**CSS 448 A2**

**Query Optimization**

**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

**Cost Calculation:**

For Select:

Based on the selection criteria stated in the Selection object, the calculate state function's goal is to establish the best method for picking data from a particular table. To accomplish this, the code first determines the current state of the table by repeatedly invoking the calculate state method on the Selection object's child node.

The function then determines the range of values for the chosen column, which is then used to calculate an estimate of how many matching tuples there are. If no prior selection has been made, the range is either derived from the previously applied selection on the column or from the database statistics. Following that, the matching tuples are utilised to determine the matching pages, or the number of pages that must be read.

The function then goes on to assess several methods for choosing the data based on matching pages. The different algorithms studied include sorted file eq, hash eq, hash range, tree eq, and table scan. The most effective method for picking the data is the one with the lowest operational costs. On the premise that the cost of reading a page is constant, the number of pages required to retrieve the matching tuples determines the operation's cost.

The calculate state function makes a number of assumptions in order to determine the best selection strategy. For example, it assumes that simple equality selection does not have any overflow buckets, that the worst-case select column is not a candidate key, and that the index accounts for about 10% of the table's total number of pages. While assessing the accuracy of the selection technique chosen by the function, it's critical to keep these presumptions in mind.

In order to maximise the selection of data from a particular table, the calculate state function is a complicated piece of code that uses a number of different data structures and techniques. It is a significant part of a larger database management system, and the function's ability to accurately choose the selection technique is essential to the system's success.

For Joins:

Calculate\_state\_join\_size and calculate\_cost\_without\_optimization are two methods in the provided code. Both of these functions are designed to figure out how much it will cost to conduct a join operation on a relational database.

A Join object and two pointers to State objects that indicate the left and right children of the join operation are passed to the calculate\_state\_join\_size function. The function establishes a new State object to reflect the state of the output and determines the size of the join operation's output. The calculate\_cost\_without\_optimization function is then called with this new State object to determine the cost of the join action.

The same Join object and two pointers to State objects that symbolize the left and right children of the join operation are passed to the calculate\_cost\_without\_optimization function.

Using three different algorithms—sort-merge join, simple nested loop with left as outer, and simple nested loop with right as outer—the function determines the cost of conducting a join operation. The function compares the cost of each of these algorithms and chooses the one with the lowest cost. The Join object is then set with the ultimate join operation cost.

In general, these functions are used to choose the least expensive algorithm to optimize join processes in a relational database.

Calculating the cost of performing a join algorithm with the left side optimized is done using the specific method calculate\_cost\_optimizing\_left\_side(). A Join object depicting the join process, along with two State objects representing the states of the left and right children of the join, are the three inputs for the method.

The method's second step determines whether pipelining in index stacked loop join is feasible. It tests to see if the join column has a hash index or tree index as it iterates through the tables contained in the left child. The cost of executing the index nested loop join is computed and compared to the current minimum cost if the join column has a hash index. The cost of executing the index nested loop join is also computed and compared to the current lowest cost if the join column has a tree index.

The method's last step determines whether it is feasible to perform a sort merge join with a pipelined left table. The expense of performing the join on the left child's primary key will increase.

For Selection:

**How to Run Project:**

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

make run

```

ASSUMPTIONS:  
  
Assume that DB ranges are inclusives

Assume input1.A and input1.A will never be joined in ambiguous case.

ASSUMPTION: there are no overflow buckets on simple equality selection

ASSUMPTION: index is about 10% of table size in number of pages

cost of reading all pages that match, and getting each tuple in

ASSUMPTION: if range not in statistics, assume uniform distribution of range from 1->100

**Input:**

**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,PRIMARY KEY(Dnumber));**

**FOREIGN KEY (EMPLOYEE(Dno) REFERENCES DEPARTMENT(Dnumber));**

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

**FOREIGN KEY (DEPT\_LOCATIONS(Dnumber) REFERENCES DEPARTMENT(Dnumber));**

**OP1 = EMPLOYEE SELECTION Dno=5**

**OP2 = OP1 PROJECTION Bdate,Dno**

**OP3 = OP2 JOIN DEPARTMENT ON Dno=Dnumber**

**OP4 = OP3 SELECTION Dno=5**

**RESULT = OP4 PROJECTION Bdate,Mgr\_ssn**

**Cardinality(EMPLOYEE) = 1000000**

**Cardinality(DEPARTMENT) = 10**

**Cardinality(DEPT\_LOCATIONS) = 100**

**SIZE(EMPLOYEE) = 50000**

**SIZE(DEPARTMENT) = 1**

**SIZE(DEPT\_LOCATIONS) = 1**

**RF(Fname in EMPLOYEE) = 0.1**

**RF(Minit in EMPLOYEE) = 0.0025**

**RF(Lname in EMPLOYEE) = 0.1**

**RF(Ssn in EMPLOYEE) = 0.0001**

**RF(Bdate in EMPLOYEE) = 0.004**

**RF(Address in EMPLOYEE) = 0.1**

**RF(Sex in EMPLOYEE) = 0.5**

**RF(Salary in EMPLOYEE) = 0.1**

**RF(Super\_ssn in EMPLOYEE) = 0.001**

**RF(Dno in EMPLOYEE) = 0.001**

**RF(Dname in DEPARTMENT) = 0.1**

**RF(Dnumber in DEPARTMENT) = 0.1**

**RF(Mgr\_ssn in DEPARTMENT) = 0.1**

**RF(Mgr\_start\_date in DEPARTMENT) = 0.3**

**RF(Dnumber in DEPT\_LOCATIONS) = 0.1**

**RF(Dlocation in DEPT\_LOCATIONS) = 0.02**

**Cardinality(Ssn in EMPLOYEE) = 10000**

**SIZE(Ssn in EMPLOYEE) = 100**

**Range(Ssn in EMPLOYEE) = 1,10000**

**Cardinality(Dnumber in DEPARTMENT) = 10**

**SIZE(Dnumber in DEPARTMENT) = 1**

**RANGE(Dnumber in DEPARTMENT) = 1,10**

**Cardinality((Dnumber, Dlocation) in DEPT\_LOCATIONS) = 100**

**SIZE((Dnumber, Dlocation) in DEPT\_LOCATIONS) = 1**

**Cardinality(Dno in EMPLOYEE) = 10**

**SIZE(Dno in EMPLOYEE) = 100**

**Range(Dno in EMPLOYEE) = 1,10**

**Height(Dno in EMPLOYEE) = 4**

**Cardinality(Dnumber in DEPT\_LOCATIONS) = 10**

**SIZE(Dnumber in DEPT\_LOCATIONS) = 1**

**Range(Dnumber in DEPT\_LOCATIONS) = 1,10**

**Height(Dnumber in DEPT\_LOCATIONS) = 1**

**Output:   
  
Total Query Cost (unoptimized): 74 I/Os**

**Cost Breakdown:**

**Operation ->RESULT:**

**Type: projection**

**Operation Description: Projection operation done on the fly.**

**Cost of Operation (in I/Os): 0**

**Number of Output Pages: 1**

**Operation ->OP4:**

**Type: selection**

**Operation Description: A full table scan was performed to match this selection.**

**Cost of Operation (in I/Os): 1**

**Number of Output Pages: 1**

**Operation ->OP3:**

**Type: join**

**Operation Description: Hash index nested join with the right table pipelined was performed, as join occurs on indexed column.**

**Cost of Operation (in I/Os): 19**

**Number of Output Pages: 1**

**Operation ->OP2:**

**Type: projection**

**Operation Description: Projection operation done on the fly.**

**Cost of Operation (in I/Os): 0**

**Number of Output Pages: 10**

**Base Table ->DEPARTMENT**

**Number of Pages: 1**

**Operation ->OP1:**

**Type: selection**

**Operation Description: Tree equality was performed to match this selection.**

**Cost of Operation (in I/Os): 54**

**Number of Output Pages: 50**

**Base Table ->EMPLOYEE**

**Number of Pages: 50000**

**Query Tree Visualization:**

**RESULT**

**|**

**OP4**

**|**

**-------------- OP3 -------------**

**| |**

**OP2 DEPARTMENT**

**|**

**OP1**

**|**

**EMPLOYEE**

**OPTMIZING QUERY ....**

**Total Query Cost (optimized): 69 I/Os**

**Cost Breakdown:**

**Operation ->RESULT:**

**Type: projection**

**Operation Description: Projection operation done on the fly.**

**Cost of Operation (in I/Os): 0**

**Number of Output Pages: 1**

**Operation ->OP3:**

**Type: join**

**Operation Description: Hash index nested join with the left table pipelined was performed, as join occurs on indexed column.**

**Cost of Operation (in I/Os): 14**

**Number of Output Pages: 1**

**Operation ->PUSHED\_OP4\_LEFT:**

**Type: selection**

**Operation Description: A full table scan was performed to match this selection.**

**Cost of Operation (in I/Os): 10**

**Number of Output Pages: 10**

**Base Table ->DEPARTMENT**

**Number of Pages: 1**

**Operation ->OP2:**

**Type: projection**

**Operation Description: Projection operation done on the fly.**

**Cost of Operation (in I/Os): 0**

**Number of Output Pages: 10**

**Operation ->OP1:**

**Type: selection**

**Operation Description: Tree equality was performed to match this selection.**

**Cost of Operation (in I/Os): 54**

**Number of Output Pages: 50**

**Base Table ->EMPLOYEE**

**Number of Pages: 50000**

**Query Tree Visualization:**

**RESULT**

**|**

**-------------- OP3 -------------**

**| |**

**PUSHED\_OP4\_LEFT DEPARTMENT**

**|**

**OP2**

**|**

**OP1**

**|**

**EMPLOYEE**