

Mordor DB

# Implementation

## Description

MordorDB is a console application written in C# .  Data is stored on disk using JSON.  The most recently accessed data will be stored in memory. In memory, objects are defined recursively. Each collection can contain any number of key-value pairs. The keys will be strings, and the values can either be primitive values (such as strings or integers) or sub-collections.

## Logging

The log is a JSON object stored in a separate file. It is held in memory, and flushed to the disk every n operation, where n is configurable (these are the automatic checkpoints). The user can also force the log to be written to disk using a checkpoint keyword. The user can rollback to the last checkpoint using a rollback command.

## Indexes

Indexes are implemented using B+ trees. The user can create an index on any collection or sub-collection. If the user’s query only contains items that are in an index, that index will be used with referring back to the base collection.

# User Manual

## Getting Started

In order to execute MordorDB, simply click on the exe file found in the debug directory of the bin folder. All files related to the database will be stored in the directory where the executable is located so creating a new folder for organization is recommended but not required.

## Commands

The following commands are available from the prompt:

**print** –

Displays a formatted view of the current data stored in memory.

**checkpoint** –

Creates a checkpoint and forces the log to write to disk

**rollback** –

Rolls the data back to its state at the last checkpoint

**help** –

Displays a list of commands that are available

**clear** –

Clears the terminal window

**exit** –

Writes the current data to file and ends the program

**Query help** –

Displays a list of common query commands with examples included.

# Query Language

MordorDB includes the capabilities to create, retrieve, update and delete via method called from command line. Results can be filtered with a where command.  In addition it includes aggregate functions for sum, max, min, average and count. All queries must end with a semicolon.

### Select

In order to retrieve data from the database the user specifies the **select** command followed by a space. Then the collection name or names and then the field name are concatenated with periods.

To retrieve the first name of a student in the students collection with an id of 1 you would enter :

**select students.1.firstname;**

To retrieve the first name of all students you would enter**:**

**select students.\*.first;**

### Insert

To create new data in the database you use the **insert** command. You must format the new data as a Json object. You must also specify the collection into which it will be inserted as well as the key for the new data.

To add the first and last names of a student to the students collection you would enter:

**insert { “first” : “Joe”, “last” : “Schmoe” } into students.jschmoe;**

### Update

To change existing data, you use the **update** command.

To update the value of the first name field of the jschmoe item in the students collection to “Joel”, you would enter:

**update students.jschmoe.first value Joel**

To update the entire item you must enter a properly formatted json string.

**update students value {"first" : "Joel", "last" : "schmoe", "email" : "jschmoe@gmail.com"}**

### Delete

To remove an item from the collection you use the **delete** command.

To delete any items in the students collection where the first name is less than George you would enter:

**delete students.\* where students.\*.first < “George”**

### Where Clause

You may limit your results by using a where clause. The where clause supports the following operators: >, < , <=, >=, !=

To only retrieve the first name of students who have the last name schmoe you would enter:

**select students.first where students.last == “schmoe”;**