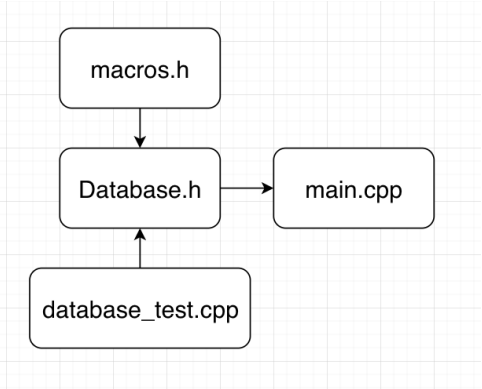
Project 1: Implementing SQL Statements

🎀 Claire Liu, CS320: Database Management Systems, Fall 2022 🎀

👷‍♀️ Description of Implemented Software

The file organization of this program is relatively simple:



- **Database** is the class that contains all the program implementation.
- **macros.h** contains some useful globals used in Database.h
- **database_test.cpp** contains the the test suite that tests the functionality of Database
- **main.cpp** contains some example sql statements where Database is used. The output is sent to the terminal. This is to demonstrate the output of the sql statements (from SELECT and UPDATE)

Within **Database**, the architecture is built around two main tables:

- db_primary
- db_attr

db_primary stores the records of all of the tables (created by the user via CREATE_TABLE) and their relevant information. More specifically, each record contains the following:

Name (not stored in record)	**Data type**	**Description**
tableName	char*	The name of the table created by the user
dbAttr	void*	A pointer to the first db_attr record associated with this table
numDbAttr	int	The number of db_attr records associated with this table
primaryKeyNum	int	Which attribute number is the primary key
dataRoot	void*	A pointer to the start of the data block associated with
dataCurr	void*	A pointer to the end of currently writing blocks, like a write head
dataEnd	void*	A pointer to the end of the block we are currently writing to
numCurrBlockCount	int	The number of blocks associated with this table
numCurrRecordCount	int	The number of records in the table

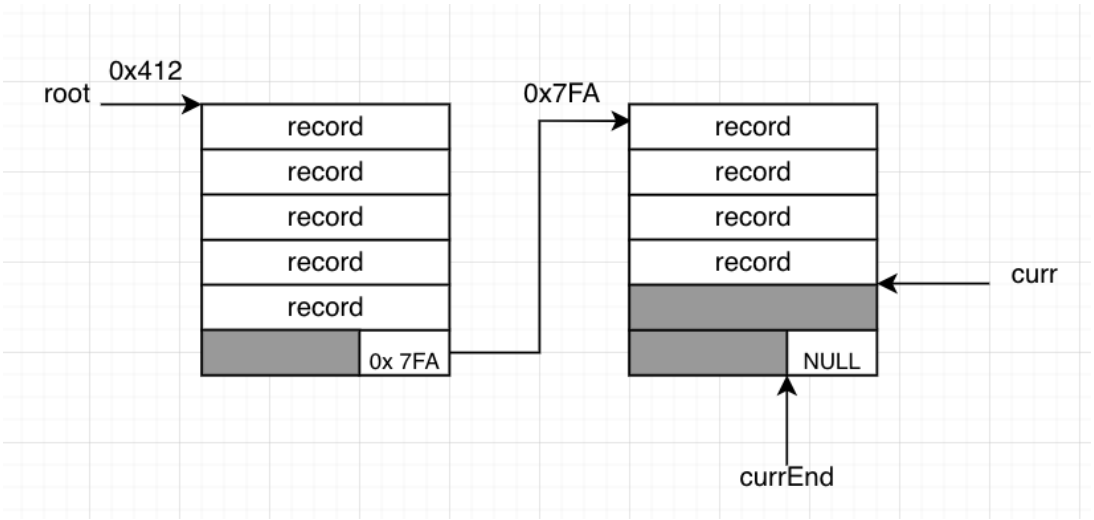
The record size for db_primary is fixed. We store the important information for db_primary as instance variables in the Database class.

```
void* db_primary = NULL; // initially set to NULL, but holds the root of db_primary
void* db_primary_curr = NULL; // write head for db_primary
void* db_primary_curr_end = NULL; // end of current block we're writing to
int db_primary_records = 0; // number of records in db_primary
int db_primary_blocks = 0; // number of blocks associated with the table
```

db_attr table stores all of the attributes and their relevant information. It serves as a lookup table so that as we traverse through a table we know what to cast the bytes into. Like db_primary, we keep the important information about db_attr as instance variables in the Database class. Each db_attr record stores the following information:

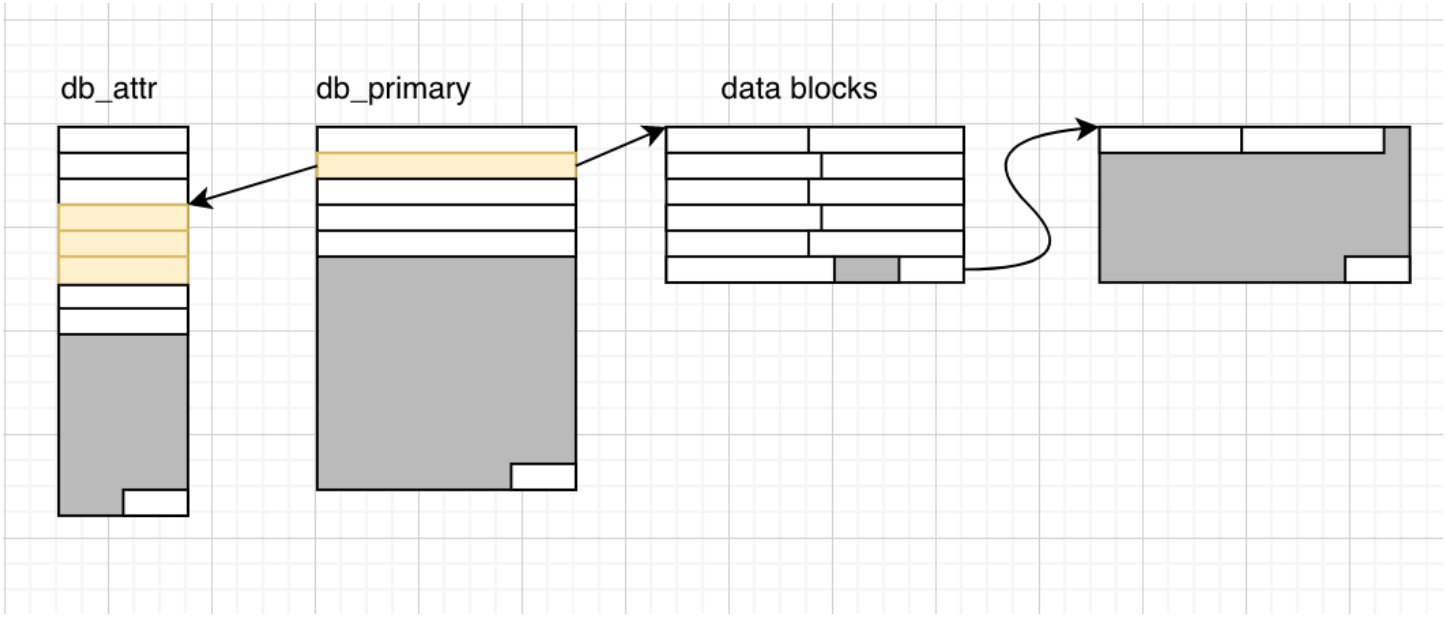
Name (not stored in record)	**Data type**	**Description**
attrName	char*	The name of the attribute
dataType	dataType (an enum)	Either VARCHAR, CHAR, SMALLINT, INTEGER, REAL
attrLength	int	The (max) length if dataType is VARCHAR or CHAR

To recap, the diagram below shows the basic format of how the db_primary, db_attr, and table data blocks work. We keep a pointer (root) to the root, a pointer (curr) to current write head, and a pointer (currEnd) to the end of the block we are currently writing to. The blocks are all initially calloc'd and left as voids. The blocks are unspanning, so there is some wasted space, but this is a trade off for cleaner and simpler implementation. In some ways, the blocks that make up db_primary act like a linked list, where the end of each block contains a pointer to the next block.



For fixed records, we generally can work off of lookups to the db_attr table and offsets calculated from those attributes. db_primary and db_ attr records are always fixed. Data records are fixed if they do not contain varchars. They are variable if they do contain varchars. For variable records, we also work off of lookups to the db_attr table, but we have to move byte by byte to navigate through the tables, looking for the END_FIELD_CHAR which is ` and the END_RECORD_CHAR which is ~.

Here's how db_primary, db_attr, and the data blocks for a table interact with each other. A record in db_primary corresponds to a table. The record contains metadata about the table, including a pointer to the first db_attr record associated with the table and a pointer to the first data block associated with the table. With hashing, an additional layer is introduced at the data block level to contain the buckets, and then the unordered lists for each bucket look just like the regular data blocks.



There is some basic error handling in place. For example, the table names, the attributes for each table, and the primary keys must be unique and this is enforced.

🛠️ How to Run the Program

To run the program, run `make` from the project root. This runs all of the configurations of the program
To see unordered inserts, run `make unordered`
To see ordered inserts, run `make ordered`
To see hashed inserts, run `make hashed`
To run the test suite, run `make test` from the project root.
Within the body of main in main.cpp, you could add sql statements in the following format. The format of these statements generally matches the project description.

```
db->create_table("movies", "id", 6,
                 "id", "integer",
                 "title", "char(32)",
                 "rating", "real");
db->insert("movies", 3, 25, "The Last Starfighter", 3.0);
db->insert("movies", 3, 26, "Princess Diaries", 5.0);
db->select("movies", 2, "id, title", "id = 27");
// update doesn't fully follow project format, see limitations section
 db->update("movies", 3, "rating", "5.0", "id > 27");
```

The git repository for this program is https://github.com/lee-anh/DB_Project1. Please request access if interested.

🚛 Understanding the System Dump

The system dump mimics the output of sql statements. printDbPrimary printDbAttr, and printTable all behave similarly to SELECT * in SQL. For printDBAttr you can see where the different block breaks are and for printDBAttr and printTable you can see the address of where each record is located.

printDbPrimary

```
name | void* db_attr | int num_db_attr | int primary_key_num | void* data_root | void* data_curr | void* data end | int numDataBlocks| int numDataRecords
movies | 0x7fdb57704490 | 3 | 0 | 0x7fdb57704690 | 0x7fdb57704ee0 | 0x7fdb57705088 | 3 | 26
test | 0x7fdb57704568 | 4 | 0 | 0x7fdb57704890 | 0x7fdb577048bf | 0x7fdb57704a88 | 1 | 1
stars | 0x7fdb57704a90 | 3 | 0 | 0x7fdb57704c90 | 0x7fdb57704cd4 | 0x7fdb57704e88 | 1 | 1
critics | 0x7fdb57704b68 | 3 | 0 | 0x7fdb57705c90 | 0x7fdb5770604f | 0x7fdb57706088 | 2 | 51
movies1 | 0x7fdb57704c40 | 4 | 2 | 0x7fdb57705490 | 0x7fdb577054c1 | 0x7fdb57705688 | 1 | 1
```

Contains lots of metadata to make sure searches, insertions, and updates can all function correctly.

printDbAttr example output:

```
void* | char(TABLE_NAME_SIZE) attrName | (int) type | (int) length
0x7fdb57704490 | id | 3 | -1
0x7fdb577044d8 | title | 1 | 32
0x7fdb57704520 | rating | 4 | -1
0x7fdb57704568 | id | 3 | -1
0x7fdb577045b0 | director | 1 | 32
0x7fdb577045f8 | title | 0 | 32
0x7fdb57704640 | rating | 4 | -1
----------------------------------------------------------------
0x7fdb57704a90 | id | 3 | -1
0x7fdb57704ad8 | fname | 1 | 32
0x7fdb57704b20 | lname | 1 | 32
0x7fdb57704b68 | id | 3 | -1
0x7fdb57704bb0 | fname | 0 | 32
0x7fdb57704bf8 | lname | 0 | 32
0x7fdb57704c40 | id1 | 2 | -1
----------------------------------------------------------------
0x7fdb57705090 | title1 | 0 | 32
0x7fdb577050d8 | director1 | 1 | 32
0x7fdb57705120 | rating1 | 4 | -1
```

For printDBAttr you can see where the different block breaks are and the address of each record.

printTable

```
movies system dump:
record address | id* | title | rating
0x7fdb57704690 | 25 | The Last Starfighter | 3
0x7fdb577046b8 | 26 | Princess Diaries | 5
0x7fdb577046e0 | 27 | Star Wars | 4.2
0x7fdb57704708 | 28 | Rouge One | 5
0x7fdb57704730 | 127 | Frozen | 4
0x7fdb57704758 | 208 | Tangled | 5
0x7fdb57704780 | 217 | Honey I Shrunk the Kids | 5
0x7fdb577047a8 | 228 | Big Hero 6 | 5
0x7fdb577047d0 | 237 | Cars | 5
0x7fdb577047f8 | 248 | Cars 2 | 5
0x7fdb57704820 | 257 | Toy Story | 5
0x7fdb57704848 | 268 | Tarzan | 5
0x7fdb57705890 | 272 | Father of the Bride | 5
0x7fdb577058b8 | 274 | Avengers | 5
0x7fdb577058e0 | 275 | Thor | 5
0x7fdb57705908 | 276 | Crazy Rich Asians | 5
0x7fdb57705930 | 277 | Little Shop of Horrors | 5
0x7fdb57705958 | 278 | Captain America | 5
0x7fdb57705980 | 280 | Iron Man | 5
0x7fdb577059a8 | 281 | Back to the Future | 5
0x7fdb577059d0 | 283 | Cheaper by the Dozen | 5
0x7fdb577059f8 | 284 | Antman | 5
0x7fdb57705a20 | 285 | Joy Luck Club | 5
0x7fdb57705a48 | 287 | Solo | 5
0x7fdb57704e90 | 288 | Cinderella | 5
0x7fdb57704eb8 | 297 | Sleeping Beauty | 5
```

For print table you can see the result of a select * on the table and the addresses of each record.

🧮 Algorithms Used and Their Implementation

Unordered

Unordered inserts could be done in constant time for fixed and variable records because they were simply copied in where the write head was.

Fixed records were more efficient when checking for the primary key because we could simply work based off and offset to the primary key for each record whereas for variable we had to iterate through all the data in the table since we could not predict how long each record would be (and thus where each primary key would be).

Also, for variable updates, all the records, plus the new record, were copied over to newly allocated blocks. However, the old data blocks were freed as we went, so it was just more costly in that instead of potentially copying just a portion of the data, we just copied all of the data over. This method was more straightforward to implement.

Unordered selects and updates were done in linear time. Selects and updates used a boolean array called isTarget to determine which records fit the WHERE criteria to select/update. isTarget essentially represented the set of records that fit the conditional. Therefore, if ANDs and ORs are added in the future, functionally can easily be expanded.

Ordered

Ordered inserts were more costly to system resources compared to unordered inserts since we had to locate an insertion point for the data. This took linear time. Theoretically, for fixed records, a binary search would have improved the search time to log n time, but I didn't get around to implementing this.

When fixed inserts were done, all the records below the insertion point were copied down. Variable insertions for ordered worked like the variable updates in unordered where all the records, plus the new record, were copied over to newly allocated blocks.

Updates for ordered worked like the insertions for ordered.

Select worked the same as unordered for ordered.

Hashed

Hashing basically introduced another layer to the unordered inserts, cutting down the search time for primary keys and updates.

I used an external hashing scheme where primary keys were hashed to buckets, which were of fixed size. The number of buckets were just how many buckets could fix on one block. If there was a collision, then an unordered chain begins. No rehashing was implemented.

Updates and selects to the buckets worked the same as unordered, but of course with the hashing layer the performance of hashed was better than unordered. The only drawback was the hashed resulted in more wasted space across the system since more data blocks had to be allocated.

Some important functions

There were a few important functions that were critical to the system's functionality.
For example, the checkSpaceAdd and checkSpaceSearch (and similar functions) made traversal and addition to the data blocks safe. Since the blocks are like linked lists, it's really easy to get out of bounds when iterating through the data. But, by calling the checkSpaceSearch before each iteration, we would make sure the next read was safe because the checkSpaceSearch would redirect the read head to the next block if we were at the end of a block.

Similarly, the checkSpaceAdd block made sure there was enough space left of the current block to add another record and allocated new space and redirect the write head if needed.

findPrimaryKeyFixed and findPrimaryKeyVariable were also important because they acted as a check to see if a record with a given primary key existed in the database, and returned the negative index of where the primary key would be if a record with the primary key didn't exist in the database already. The negative index could then be multiplied by -1 to get the insertion point, and that was used by the ORDERED inserts.

🧪 Verification Plan

To ensure that the system functions correctly, I wrote some unit tests using gtest.
All tests are in the same testing suite, called database_test.
Tests are separated out into 4 classes:

- DatabaseTest - tests functions whose performance doesn't depend on the insertion method
- DatabaseTestUnordered
- DatabaseTestOrdered
- DatabaseTestHashed

The tests were designed with the following in mind:

- Different insertion methods
- Different data types - VARCHAR, CHAR, SMALLINT, INTEGER, and REAL
- Fixed vs. variable records

Insertion test cases were written with edge cases in mind such as empty insert, insert to beginning, insert to middle, and insert to end.

Load testing was done to make sure that the different blocks linked for data, db_primary, and db_attr.

Update was tested through the updateTest function, which is the same as update(,) except it returns the number of rows updated.

Select shares a lot of the same functionality and implementation as update, so it is tested by updateTest indirectly in the unit tests. We visually inspect the result of select in the system dump.

```
claireliu@Claires-MacBook-Air p1 % make test
cmake -S . -B build
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/claireliu/Developer/Databases/p1/build
cmake --build build
Consolidate compiler generated dependencies of target gtest
[ 18%] Built target gtest
Consolidate compiler generated dependencies of target gtest_main
[ 36%] Built target gtest_main
Consolidate compiler generated dependencies of target database_test
[ 45%] Building CXX object CMakeFiles/database_test.dir/database_test.cpp.o
[ 54%] Linking CXX executable database_test
[ 63%] Built target database_test
Consolidate compiler generated dependencies of target gmock
[ 81%] Built target gmock
Consolidate compiler generated dependencies of target gmock_main
[100%] Built target gmock_main
cd build && ctest --output-on-failure
Test project /Users/claireliu/Developer/Databases/p1/build
      Start  1: DatabaseTest.CreateTable
 1/54 Test  #1: DatabaseTest.CreateTable ..............................   Passed    0.01 sec
      Start  2: DatabaseTest.RetrieveDBPrimaryRecordOnly
 2/54 Test  #2: DatabaseTest.RetrieveDBPrimaryRecordOnly ..............   Passed    0.01 sec
      Start  3: DatabaseTest.CheckGetPrimaryKeyType
 3/54 Test  #3: DatabaseTest.CheckGetPrimaryKeyType ...................   Passed    0.01 sec
      Start  4: DatabaseTest.RetrieveDBPrimaryRecordFirst
 4/54 Test  #4: DatabaseTest.RetrieveDBPrimaryRecordFirst .............   Passed    0.01 sec
      Start  5: DatabaseTest.RetrieveDBPrimaryRecordMiddle
 5/54 Test  #5: DatabaseTest.RetrieveDBPrimaryRecordMiddle ............   Passed    0.01 sec
      Start  6: DatabaseTest.RetrieveDBPrimaryRecordEnd
 6/54 Test  #6: DatabaseTest.RetrieveDBPrimaryRecordEnd ...............   Passed    0.01 sec
      Start  7: DatabaseTest.NoDuplicateTableNames
 7/54 Test  #7: DatabaseTest.NoDuplicateTableNames ....................   Passed    0.01 sec
      Start  8: DatabaseTest.NoDuplicateTableAttributes
 8/54 Test  #8: DatabaseTest.NoDuplicateTableAttributes ...............   Passed    0.01 sec
```

… more tests here …

```
      Start 43: DatabaseTestHashed.UpdateFixedEqual
43/54 Test #43: DatabaseTestHashed.UpdateFixedEqual ..................   Passed    0.01 sec
      Start 44: DatabaseTestHashed.UpdateFixedNotEqual
44/54 Test #44: DatabaseTestHashed.UpdateFixedNotEqual ...............   Passed    0.01 sec
      Start 45: DatabaseTestHashed.UpdateFixedGreaterThan
45/54 Test #45: DatabaseTestHashed.UpdateFixedGreaterThan ............   Passed    0.01 sec
      Start 46: DatabaseTestHashed.UpdateFixedGreaterThanOrEqualTo
46/54 Test #46: DatabaseTestHashed.UpdateFixedGreaterThanOrEqualTo ...   Passed    0.01 sec
      Start 47: DatabaseTestHashed.UpdateFixedLessThan
47/54 Test #47: DatabaseTestHashed.UpdateFixedLessThan ...............   Passed    0.01 sec
      Start 48: DatabaseTestHashed.UpdateFixedLessThanOrEqualTo
48/54 Test #48: DatabaseTestHashed.UpdateFixedLessThanOrEqualTo ......   Passed    0.01 sec
      Start 49: DatabaseTestHashed.UpdateVariableEqual
49/54 Test #49: DatabaseTestHashed.UpdateVariableEqual ...............   Passed    0.01 sec
      Start 50: DatabaseTestHashed.UpdateVariableNotEqual
50/54 Test #50: DatabaseTestHashed.UpdateVariableNotEqual ............   Passed    0.01 sec
      Start 51: DatabaseTestHashed.UpdateVariableGreaterThan
51/54 Test #51: DatabaseTestHashed.UpdateVariableGreaterThan .........   Passed    0.01 sec
      Start 52: DatabaseTestHashed.UpdateVariableGreaterThanOrEqualTo
52/54 Test #52: DatabaseTestHashed.UpdateVariableGreaterThanOrEqualTo ...   Passed    0.01 sec
      Start 53: DatabaseTestHashed.UpdateVariableLessThan
53/54 Test #53: DatabaseTestHashed.UpdateVariableLessThan ............   Passed    0.01 sec
      Start 54: DatabaseTestHashed.UpdateVariableLessThanOrEqualTo
54/54 Test #54: DatabaseTestHashed.UpdateVariableLessThanOrEqualTo ...   Passed    0.01 sec

100% tests passed, 0 tests failed out of 54

Total Test time (real) =   0.53 sec
```

Hooray! All of our tests pass! 🙌🏼

In addition to the gtests, we need to do a visual inspection of the system dump. The system dump is demonstrated from main.cpp and outputs to the terminal. This allows us to see that the prints, selects, and updates all output coherent and correct data to the user.

🚧 Program Limitations

- For the WHERE clause of SELECT and UPDATE statements, the program cannot parse ANDs and ORs. It can only handle one clause with one comparison at a time. No strings with spaces are allowed for now.
- Update can only update one field at a time.
- Update takes all the arguments as char*, even the value to set, deviating from the project specs a little.
- Should not update the primary key since the order of the records cannot be maintained.

- There is limited input error checking for some functions, so a segmentation fault may result for incorrect input. The system is not fail-safe.
- Ideally, a binary search would have been implemented for finding the primary keys of the fixed, ordered tables.

Thank you for your understanding 🙏 I tried really hard 🥲