Project Report Phase 2: Extending features to H2 Database

Shikha Rajen Soni Ketki Gorakhnath Trimukhe Dinesh Gudi

ABSTRACT

This paper will present a summary of the main features of H2 Database, the open source engine we decided to improve this semester, by implementing new features. We will first explain the reasons that led us to choose such a software, by explaining how files are stored and indexed. Then we will describe two database features we wish to add to the engine. Eventually, we will give information about how we are going to implement the features, and how which of this feature will improve the database usage for the end user.

1. PRESENTATION OF THE ENGINE

H2 Database in an open source relational database system. The source code of H2 is written purely in Java and the engine supports both SQL and Java Database Connectivity. The database can be used in applications and also with the web browser. In addition H2 has strong security features. It uses the SHA-256 for password encryption and AES-128 algorithm for encryption of data files. The database performs better than other databases written in pure Java like HSQLDB and Derby as indicated by the performance benchmarks. In this paper we studied the database in detail and implemented useful features which are not in present in the current version of the database.

The rest of the paper is organized in the following way. In Section 2, we present an in-depth study of various features of H2. In section 3, we point out the limitations of the current version of the database. Section 4 gives the details of the features that we chose to implement and the implementation. In Section 5, we give the motivation behind choosing to implement these features. Our approach is described in Section 6 and the Current status and future plan is presented in Section 7.

2. DESIGN AND FUNCTIONALITY

In this section we present the analysis of H2 features categorized into (a) relations and indexes, (b) record insertion into files, (c) key values and addresses insertion into the indexes, (d) page fetching, (e) query processing (and any optimization) steps

2.1 Relations and indexes

H2 database supports SQL queries for creating relationships between two entities. Relations between entities is captured by using foreign keys. H2 supports foreign keys with cascading and check constraints. Indexes for foreign keys are also supported. Shown below is an example of a query to create a relation

```
FOREIGN KEY(ID) REFERENCES TEST(ID);
```

Indexing[3] is used to speed up the query processing. Indexing in H2 is done using B+ trees indexing and hash based indexing. H2 database has a default indexing technique on the row ID; that is the primary key or some unique ID to determine the tuple if nothing specified. Indexing in h2 can support both equality and range look up queries. It also supports multi-column indexes on attributes. H2 also supports clustered indexing on a given column of a table. To create manually the indexes 'CREATE INDEX' query is used. By default, the index in H2 is B+ tree index. An example code shows the creation of an index:

```
CREATE INDEX INDEX_PLACE ON ADDRESS(CITY,
     NAME,FIRST_NAME);
```

The SQL statement for deleting an index is:

```
DROP INDEX INDEX_NAME
```

The "test.h2.db" file contains the indexes for the tables in the database[1]. In H2, if the index contains unique records, then NULL is distinct[1]. This allows only one record with a null value in one column.

2.1.1 *In-memory Hash Indexes*

In-memory indexing, in particular, hash indexing, improves the performance of querying and data manipulation[1]. By default, in-memory indexes are automatically created for in-memory databases[1]. However, we can customize the creation of in-memory hashes for persistent databases too. This can be done by using CREATE MEMORY TABLE[1]. Usually, this command causes the records to be stored in memory which can cause a serious problem if the table contains many rows. The in-memory has indexes perform much better than ordinary hash indexes. But, the drawback for both in-memory and ordinary hash index is that they are

useful only in the case of equality search and are not good for ranged searches. The hash indexes can be created by using[1]

```
CREATE UNIQUE HASH INDEX
```

2.1.2 Multiple Indexes

We cannot use multiple indexes for searching in H2. A query containing a single predicate can at most use only one index. However, we can use the union operator to combine two queries so that each query uses a separate index[1].

2.1.3 Function Based Indexes

In H2, function based are not supported but they can be created indirectly by using the values in the column obtained after computing them using the statement

```
CREATE TABLE TABLE_NAME(attribute1 var1, attribute2
    var2,âĂę..)
```

The following statement can be used after obtaining the result from the above statement using CREATE INDEX. This is an indirect way of creating a function based index. Other databases like HSQLDB, MySQL, Derby and PostgreSQL support the function-based indexing[1].

2.1.4 Multidimensional Indexes

In H2, there is no spatial index. But multi-dimensional ranged queries are performed using B-Tree[1]. This is done by creating a single value for a multi-dimensional search key. The obtained value is used for spatial queries. In the current implementation of H2 database, Z-order or Morton order is used for spatial indexing[1]. The other alternative available for such kind of indexing is the Hilbert curve. But this kind of implementation is much difficult. The Z-order uses the bit-interleaving algorithm for spatial indexing[1]. As mentioned above, B-tree is then used for indexing the key value created from a multi-dimensional key[1].

2.2 Record insertion into files

In H2, all the records are stored in a single file. Each file is further divided into multiple chunks and two file headers[4]. Each header is of size one block which is 4096 bytes. The chunks are comprised of at least one block and are usually of around 200 blocks[4]. The chunks contain data in log structured storage in which the data and metadata are written to logs. Each chunk is made up of many B-tree pages. All the records and the indexes are by default stored as B-tree pages[4]. The root node and the non-leaf nodes in the b-tree are pointers to other pages[4]. The leaf nodes are the pages with keys and entries.

2.2.1 Maps

The tables in H2 are represented by maps composed of unique keys and the corresponding rows[4]. The advantage of using maps is that the index lookup is uncommonly fast.

2.2.2 File Header

The two file headers are copies of one anothe[4]r. The duplicate file header serves as a backup when one of the headers is corrupted[4]. The file header contains the data regarding the block location, block size, chunk details, time created, file format details, version number and checksum[4]. The checksum in both the file headers is verified for correctness.

2.2.3 Chunk Format

Each chunk is composed of a header, the pages that were modified in that version (snapshot of all the pages that are made up of maps) and a footer[4]. The records of the database are stored in pages. One chunk is made up of about 200 blocks[4]. The footer contains the data which helps the database to verify whether the chunk is written completely. The header and the footer of a chunk contain the data regarding the location of the first block, the number of live pages, the size of the chunk, the address of the next chunk, checksum, version number. The chunk contains the pages that are modified in that version along with the parents of those pages recursively up to the root node[4]. If a map in a page is updated, then that page is copied into the next chunk and is modified[4]. This decreases the number of live pages in the old chunk. This kind of storage mechanism is called copy-on-write[4]. A chunk is no longer useful if it does not contain live pages. If it does not contain any live pages, it is marked as free so that new chunks can use the free space [4].

2.2.4 Page Format

The maps are stored in the form of b-trees, and the nodes of a b-tree are the pages that contain the map data[4]. The leaf pages of the b-tree consist the key value pairs of the maps. The non-leaf pages are made up of keys and corresponding pointers to the leaf pages. The root can be a leaf node or a non-leaf node based on the data [4]. Each page contains details about the length of the page, checksum, id of the map that the page belongs to, the number of keys in page, address of children, the number of children, keys, and values(in the case of leaf pages only [4]). The storage of pages follows a particular order: the root page first, the storage of the internal pages follows and later the storage of the leaf pages.

2.2.5 Insertion Queries

Data insertion[3] in H2 is same as any other relational database system.It uses the 'INSERT INTO' query to insert a new row into the table [1]. Insertion in H2 can be either DIRECT or SORTED. Using DIRECT keyword will insert the new entry directly into the table without any steps further, while SORTED will make the B-tree pages split up at the point where the entry needs to be inserted. This feature helps improve performance and optimize the disc space.

Simultaneously, the B+ tree index for the database is created in the form of pages [1]. The default database size when it is first created is 2KB. We can explicitly change the initial database size.

In H2, the record insertions into files is handled by "FileStore.java". There are four access modes to access the records in files namely "r", "rw", "rws" and "rwd"[1]. Additionally, the files are encrypted for security purposes.

2.3 Key values and addresses insertion into the indexes

In H2, the records are organized in the order of the primary key of type INT[1]. This makes the indexes on the primary key to be of the same order as that of the original table. This is known as "clustered index". However, there

can be indexes on columns other than the primary key. In such cases, the index would be un-clustered.

By default, the records in the data files and the indexes are stored in the form of B-trees[1]. Each element in the B-tree is composed of a list of unique key and the data[1]. The records in H2 are always organized in the form of data B-trees with the unique key value of type long[1]. If the primary key of type INT, TINYINT, SMALLINT or BIGINT is specified and it is of only one column, then this key serves as the unique key of the data B-tree[1]. In cases where the primary key is not mentioned, the primary key is composed of multiple columns or if the primary key is not of any of the type INT, TINYINT, SMALLINT or BIGINT, then an auto increment key called as the row id is used as the key for the data B-tree[1]. The only exceptions to the data columns that are not stored in the B-tree are the BLOB and CLOB columns that are explained later. These columns are stored externally[1].

Each index created on the table is also a B-tree by default[1]. The entries of this index are composed of the key of the data B-tree in addition to the column on which the table is to be indexed[1]. If the primary key of the table is created after the creation of the index, the primary key is composed of multiple columns or if the primary key is not of any of the type INT, TINYINT, SMALLINT or BIGINT or if the primary key is of multiple columns, then a new b-tree index is created[1].

2.3.1 *Row id*

In H2, each data record is uniquely identified by a row id[1]. This row id attribute acts as a key for the table in the database. The row id is represented as a long integer[1]. The database usually generates the row id, but in cases where the table contains only one column, and the datatype of the column is an INT or BIGINT, the column values become the row id for individual records. We can access the row id by using the _ROWID_[1] macro, although its use is not suggested and is a non-standard way of accessing the row id. Using row id to access the data is fast since the data is automatically sorted by row id[1]. It is important to remember that we cannot access the row id of the record until the record is created. If the predicate in the query does not contain the row id and there is no index available, and then all the rows are scanned. Such a scan goes through the entire table is the same order as the row id. We can use EXPLAIN SELECT to know the strategy the database uses to retrieve data.An example is shown below[1]:

```
SELECT * FROM ADDRESS WHERE NAME = 'Shikha';

EXPLAIN SELECT PHONE FROM ADDRESS WHERE NAME = 'Shikha';
SELECT
    PHONE
FROM PUBLIC.ADDRESS
    /* PUBLIC.ADDRESS.tableScan */
WHERE NAME = 'Shikha';
```

2.4 Page fetching

Page fetching[2] is basically data retrieval from the database for modification or read only purpose. H2 is supported by a persistent key-value type storage structure namely MVStore[4]. MVStore is a default storage engine for H2 versions of 1.4 versions and above. It provides support for SQL, JDBC and transactions, but it can also be used directly within the application[4].

The MVStore , multi-version store stores key-value pairs in a number of maps, which uses the java.util.Map interface[4]. It supports file-based persistence as well as in-memory operation[4]. It provides the feature of file abstraction to provide file encryption and zipping files. The data read and write is basically done using java code MVStore provides an interface known as MVStore Builder to provide options to perform various operations[4]. Some of the operations are as follows:

- readOnly: open the file in read-only mode[4].
- compress: compress the data when storing using a fast algorithm (LZF)[4].
- encryptionKey: the key for file encryption.

Now for efficient fetching of data MVStore provides some main implementations according to the type of data pages[4]. The MVRTreeMap is an R-tree implementations which supports fast data retrieval using spatial queries[4]. As mentioned above MVStore is a key-map structured data storage medium, it supports common lookup operations including access to first and last key. It can be used to iterate over the map keys. It supports a rare feature for maps, that is, fast index loopkup. Each map is arranged internally in the form of a counted B+ tree which helps faster and efficient performance as well as calculations like median fo keys in the maps. It is feasible to take a snapshot of the data in the maps at a given point of time. This snapshot is known as a version and is a fast process since only the pages that are changed are copied in the next version and the older version becomes a read-only version. It supports rollback from an older version[4]. The In-memory performance operations are 50 percent slower as compared to using TreeMap interface of java. In case the file name is not specified, the MVStore works purely in-memory. MVStore does not have any specific size limits, so the pages, key-values and data chunks can be large enough[4]. Also there is a provision to store large binary objects by splitting them into small blocks[4]. There is support for concurrent read and writes where many read operation can take place parallel and write operations are synchronized. Caching of data takes place which is done at page level.

The file format for the stored data file contains two file headers and data chunks. The chunk contains a number of B+ tree pages and is stored in the form of log structured storage. Pages in the following chunk have a reference to the pages in the preceding chunk. Each chunk contains a header and number of pages and footer. The pages are B+ tree pages as mentioned above and contain key-value pairs of the map data. These are mainly leaf pages. Also there are internal nodes which contain keys and pointers to these leaf pages. The leaf or any of the internal nodes can be the root of the tree. The pages are not human readable and are basically stored in an array of bytes. In every map the root page is stored first, then the internal nodes and then the leaf pages. This speeds up the read operation and makes the sequential reads faster than the random access reads. The page pointers are stored in long in a special format.

The information that whether the page is a leaf page or an internal node is encoded, so that when deleting the data

map the leaf pages do not have to be read. The length code ranges from 0-31 where 0 signifies the maximum length of the page that is 32 bytes and 31 means longer than 1 MB[4]. So in this way reading the page requires only one read operation. The chunk metadata stores the result of addition of the maximum length of all the pages and when any page is removed or marked as removed then the maximum length is adjusted accordingly. This helps in figuring out the free space in the block and the number of free pages available. Also to support efficient range counting, skipping of pages and looking up pages by index, the total number of entries in child pages is preserved. These pages form a counted B+ tree .
[3]

- First we have to determine the hash of the item name using hash function like MD5
- Next, we take into consideration the last two bits of the resulting hash value and place each item in the domains depending on it. 00 gets Domain0, 01 gets Domain1, 10 gets Domain and 11 gets Domain4.

2.5 Query processing and Optimization

Initially, the queries are parsed using the "parser.java" file. The parser is used for performing appropriate functions and also detecting errors in the syntax. The parser first stores the query as a string in a variable called "sqlCommand". It then breaks and reads each character and stores the type of each character as an integer in an array based on whether it is character, numerical, quote, space or a special character. An integer value corresponds to each of the character type and is stored in the charecterType array. The parser then breaks the query into tokens like "SELECT", "FROM". Each token is identified as a Keyword, identifier, parameter or value and various functions corresponding to these tokens are performed. The parser also checks whether the tokens are syntactically correct. The identifiers like names of databases and the columns are checked by the parser if they actually exist. The parser throws appropriate exceptions for any errors in the syntax.

H2 uses cost base optimization for query processing[3] and optimization process. Query processing involves processing simple as well as some complex queries. Queries which are simple or medium in complexity, the running time of the query is the lowest of all the possible running times of all possible plans. Also there are an extremely complex query for which an algorithm is used which tries all possible combinations for starting few tables and later continues with greedy algorithm. Later a genetic algorithm tests at most 2000 distinct plans out of which only left deep plans are evaluated. The queries and expressions after parsing may be simplified automatically if possible. The functions are also optimized, for example if the WHERE clause is always false then the table will not be accessed. Also for SELECT queries where, no WHERE clause is used, the data is not accessed at all. Thus the query only counts all the rows of the table and does nothing else. This phenomenon is known as count optimization. In query execution process at most one index can be used for each join operation.

3. LIMITATIONS

H2 Database Engine suffers from several limitations. First of all, the software does not implement multi-threading. As it is developed in Java, we could imagine it using the class Threads. It would certainly permit to compute queries more efficiently. The engine however is fully compatible with programs using multi-threading. Besides, H2 Database's features mostly consist in Information Retrieval tools: DDL and DML. We can select and retrieve data by interrogating tables, do projections, join. Yet it would be interesting that a user can do some data mining directly in H2's console such as classification or clusterig. Finally there are several indexing methods, such as trees and Index. But it is most probably perfectible. For example, adding adaptive indexing would help a database administrator to create the index more quickly

4. IMPLEMENTATION

4.1 K-Means clustering

4.1.1 What is it? What is the logic of clustering?

Clustering consists in analyzing data and finding similarities between tuples in order to sort them into clusters and eventually find a structure inherent to the data. Thus rows belonging to the same clusters will have bigger closeness than rows from two different groups.

Several clustering methods exists: Otsu's method, hierarchical clustering, etc. K-Means is one of them.

K-Means method consists in two things. First, one needs to find K, the number of clusters that best describe the data. There is no point (and no interest in) considering using one cluster for each row. That is why one needs to use some algorithms to compute the K value. In parallel one wants to sort each tuple in the corresponding cluster. To do so, KMeans most common measure is the Sum of Squared Error (SSE) that compute, for each point, the error in the distance to the nearest cluster.

4.1.2 Why adding such a feature?

Most features of H2 Database consist in data retrieval. And their main purpose are finding the data corresponding to a given query in the database as quick as possible.

K-Means clustering however does not work like that. It does not correspond to record data, but it tries to enrich it by analyzing it. Thus it permits to add new information that was not initially in the database.

For this reason, such a feature seemed very powerful for us and that is why we wanted to add it to H2 engine.

4.1.3 What Query will be inserted

We will need to create a new query called K MEANS. The user will be able to use the new feature by typing the following command on the H2 console:

```
select K_MEANS(column_name) from table_name;
```

4.1.4 What changes will we do in the code?

Adding this new feature will oblige us to do some modification in H2 Database engine source code.

First, we are going to create and instantiate a new command. Each H2 DML command is located in a corresponding command name.java file within the following package: org.h2.command.dml. So we will create a kMeans.java file that will inherit the abstract class query.

Moreover, as we want to implement K-Means method, we will need to name a new package: org.h2.clustering.kmeans, in which we will right every files and lines of code concerning our model: Sum of Squared Error (SSE) computation, algorithms, etc.

4.1.5 What will be the output of the query

Using K-Means method on a table will permit the user to sort data into clusters. To do so, the output of the K MEANS query will be a graph representing the the clusters and their centroids.

4.2 FIRST() and LAST() functions

4.2.1 What does it do and how?

The first query returns the first value of the column selected. Last like first returns the last value of the column selected, like in SQL. This can be done by storing the first and the last entry into the table, and then returning the the column value. The last entry of the table is replaced each time a new row insert takes place. The code area to change would be

- org.test.synth.sql, where a new command for FIRST and LAST would be added and given a int value.
- Add 2 new class in the src/main org h2 command dml as First.java and Last.java.
- As soon as the insert command is processed if the record ID is 1, it is permanently stored in the class First.java
- The insert query is in the same package as the src/main org h2 command dml, where in the function insert rows will keep an instance of the last inserted row.

4.2.2 What will be the query?

This extension is same as FIRST() and LAST() in sql using the column name.

```
select FIRST(column_name) from table_name;
select ROW(column_name) from table_name;
```

This command will return the value of the column in the first and last row respectively.

5. MOTIVATIONS

We chose H2 database to work on as it is mainly written in Java and hence it can be embedded in Java applications or can be run in client-server mode. The source code is available as open source software under modified version. Though there are numerous other open source databases that are frequently used, we chose H2 for the project due to some of the advantages of H2 over other databases. H2 is an embeddable database in nature and seemed the right choice for the kind of extensions we wanted to propose. The extensions we have provided are K-means clustering, adaptive indexing and First() and last() functions. Clustering is a popular data mining and statistical analysis technique. If there are large number of variables, then K-Means can prove to be computationally faster than hierarchical clustering. This technique though significant seemed missing in H2 and hence motivated us to choose this as an extended feature. Adaptive indexing as mentioned in the extensions uses dynamic hash indexing. This will give us an opportunity to explore actual implementation of indexing and difference between various other indexing techniques. The first() and last() functions will add an extra functionality and will save efforts of writing the whole function again while returning the first and last value of the column selected.

6. APPROACH AND METHODS

6.1 Project tools and management

In this section we will discuss about the environment, tools and management methods that we did since the beginning of this project and that we are going to use to complete it.

H2 Database Engine code is available on H2's website, and one can also find a link to the GIT repository of the project. Thus one of the first things we have done is to fork this repository so as to get our own version of the files on which we could develop our new features.

H2 is developed in Java, and every person in the group felt comfortable with using the Eclipse IDE for Java development. As a consequence, we then had to associate the forked GIT repository to a plain Eclipse project so that we can write code, add new files and compile on this IDE. It demanded us to rebuild some dependencies and add few modifications to the source code to compile the project.

We used the Eclipse debugging tool to track the code. This tool was very helpful in going through the required parts of the code quickly.

6.2 Source code comprehension

To do the analysis of what is existing, we were able to get the Javadoc, some tutorials and features explanation on H2 Database website. We also did cross comparisons with other existing engine so as to understand H2's specifications.

The main method in H2 is present in the "Console.java" file which is present in org.h2.tools package. The queries are parsed by using the "Parser.java" file in the org.h2.expression package. The database connection is taken care of by "Database.java", "JDBCDatabaseMetaData.java", "JDBCPreparedStatement.java" and by the org.h2.jdbc package. The class for the tables is "Table.java" in org.h2.table package. The web browser is implemented in the classes "WebApp.java" and "WebServer.java" in the h2.server.web package. The aggregate functions are implemented in the org.h2.expression package.

6.3 Team management

To enhance communication inside the group, we did regular weekly meetings in order to discuss about the progression or some particular issue that we needed to overcome.

However, it did not seem enough in everyday, that is why we also chose to create a slack team. Slack is a web application that permits us to chat, to create channels so as to discuss some particular topic, share documents and web links. On this platform, we shared code documentation and tutorials, we discussed about every assignment, shared some data samples, etc.

6.4 Report assignment

In this project, we are to submit several reports that must be written using LaTeX. To do so, we chose to use the website overleaf.com that permits to write LaTeX, online. Be-

sides it allows every user to modify simultaneously the document which is a gain of time.

6.5 Versioning

As we will have to develop and collaborate simultaneously on the source code, we need to use a versioning tool. For this reason we used a common GIT repository hosted on github.com. We used a Feature Branch Work- flow, that isolates new features into branches and used pull requests as a means to discuss changes before they were integrated into a project. This workflow helped us to keep things organized while always having a working version in branch master.

7. FUTURE WORK

The implementation of the extensions we proposed to be added to the existing source code of h2 are currently accomplished. So, currently we have successfully implemented the first() and the last() functionalities. This code works well for all the datatypes. Kmeans-clustering has also been successfully implemented such that the cluster graph is an output in the Eclipse editor. We have calculated the number of clusters from 1- 20 and the SSE. The output obtained is the efficient clusters formed from the given data. The H2 console displays the number of clusters. In the future we look forward to work on embedding the clustering graph in the H2 console itself. The user can thus get the graphical representation on the click of a button on the H2 Console.

8. REFERENCES

[1] H2 database features. `http://www.h2database.com/html/features.html`.

[2] H2 database Indexing. `http://www.h2database.com/html/mvstore.html#differences/`.

[3] H2 database performance. `http://www.h2database.com/html/performance.html/`.

[4] H2 Mvstore. `http://www.h2database.com/html/mvstore.html`.