Phase II: Implementation of RDF DBMS

Group 19 Members

Austin Gilmore

Zuwei Guo

Parker Hoang

Ryan Hoang

Adam Omais

Abhash Shrestha

Abstract

The purpose of this project is to implement an RDF DBMS using the modules of MiniBase as building blocks. Compared to traditional DBMS, an Resource Description Framework (RDF) DBMS is a framework that stores the information about Web resources in a detailed and structured manner. To accomplish this goal the RDF databases store the data in the form of triples instead of tuples. In this project we extend the definition of triples to quadruples where ‘confidence’ is added to the triples. We further modify the MiniBase labeling and indexing mechanisms so that it supports the storing and retrieving of RDF data.

Keywords:

Resource Description Framework (RDF), Quadruple Data, DBMS, MiniBase

I. Introduction

Phase 2 of this project consisted of implementing an RDF database using minibase with a modification to the data tuples in minibase to include the fields subject, object, predicate, and confidence to create a quadruple. To implement this modification, changes to different modules of minibase were required including the heapfile, btree, disk manager, stream, and iterator. In minibase, data tuples are stored as tables inside of heapfiles and are modified for the RDF database by storing quadruples and labels in separate heap files instead of tuples. The btree is a data structure used to index the data tuples stored in the heapfiles and in the RDF database implemented instead indexes the newly created quadruples and labels from their respective heap files. Disk manager is a module with many different functions to manage the files for the heapfile and btree and in the conversion to an RDF database required changes to functions to accommodate for the quadruple heapfile, label heapfile, and btree changes. The stream streams the quadruples based on a set of filters on the subject, predicate, object, and confidence and an indicated order. The iterator is used to sort tuples and is modified to instead sort quadruples in a given field as well as in all fields. A batchinsert program was then created to store quadruples into the database according to an index with 5 different indexing schemes implemented.

II. Description of the Proposed Solution / Implementation

The implementation of quadruples to replace tuples required modifications to a large number of classes and functions that were part of different modules in minibase. We modified the heapfiles to accommodate the change from tuples to quadruples starting with creation of ID classes for quadruples (QID), labels (LID), entities (EID), and predicates(PID) and tuple was modified into quadruple with a fixed length byte array of 28 bytes and the 4 fields subject, object, predicate, and value (confidence). Within the byte array, subject, predicate, and object are stored as pointers using slotNo and pageNo. We created overloaded constructors for quadruple, including one which passed in the the size parameter included in the old tuple constructor so that modifications for the use of quadruples with a size of 12 bytes in DatePageInfo could be made. In this way, we eliminated the use of tuples altogether. Related functions for the new quadruple class such as get and set methods for the quadruple fields were also modified. Set methods set the respective class field and also write into the byte array. Other quadruple functions include a function to print out the quadruple, return the length of the quadruple, copy the quadruple, an init method to initialize a quadruple, and a method to set a quadruple given a byte array and offset. The heap package was modified to quadrupleheap to store quadruples in the quadruple heap file rather than tuples. This included changes to the methods for insertion and deletion of quadruples into the heapfile, methods to return the number of quadruples in the heapfile, a method for performing a sequential scan of the heapfile, and a method for updating a specific quadruple in the heap file. Also created was the labelheap package which modified heap to store a newly created class, labels, into label heapfiles. This

labelheap stores the quadruple's fields as labels into a new heapfile separate from the quadruples.

Next, the btree package which previously used the btree data structure to index records in the heapfiles was modified according to the changes to the heapfiles to now index by quadruples and labels using quadruple ID (QID) and label ID (LID) rather than record ID (RID).

In the disk manager module, modifying to an RDF database required changes to many functions to manage the new files for label heapfiles, quadruple heapfiles, and btree index files. An rdfDB class was created by modifying the original DB class where each RDF database contains a heap file to store quadruples and two label heapfiles to store entity labels and subject labels. Functions related to this change included get methods for retrieving the number of quadruples, entities, subjects, predicates, and objects, and insertion and deletion functions for quadruples, entities, and predicates. The diskmanager stream for the RDF streams the quadruples in the RDF database and accesses them based on filters for subject, predicate, object and predicate and a given order. Functions were added to count reads and writes to the database as well.

The iterator for tuples was modified to sort quadruples and functions for comparing quadruples to each other based on a field provided as well as on all fields were created.

A program batchinsert was created to store the quadruples in the RDF database indexed according to 5 different indexing schemes.

A command line program report is used to output statistics on the RDF database.

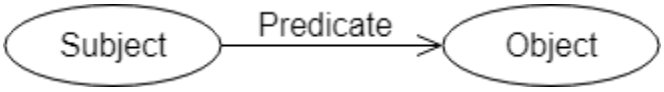
2.1 Quadruple ID Class

The quadruple class is derived from the tuple class which was originally implemented in MiniBase. Similar to the tuple indexing that utilizes a RID for each row, the quadruple objects are indexed through QID. Each QID object stores the storage location specified by pageNo and slotNo. The class also has three methods: copyQid, writeToByteArray and equals. The implementation details are as follows:

- copyQid: This method makes a copy of the given QID by its pageNo and slotNo.
- writeToByteArray: This method writes the QID into a byte array at offset.
- Equals: This method compares two QID objects by checking the pageNo and slotNo.

2.2 Label ID Class

The original RDF triples consist of subject, predicate and object. Each triple is represented as a node-edge-node graph as shown below. [2]



For each node in the graph it is associated with a label. The label ID class (LID) is used to store the labels and provides methods similar to the QID class. The only two new methods added are returnEID and returnPID which returns the entity ID and predicate ID respectively.

2.3 Entity ID Class and predicate ID class

RDF consists of triples. These triples are based on an Entity Attribute Value (EAV) model, in which the subject is the entity, the predicate is the attribute, and the object is the value. The objects and methods are similar to the QID class.

2.4 LabelHeapfile

The Labelheapfile modified the existing Heapfile to use labels instead of tuples. The only modifications made were changing references from Tuple to LID.

2.5 Label

Label was created based off Tuple and adds a getLabel and setLabel method. We can access labels through the LabelHeapfile.

2.6 Btree package

QBTreeFile and LBTreeFile were created which both derive from btreefile but were modified to create Quadruple btree files using QID and Label btree files using LID respectively. The classes QLeafData and LLeafData which define btree node data were made based on leafdata but now use QID and LID. KeyDataEntry had extra overloaded constructors added which handled cases for QID and LID data entry. No new logic was added except for renaming

2.7 rdfDB Disk Management

The rdfDB class was derived from the old DB and is modified to maintain the additional heap files and btree based index files. The class attributes are instantiated in parts, where the class is constructed with a particular index order selected, and later initialized to avoid circular calls with QuadrupleHeapfile and LabelHeapfile objects within the rdfDB class.

2.8 batchinsert Program - Five indexing alternatives and the performance analysis

The five indexing alternatives we chose are based on the secondary index method in which two keys are combined to facilitate a search.

(1) Subject, Predicate
(2) Subject, Object
(3) Object, Confidence
(4) Subject, Confidence
(5) Predicate, Object

Since the number of keys can range from 0 to 4 in a query, there are three situations regarding the key index matches.

- The keys match the index exactly:
  In this situation, we first retrieve the QIDs matching the key number one. Then among these QIDs we filter out the ones that match key number two.
- The keys match part of the index:
  In this situation, we can not filter the key because it does not match the index. For this case, we just retrieve the whole quadruple heap and filter according to the key.
- The number of keys is greater than two:
  In this situation, we would retrieve the quadruples that match the keys not in the index. After that we use the same strategy as situation one.

III. Interface specification

For the BatchInsert program, 3 command line arguments are provided. The first argument will represent the name of the data file containing the quadruple data to be inserted. The second argument will provide the indexing option based on the inserted quadruples. The third argument will represent the name of the database that will be generated from the data. From these 3 arguments, the necessary heap files/index files are generated.

IV. System requirements and execution instructions

Software requirements: Eclipse or IntelliJ IDE. These are especially useful due to their built in java debugging capabilities.
hardware requirements: None to specify, all of our computers were able to run the MiniBase and the end deliverable.
Execution Instructions: The MiniBase is acquired from our GitHub repository. It should be loaded into an IDE that supports java development such as Eclipse or IntelliJ. VSCode is usable but not ideal for our development purposes, though it should run the deliverable MiniBase as intended. After loading into an IDE, the user should build and run the project files, and can interact with the MiniBase through the console.

V. Related Work

https://rdf4j.org/documentation/programming/model/
https://web.stanford.edu/class/cs520/2020/notes/What_Are_Graph_Data_Models.html
https://www.ontotext.com/knowledgehub/fundamentals/what-is-rdf-triplestore/

VI. Conclusions

Phase 2 modifies the minibase provided to us into an rdf database by changing the given data tuple into quadruples and adding additional indexing functionality. This change required a set of modifications to many minibase modules. This gave insight into the way the different databases modules work together with storing data into tables as heapfiles, use of the btree data structure in indexing the data in these heapfiles, how disk manager manages these heap and btree files, the streams use in streaming these quadruples from the heap file with a set of filters, how the iterator works to sort the data in these heap files by comparing the quadruples, and how the batchinsert works to insert data into the rdf database.

Bibliography

[1] Candan, Kasim. Liu, Huan. Suvarna, Reshma. *Resource Description Framework: Metadata and Its Applications*, SIGKDD Explorations. pg. 6-17, 2001.

[2] Vinay K. Chaudhri. *What are some Graph Data Models?* CS520 Knowledge Graphs, Department of Computer Science, Stanford University, Spring 2021, https://web.stanford.edu/class/cs520/2020/notes/What_Are_Graph_Data_Models.html

Appendix

Our team worked collaboratively throughout each phase of this project including development of the project and all minibase module modifications as well as writing of the report.