$.
- Both tables are stored compactly on disk in no particular order.
- No indexes are available.
- 52 memory blocks are available for query processing.

(a) What is the expected cost of joining $R$ and $S$ using a block-based nested-loop join, with $R$ as the outer table?

(b) What is the expected cost of joining $R$ and $S$ using a block-based nested-loop join, with $S$ as the outer table?

(c) What is the expected cost of joining $R$ and $S$ using a sort-merge join? What is the minimum number of memory blocks required for this cost to remain unchanged?

(d) What is the expected cost of joining $R$ and $S$ using a hash join? What is the minimum number of memory blocks required for this cost to remain unchanged?
Problem 3.

Consider a hash join of two tables $R$ and $S$. $R$ occupies 20000 blocks and $S$ occupies 15000 blocks. Unfortunately, we have an imperfect hash function that consistently assigns twice as many tuples to even-numbered partitions as to odd-numbered partitions (although all even-numbered partitions are of the same size, and so are odd-numbered partitions).

(a) Using this hash function, what is the minimum amount of memory required for the basic hash join algorithm described in lecture to complete in two passes?
(b) Can you come up with a strategy to make the required amount of memory nearly as low as the amount required by a perfect hash function?

Problem 4.

The mode of a list of values is the most common (frequent) value. There can be more than one mode for a list. For example, the list 1, 2, 2, 3, 4, 4, 2, 1, 1 has two modes 1 and 2.

(a) Assume that no index is available. Design the best algorithm you can come up with to compute one mode (any one) of a column in a table. Analyze the worst-case performance of your algorithm in terms of number of I/O's.
(b) Consider the following query that compute one mode for a subset of a table:

\[
\text{SELECT mode(r.A) FROM R r WHERE condition(r);}
\]

Briefly discuss how value-list index, project index, and bit-sliced index on $R(A)$ might be used to process the above query.

Problem 5.

For this problem, you may work either independently or in groups of two or three. If you work in groups of three, you must complete 5(c). Otherwise, 5(c) is optional. If you are not familiar with C++ programming, you may substitute Problems 6 and 7 for this problem.

Like in Homework #2, you are going to build a small piece of a database management system dubbed “DB216.” This time, the focus is query processing. To get started, copy the source files to your home directory using “cp –r ~cps216/db216-hw3/ ~/” on rack40. Go into directory db216-hw3/ and type “make clean”, and then “make”. After compilation completes, you will find an executable file db. Typing “./db” creates a database file test.db (if one does not exist already) and runs a number of tests on it.

For this problem, turn in a pointer to the directory containing your documentation, source code, and executable. Make sure the directory is readable to the TA after the homework is due. The directory should contain a README file documenting any known bugs/limitations, additional test cases or nifty features you have implemented, and instructions to compile and run your program. In the code, clearly indicate which parts are new or modified.

(a) Implement index nested-loop join.

Your code should go in src/qp/IndexNLJoinIter.{h,cpp}. See
src/qp/Iterator.h for the documentation of the iterator interface, which is slightly different from the one discussed in class. For example, iterators, refer to src/qp/{TableScanIter, PrimaryIndexScanIter, NLJoinIter, BlockedNLJoinIter, MergeSort}.h, cpp. To test your code, uncomment appropriate lines in src/main.cc and compile and run ./db. You are also encouraged to develop your own test cases in test/hw3.{h, cpp}.

(b) Implement merge join.

Your code should go in src/qp/MergeJoinIter.{h, cpp}. You can assume that both of its inputs have already been sorted in ascending order. A particularly messy case that you will have to deal with is when many rows from both inputs all have the same value for their join columns. This case requires a “mini” cross product between the joining input rows. Hence, you may need to allocate some memory to hold these joining input rows. Use new/delete for memory allocation/deallocation. To make your life easier, you do not have to handle the case when the joining input rows do not fit in main memory. Again, to test your code, uncomment appropriate lines in src/main.cc and compile and run ./db.

(c) Augment the sort iterator to handle multi-pass external merge sort. (Optional unless you work in a group of three.)

You should modify the code in src/qp/MergeSortIter.{h, cpp}. Currently, the iterator can only handle the case when the entire input fits in the pre-allocated memory buffer. Augment the implementation so that it can handle input of arbitrary size. Use the pre-allocated memory buffer and do not make any additional new requests. Use BDBRecno to store intermediate runs. Again, to test your code, uncomment appropriate lines in src/main.cc and compile and run ./db.

Problem 6.

For this problem, you may work either independently or in groups of two. If you are not familiar with C++ programming, you may substitute this problem and Problem 7 for Problem 5.

For this problem, you are going to explore the concept and implementation of materialized views. Essentially, materialized views are pre-computed query results. Frequently asked queries (or subqueries) are good candidates for creating materialized views. When the same query (or subquery) is asked for the second time, we can simply return the pre-computed result in the materialized view, without re-computing it from scratch. The downside of materialized views is that they must be maintained up-to-date with respect to the base tables. Whenever the base tables are modified, materialized views need to be modified accordingly.

To get started, copy the necessary files to your home directory using “cp –r ~cps216/examples/trigger-matview/ ~” on rack40.

Consider two tables $R(A, B, C)$ and $S(C, D)$ (created in create.sql). Implement the following materialized views:
For each of the materialized view that you need to implement:

1. Write a CREATE TABLE statement to create a database table to hold the content of the materialized view you are implementing.
2. Write an INSERT statement to initialize the content of the materialized view from the contents of the base tables.
3. Write CREATE TRIGGER statements to monitor the modifications on R and/or S and modify the table created in (1) accordingly. To make your life easier, you do not need to handle UPDATE statements on R and S (but you do need to handle both INSERT and DELETE). Make your trigger as efficient as possible; in particular, do not simply recompute the materialized view whenever base tables are modified.
4. Demonstrate that your materialized view is working correctly. Specifically, start by issuing the following statements (assuming that the table created in (1) is named V):
   ```
   DELETE FROM R;
   DELETE FROM S;
   SELECT * FROM V;
   ```
   Then, write a sequence of SQL modification statements to modify R and/or S. Follow each statement by the following statement:
   ```
   SELECT * FROM V;
   ```
   Verify that V is always maintained consistently with the current state of R and S.

The file distinct.sql contains an example of implementing “SELECT DISTINCT B FROM R,” as a materialized view. Use it as an example to structure your solution. For this problem, turn in a pointer to the directory containing your solution.

Problem 7.

For this problem, you may work either independently or in groups of two. If you are not familiar with C++ programming, you may substitute this problem and Problem 6 for Problem 5.

In this problem you are asked to build a Java application program DocIndex that supports efficient keyword searches on documents. Specifically, it should accept following command-line inputs:

- `java DocIndex create indexName`
  Create an index named `indexName`.
- `java DocIndex insert indexName pathToTextFileToBeIndexed`
  Index the specified document.
- `java DocIndex lookup indexName keyword_1 [keyword_2 keyword_3 ...]`
  Search the index for documents containing *all* of the specified keywords. The result should be a list of paths to documents satisfying the search condition. The paths
should be those that were supplied when these documents were inserted into the index using the insert command.

You should use either Berkeley DB or IBM DB2 as your backend database to store and manage and index. Turn in a pointer to the directory containing your documentation, source code, and executable. Make sure the directory is readable to the TA after the homework is due. The directory should contain a README file documenting any known bugs/limitations, additional test cases or nifty features you have implemented, and instructions to compile and run your program. You should also make a subdirectory that contains some plain-text files on which both you and the TA can test your program.