**CSS 448 A1**

**Converting SQL Schema to NoSQL Schema**

**Stefan Vercillo**

**Feb. 7, 23**

**Motivation:**

The goal of this assignment is to convert a SQL schema into a NoSQL schema. The NoSQL schema that I have elected to convert into is JSON. My program takes a “pseudo-SQL schema” file as defined in the question as input, parses the strings of this file into a data structure, and then convert this data structure into a valid JSON string which is outputted to a file.

Hence there are two key components of this project, the parsing of input and the dumping of the data in a NoSQL type format. In order to understand both of these components, it is important to understand the underlying data structure and the hierarchy that data is stored in. This will give insight into how the data is stored, and how it efficiently stores data for purposes of this exercise.

**Data Structure:**

The lowest level of storage is a Column. In SQL, a table is represented by records which adhere to a particular table schema. Each record is a tuple of column values in the order specified by the table schema. Additionally, the type of each column must adhere to the type specified by the table schema. However, in this assignment there is no type specified with columns by default. Below is my representation of a column via the Column struct:

struct Column

{

public:

std::string column\_name;

ColumnTypes column\_type;

bool is\_nullable;

Column(std::string column\_name){

this->is\_nullable = true;

this->column\_name = column\_name;

this->column\_type = COLUMNTYPES\_UNKNOWN;

}

Column(std::string column\_name, ColumnTypes column\_type){

this->is\_nullable = true;

this->column\_name = column\_name;

this->column\_type = column\_type;

}

Column(){}

};

Each column has a name which characterizes the data that it represents. In addition to the column name, there is a column type which was mentioned previously and a nullable flag. This flag distinguishes whether or not a record’s data for the given column is necessarily not nullable. I have interpreted the assignment such that by default, all columns are nullable, which is the default behaviour in SQL. Column types are an Enum that represents the possible column type values.

For a given schema, there are also keys:

struct Key

{

public:

std::vector<Column \*> key\_columns;

};

A key is a virtual class that contains key columns. A key is inherited by primary keys and foreign keys. Key columns represent the columns that a key span, in order represented by a vector. This is part of the bonus part of this lab. As defined in the sample schema, keys (both primary and foreign) only span a single column, however this design allows keys to span multiple. A vector is used tor represent key columns because it preserves the insertion ordering.

struct PrimaryKey : public Key{

public:

bool is\_present;

PrimaryKey()

{

is\_present = false;

}

};

A primary key is the simplest implementation of a key, and the only additional field that exists is a flag indicating whether or not a primary key is present or not. This feature also extends into the bonus material of this assignment. I have elected to represent primary key as part of a table which is a higher-level data structure.

struct ForeignKey : public Key

{

public:

std::vector<Column\*> reference\_columns;

Table \* reference\_table;

Table \* table;

ForeignKey() {}

};

Foreign keys, like primary keys, span a certain number of columns (in a particular order). However, they additionally reference another table and its columns. Each column that spans a foreign key must match reference a column in a secondary reference table. The reference table is represented as a pointer to a table object. In addition to key columns, there are reference columns as mentioned. Foreign keys as represented in my project, exist outside of the context of a table, because foreign key statements can be separate from the table creation statement.

struct Table{

public:

std::vector<Column\*> column\_vals;

std::unordered\_map<std::string, Column \*> columns;

PrimaryKey \* primary\_key;

std::string table\_name;

Table(){}

Table(std::string table\_name){

this->table\_name = table\_name;

}

};

Tables are represented as above. There is a vector of column values, in addition to a map of columns. This allows columns to preserve their schema ordering while maintaining constant time access to these columns. Columns maps use column names as an index so we can see if a column name already exists in a table. There is a pointer to the primary key of the table. The foreign keys are not stored here but in the schema.

The highest level of storage is a Schema. A Schema pertains to a particular file and all the data and meta-data associated with it:

struct Schema{

public:

std::vector<Table \*> table\_vals;

std::unordered\_map<std::string, Table \*> tables; // tables accesible by name

std::unordered\_map<std::string, std::vector<ForeignKey\*> > foreign\_keys; // foreign keys

int schema\_id;

Schema() : schema\_id(1) {}

};

Each schema pertains to a particular file. Each schema can have multiple tables that are stored in order. Additionally, there is tables map that allows for constant lookup time on a table name. There is also a map of vectors of foreign keys. This allows to access a list of foreign keys per table name.

**Algorithms Used:**

Parsing: -> Fill Schema From Text File

A file is passed to the relational schema parser, which fills a schema. In order to do this as efficiently as possible, this parser fills out the schema data structure in one pass copying data as infrequently as possible.

The parser iterates through every character in the file passed. Based on a set of delimiters the bounds of a word are established. Words are then taken as a substring from the initial contents. In order to determine what type of word a word is, a state machine using the last word and Boolean flags is used. This state machine / directed graph looks at a word and determines the type of word it is, and then does the appropriate action with this word. For example if the word value is “TABLE” then we know that the next word is going to be the name of a new table.

The state machine I implemented is particularly complex because it incorporates the possibility of foreign keys being present in a table definition. Additionally, I have written this state machine to allow for primary and foreign keys that span multiple columns. For example:

*TABLE DEPARTMENT*

*(Dname,Dnumber,Mgr\_ssn,Mgr\_start\_date,*

*FOREIGN KEY (Mgr\_ssn, DNAME) REFERENCES EMPLOYEE(MGR\_SSN, DNAME),*

*FOREIGN KEY (DNAME) REFERENCES EMPLOYEE(DNAME),*

*PRIMARY KEY(Dnumber));*

the above schema is considered valid in my project to allow for a wider breadth of schemas to be possibly interpreted. This is part of the bonus question as it extends upon the ideas initially suggested in the assignment.

Additionally:

*TABLE DEPARTMENT*

*(*

*PRIMARY KEY(Dnumber),*

*FOREIGN KEY (Mgr\_ssn, DNAME) REFERENCES EMPLOYEE(MGR\_SSN, DNAME),*

*FOREIGN KEY (DNAME) REFERENCES EMPLOYEE(DNAME),*

*Dname,Dnumber,Mgr\_ssn,Mgr\_start\_date*

*);*

is also considered a valid schema. This demonstrates how the order of the columns does not matter, the primary and foreign keys are still considered valid if they are defined before the columns of a table are defined.

On the other hand, for the below the relation schema:

*TABLE EMPLOYEE(Ssn, PRIMARY KEY( COLUMN\_THAT\_DOES\_NOT\_EXIST)));*

Will halt and throw an error:

new@MacBook-Pro a1 % make FILE\_SUFFIX=5 run

g++ -std=c++11 -c src/main.cpp -o obj/main.o -Wno-c++11-extensions

g++ obj/json\_dumper.o obj/main.o obj/relational\_schema\_parser.o -o bin/main

./bin/main data/relational\_schemas/relational\_schema5.txt data/json\_schemas/json\_schema5.json

PRIMARY KEY COL COLUMN\_THAT\_DOES\_NOT\_EXIST IS NOT A VALID COLUMN

make: \*\*\* [run] Abort trap: 6

new@MacBook-Pro a1 %

Things exemplifies how in order to create a primary key; the primary key cols must exist on the table. The above is true as well for foreign keys. However there is an additional check on reference columns as well:

i.e. for the schema

*TABLE EMPLOYEE(EID);*

*TABLE DEPARTMENT(MGR\_SSN);*

*FOREIGN KEY (DEPARTMENT(Mgr\_ssn) REFERENCES EMPLOYEE(Ssn));*

The below error is thrown:

./bin/main data/relational\_schemas/relational\_schema6.txt data/json\_schemas/json\_schema6.json

FOREIGN KEY COL SSN IS NOT A VALID COLUMN

make: \*\*\* [run] Abort trap: 6

The is also an error that is thrown if the foreign key is created on a table that does not exist.

I.e. for the schema

*TABLE EMPLOYEE(EID);*

*TABLE DEPARTMENT(MGR\_SSN);*

*FOREIGN KEY (DEPARTMENT(Mgr\_ssn) REFERENCES TABLE\_DOESNT\_EXIST(EID));*

The following error is thrown:

./bin/main data/relational\_schemas/relational\_schema6.txt data/json\_schemas/json\_schema6.json

FOREIGN KEY REFERENCES TABLE WHICH DOES NOT EXIST

make: \*\*\* [run] Abort trap: 6

**Dumping: -> Converting the Data Structure to JSON**

Below is the JSON schema structure that I have chosen to serialize my data into.

{

"tables" : [

{

"table\_name" : "$table\_name",

"columns" : [

{

"column\_name" : "$column\_name",

"not\_null" : true,

"column\_type" : "$column\_type"

}

....

],

"primary\_key" : {

"present" : true,

"column\_names" : [

"$column\_name1",

...

]

},

"foriegn\_keys" : [

{

"key\_columns\_names" : [

"$column\_name"

..

],

"reference\_table" : "$table\_name",

"reference\_column\_names" : [

"$column\_name"

...

]

}

]

}

]

}

Primary keys have an array of columns which represents the columns that a key span in the order specified by the relational schema. This is true also for foreign keys key columns and their reference columns. If a primary key exists, then the present flag is set to true. Additionally, each column contains information about the type of column. If this information is not known, then the column type appears as unknown.

**Supported Query Examples:**

Below are some examples of input relational schemas that will be supported and the output schema that the program will produce.

INPUT 1:

*TABLE EMPLOYEE(Ssn, MGR\_SSN, DNAME, PRIMARY KEY(Ssn));*

*TABLE DEPARTMENT*

*(Dname,Dnumber,Mgr\_ssn,Mgr\_start\_date,*

*FOREIGN KEY (Mgr\_ssn, DNAME) REFERENCES EMPLOYEE(MGR\_SSN, DNAME),*

*FOREIGN KEY (DNAME) REFERENCES EMPLOYEE(DNAME),*

*PRIMARY KEY(Dnumber));*

OUTPUT 1:

{

"schema\_id" : 1,

"tables" : [

{

"table\_name": "EMPLOYEE",

"columns": [

{

"column\_name" : "SSN",

"not\_null" : "true",

"column\_type" : "unknown"

},

{

"column\_name" : "MGR\_SSN",

"not\_null" : "true",

"column\_type" : "unknown"

},

{

"column\_name" : "DNAME",

"not\_null" : "true",

"column\_type" : "unknown"

}

],

"primary\_key" : {

"present" : "true",

"column\_names" : [

"SSN"

]

},

"foriegn\_keys" : [

]

},

{

"table\_name": "DEPARTMENT",

"columns": [

{

"column\_name" : "DNAME",

"not\_null" : "true",

"column\_type" : "unknown"

},

{

"column\_name" : "DNUMBER",

"not\_null" : "true",

"column\_type" : "unknown"

},

{

"column\_name" : "MGR\_SSN",

"not\_null" : "true",

"column\_type" : "unknown"

},

{

"column\_name" : "MGR\_START\_DATE",

"not\_null" : "true",

"column\_type" : "unknown"

}

],

"primary\_key" : {

"present" : "true",

"column\_names" : [

"DNUMBER"

]

},

"foriegn\_keys" : [

{

"key\_column\_names" : [

"MGR\_SSN",

"DNAME"

],

"reference\_table" : "EMPLOYEE",

"reference\_column\_names" : [

"MGR\_SSN",

"DNAME"

]

},

{

"key\_column\_names" : [

"DNAME"

],

"reference\_table" : "EMPLOYEE",

"reference\_column\_names" : [

"DNAME"

]

}

]

}

]

}

INPUT 2:

*TABLE EMPLOYEE(Fname,Minit,Lname,Ssn,Bdate,Address,Sex,Salary,Super\_ssn,Dno,PRIMARY KEY(Ssn));*

*TABLE DEPARTMENT(Dname,Dnumber,Mgr\_ssn,Mgr\_start\_date);*

*FOREIGN KEY (DEPARTMENT(Mgr\_ssn) REFERENCES EMPLOYEE(Ssn));*

*TABLE DEPT\_LOCATIONS(Dnumber,Dlocation,PRIMARY KEY(Dnumber, Dlocation));*

{

"schema\_id" : 1,

"tables" : [

{

"table\_name": "EMPLOYEE",

"columns": [

{

"column\_name" : "FNAME",

"not\_null" : "true",

"column\_type" : "unknown"

},

{

"column\_name" : "MINIT",

"not\_null" : "true",

"column\_type" : "unknown"

},

{

"column\_name" : "LNAME",

"not\_null" : "true",

"column\_type" : "unknown"

},

{

"column\_name" : "SSN",

"not\_null" : "true",

"column\_type" : "unknown"

},

{

"column\_name" : "BDATE",

"not\_null" : "true",

"column\_type" : "unknown"

},

{

"column\_name" : "ADDRESS",

"not\_null" : "true",

"column\_type" : "unknown"

},

{

"column\_name" : "SEX",

"not\_null" : "true",

"column\_type" : "unknown"

},

{

"column\_name" : "SALARY",

"not\_null" : "true",

"column\_type" : "unknown"

},

{

"column\_name" : "SUPER\_SSN",

"not\_null" : "true",

"column\_type" : "unknown"

},

{

"column\_name" : "DNO",

"not\_null" : "true",

"column\_type" : "unknown"

}

],

"primary\_key" : {

"present" : "true",

"column\_names" : [

"SSN"

]

},

"foriegn\_keys" : [

]

},

{

"table\_name": "DEPARTMENT",

"columns": [

{

"column\_name" : "DNAME",

"not\_null" : "true",

"column\_type" : "unknown"

},

{

"column\_name" : "DNUMBER",

"not\_null" : "true",

"column\_type" : "unknown"

},

{

"column\_name" : "MGR\_SSN",

"not\_null" : "true",

"column\_type" : "unknown"

},

{

"column\_name" : "MGR\_START\_DATE",

"not\_null" : "true",

"column\_type" : "unknown"

}

],

"primary\_key" : {

"present" : "false"

},

"foriegn\_keys" : [

{

"key\_column\_names" : [

"MGR\_SSN"

],

"reference\_table" : "EMPLOYEE",

"reference\_column\_names" : [

"SSN"

]

}

]

},

{

"table\_name": "DEPT\_LOCATIONS",

"columns": [

{

"column\_name" : "DNUMBER",

"not\_null" : "true",

"column\_type" : "unknown"

},

{

"column\_name" : "DLOCATION",

"not\_null" : "true",

"column\_type" : "unknown"

}

],

"primary\_key" : {

"present" : "true",

"column\_names" : [

"DNUMBER",

"DLOCATION"

]

},

"foriegn\_keys" : [

]

}

]

}

INPUT 3:

*TABLE*

*EMPLOYEE(*

*Fname INT,*

*Minit FLOAT,*

*Lname TEXT,*

*Ssn FLOAT,*

*Bdate BYTE,*

*Address VARCHAR,*

*Sex VARCHAR,*

*Salary INT,*

*Super\_ssn INT,*

*Dno INT,*

*PRIMARY KEY(Ssn)*

*);*

OUTPUT 3:

{

"schema\_id" : 1,

"tables" : [

{

"table\_name": "EMPLOYEE",

"columns": [

{

"column\_name" : "FNAME",

"not\_null" : "true",

"column\_type" : "INT"

},

{

"column\_name" : "MINIT",

"not\_null" : "true",

"column\_type" : "FLOAT"

},

{

"column\_name" : "LNAME",

"not\_null" : "true",

"column\_type" : "TEXT"

},

{

"column\_name" : "SSN",

"not\_null" : "true",

"column\_type" : "FLOAT"

},

{

"column\_name" : "BDATE",

"not\_null" : "true",

"column\_type" : "BYTE"

},

{

"column\_name" : "ADDRESS",

"not\_null" : "true",

"column\_type" : "VARCHAR"

},

{

"column\_name" : "SEX",

"not\_null" : "true",

"column\_type" : "VARCHAR"

},

{

"column\_name" : "SALARY",

"not\_null" : "true",

"column\_type" : "INT"

},

{

"column\_name" : "SUPER\_SSN",

"not\_null" : "true",

"column\_type" : "INT"

},

{

"column\_name" : "DNO",

"not\_null" : "true",

"column\_type" : "INT"

}

],

"primary\_key" : {

"present" : "true",

"column\_names" : [

"SSN"

]

},

"foriegn\_keys" : [

]

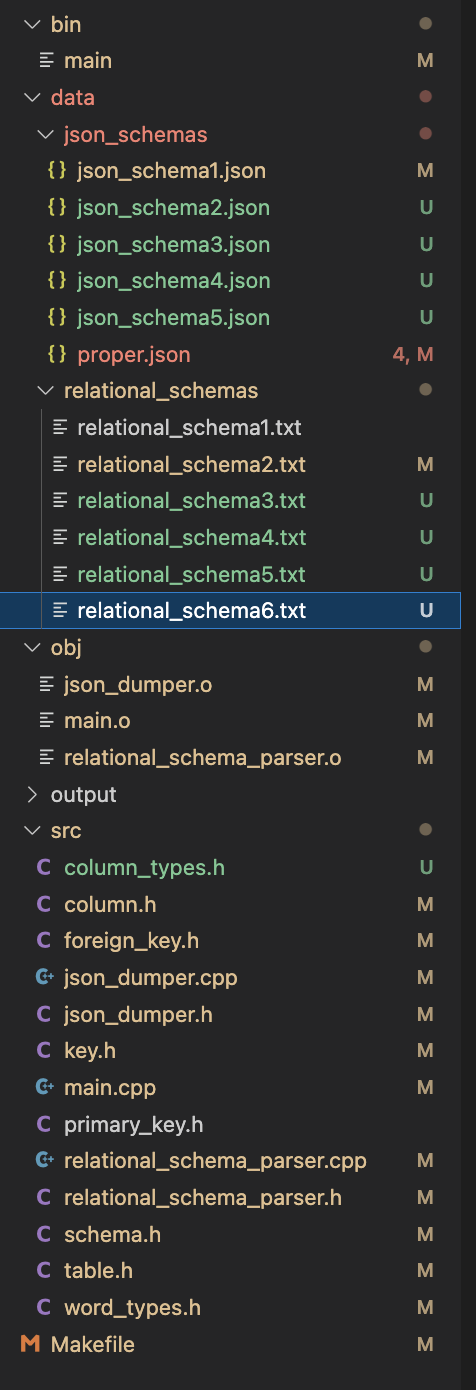
}

]

}

**How to Run Project:**

Given the project files attached, create the following file structure:

****

Alternatively, the zip file that has been provided can be unzipped. Ensure that g++ is installed and this project is run on a Linux environment. Then from the root directory, run the following command:  
  
  
```

make FILE\_SUFFIX=${FILE\_SIFFIX} run

```

where FILE\_SUFFIX is a number between 1 and 6, indicating which of the relational schema files that will be interpreted as the input. The FILE\_SUFFIX also dictates where the output is written to.

For example, the command:

```

make FILE\_SUFFIX=1 run

```

will generate an output file at “data/json\_schemas/json\_schema1.json” given there are no errors with the input schema.

**Additional Bonus Material:**

All of the bonus material has already been covered with strong justification. I believe by this point it is evident that the features of my implementation go well above and beyond the expectations of this assignment.

**External Algorithms Used:**

No external sources were used in the making of the project. Novel algorithms were used.

**Partners:**

I did not have a partner for this lab as I joined late and was unable to find one. All verification was done myself.