Database – Report

The report is divided into sections, discussing each part of the assignment.

TO RUN

Please use the zipped Final Version that is uploaded to make and examine the code. I have also included a zipped file containing folders of all the individual sections to show the database in progress for each step as suggested.

To run, please start with $make
Then, access the user interface with $java DbManager

To view a database	$USE $<databaseName>
To see tables	$list
To view a table	$view <tableName>
To view column Types	$type $<tableName>
To view a column	$SELECT <columnName> FROM <tableName>

This will build 3 sample databases that shows how the system builds a folder structure and populates it with data. These tables may also be tested to show the different results of commands on tables, and that user input/updates can be stored in the database system after shutdown. All text files I uploaded also need to be included in the folder, as some of these are required for the tests (building the database from files) and some store the sample databases.

SUMMARY OF EXTENSIONS

After completing all required sections, I carried on working on the following extensions: Type, Constraints, and Catalogues. I also started the User Interface, but it is basic and I focused more on making a robust system with the previous extensions.

Type: I placed cell values inside their own Item class, which has constraints placed on the value using a type enum (currently Integer, String, Boolean). Integers may contain only numbers (negative or positive), booleans may contain either 0 or 1, and strings may contain anything. The type information is stored so that when a database is written to a text file, the type information is not lost. Further, when a database is built from its storage text file, item accesses and re-builds the type of each cell. I do this by adding a type indicator to the start of each value, so if the value is “1456” it would be stored as “i%1456” in the text file. The Item object stores its enum Type as Type.INT, checks the constraints, and splits the string to store “1456” as the value. Columns must be the same type, with the exception of the column name, which defaults to a string.

Foreign Keys: The Item class also tracks foreign information. If a cell is a foreign key, it stores the name of the primary table and column it references. It also has a pointer to the correct Record it references (added later by its database). The validity of the foreign key is checked by the database, which has access to all of its tables to confirm that tables, columns, and values match as they should. In this way the order of tables added is important. If a user attempts to add a table with a foreign key pointing to a table that does not exist, the database will display an error and prevent the action. For an example, please see the Robot table, which contains tests/examples of a foreign key and type constraints. When a database has no foreign keys, the FKIndex.txt will exist but will be blank.

Constraints: When a database is built from a stored file, it checks a special file called the FKIndex.txt (the foreign key index), which stores information for each table containing a foreign key. This information allows the database to first check through its tables, to confirm that each foreign key references a table and column that exist, and that its value actually matches as it should. Then, the database adds a pointer from the foreign key item to the correct Record in the table, to allow for quicker joins later on. When writing the database to files for storage at the end of the session, the database will compile an updated FKIndex file, by going through its tables and gathering all foreign key data and storing it in the file. The format of the FKIndex is that each row contains: primaryTableName

primaryColumnName foreignKeyTableName foreignKeyColumnName. There is one row per table that requires foreign keys.

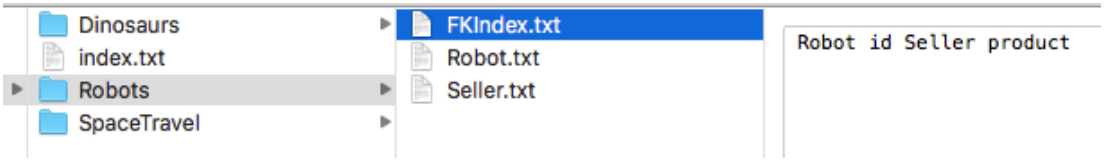

Figure 1 Foreign key index for Robots database which has one table with a foreign key

Catalogues: Inside the file system created by the database manager class for loading and storing databases is an index text file. This index stores the names of all databases in the system. Users may query the names of all databases in the system, and the index is used to easily provide this information. The file is automatically updated as databases are added or deleted. Further, once a database is selected, users may print out the tables inside, select a table and display it in full, select specific columns to display from a table, or display the type of each column in a table.

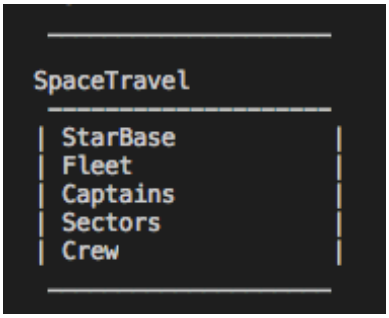

Figure 2 A database and the table names it contains

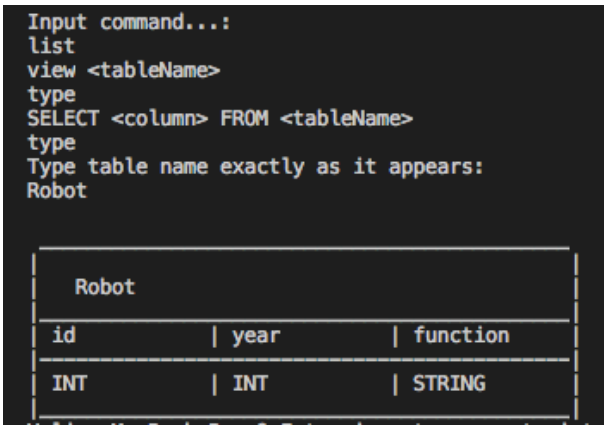

Figure 3 A display showing the types of each column

Figure 4 Index file listing database names

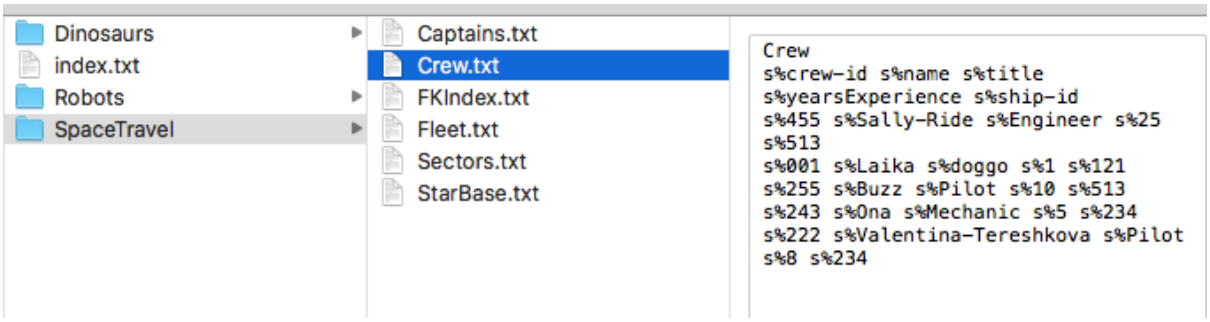

Figure 5 File system storing all table information – including type

User Interface: The user interface is limited, because I have not made a complete SQL parser, so not all of the database functionality is accessible through the user interface. For now, users may see all the database names as a printed chart, may select a database, view the table names within the selected database as a chart, select individual tables and display them in full or by specific columns, and view column types of a selected table.

```
Here are the Databases:
> SpaceTravel
> Dinosaurs
> Robots


Enter database name (exactly as it appears):

SpaceTravel
WORKING WITH DATABASE: SpaceTravel
```

Figure 6 User interface for selecting a database from the system

```
Input command...:
list
view <tableName>
SELECT <column> FROM <tableName>
view Crew

┌──────────────────────────────────────────────────────────────────────┐
│   Crew                                                               │
├──────────────┬──────────────┬──────────────┬──────────────┬──────────┤
│ crew-id      | name         | title        | yearsExperie | ship-id  │
├──────────────┼──────────────┼──────────────┼──────────────┼──────────┤
│ 455          | Sally-Ride   | Engineer     | 25           | 513      │
│ 001          | Laika        | doggo        | 1            | 121      │
│ 255          | Buzz         | Pilot        | 10           | 513      │
│ 243          | Ona          | Mechanic     | 5            | 234      │
│ 222          | Valentina-Te | Pilot        | 8            | 234      │
└──────────────┴──────────────┴──────────────┴──────────────┴──────────┘
```

Figure 7 A table displayed using VIEW Crew

```
SELECT name FROM Crew

| name         | Sally-Ride   | Laika        | Buzz         | Ona          | Valentina-Te
```

Figure 8 Selecting a single column from the Crew table

```
SELECT title FROM Crew

| title        | Engineer     | doggo        | Pilot        | Mechanic     | Pilot
```

Figure 9 Selecting a single column Title from the Crew table

SUMMARY OF REQUIRED SECTIONS

Here is a brief summary of my thinking for the project. I had to refactor a lot to get to the final datatypes and design used. Here is a brief overview of where I ended up:

- **Testing** Unit testing was used throughout development. All but the display class and DatabaseManager class (user interface/ input) were heavily tested automatically. These two were carefully separated out so that all other classes could be tested independently of input and display.

- **Item** class to store values in a Record. This adds the ability to track and add type constraints, as well as connect Items vertically for easy column searching. The item object may also store foreign key information, and stores a pointer to the record it points to, as well as the table name and column name of the table. This allows the database to check for the validity of the foreign keys as it builds its tables from files or adds new tables. Item class does not depend on any other classes.

- **Record** class uses a Linked list to store Items. The first item value automatically becomes the primary key. The record class only depends on the Item class. The Record class can link two records together (to allow items to be accessed as columns), return its primary key, return the type of its items, add items and check types, remove items, and check if it contains a given value.

- **Table** Linked Hash Map to store rows. Primary keys serve as the keys in the hash map, and are guaranteed to be unique. I also track pointers to all the column names using a hash map (the pointers prevent duplication of data) and the data structure further assures that the column row names are unique. This column structure, combined with the downward pointers in the Item class, allow for a single column to be easily extracted from a table. The table also stores if any of its columns are foreign key columns, and if so, stores the data needed to easily check this information is accurate.

  The table can add rows, delete rows, add multiple rows, add columns, add multiple columns, insert columns at a specific place (mostly for display concerns), completely clear itself, update itself, check for values it contains, return a list of its keys, remove a specific column, remove a specific row (via the primary key), update a specific column or row, add a row (and link the items to the items in the row above, for easily accessed column behaviour), and return the names of all its columns.

- **Database** The database class has an Array List of all of its tables. These tables have all the functionality of the above tables. Added to this, the database performs foreign key constraint checking, and assures all tables added with foreign keys are legal (the other tables, values and columns exist). The database pulls in several other classes and uses them as components. Thus is has a display object to show its tables, a Read Write object to access tables stored as files and to write out its tables and store the data (as well as type and foreign key information) between sessions.

  The database can add, delete, and update its tables, and updates the file system as needed. It also deals with constraints.

- **Display** Currently feeds to the console, but could be updated and replaced with a graphical display. Everything that appears in the console passes through this class (there are no system.out.println() anywhere else!).

- **ReadWrite** This class is responsible for reading / writing tables and databases to file, writing the index, the foreign key index, and creating the folder structure on command. My goal was to make a robust class that utilizes the try / catch block effectively to gracefully handle missing files.

- **Input** This class handles and processes all user input. It is used by the database Manager.

- **Database Manager** The database manager is a class that I used to wrap up all the other classes into a working program. It deals with the main program loop, and prompts / deals with user queries. On command, it will show the databases in the system, allow users to add new databases or select an existing database loaded from file.

A NOTE ON FILES: The database Manager creates a file system for storage (as in the files section). If it is the first time the program is run, a folder called "databases" is created. Inside this folder, each database gets its own folder with its name. An index file is created that contains the names of every database in the system. Inside each database folder there is a text file for each of its tables, which contains the table name, data and type information. There is also a special file that tracks foreign key relationships, called the FKIndex. This allows databases to be loaded from saved files made previously by the database manager. When changes are made to a database or table, it writes to these files and updates them.

REQUIRED SECTION 'MINI REPORTS'

Below I provide 'mini reports' for my thinking as I worked through each section. Some decisions were later changed, and there was a lot of refactoring between sections.

RECORDS

DESIGN DECISIONS

I decided to use Varargs, since a record will have some unknown number of items associated with it, and it makes sense to set them all up with the constructor at the start.

The second choice is to use either an array or linked-list. Since varargs allows access to the number of items input to the function, an array would be possible of that size. However, I chose a linked-list because this will allow for the flexibility of altering record items after a record is created. This may or may not be a good idea (to allow for alterations) but I will design with alterations in mind, because I know that in SQL it is possible to alter a table, and so it is a feature that users want and need.

One thought is that a way around this would be to require users to delete old tables and recreate them to add columns, but that might be messy.

QUESTIONS / IMPROVEMENTS

I am also considering using a map, with the strings as keys. I also considered sets, but that would not work because while there would be no duplicate keys, there also would be no duplicate anything else, even when the data called for it.

POSSIBLE ISSUES

I saw a potential danger with a method that gives access to the array itself (because it could be altered) but no danger in passing out the actual strings, as they are immutable. For this reason, I chose to clone the internal LinkedList before passing it out.

Another possible issue is letting users input items with spaces. For now, it is not a problem, but perhaps I will want to add a check that no Strings contain spaces.

TABLES

DESIGN DECISIONS

Because the primary methods will involve insertion and deletion, I chose a LinkedList over an Array to start.

First thoughts about the problem of the cycle between a Record and the Table it's in: perhaps I should refactor the Record class such that each item is entered as a tuple of Strings, like (ColumnName: Item) so that each record 'knows' which column each of its items are in. On creation of a record (in the table), it will simply specify the column name. Some problems with this could be errors caused by the

repetition of the column names, if something gets entered incorrectly. Is the best way to do this a map? The 'key' would be the item, and the value the column name - except this won't work because items aren't guaranteed to be unique. Column names are unique except they cannot map to multiple items.

I decided to use getters even for immutable String name field, so that users can later choose to alter the name the of the table.

REFACTOR
Updating Record class to be able to return specific item by index number.
Update Record to replace all values

QUESTIONS / IMPROVEMENTS
I perform a check to make sure column names are unique - achieve this by comparing to length of set of same data

NOTE
Number of rows is the same as index of last row (7 rows, last row will have index 7) because the first row (row 0) is the Column Name row, and it is not counted. So a table with 1 row has 1 Column name row AND a row of data. The column name row cannot be deleted, but it can be altered/updated

REFACTOR TABLE

REFACTOR
Have decided to refactor both record and table to create a linked grid, so that any combination of rows/ columns may be selected by the user. This will also make it easier to add or delete columns.

My thinking is that this operation will be faster to find items, even if it is slightly slower to initially setup. The downward link will make accessing columns fast, whereas before, using a list of linked lists, I would have had to go through all items in every row up to the column index to find the one below.

Additionally, having a class Item will make it easy to change the storage type for an item later on, or to expand to have different types of items within a single table.

I will use a hash table to store the row index based on the primary key. I will use either a map or another hash table to store the column name to the column index.

I refactored the Record class to use a new Item class. Each item has a value and a down pointer (pointing down to the next item within the same column).

QUESTIONS / IMPROVEMENTS
I added a check to make sure column names are unique.

FILES

DESIGN DECISION
For reading and writing to file, I've decided to make a dedicated class, that knows about Tables and Records. This way feels clean and easy to use, and should connect well to having an overall manager class. It also allows me to keep Record and Table from knowing anything about reading/writing files.

I chose scanner class for reading, and fileWriter for writing.

Currently, I'm thinking that no item may contain spaces (so all column names and item entries are single words). The table is stored as:

```
tableName
colName1   colName2   colName3 ...
item1      item2      item2   ...
item1      item2      item2   ...
```

REFACTOR
Need to refactor Table to be constructed with a collection of strings, and the same for Records.

QUESTIONS / IMPROVEMENTS
Should the input/output relating to files be handled directly by Table.java, or a new class? Perhaps a new input/output to file class, that Table.java can use as an object.

PRINTING

DESIGN DECISION
I decided to make a class that handles display, and it knows about tables. To start, I will focus on console display (but this design should make it easier to add a GUI later). As it is, all messages (including errors) are passed through the display class to make it easier to add graphics later on.

I have chosen to format the display so that very long items will be truncated after a set amount of characters. I have used alignment to keep the rows matched up cleanly, with lines dividing the title (cantered), column names, and data.

KEYS

DESIGN DECISION
Each record has a primary key, which is automatically set as the first item entered on construction. This primary key will then be used by the Table class as a key in a hash table.

I chose to use a Linked Hash Map for the rows (after a few other experiments). This takes Strings as primary keys, and stores a Record (row) for each one. I also use a HashMap to track columns by column name (and so column names must also be unique).

The uniqueness is assured by the data structure, but I add extra precaution that prevents users attempting to add duplicate column names or row primary keys, to prevent a user accidently 'updating' a primary key rather than adding a new item.

REFACTOR
For the table class, I refactored the list of Records into a hash table, so that each row may be accessed with its primaryKey as a hash key.

I also made a hash table for the columns, using the column name as the key - the hash table contains pointers to the first item in every column, so it does not duplicate any information.

I changed from Hashtable to Hashmap, and then changed to a LinkedHashMap so that I could track the order of items added - this proved useful when adding new columns, so that I could quickly LinkedHashMap items downwards in a new column. This feature allows for specific columns to be worked with in isolation, with in mind joining table later on.

I had to refactor the Display class to work using primary keys rather than indices.

I also refactored the ReadWrite class to use primary keys rather than indices, and made use of some hashMap methods to make testing more robust - comparing the collections of values of a table that has been written to/read from and also comparing keys.

QUESTIONS / IMPROVEMENTS
While I can access the order of the items added using LinkedHashMap.setKey(), I also chose to store a Record lastAdded pointer to improve performance, and use this when only the most recent row is needed to link it vertically to the row being added. In other cases (such as linking up a whole column) I do use the setKeys() method.There might be a faster way to access the order of added elements in a linkedHashMap.

DATABASES

DESIGN DECISION

Have created a Database class that has components (Display, ReadWrite, List of tables) inspired by the component architecture system.

On construction, the database class creates a folder 'databases' if it doesn't exist, and places another folder inside with the database name. Inside this folder it stores all tables as .txt files (the convention is the table name is the file name). When constructed, if there are text files in the database folder, it loads all files, generates tables, and stores them in a list of tables for easy access.

The database may be updated (tables added, information updated within tables, tables removed). When a table is removed, the text file (if it exists) is deleted. When a whole database is deleted, the folder and all files within it are deleted.

At closing, the database writes/overwrites all table files within its folder system for storage.

For content and structure of the Database class, I have chosen to largely keep the logic related to tables and folder structure, with a getTable() method used to access individual tables to then alter tables directly. Thus, the database deals with creating folders, making/removing tables, and loading/saving tables (via the ReadWrite class).

With this design, I plan to add a Database Manager that holds a list of databases in the system, and which has an Input component to interface with the user.

REFACTOR
Updated ReadWrite class to deal with folders and accessing filenames within folders.

TESTING
Throughout, I have used unit testing. I also make methods Boolean where possible for testing and for normal use, as issues with reading/writing files can stop various processes.

QUESTIONS / IMPROVEMENTS
I want to add a separate Input class to take in user input, and this will later hold the logic for parsing, to allow for expansion into a strict/loose version of SQL for communicating with the database. I would also like to make a further class, DatabaseManager, that deals with switching between databases, and ultimately will have the Input component for user input.

EXTENSIONS

MANAGER: To start, I made a database manager class, that wraps up the database, user input, display, and reading/writing files. It handles the database system, allowing for multiple databases to co-exist. It also handles the current active database and program loop, and saves it to file after actions/updates.

INPUT: I made an input class to handle user input and feed it back to the Database manager. If I had more time, I would like to have also made a parse class, that takes user queries and translates them into actions on databases / tables.

TYPES: I added a type enum class to hold INT, STRING, and BOOL. I then added fields to the Item class to track type. This step required a lot of refactoring in various classes, including Record, Table and Database, to account for the ability of users to declare type. I was also careful to add type checking and constraints on types (all types in a column must match, and rules about what types may contain). The ReadWrite class also needed to be refactored look for type, and so I created a type indictor for storing and reading tables from/to files. The Record class needed to be refactored to check for the type indicator and initialize items with correct types. The item class handles dividing this information into its separate fields of ‘value’ and ‘type’.

CONSTRAINTS: For this, I added a foreign key field to the item class. If an item is a foreign key, it stores the name of the table and column it points to, as well as a pointer to the item (added later by the database). This step also required a lot of refactoring, because in addition to adding the information,

the database needs to check that the table and columns exist for the foreign key to point to. I also needed some way to keep this information accessible when reading and storing databases to files.

Thus, I created a special file called FKIndex to store foreign key relationships. This made it easy for a database to load itself from files, and check over the FKIndex to confirm that all tables/columns exist as they should, and further to create the links between the foreign keys for quick joins. If a new table is added that has foreign keys, the database only allows the addition if it meets the requirements. Thus, the order that tables are added matters. Overall, the system prevents users from adding incorrect foreign keys.

The database constructor needed a lot of additional information and checks to handle foreign keys, and the table class also needed additional constructors for foreign key tables.

CATALOG: For catalog, I began by creating a special index file that gets created with every new databases. This file simply contains the names of all databases in the system, and it is updated when new database are added/updated/deleted. I then created a way for users to query all the names of database in the system, and then to access the names of all tables within a databases, to view specific tables in full or by column, and finally to view the tables with their column types displayed.

USER INTERFACE: The user interface is currently console based, but as all print commands are passed through the display class, this could be easily updated for a graphics display. On start, the program access the database index and prints out all available databases in the system. It prompts the user to either create a new database or use an existing one. Once a database is selected, the user may list tables, view tables, view column types, or select specific columns from tables.

NEXT STEPS: If I had more time, I would like to make an SQL (or equivalent format) class to act as a parser. I then extend my file system to clone itself and update the clone with every change during a session, so that sudden crashes won't result in a loss of data. I would also like to use the echo command to capture all user input, and thus create a replicate-able record of the session to serve as a transaction for added data protection against crashes.