# CSE 510 Phase 2 Report

## Group Members

Pavan Chikkathimmegowda
Krima Doshi
Tanner Greenhagen
Jack Myers
Christian Seto
Shubham Verma

March 21, 2022

**Abstract**

This phase is an extension of Phase 1 on MiniBase file, database which stores data as tuples (individual records). Here, we will extend this definition of tuples to Quadruples (subject, predicate, object and confidence). This extension is done on RDF triples by introducing the concept of "confidence" to assert the confidence level of the resulted quadruple consisting of (subject, predicate, object, confidence). The provided packages like btree, buffermgr, catalog, iterator and heap also need to be modified, along with addition of several wrapper classes to work with Quadruple Files.

Keywords:*Database Management System Implementation, DBMS, Resource Description Framework, RDF, Probabilistic RDF, pRDF, buffer manager, iterator, quadruple, btree, stream, sort, heapfiles.*

# 1 Introduction

Quadruples are data types that store the IDs of Subject, Predicate and Object of fixed length along with the confidence value that tells the confidence level of the respective quadruple.

The subject, Predicate and object together are popularly known as an RDF. RDFs are commonly used in a object-oriented context to solve real world problems. They are used because of their human-friendly syntax, easy data integration and aggregation compatibility (using transitive rules), yes/no queries, etc.

A predicate defines the relationship between the subject and object, while the confidence denotes how much confidence this RDF triple has. It can be used in a probabilistic scenario to query triples.

## 1.1 Terminology

- **Quadruples** : They are data structures that store IDs for Subject, Predicate and Object of fixed length along with the confidence value.
- **Labels** : A label is a class that defines an ID and a String value pair to uniquely identify an element in the database. It is used when storing such string elements and creating btrees for them.
- **Entities** : They are real world objects that are distinguishable from other objects. In the quadruple use case, entities are generally the subjects and objects for which the relationship is defined.They have a String label and hence, can be stored with the label storage and access structure.
- **Predicates** : Predicates in the quadruple define the relationship between the two given entities and the confidence of that relationship. It can also be defined as the property of the entities given. They, like entities, are a String attribute and can similarly use the Label structure.
- **Confidence** : It is the value of percentage of tuples that satisfy a rule and a condition which is that fact asserted by that quadruple.

## 1.2 Goal description

The java minibase provided has a relational database management implementation for tuples. It cannot handle quadruples with the format that it is given in. The goal is primarily to create a working QuadDB based on a RDF triple-confidence pair, aka quadruples, which can insert records into files and query these records. The aim is also to create Indexes using Btrees and observe behavior of the the disk accesses made for inserts and queries for each index and order type specified.

## 1.3 Assumptions

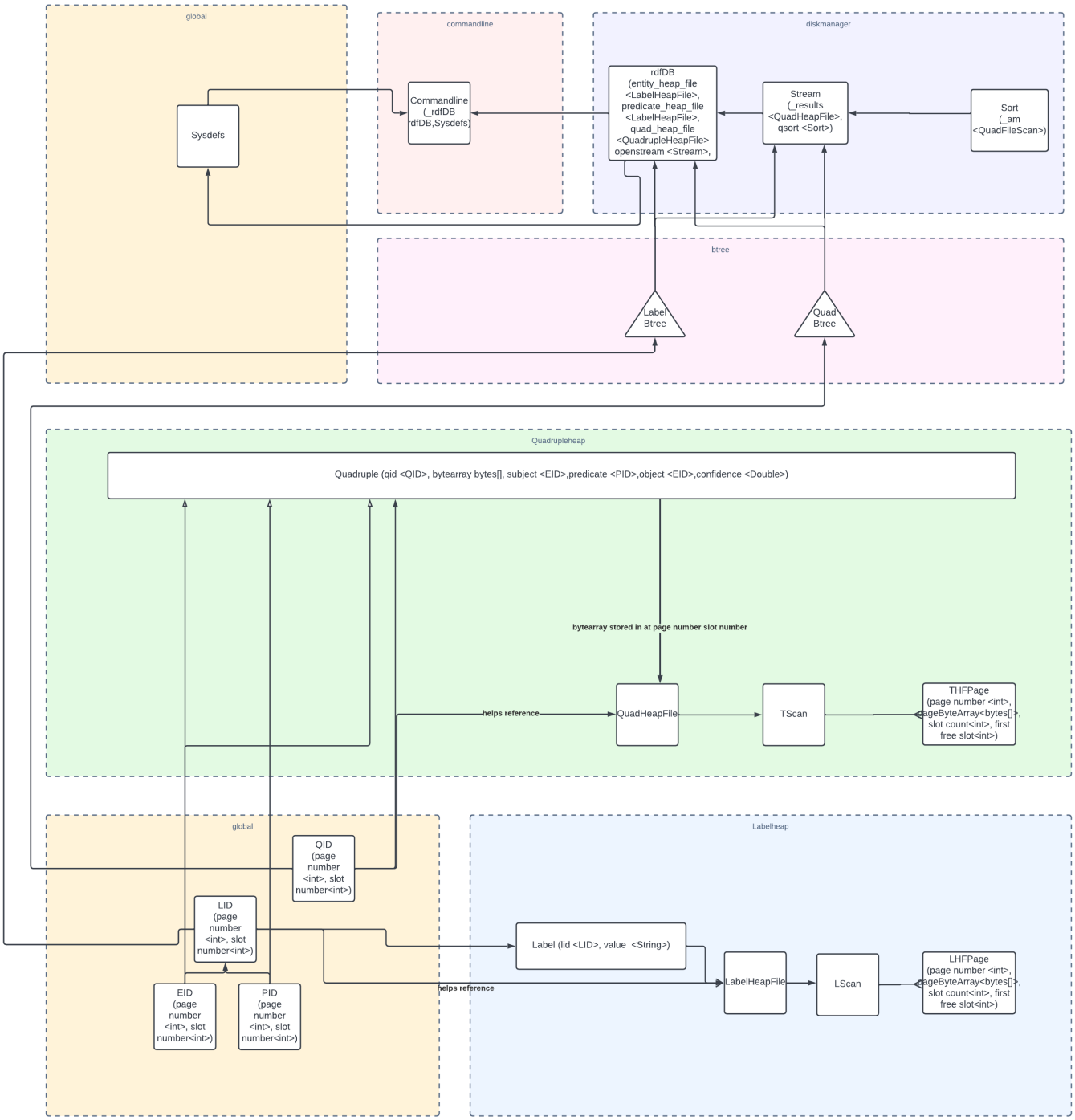
We make the following underlying assumptions for our database:

1. We assume that the user will always provide valid commandline input, with no errors or mistakes.
2. It is assumed that any input file will have a name without any spaces.
3. It is assumed that the subject, predicate, and object are all separated by colons. It is also assumed that the object and the confidence are separated by tabs.
4. The user knows the command input order and corresponding type in the command.
5. Any corrupt row is not entered into the database by the batchinsert command.
6. The buffer number input is for the overall implementation and the number of buffers used for sort is fixed.

## 2 Description of the Proposed Solution/Implementation

This is the Phase 2 of CSE 510 : Database Management Systems Implementation project. We implemented Quadruples consisting (subject, predicate, object, confidence) which are of fixed length by extending and modifying several packages of MiniBase which originally stores data in the form of Tuples having variable length. Every quadruple object consists of four elements : Subject, Object, Predicate and Confidence and then they are stored in the database as an array of bytes in a Heap File. Subjects and Objects are of class "entity" and Predicates are of class "predicate". Entities and Predicate values are stored in the class Label and they both have an ID associated with them which provides the slot number and page number where the value is stored (in the pages of Heap Files).

Wrapper classes are built on top of these Heap Files and Quadruples to access, modify and delete these tuples. The rdfDB class is used to insert and delete stored quadruples. The Stream class is used to query for Quadruples using the filters, index and sort order specified. The sort class consists of Heap sort for the first run and then it uses Merge Sort for the remaining runs. The architecture workflow of the proposed solution is given in the figure below:




## 2.1 Global

The global directory contains classes which are used throughout the database in various other components.

### 2.1.1 QID

The QID contains the QID of the quadruple which contains information used to access records in the QuadHeapFile. It consists of the page number and slot number which are used by the the Scan to get the page of the stored information and the address where it is stored on that page. It also has methods to allocate

a new page number and slot number when creating a new QID, copy the given QID, write the quadruple byte array (32 bytes) to the position specified by slot number on the given page array and compare two qids.

#### 2.1.2 LID

This class is created for labels. It stores the LIDS with the page and slot number for Labels. It performs the same functionalities as QID but with reference to Labels. This class also has a function to return the corresponding EID or PID.

#### 2.1.3 PID

The PID class is created for page access for predicates and is similar to QID. In addition, it can create a new PID with the given LID and return a new LID corresponding to the same page number and slot number as it.

#### 2.1.4 EID

The EID class is created for page access for enitities and works the same as PIDs, just the values in it are entities.

#### 2.1.5 SystemDefs

This file consists of system definitions used throughout the project. Currently, it has variables to access the rdfDB and set buffer pages for the minibase. It has to be initialised before running any command as it is responsible to allocate pages and buffer pages to execute the query for the system.

#### 2.1.6 Design Decision:

- We are fixing the buffer size to 1000 at the time of batchInsert
- We expect the user to input a sufficiently large number of buffers corresponding to a large data size.

### 2.2 QuadrupleHeap

Quadruple consists of pointers to Entity, Predicate, Object and contains value of the quadruple. This class implements all the methods of quadruple described in the project phase description.

#### 2.2.1 QuadrupleHeapfile

QuadrupleHeapfile contains an set of Quadruple records. This class supports insert quadruple into page, delete quadruple, get quadruple using quadruple ID. Quadruple heapfile contains directory which is of type HFPage. Each entry in HFPage is stored as Tuple. Tuples is are linked to each other with double linked list data structure.

#### 2.2.2 QuadrupleUtils

This class contains compareQuadrupleWithQuadruple method compare two quadruples with a given fields (which could be some combination subject, predicate, object, confidence) and return an integer. The sorting order is described above and is represented by an array. Quadruples attributes are compared based on the sorting order until attributes are different or all selected attributes have been compared. Method equals checks whether two Quadruples are equal based on all of their attributes returns a boolean. Method setValue sets field from one quadruple to another quadruple. This is used by the Sort class to set the values of a dummy Quadruple to the values of another quadruple. It is constantly updated to the values of the previously returned tuple (next tuple to be returned in terms of either Ascending or Descending sorting order). There is also another version of setValue that is for initializing the dummy Quadruple to either have initial attributes greater than every other quadruple or attributes less than every other quadruple to allow for returning elements starting from either the smallest or largest quadruple depending on the sorting order.

## 2.3 LabelHeap

Label consists of a String, stored as a byte array. This class implements all of the methods of Label as described by the phase description.

### 2.3.1 LabelHeapFile

The LabelHeapFile class contains methods to initialize and delete a heap file for the label class. The main difference between this and the HeapFile class is that it is designed to work with LIDs and Labels rather than RIDs and Tuples. Methods are also included to insert, delete, update, and get Labels from the file. it can also initiate a scan on itself and get the number of labels it is storing.

### 2.3.2 LabelUtils

This class has the compareLabelWithLabel method, which compares two labels and returns if they are the same or if one of them is greater than the other. Equal will return if the labels are equal or if they are not equal. Value will return the Label's string, offset, or length value. SetValue will set the Label to a specific value.

## 2.4 Sort

The major change from the original Sort class was just to change any data access from working with tuples of arbitrary size and attribute types to quadruples with fixed lengths and attribute types. This mainly involved removing setting the tuple size and number of attributes to dealing with the 32 byte long quadruple that had four fixed attributes. Beyond changing from tuples to quadruples the big change was allowing for sorting to occur based on multiple attributes rather than just one. For example if the desired order was option 1, we would have to sort based on subject first, if subjects were then equal we would sort the equal subjects by predicate, if the subjects and predicates were equal we would sort by object (technically we would sort on confidence if the subject, predicate, and object were equal, but since quadruples can not have the same subject, predicate, and object this would never occur). In the sort class an order was passed that gets represented by an array. For example if the sorting order was one, and array with values 1,2,3,4 would be used for sorting where 1 represents subject, 2 represents predicate, 3 represents object, and 4 represents confidence. The more left it is the more major the attribute is for sorting. If we were sorting based off just subject and confidence then it would be 1,4-1,-1 where -1 means there are no additionally sorting attributes so if two quadruples are equal up till then they are considered equal for sorting purposes. It was also decided that the case of the strings will impact the order. Capital letters are considered to be before lowercase letters if the sorting is being performed in ascending order. We decided that it could be desirable for the user to be able to try and get some results to appear earlier than other attributes to show importance. A way to do this in our system is to capitalize the string values.

## 2.5 btree

The pre-implemented btree package was working with tuples to create btree data structures to index all the records into heap files using RIDs. Rather than alter any of these existing files, we chose to add files that will have very similar functionality but will work with Quadruples and Labels rather than Tuples.

### 2.5.1 Quadruple btree files

The added Quadruple btree files are designed to index the QuadrupleHeapFiles by using their associated QIDs (Quadruple IDs). In total, nine classes were created for the Quadruples. These classes are QuadBT, QuadBTFileScan, QuadBTIndex, QuadBTLeafPage, QuadBTreeFile, QuadBTreeHeaderPage, QuadBTSortedPage, QuadIndexFile, and QuadLeafData. The core functionality of these files is the same as that of the provided nine, except that these will work with QIDs and Quadruples rather than RIDs and Tuples.

#### 2.5.2 Quadruple btree Index Options

There are five different indexing options that the user can choose to use. This index will be the key that they btree will use to index its entries by. The five index options are:

1. Object
2. Predicate
3. Subject
4. Object + Predicate
5. Predicate + Subject

#### 2.5.3 Label btree files

The modified Label btree files are similar to the Quad btree files, except they leverage LIDs and Labels rather than QIDs and Quadruples. Nine classes were also created, and they are LabelBT, LabelBTFileScan, LabelBTIndex, LabelBTLeafPage, LabelBTreeFile, LabelBTreeHeaderPage, LabelBTSortedPage, LabelIndexFile, and LabelLeafData. Again, the core functionality of these classes was not changed, they are just slightly adapted to be able to work with Labels.

#### 2.5.4 Design Decision:

- For the LabelBTree, the BTree is indexed on the String value of the Label
- For the QuadBTree, we are indexing on one of the 5 index options selected
- The confidence is a column we did not consider in the index as it has more possibility to be updated, due to which case the Index would have to be modified every time. So in order to decrease the number of disk accesses at insertion, we consider the other three column over confidence.

### 2.6 rdfDB

The rdfDB class is responsible for creating and maintaining all of the relevant files to the database. The first two files are the QuadrupleHeapFile and the QuadrupleBTreeFile, which are used to store the Quadruples and iterate through them. There are also the Entity LabelHeapFile and Entity LabelBTreeFile, which are used to store all of the entities that are encounted. Subjects and Objects from the Quadruple are considered Entities and stored here. Lastly, there are the Predicate LabelHeapFile and Predicate LabelBTreeFile to store the Predicates from the Quadruples. From these files, the rdfDB has methods to return the number of Quadruples, Entities, and Predicates in the database. It can also count the number of distinct subjects and objects that are in the database. The rdfDB has methods to insert and delete Entities, Predicates, and Quadruples into there respective files. It also can open an instance of a Stream class on its files.

#### 2.6.1 Design Decision:

The report contains the following components:

- We have created the rdfDB to be singleton, so only one instance of the object would be created in the run and all other places would reference that same instance.

### 2.7 Stream

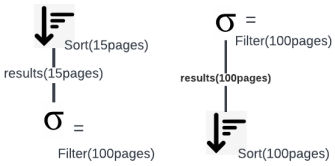
The Stream class is responsible to feed in quadruples back to the rdfDB after querying for them one by one. The commandline parses the input query calls the openstream function of rdfDB. The openstream function creates a new stream object and forwards the filter and order value to the Stream object.

The Stream class, first, detects the '*' filters and sets the nullFilter(no filter) value of that particular attribute to true and then sets the string filter values of the attributes. After setting the values of the filters, it checks if the index is valid for the current filter, i.e. the nullFilter is not set. If it is, it performs a normal full BTree scan, else it uses the predefined indexed BTree scan. In the scan, it gets the EIDs and PIDs from their corresponding FileScans and sets the bytevalue gained from them. If any of them(subject entity or object entity or predicate) are not found, it returns a not found message. If found it queries for them and applies the filters. For each match, it appends the quadruple to a QuadrupleHeapFile. It returns the results. If the results are not null, it sorts them.

After the querying of the Stream class, the results are streamed using the getNext function. At the end, the stream needs to be closed.

### 2.7.1 Design Decision:

- Filter the results first before sorting them. Filtering is a scan operation on the entire BTree to filter out the matching records and return the results. Filtering the records would decrease the amount of records which need to be sorted, which is expensive.As opposed to sorting on all the records and then filtering them out. We can gain more insight about it from the following figure.




## 2.8 Command Line Interface

The CommandLine class is the entry point for the user to perform operations on the database. It is responsible for handling user input and performing the program. It is in charge of running and setting up the system to work with the batchinsert, query, and report programs. It checks that the entered command actually corresponds to one of the programs and that it has the correct number of options associated with it. It also writes the batchinsert and query commands with their options to a log file that can be viewed with the report command.

### 2.8.1 Design Decision:

The report contains the following components:

- We assume the user to input the command in the correct format and type for it to work, otherwise it would throw an error and end the current run.
- The number of buffers entered by the user is sufficiently large to be able to execute the query.

### 2.8.2 batchinsert

The batchinsert program takes three arguments: DATAFILENAME, INDEXOPTION, and RDFDB-NAME. The program will read in all of input from the data file, and will insert it into the specified rdfDB using the provided index option. It will perform this by creating SystemDefs instance, and then opening an rdfDB within that instance. All of the data will be parsed, and then inserted as a Quadruple into the database. The name of the file that ends up representing the database is represented by RDFDB-NAME_INDEXOPTION. The SystemDefs we end up using are 1000 pages and a buffer pool size of 100. These numbers seem to accommodate large datasets and allows for read and write page amounts to differ heavily based on the data size. This is important when we start comparing index options. The subject,

predicate, and object are split apart by having a colon between them. The object and confidence are split apart by having a tab between them. If there are any fields with no values, the row is discarded as we believe every quadruple should have a defined subject, predicate, object, and confidence. We also do accept confidence values of over 1 as we assume the user is providing correct input as long as the field has a value. We then take the three label fields (subject, predicate, object) and insert them into the database via the insertEntity and insertPredicate methods. These methods also return their respective EIDs and PID which are really just used to store the page number and slot number of where the strings are stored in the entity and predicate heap files. During this process the labels are added to the corresponding LabelHeapFiles. We then convert the page ids and slot numbers of the three ids into bytes. Each page number and slot number can be represented by four bytes each for a total of 8 bytes per id. The confidence is then represented by 8 bytes as well for a total of 32 bytes. We then insert these 32 bytes into the quadruple heap file via the insertQuadruple method of the rdfDB. The buffer manager then has all of its pages flushed after the operation is completed to allow for another command to be run with a clean set of buffer frames.

#### 2.8.3 Design Decision:

The report contains the following components:

- Corrupt/incorrect records are not inserted into the database.

#### 2.8.4 query

The query program takes eight arguments: RDFDBNAME, INDEXOPTION, ORDER, SUBJECTFILTER, PREDICATEFILTER, OBJECTFILTER, CONFIDENCEFILTER, and NUMBUF. It will open the rdfDB instance with the provided rdfDB name and index option. It will return all Quadruples that have attributes equal to each of the the filters. The exception is CONFIDENCEFILTER, which is a minimum confidence. All returned quadruples will have a confidence greater or equal to this filter. NUMBUF is the number of buffers to be used and is set via the SystemDefs declaration. It has been found that to really guarantee that execution works properly there must be a minimum of 1000 frames allocated to make it through the sorting process properly. Otherwise the sorting will likely fail. A stream object is then retrieved from the rdfDB instance that took the filters as input. By iterating through the stream object we get all of the quadruple ids for the quadruples in the specified sorting order. We then give the QID to the rdfDB that then gets all of the label strings from the three sets if 8 bytes that each contain a page number and slot number where the strings are stored. A string is then printed that has the original quadruple values for the user's understanding. The buffer manager again has its pages flushed to allow for another command to be run with a clean set of buffer frames. The stream object is also closed to delete the sort object and temporary QuadrupleHeapfile that stored the query results.

#### 2.8.5 report

The report program outputs various statistics for a provided database file name. This includes the counts for each of the number of labels, subjects, predicates, objects, and quadruples contained within the database. These numbers can be used to judge whether the insert options are correctly eliminated duplicate quadruples and only inserting each label type once in the heap files. The program also prints the contents of the logfile to the console. The contents include all of the batchinsert and query commands along with the read and write values for each of the commands. The execution time is also included, but it varies greatly even between running the same command so it is not a very reliable measure.

#### 2.8.6 Design Decision:

The report contains the following components:

- The logs of the disk reads, writes and execution of the past commands used so as to be able to track back and compare the past queries.
- The count of each subject, predicate and object in the database so as to get a better idea of the data stored in the database.

#### 2.8.7 PCounter

This class is responsible for counting the number of reads and writes that are performed by each operation on the database. These counts are displayed after the operation is completed, and are logged as well. The values are actually cleaned after each batchinsert and each query. The reads and writes are incremented within the read_page and write_page methods of the DB class which rdfDB extends from.

## 3 Interface Specifications

To run the actual code outside of an IDE a jar must be created. In our case we used the Gradle build system (version included in System Requirements) to package our jar. If you navigate into the CSE510_Project directory, you can then run the command ./gradlew build. This will actually create the jar which can then be found in the build/libs directory named CSE_510_Project.jar. To run the jar, assuming you are in the build/libs directory, can be done via the command java -jar CSE_510_Project.jar.
Once the jar is running there are really four commands (items in all capital letters need to be replaced with actual values when running the commands):

1. batchinsert DATAFILENAME INDEXOPTION RDFDBNAME
2. query RDFDBNAME INDEXOPTION ORDER SUBJECTFILTER PREDICATEFILTER OBJECTFILTER CONFIDENCEFILTER NUMBUF
3. report FILENAME
4. exit

The INDEXOPTION, ORDER, and NUMBUF should be whole numbers.

The INDEXOPTION should be values from 1 through 5 with their meaning being outlined below:

1. Object
2. Predicate
3. Subject
4. Object + Predicate
5. Predicate + Subject

The ORDER value should be from 1 through 6 denoting the sorted order of the returned quadruples. The quadruples are returned in an ascending order:

1. subject, predicate, object, confidence
2. predicate, subject, object, confidence
3. subject, confidence
4. predicate, confidence
5. object, confidence
6. confidence

The NUMBUF value should be kept at values that are greater than 1000 to actually allow for the sorting to be able to accommodate the required page allocations.

The CONFIDENCEFILTER should be integer values where everything returned will be greater than or less than that value. There are no quotes required around any of the strings and there is also no spaces

accepted in any of the values. Any space when entering the commands is expected to denote a separate parameter.

In report, the FILENAME is just the RDFDBNAME_INDEXOPTION of a previously established in a batchinsert.

Also the RDFDBNAME in query is also RDFDBNAME_INDEXOPTION established in batch insert. The ending value of the RDFDBNAME in query should match the INDEXOPTION provided in query or the system will fail.

The exit command is then used to actually stop running the program. It is important to note that the database files to persist exiting the program so exiting on accident should not actually cause any issues.

# 4 System Requirements

## 4.1 Required Tools/Software

1. Java 8 or greater : Lower versions of Java is not compatible with Gradle which affects automation in the project.
2. JDK 1.8 or Greater : JDK version less than 1.8 would not support Java 8 or above which can cause the project to not run properly.
3. Gradle 7.4.1 : Helps in fetching required libraries and plugins for Java projects automatically. Gradle version 7.4.1 is compatible with Java 8 which is used in this project.
4. Minibase : Minibase folder provided to us in Phase 1 has been modified to work on quadruples. The tar file is unzipped and then opened in any of the IDEs. We have used VS Code for our build.
5. Linux : The project will work only in a linux system or in Windows with WSL (Windows Sub-system for Linux).

## 4.2 Execution Instructions

1. Go to the project home directory "CSE510_Project"
2. Run the command "./gradlew build". This will produce the artifact "build/libs/CSE_510_Project.jar"
3. Run the command "java -jar build/libs/CSE_510_Project.jar" to execute the application.
4. Enter the desired command from 3 for the application to perform.

# 5 Build System

## 5.1 Gradle

We replaced the Makefile build system with Gradle. Gradle is the modern build system for Java programming language which is the language the Minibase base project is written in. The advantage of having Gradle build is that it turns our build step into just one command i.e `./gradlew build`. Anytime we add a new package into the code base we don't need to make changes to makefiles.

## 5.2 Continuous Integration

We also have continuous integration built into the Github repository. Anytime we raise a pull request, continuous integration will automatically run the build step and verify for any compilation errors. It also produces the jar file as the artifact.

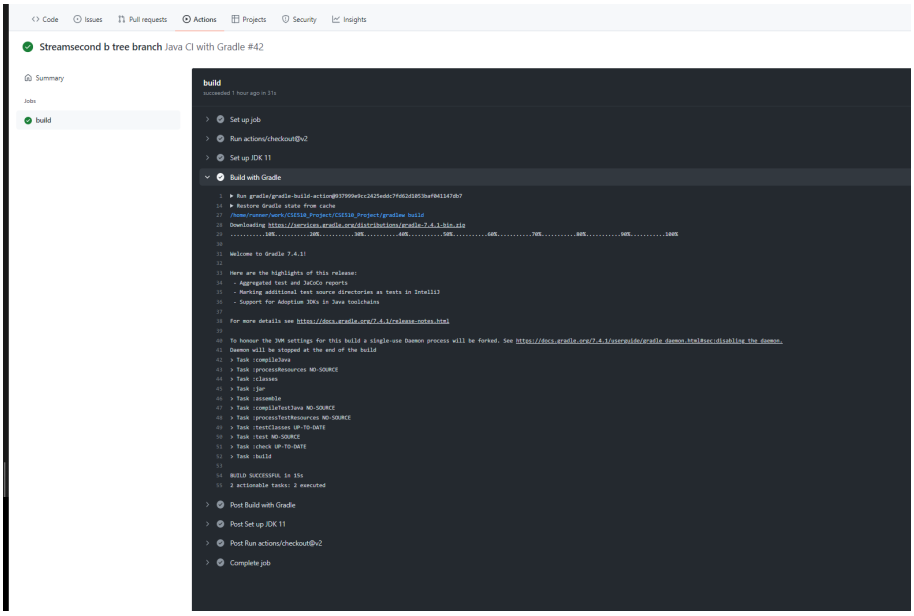



Figure 1: Continuous Integration

# 6 Related Work

In phase two of the project, we were asked to create a modified version of an RDF. In particular, we were tasked to add a confidence value to each RDF triple such that it became a quadruple, changing the data from `(subject, predicate, object)` to `(subject, predicate, object, confidence)`. As a result, some related work associated with RDF includes the original W3C documentation for RDF [2], which outlined the specifications for a data framework that made computer to computer interaction with the internet easier. Dr. Eric Miller's article titled "An Introduction to the Resource Description Framework" [3] describes the background and general details of RDF and why it is important for the internet. Miller summarizes RDF as "an application of XML that imposes needed structural constraints to provide unambiguous methods of expressing semantics for the consistent encoding, exchange, and machine processing of standardized metadata" [3]. Dr. K. Selçuk Candan et al's article titled "Resource Description Framework: Metadata and Its Applications" [1] goes on to describe the past, present, and future of RDF and the semantic web. Specifically, the authors describe some of the precursors of RDF such as the `<META>` tag in HTML; how RDF works at the present; current RDF tools; and future challenges for RDF and the semantic web including security, ease of use, and compatibility [1].

The extended RDF with quadruples is a similar to Probabilistic RDF, or pRDF. pRDF adds probabilities to the traditional RDF which adds uncertainty to the data and allows for nuanced querying. Dr. Octavian Udrea et al developed pRDF, outlining the syntax, theory, and query algorithms in their article titled "Probabilistic RDF" [4].

# 7 Conclusions

## 7.1 Database Metrics

### 7.1.1 Batch Insert and Report Statistics

After integrating and testing our database, we collected statistics for various interface aspects. The number of pages we set was 1000, and the number of buffers we used was 100 when performing `batchinsert`. For `batchinsert`, figure 2 shows the number of reads and writes to the database. Of particular note is that index type 5 (predicate/subject) had the least number of reads and writes, with 11,1844 reads and 7,049 writes. After performing `batchinsert`, the `report` method returned the number of labels, subjects,

predicates, objects, and quadruples. These values should be the same across all the index types, as seen in figure 3. However, index type 2 (predicate) returned 3,014 quadruples instead of 3,013 quadruples. We are uncertain of this result. Then, we performed two different query combinations: one focused on index vs. order type with no filters and another focused on some basic filters.

#### 7.1.2 Index vs. Order Type Query Statistics

We performed the index vs. order type with no filters queries for a buffer size of 1000 (see figure 4) and 100 (see figure 5). The cells highlighted yellow are results which gave the appropriate result but gave an error at the end. This did not inhibit the command line interface process (i.e., the user could still input another command). Of particular note for these result is how the order type did not significantly change the number of reads and writes for any given index (excluding some of the yellow results). This illustrates two facts:

1. The fixed size of our entries (in this project was quadruples) resulted in predictable queries.
2. The quadruple B-tree also helped in making queries more predictable.

Another important result in these statistics is that buffer size has a significant effect on the read/write statistics (100 buffer queries had more than 2 orders of magnitude reads/writes compared to 1000 buffer queries). More testing may be done in the future to plot the buffer size versus the number of reads/writes.

#### 7.1.3 Filter Query Statistics

Since order type did not have an impact on our index type results, we decided to fix order type to 1 and run various filter queries. In addition, we decided to run at 1000 buffers. For basic filter queries, we decided to filter on the largest number of entries for subject, predicate, and object. These are as follows:

1. **Subject:** crschmidt (100 instances)
2. **Predicate:** name/type (494 instances)
3. **Object:** Person (494 instances)

Our reasoning for this is that these results would potentially give the worst case scenario in terms of reads and writes. In addition, we performed queries on four different values of confidence: 0.2, 0.4, 0.6, and 0.8. Finally, we performed combination filters, specifically:

1. **Subject+Predicate (SP):** crschmidt+name
2. **Subject+Object (SO):** crschmidt+Person
3. **Predicate+Object (PO):** type+Person
4. **Subject+Predicate+Object (SPO):** crschmidt+type+Person

The read/write statistics for these filter queries can be found in figure 6. Similar to the previous section, the cells highlighted yellow are results which gave the appropriate result but gave an error at the end.

There are several interesting results in the filter queries. One is that the combination index types that were close to the combination filter query performed very well. For example, index type 5 is based on a subject/predicate key, and the subject and predicate combination filter query only resulted in 87 reads and 7 writes. This is in contrast to index type 4 (i.e., subject/object key) on the same query, which required 297 reads and 158 writes. It should be noted that this query only had one result. Another interesting result was the fact the the more specific the query became, the less number of reads and writes were required, with the subject/predicate/object query producing the smallest reads and writes across all index types. Further, the confidence filters did not result in smaller reads and writes across each index. This is because the scan still had to be performed on every single confidence value to determine if it was greater than or equal to the given filter.

# References


[1] K Selçuk Candan, Huan Liu, and Reshma Suvarna. "Resource description framework: metadata and its applications". In: *Acm Sigkdd Explorations Newsletter* 3.1 (2001), pp. 6–19.

[2] Ora Lassila, Ralph R Swick, et al. "Resource description framework (RDF) model and syntax specification". In: (1998).

[3] Eric Miller. "An introduction to the resource description framework." In: *D-lib Magazine* (1998).

[4] Octavian Udrea, VS Subrahmanian, and Zoran Majkic. "Probabilistic rdf". In: *2006 IEEE International Conference on Information Reuse & Integration*. IEEE. 2006, pp. 172–177.


# A Roles of Group Members

1. Pavan Chikkathimmegowda
   a) Created build setup using gradle replacing Makefile to make everyone more productive
   b) Created continuous integration setup on Github to validate sanity of PRs
   c) Integrated individual components to get code to compile fixing 100+ compilation errors
   d) Created IDE setup to aid in debugging
   e) Implemented `Quadruple` class
   f) Implemented `QuadrupleHeapFile` class
   g) Implemented `QuadrupleUtils` class
2. Krima Doshi
   a) Created the `QID`, `EID`, `PID` and `LID` classes
   b) Developed the code for the `Stream` class the database on top of the rdfDB, scanning and filtering records and returning the results in a sorted order
   c) Created `QuadrupleOrder` as per the order requirements specified
   d) Created the design of the integrated architecture of the Quadruple minibase and noted the design decisions for the report.
   e) Participated in meeting discussions, integration and debugging sessions
3. Tanner Greenhagen
   a) Adjusted the Sort Class and helper classes to allow for Quadruple Sorting
   b) Setup the CommandLine Class to take in user input and run appropriate rdfDB functions
4. Jack Myers
   a) Modified the `HeapFile` class to create `LabelHeapFile`
   b) Refactored Shubham's work on the `btree` package to work the `Quadruple` class
   c) Modified several classes from the `btree` package to work with the `Label` class
   d) Designed and implemented the `rdfDB` class
   e) Participated in meeting discussions
5. Christian Seto
   a) Copied and modified the `Tuple` class into `Label`
   b) Copied and modified the `HFPage` class into `LHFPage`
   c) Copied and modified the `Scan` class into `LScan`

d) Copied and modified the `iterator.TupleUtils` class into `iterator.LabelUtils`

e) Made minor changes to rdfDB

f) Performed scheduling of meetings and participated in meeting discussions

g) Wrote the "Related Works" portion of the report

h) Worked on integration of entire system

6. Shubham Verma

a) Copied and modified the `btree` package to work with quadruples

b) Copied and modified `catalog`, `chainexception`, `global`, `index` and `iterator` packages to work with `btree` package

c) Wrote "Abstract", "Introduction" and "System Requirements" part in Report

d) Worked with Krima to create the design of integrated architecture of the Quadruple minibase for the report.

e) Participated in meeting discussions

# B Database Statistics

| | Read/Write | |
|---|---|---|
| Index Type | Read | Write |
| 1 | 253632 | 11406 |
| 2 | 744152 | 21453 |
| 3 | 381098 | 12198 |
| 4 | 232821 | 11202 |
| 5 | 111844 | 7049 |

Figure 2: Batch insert read/write statistics

| | TestDB_1 | TestDB_2 | TestDB_3 | TestDB_4 | TestDB_5 |
|---|---|---|---|---|---|
| **Label** | 1889 | 1889 | 1889 | 1889 | 1889 |
| **Subject** | 441 | 441 | 441 | 441 | 441 |
| **Predicate** | 63 | 63 | 63 | 63 | 63 |
| **Object** | 1875 | 1875 | 1875 | 1875 | 1875 |
| **Quadruple** | 3013 | 3014 | 3013 | 3013 | 3013 |

Figure 3: Record statistics for each index type

| Reads | Order Type | | | | | |
|---|---|---|---|---|---|---|
| **Index Type** | **1** | **2** | **3** | **4** | **5** | **6** |
| **1** | 287 | 288 | 288 | 288 | 288 | 288 |
| **2** | 267 | 268 | 268 | 268 | 268 | 268 |
| **3** | 279 | 279 | 280 | 280 | 280 | 280 |
| **4** | 312 | 313 | 313 | 313 | 313 | 313 |
| **5** | 298 | 299 | 299 | 299 | 299 | 299 |

| Writes | Order Type | | | | | |
|---|---|---|---|---|---|---|
| **Index Type** | **1** | **2** | **3** | **4** | **5** | **6** |
| **1** | 307 | 132 | 132 | 132 | 131 | 132 |
| **2** | 112 | 111 | 112 | 111 | 112 | 112 |
| **3** | 123 | 124 | 123 | 124 | 124 | 124 |
| **4** | 332 | 157 | 157 | 157 | 156 | 157 |
| **5** | 270 | 142 | 143 | 143 | 143 | 143 |

Figure 4: Index vs. order type read/write statistics (buffer size of 1000)

| Reads | Order Type | | | | | |
|---|---|---|---|---|---|---|
| Index Type | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 48725 | 48719 | 48699 | 48673 | 48702 | 48667 |
| 2 | 39983 | 39981 | 39979 | 39959 | 39992 | 39964 |
| 3 | 46705 | 46700 | 46627 | 46641 | 46670 | 46636 |
| 4 | 46851 | 46843 | 46846 | 46798 | 46850 | 46796 |
| 5 | 49846 | 49810 | 49868 | 49780 | 49803 | 49791 |

| Writes | Order Type | | | | | |
|---|---|---|---|---|---|---|
| Index Type | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 1256 | 1257 | 1257 | 1255 | 1239 | 1253 |
| 2 | 1101 | 1092 | 1102 | 1088 | 1104 | 1099 |
| 3 | 1223 | 1244 | 1222 | 1240 | 1242 | 1239 |
| 4 | 1231 | 1232 | 1231 | 1229 | 1214 | 1229 |
| 5 | 1291 | 1275 | 1290 | 1281 | 1294 | 1292 |

Figure 5: Index vs. order type read/write statistics (buffer size of 100)

| **Reads** | **Query** | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | **Confidence Filter** | | | | | | | | |
| **Index Type** | **1 Subject Filter** | **1 Predicate Filter** | **1 Object Filter** | **0.2** | **0.4** | **0.6** | **0.8** | **SP** | **SO** | **PO** | **SPO** |
| **1** | 272 | 289 | 183 | 287 | 288 | 288 | 288 | 272 | 158 | 186 | 160 |
| **2** | 253 | 180 | 270 | 267 | 268 | 268 | 268 | 163 | 245 | 184 | 158 |
| **3** | 113 | 281 | 282 | 279 | 279 | 279 | 279 | 112 | 105 | 284 | 107 |
| **4** | 297 | 314 | 315 | 312 | 313 | 313 | 313 | 297 | 290 | 190 | 164 |
| **5** | 283 | 300 | 301 | 298 | 298 | 298 | 298 | 87 | 275 | 303 | 60 |

| **Writes** | **Query** | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | **Confidence Filter** | | | | | | | | |
| **Index Type** | **1 Subject Filter** | **1 Predicate Filter** | **1 Object Filter** | **0.2** | **0.4** | **0.6** | **0.8** | **SP** | **SO** | **PO** | **SPO** |
| **1** | 138 | 132 | 24 | 132 | 131 | 131 | 131 | 133 | 25 | 25 | 26 |
| **2** | 118 | 22 | 112 | 112 | 111 | 111 | 111 | 23 | 113 | 23 | 24 |
| **3** | 17 | 124 | 124 | 123 | 123 | 123 | 123 | 12 | 12 | 125 | 13 |
| **4** | 163 | 157 | 157 | 157 | 156 | 156 | 156 | 158 | 158 | 29 | 30 |
| **5** | 149 | 143 | 143 | 237 | 142 | 142 | 142 | 7 | 144 | 144 | 8 |

Figure 6: Query filter read/write statistics (order type of 1, buffer size of 1000)