Dongpeng Xia, Frank Zhang

NakedDB

Brief project structure overview:

The entry point to the project is coreOS.NakedDatabase. This holds the main function, the input, and a function that handles basic parsing. It determines the general type of SQL statement and passes it to either DBHelper or QueryTree. All SQL SELECT statements are passed to QueryTree. Everything else is passed to DBHelper for parsing. The database is backed by the package relation, which holds the Table class. The major data structure we used for the table and the index was a RedBlack Tree (java.util.TreeMap). Each table and index is saved in a binary file (using java.serialization). A central file Manifest.txt has a list of all the table file names. The database is managed through two singleton overseer classes, TableGolem and IndexGolem.

General behaviors:

The database does not save tables to disk unless called to do so using the command “COMMIT”, or if the user exits the database using “EXIT”. Indices are saved immediately. The saved database is loaded when the program starts. Rule-based optimization is implemented through the query.Optimizer class. Null values are printed as “nAN”. Each table must be created with a primary key. Keyless tables are not allowed. Tables and result tables are automatically sorted by default. The input is a simple command line input. Each line in the console must correspond to one SQL statement. Lines must end in semicolons.

SQL Grammar:

All statements are case insensitive. [Bracketed arguments are optional]. WHERE conditions support the following comparison operators: >, <, =. Multiple conditions may be chained together using AND or OR. See appendix for examples.

SELECT <*attrname*> ... FROM <*tablename*> WHERE <*condition*>*...*

- Our grammar supports multiple nested selects, multiple joins and multiple where conditions. Refer to the appendix for working examples.
- Column names can be written as *Tablename.attrname* if tables share similar column names, i.e during joins.

SELECT … FROM <*tablename*> JOIN <*tablename*> *WHERE ...*

- The JOIN syntax has been modified from standard SQL. Instead of using JOIN ON (condition), our grammar uses the WHERE clause instead. The user should provide the joining condition in the where clause.
- Our grammar supports multiple table joins, and also self joins.
- Instead of using FULL OUTER JOIN, the user can simply create a JOIN without a joining condition in the WHERE clause.
- To make an INNER JOIN, simply specify the joining condition in the WHERE clause.

CREATE TABLE <*tablename*> VALUES (<attrname> <attrtype> [NOT] [NULL] [PRIMARY KEY], ...)

- The user can provide an optional argument of either NULL, NOT NULL, or PRIMARY KEY. If no argument is provided, then by default the attribute will not be nullable. Marking an attribute as NULL allows the value to be nullable, and marking the value as NOT NULL specifies that it cannot be null.
- The optional argument PRIMARY KEY signifies the primary key. Each CREATE TABLE statement must mark a column as the primary key, otherwise it is rejected.
- The following attribute types are supported: char (same as string), int, long, boolean, float, double.

DROP TABLE [IF EXISTS] <*tablename1*>*, <tablename2*> ...

- This clears the target table then removes it from the database. The IF EXISTS argument silences error messages if the table does not exist. The user can delete multiple tables with a single command.
- Note that dropped tables are not deleted from the disk until "EXIT" or "COMMIT" is called.

INSERT INTO <*tablename*> VALUES (<*v1*>, <*v2*>....)

- The values provided must match the table's attributes. If there are too many or too few values provided, or if the values do not match the table's respective types, then the command is rejected.

INSERT INTO <*tablename*> (<*attr1*>, <*attr2*>...) VALUES (<*v1*>, <*v2*>....)

- This allows the user to choose which values to insert, and all unnamed values are inserted as null. If a non-nullable column or the key is not included, then the command is rejected.

DELETE FROM <*tablename*> [WHERE <*condition1*>]

- If no WHERE condition is stated, then the table is cleared.
- Multiple WHERE conditions joined by AND or OR may be used.

CREATE INDEX ON <*Tablename*>.<*attributename*>

DROP INDEX ON <*Tablename*>.<*attributename*>

- Our database provides single column indexing. Indexes are held in disk and loaded in memory only on access.

<u>Additional commands:</u>

DUMP

- This prints all tables to console.

COMMIT

- This writes all tables to disk

CLEAN

- This cleans all intermediate and temporary tables generated from SELECT statements

EXIT

- This command first writes all tables to disk before gracefully exiting. It is not case sensitive.

SOURCE <*filename*>

- This executes all commands in the given file. The file should be a text file, and each line should contain only one command.

Functionality:

Query Input Manager

The user provides input via a command line interface. Each command must end in a semicolon. The input is parsed depending on its type. If it is a SQL SELECT statement, then it is handled via the packages query, query_tree_nodes, and query_where_filters. Otherwise, the statement is handled in the class DBHelper. The project does not use recursive descent for parsing. Instead, commands are first parsed and split into their respective clauses. Then each clause is parsed in a loop by their respective handling method.

Data Definition Language: see SQL Grammar

All four methods, CREATE TABLE/INDEX, DROP TABLE/INDEX are implemented in DBHelper. When a table is created, it is assigned an auto-generated internal ID to enable easy identification. Tables must also have unique table names. Each table also ensures that it has a primary key, and that all its columns have unique names. Indexes are stored in disk and are not held in memory. When an index is used, it must first be loaded from disk. To use more than one index, the currently open one must first be closed.

Data Manipulation Language: see SQL Grammar

The methods INSERT, DELETE, and SELECT are all implemented. INSERT and DELETE are implemented in DBHelper, whereas SELECT is handled in the three query packages. These packages parse the input and generate a query tree, then they process it. In order to preserve referential integrity, duplicate keys and attribute names within a table are not allowed. Additionally, all tables must have unique names. Attribute names are internally renamed in the form <*Tablename*>*.Attributename.* Users can also directly use this notation within SQL statements. This allows joins between two tables with similar attribute names. The program has a powerful parser, however, the grammar is restricted. See appendix for advanced examples, in addition to SELECT syntax differences.

Main Memory Execution:

Because every table is backed by a RedBlack Tree, all tables are sorted by default. This includes the results from SELECT statements. Our syntax for joins varies from standard SQL syntax. Outer joins are detected when a JOIN does not have a matching WHERE condition specifying the joining condition. If a joining condition is specified in the WHERE clause, then an inner join is performed. Additionally, the WHERE clause in a SELECT statement can have multiple conjunctions and/or disjunctions.

Storage Structures - Disk:

The storage structures of this database are primarily optimized for fast insertion and flexibility over storage efficiency. The primary data structure used is a RedBlack Tree. RedBlack trees implement Java's Serializable class. This allows us to efficiently save to and read tables from the disk. The program does not automatically perform writes, and the user must specifically call "COMMIT" to write to disk, or to exit the program. Each value is wrapped in an Item object which allows for more simplicity and flexibility at the cost of greater storage space.

Optimization - Potential:

We have implemented Rule-based optimization via the query.Optimizer class. This shuffles our query tree so that SelectionNodes are pushed to the leaves, thus performing selections as early as possible.

Architecture: Project Structure

Package.coreOS: This package holds the main methods for the project.

- NakedDatabase: This holds the main function (which is just the console input) and a master parsing method which determines which class should handle further parsing.
- DBHelper: NakedDatabase passes all non-SELECT commands to DBHelper to parse.
- IndexGolem: a singleton overseer class that handles indexes. It should be accessed using the openIndex() and closeIndex() method, which reads and writes indexes from the disk. In order to use another index, the currently open one must be closed first. Index files are stored as <*Tablename.Attributename.idx*>.

Package.miscellaneous.Helpers

- This contains static helper methods used internally. These methods are generated due to similarities between the major parsing methods and as a result allow for more code reuse.

Package.query: holds methods to parse and optimize SELECT Statements.

- Checker: This class verifies the correctness of a SELECT statement.
- Optimizer: This implements rule-based optimization.
- Parser: A static class that contains helper methods for parsing. These methods are used throughout Package.query_tree_nodes.
- QueryTree: This class provides end to end parsing and processing of SELECT statements and is called when the master parser detects "SELECT". It holds the root of the query tree. The resulting selection is added to a table and placed in TableGolem. This method generates several intermediate tables which are automatically cleared.

Package.query_tree_nodes: used by class QueryTree.

- QueryActionNode: This is the base node class, and other classes in this package extend it.
- CrossProductNode: This class handles all joins.
- FromNode: This handles the FROM/WHERE clause.
- SelectionNode: This handles SELECT clause.
- GetTableNode: This grabs a table. This is the leaf node.

Package.query_where_filters: handles filter logic in FromNode

- WhereFilter: This is the base filter class. ComparatorFilter and CompoundFilter extend this. The main method is the compare() function, which takes in the operands and operator and returns true or false.
- ComparatorFilter: This handles comparisons of =, <, > for all supported types.

- CompoundFilter: This handles multiple comparisons linked by AND or OR

Package.relation: the backbone of the database that handles internal storage.

- Table: a wrapper for a RedBlack Tree. This stores the attributes (table columns) and the rows. A table must have a key in order for a user to insert rows. Each table has a unique ID for internal use. Tables support writing to disk as *<tablename>.table* and loading from disk. The table maps between the key (String) and each row (ArrayList<Item>). Numeric types are padded with 0s and then converted to a string in order to preserve number ordering of keys.
- Attribute: These are the columns of the table. The attribute holds the name of the column, the datatype, the id of the table it belongs to and whether or not it can be nullable.
- Item: This wraps each data type and allows them to be inserted into an ArrayList to create each table row. It supports methods to seamlessly handle all types.
- TableGolem: a singleton class that manages all tables in the database. TableGolem has two RedBlack trees. One maps between a unique internal table ID and each table, and the other maps between table names and table IDs.

Unit Tests:

- A full battery of JUnit tests is provided.
- DBHelperTest: This tests self joins on the tables we were told to preload in class (i.e. table A with keys 1-100 and values 1-100).
- FunctionalTests: this provides holistic testing for the whole program by processing test SQL inputs.
- TableTest: this specifically tests the Table class, but also tests much of the relation package.
- IndexGolemTest: this specifically tests opening and writing indexes for a table.

Appendix

Potential Improvements:

Our database lacks several features. Our SELECT grammar is fairly limited and does not include many keywords. For example, support for the following could be added:

- *, Aggregate operators, IN, AS, ORDERBY, LIKE, DISTINCT, HAVING, EXCEPT, MINUS, NOT/EXISTS, NOT/UNIQUE, SOME, ANY, ALL, WITH.

Additionally, we lack support for nested SELECT clauses inside INSERT and CREATE TABLE statements. Another omission is the lack of foreign key support. Currently the user would have to manually traverse through the tables and enforce foreign key dependencies themselves.

Example Commands: see FunctionalTests to see them run and see expected outputs

- CREATE TABLE Relation (key int PRIMARY KEY, v1 int NULL, v2 int, v3 int, v4 double)
- INSERT INTO Relation VALUES (5, 6, 7, 8, 9)
- INSERT INTO Relation (Key, V2, V3, V4) VALUES (30, 31, 32, 33)
- SELECT attr2, attr1, attr3, attr4 FROM tbl1 JOIN tbl2
- SELECT attr2, attr1, attr3, attr4 FROM RESULT_TABLE9 WHERE attr2 = attr3
- SELECT attr2, attr1, attr3, attr4 FROM RESULT_TABLE9 WHERE ((attr2 < attr3) AND (attr4 > attr1))
- select attr2, attr1, attr3 from (select attr1, attr2, attr3, attr4 from (select attr4, attr3, attr2, attr1 from tbl1 join tbl2 WHERE attr3 < attr4)) WHERE (attr2 < attr3 AND attr4 > attr1) OR attr4 = 3 OR attr4 = 94
- select attr1, attr4, v1 from tbl1 join tbl2 join Relation
- DROP TABLE result_table42
- DROP TABLE IF EXISTS tabledoesnotexist
- DROP TABLE Result_table41, RESULT_table40, RESULT_TABLE39
- DROP TABLE IF EXISTS doesnotexist, RESULT_TABLE38, result_table37, notthisone, ResULT_TAbLe36
- CREATE INDEX ON tbl1.attr2
- DROP INDEX ON tbl1.attr2
- DELETE FROM Relation WHERE v2 < 20 or v3 > 25
- DELETE FROM Relation