CS165 Final Report

Kevin Zhang

December 22, 2016

1 Feature Overview

Almost all of the basic features requested of this project have been implemented. The last features remaining at the moment are nested-loop joins and updates. The list of features includes:

- Creates - create a database, table, column, or index.
- Inserts - insert a row into any table.
- Selects - select data using a range operator on any column.
- Fetches - fetch data after selecting.
- Loads - load data from files into the database.
- Indexes - create indexes over columns for faster selects.
- Joins - join values across tables to construct additional tuples.

In addition, deletion in B+ trees has been implemented and tested to work (but not incorporated into updates yet).

Code has been organized according to project components; the Makefile used looks in several pre-defined folders for source and header files. Header files mirror the layout of the source files. In addition, a `data` folder stores all database metadata and values (persisted upon server shutdown and loaded upon server startup). This folder (along with subfolders) is automatically created when needed. A `catalog` file at the root contains database metadata.

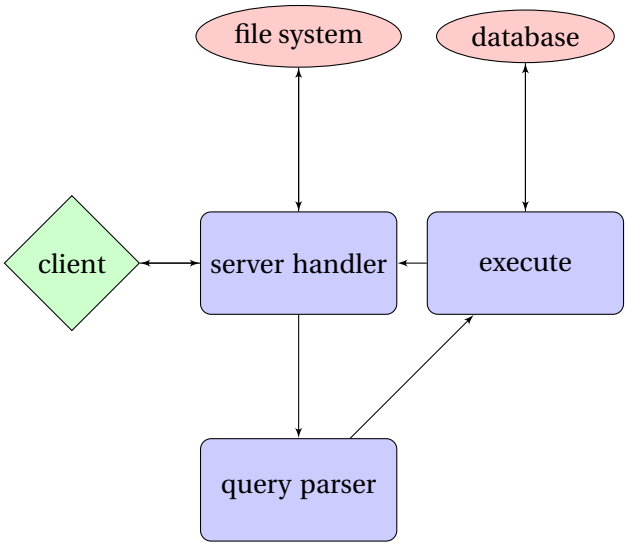
Extensive logging has been implemented; when turned on, all queries, server responses, client

messages, and more are printed to stdout. In addition, after execution of each query and on startup, the current database and client context are printed to help debug.

---

2 System Overview

The diagram below details how queries are handled by the server.



Green nodes indicate sections the user interacts with, red nodes indicate data sources, and purple nodes indicate server functions. On a client request, the following actions are taken:

1. Request is forwarded from server handler to query parser.
2. Query parser constructs a `DbOperator` object and passes it to the executor.
3. Executor interacts with the database to build a `Result` object if necessary.
4. Executor returns a string response to the server.
5. Server responds to the client if appropriate.

The `DbOperator` object contains all metadata required to execute queries; a `ClientContext` object is used to store all `Result` objects created from queries.

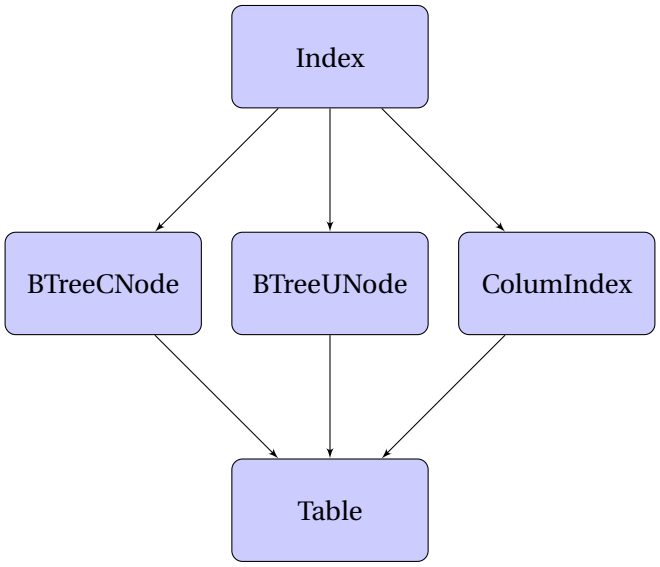
---

3 CREATES

Handling create queries is fairly straightforward; on request, corresponding objects are created. For a database, this is a `Db` object; for a table, a `Table` object; for a column, a `Column` object; and for an index, an `Index` object. The `Db, Table, Column` objects contain straightforward fields describing the contents of each, along with pointers to payloads and contained objects.

`Index` objects are a bit more complicated; these contain a `IndexType` identifying the `Index` as a B+ tree or column and a `clustered` descriptor. `Index` objects additionally contain pointers to their respective `Column` objects (for ease of data lookup) and a `IndexObject`, which is either a `BTreeUNode` (unclustered B+ tree), `BTreeCNode` (clustered B+ tree), `ColumnIndex` (containing integer arrays of values and indices), or `NULL`, signifying a clustered, sorted index (no additional data is needed to maintain this type of index).

`Index` objects are stored in the corresponding `Table`. Upon creation of a clustered index, no data is stored (it is assumed that clustered indices will be declared before any data is inserted into the database). However, upon creation of unclustered indexes, all data is inserted into the indexes. Thus, queries can immediately make use of the indexes (after insertion completes, of course).



---

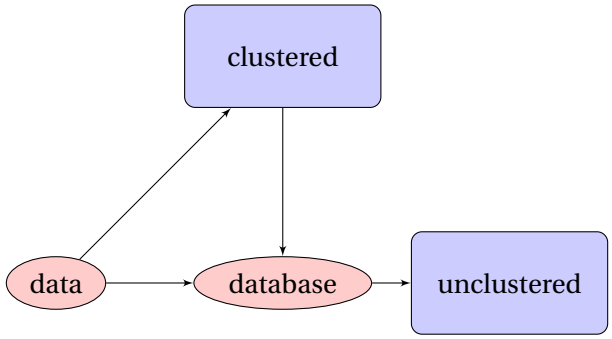
4 INSERTS

Inserts without indexes are easy; we append to the end of the corresponding `Column` objects in the given `Table`. Handling `Index` objects is harder.

The system completes the following process whenever inserting data:

1. Search for clustered indexes on the table.
2. If one is found, insert into the clustered index first (no more than one clustered index may exist). Retrieve the indices of the inserted values, then shift all values in each column one by one (to avoid unnecessary random access) as necessary to insert new data.
3. If none are found, append all values to the end of the table, then insert the new `(value, index)` pairs into each of the unclustered indices.

The B+ tree implementation is critical for this portion of the project (and probably the most time-consuming). The optimal parameters for B+ trees have yet to be determined.



---

5 SELECTS

Selects without indexes are also easy; we simply scan through the entire column and append to a results array contained in the corresponding `ClientContext` object. When we have a usable `Index`, we search through the `Index` to find the smallest value greater than or equal to the minimum of a select query. Depending on the type of index, we then:

- Clustered B+ Tree: search for the maximum, then create an array containing all values between the two indexes found.
- Clustered Sorted: binary search for the maximum, then create an array containing all values between the two indexes found.
- Unclustered B+ Tree: iterate forward through the leaves until we find a value greater than the maximum, and accumulate an array of encountered indices.
- Unclustered Sorted: iterate forward through the array until we find a value greater than the maximum, and accumulate an array of encountered indices.

For batched selects, this behavior is slightly modified:

- Clustered indexes: iterate across select queries first, building result arrays one by one.
- Unclustered indexes: iterate down the index and build result arrays along the way.

For clustered indexes, we avoid leapfrogging between result arrays and instead write all values to each result array first. This avoids some degree of random access. For unclustered indexes, we avoid scanning the same data more than once and instead accumulate values as we go, avoiding multiple data scans but incurring costs from leapfrogging across result arrays instead.

Normal queries took the following amounts of time (50000 tuples, run in background):

Select %	# us
9.1%	347
18.2 %	384
27.3 %	355
36.4 %	388
45.5 %	543
54.6 %	518
63.7 %	563
72.8 %	540
81.9 %	708
90.9 %	635
100 %	634

In total, these queries run independently would take 5615 microseconds. Batched queries were able to consistently complete these queries in under 5000 microseconds; we would likely see a larger margin of improvement with a very large number of queries. The largest number of concurrent queries we can run depends on how much space we have available for results.

---

6 FETCHES

All fetches are done in the same way in this project; we iterate across the array of indices from a select query, and retrieve the corresponding values from the column in question. The constructed `Index` objects are not as useful here, since keys into the indexes are values instead of indices. An alternative implementation might maintain two sets of `Index` objects such that retrieving values using an indice is possible as well. This incurs additional overhead, and does not solve the problem of random access.

---

7 LOADS

Loads are done simply by sending row after row to the server. An alternative implementation would batch all data, and utilize all data at once to construct the indexes. This would decrease `Index` construction by a constant factor, since construction of each node is still required (this method also requires fairly expensive computation).

---

8 INDEXES

Selects, inserts and fetches have been described for indexes above. Sorted column indexes are straightforward. There was not enough time to perform experiments to determine the optimal fan out, but, based off calculations on the page size (assuming 64K) and struct size, the optimal number of values stored in each node is around 2700.

We also recorded data on scan vs. index select performance. Some of the results are shown below:

Select %	Scans	Indexes
0 %	220	1
9.1 %	226	94
18.2 %	197	71
27.3 %	220	120
36.4 %	349	188
45.5 %	411	198
54.6 %	530	238
63.7 %	629	277
72.8 %	704	347
81.9 %	303	371
90.0 %	325	415
100.0 %	304	425

Table 8.1: Runtime in milliseconds, 50000 tuples

There is a clearly linear trend upwards for the indexes, while runtime seems to be mostly linear for scans up until around 80% selectivity, when runtime drops again. This is expected; runtime for indexes are dependent entirely on the number of values that need to be saved. Scans are a bit more complex; deciding whether a value needs to be saved or not requires a branch, which can result in very slow behavior when values are randomly distributed and are close to 50% selectivity. This is what we see in our data; runtime increases up but then drops when branching behavior becomes more consistent.

Overall, indexes seem to perform better for lower selectivities, as expected.

---

9 JOINS

Only hash joins have been implemented for now (nested joins are fairly trivial to implement). These are implemented as follows:

1. Insert all values from one select / fetch pair into a hashtable.
2. Iterate over all values from the other select / fetch pair. For each value and index:
    - Retrieve all indices from the hashtable with a matching value.
    - Insert all indices into the first result handle.
    - Insert as many copies of the index into the second result handle.

Again, there was not enough time to compare hash join performance with that of nested loops, but it seems reasonable to assume that, for low selectivities, hash joins are much more efficient than nested loop joins. Experiments would be required to determine an exact boundary as a function of the % selectivity and number of tuples.