BSQL

Instruction Manual

A query language for business

Developed by: Kyle Sokol

Background:

I wanted to create a domain-specific language written in C++. I decided to create BSQL as a friendly query tool for business professionals. As a computer science major, we learn about how to use SQL, but this tool has applications in professions far beyond software engineering. SQL is used by financial professionals and accountants, data scientists, manufacturing plant managers, and many more. Despite, the widespread use of SQL, proper training of SQL is heavily skewed toward those with technical degrees like computer science.

I got inspiration for this idea from my twin brother who is a business major at the University of Kentucky. He uses Microsoft Excel and Salesforce on a daily basis both for school and his internship. He has become very fluent with these tools, but they have their limitations. Some of the problems he will come to me with are very simple SQL queries, but a very complicated Excel formula. Because he has not been taught SQL in school, he sticks to Excel.

Goal:

My goal is to provide a much-more user-friendly syntax for SQL to make it easier for non-technical people to adopt. This language will implement a lot of the major features of SQL but not all of them. I only choose the features I thought that would be feasible to implement in the given time frame and would be of most use to business majors. Furthermore, the syntax may be a bit more verbose than SQL, but this language is intended as a learning tool, so the extra typing is justified for clarity’s sake.

How it works:

My program takes in user queries and converts them to an SQL query. If the user would like to run a function that is not possible in BSQL, they can modify the SQL statement that is outputted. Since SQL Database management systems are so widespread, I wanted BSQL to be compatible with these technologies so that’s why I convert the BSQL to SQL. Furthermore, SQL does not normally provide a REPL. In BSQL, if the user chooses to use stdin as their input, the REPL will guide them through the process of generating the query. However, if the user would like to run queries using a file, that is an option as well.

Supported Functionality:

While the user does not type SQL syntax, it is converted to SQL for execution. The purpose of this is so that this language can be used with existing SQL services that are widespread. The equivalent SQL functions that BSQL supports are SELECT, FROM, WHERE, INNER JOIN, ORDERBY, GROUPBY, HAVING, INSERT INTO, ALTER, DELETE, CREATE.

For joins, I only support inner-joins as this is the most common join from my experience. Adding the functionality for other joins would complicate the syntax more than desired. Again, any omission from BSQL can be accounted for by modifying the resulting SQL query.

BSQL also supports aggregate functions including: count, sum, max, min, and avg

SQL Equivalencies:

While the syntax for BSQL is quite a bit different from SQL, the syntax is intended to be equivalent to SQL. Here are the equivalencies:

* QUERY\_TYPE -> SELECT, DROP, DELETE,ALTER, CREATE…
* TABLES/TABLE\_NAME -> FROM
* MATCHING\_COULUMNS -> INNER JOIN ON
* FILTERS -> WHERE / HAVING
* RESULTS\_ORDER: ORDER BY
* COLUMNS\_DISPLAYED: SELECT
* COLUMN\_GROUPING: GROUP BY

Other Notes:

* All keywords are case insensitive.
* No commas are used in the language, whitespace is used instead.
* All columns must be fullly-identified, meaning all column names will have their table name precede them (eg. Employee.ssn vs ssn).
* No comments are allowed.
* For join statements, the tables listed must be in the same order as they appear in matching columns or you will get invalid results.
* Nested queries are not supported.
* Nested logical operators are not supported (e.g. (true and (true or false)) \*would like to implement this but ran out of time

Grammar:

High-Level Grammar

<QUERY> = "QUERY\_TYPE:" <QUERY\_TYPE>

<QUERY\_TYPE> = <QUESTION> | <DELETION> | INSERTION

Question/Select Statement Grammar

<QUESTION> = "QUESTION: " <TABLES> <MATCHING\_COLUMNS> <FILTERS> <RESULTS\_ORDER> <COLUMNS\_DISPLAYED> <COLUMN\_GROUPING>

<TABLES> "TABLES: " (<IDENTIFIER>)+

<MATCHING\_COLUMNS> = "MATCHING\_COLUMNS:" (<COLUMN\_RELATION>)+ | "MATCHING\_COLUMNS: NONE"

<FILTERS> = "FILTERS:" (<COLUMN> <CONDITIONAL\_OPERATOR> <LITERAL\_VALUE> <OPTIONAL\_FILTER>) |

"FILTERS:"( <COLUMN> <CONDITIONAL\_OPERATOR> <COLUMN> <OPTIONAL\_FILTER>) |

"FILTERS: GROUPING(" <COLUMN> <CONDITIONAL\_OPERATOR> <LITERAL\_VALUE> <OPTIONAL\_FILTER> ")") |

"FILTERS: GROUPING(" <COLUMN> <CONDITIONAL\_OPERATOR> <COLUMN> <OPTIONAL\_FILTER> ")")|

"NONE"

<OPTIONAL\_FILTER> = <LOGICAL\_OPERATOR> <FILTERS> | empty <COLUMN\_RELATION> = ("(" <COLUMN> "=" <COLUMN> ")")

<RESULTS\_ORDER> = "RESULTS\_ORDER:" "DEFAULT" |

("("<COLUMN> <ORDER\_TYPE>")")+

<ORDER\_TYPE> = "asc" | "desc"

<COLUMNS\_DISPLAYED> = "COLUMNS\_DISPLAYED:" (<COLUMN>)+ |

"COLUMNS\_DISPLAYED:" ("DISTINCT" <COLUMN>)+ | "COLUMNS\_DISPLAYED:" (<AGGREGATE> "(" <COLUMN> ")")+ | "COLUMNS\_DISPLAYED:"(<AGGREGATE> "(DISTINCT" <COLUMN> ")")+ | "COLUMNS\_DISPLAYED: ALL"

<COLUMN\_GROUPING> = "COLUMN\_GROUPING:" (<COLUMN>)+ | "NONE"

<COLUMN> = <IDENTIFIER>"."<IDENTIFIER> | "ALL"

<LOGICAL\_OPERATOR> = "and" | "or"

<CONDITIONAL\_OPERATOR> = "=" | ">" | "<" | ">=" | "<=" | "<>"

<LITERAL\_VALUE> = <STRING> | <NUMBER> | “NULL”

<AGGREGATE> = "COUNT" | "MIN" | "MAX" | "AVG" | "SUM"

Delete Grammar:

<DELETION> = "DELETE DELETE\_OBJECT:"<DELETE\_OBJECT>

<DELETE\_OBJECT> = <IDENTIFIER> <FILTERS> | <COLUMN>

Insert Grammar:

<INSERTION> "INSERT INSERT\_OBJECT:"<INSERT\_OBJECT>

<INSERT\_OBJECT> = <INSERT\_TABLE> | <INSERT\_ROW> | <INSERT\_COLUMN>

<INSERT\_ROW> = "ROW" <ROW\_COLUMN\_INSERTION> <VALUE\_INSERTION>

<ROW\_COLUMN\_INSERTION> = "TABLE\_NAME: " <IDENTIFIER>

<VALUE\_INSERTION> = "VALUES:" ("(" <COLUMN> <LITERAL\_VALUE> ")")+ | "VALUES:" ("(" <COLUMN> "NULL)")+

<COLUMN\_INSERTION> "COLUMNS:" ("(" <IDENTIFIER> "LETTERS )"+ | "(" <IDENTIFIER> "NUMBER )")+

<SINGLE\_COLUMN\_INSERT> "COLUMN:" ("(" <IDENTIFIER> "LETTERS )" | "(" <IDENTIFIER> "NUMBER )")

<INSERT\_TABLE> = "TABLE TABLE\_NAME:"<IDENTIFIER> <COLUMN\_INSERTION>

<INSERT\_COLUMN> = "COLUMN" <ROW\_COLUMN\_INSERTION> <SINGLE\_COLUMN\_INSERT>

Basics:

<IDENTIFIER> → ALPHA ( ALPHA | DIGIT )\* ;

<ALPHA> → "a" ... "z" | "A" ... "Z" | "\_" ;

<DIGIT> → "0" ... "9" ;

<NUMBER> → DIGIT+ ( "." DIGIT+ )? ;

<STRING> → "\’" <any char>\* "\’" ;

Note\*: Examples of the syntax have been included in the next section

Examples:

In this section, we review some example BSQL queries and their corresponding SQL query that is produced by running the program. I also show the results of some of these queries. The data tables I use for this query can be found under the *data\_tables* directory.

QUESTION EXAMPLES:

1.

SQL Query: SELECT \* FROM EMPLOYEE;

QUERY\_TYPE: QUESTION

TABLES: EMPLOYEE

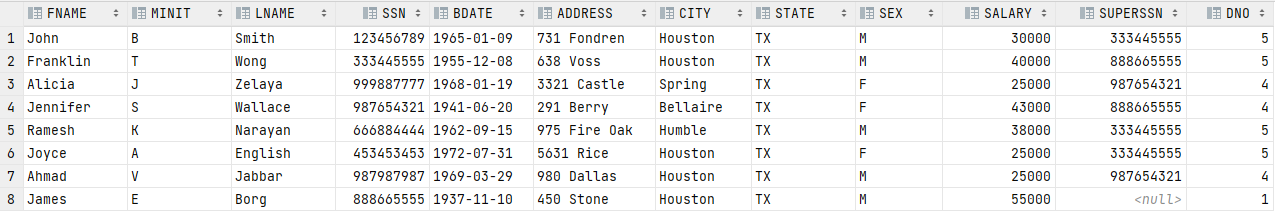
MATCHING\_COLUMNS: NONE

FILTERS: NONE

RESULTS\_ORDER: DEFAULT

COLUMNS\_DISPLAYED: ALL

COLUMN\_GROUPING: NONE



2.

SQL Query: SELECT EMPLOYEE.LNAME, EMPLOYEE.FNAME, EMPLOYEE.SALARY FROM EMPLOYEE where EMPLOYEE.DNO = 5

QUERY\_TYPE: QUESTION

TABLES: EMPLOYEE

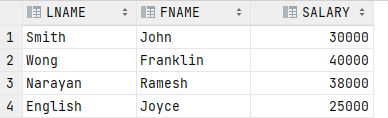
MATCHING\_COLUMNS: NONE

FILTERS: EMPLOYEE.DNO = 5

RESULTS\_ORDER: DEFAULT

COLUMNS\_DISPLAYED: LNAME FNAME SALARY

COLUMN\_GROUPING: NONE



3.

SQL Query: SELECT \* FROM EMPLOYEE WHERE EMPLOYEE.DNO = 5 AND EMPLOYEE.SALARY >= 30000 ORDER BY EMPLOYEE.SALARY DESC, EMPLOYEE.LNAME ASC

QUERY\_TYPE: QUESTION

TABLES: EMPLOYEE

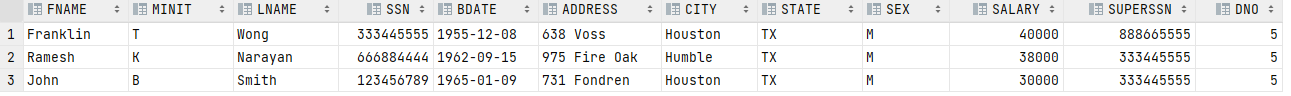
MATCHING\_COLUMNS: NONE

FILTERS: EMPLOYEE.DNO = 5 AND EMPLOYEE.SALARY >= 30000

RESULTS\_ORDER: (EMPLOYEE.SALARY DESC)(EMPLOYEE.LNAME ASC)

COLUMNS\_DISPLAYED: ALL

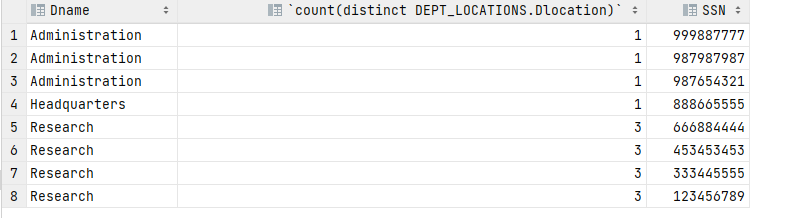
COLUMN\_GROUPING: NONE



4.

SQL Query: SELECT DEPARTMENT.Dname, count(distinct DEPT\_LOCATIONS.Dlocation), EMPLOYEE.SSN FROM DEPARTMENT JOIN DEPT\_LOCATIONS ON DEPARTMENT.Dnumber = DEPT\_LOCATIONS.Dnumber JOIN EMPLOYEE ON DEPT\_LOCATIONS.Dnumber = EMPLOYEE.DNO GROUP BY EMPLOYEE.SSN, DEPARTMENT.Dname ORDER BY EMPLOYEE.SSN desc, DEPARTMENT.Dname asc

QUERY\_TYPE: QUESTION  
TABLES: DEPARTMENT DEPT\_LOCATIONS EMPLOYEE  
MATCHING\_COLUMNS: (DEPARTMENT.Dnumber = DEPT\_LOCATIONS.Dnumber) (DEPT\_LOCATIONS.Dnumber = EMPLOYEE.DNO)  
FILTERS: NONE  
RESULTS\_ORDER: (EMPLOYEE.SSN desc) (DEPARTMENT.Dname asc)  
COLUMNS\_DISPLAYED: DEPARTMENT.Dname count (distinct DEPT\_LOCATIONS.Dlocation) EMPLOYEE.SSN  
COLUMN\_GROUPING: EMPLOYEE.SSN DEPARTMENT.Dname



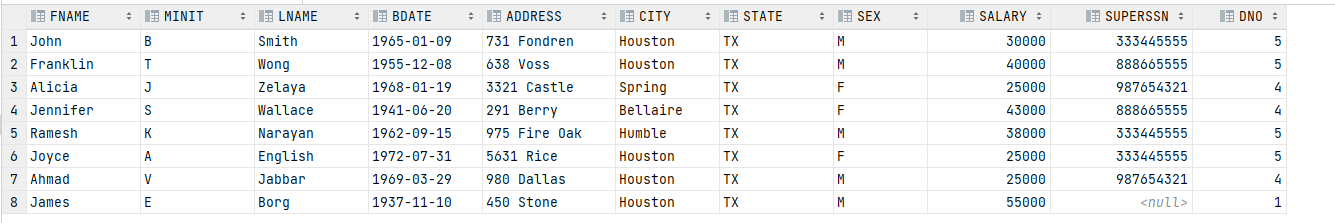
DELETE EXAMPLES:

5.

SQL Query: ALTER TABLE EMPLOYEE DROP COLUMN SSN

QUERY\_TYPE: DELETE

DELETE\_OBJECT: EMPLOYEE.SSN



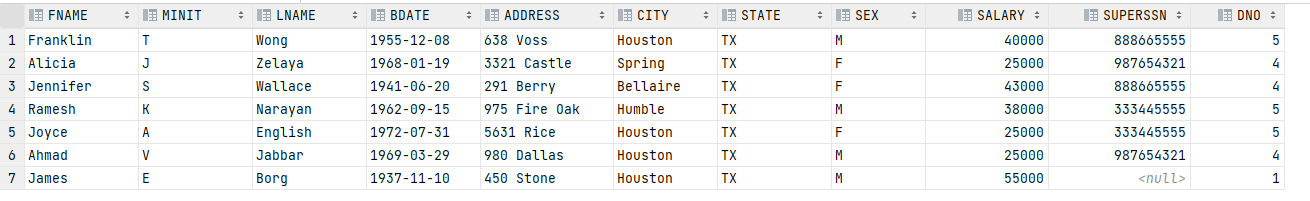
6.

SQL Query: DELETE FROM EMPLOYEE WHERE EMPLOYEE.FNAME = 'JOHN'

QUERY\_TYPE: DELETE

DELETE\_OBJECT: EMPLOYEE

FILTERS: EMPLOYEE.FNAME='JOHN’



7.

SQL Query: DROP EMPLOYEE

QUERY\_TYPE: DELETE

DELETE\_OBJECT: EMPLOYEE

INSERT EXAMPLES:

8.

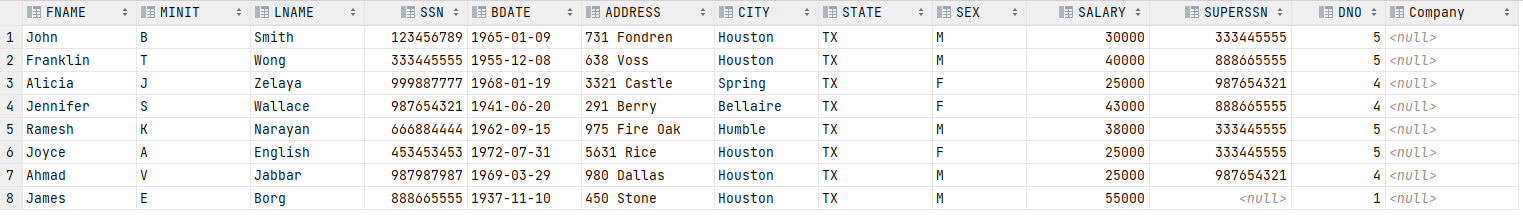
SQL Query: ALTER TABLE Employee ADD Company varchar(255)

QUERY\_TYPE: INSERT

INSERT\_OBJECT: COLUMN

TABLE\_NAME: Employee

COLUMN: (SSN LETTERS)



9.

SQL Query: CREATE TABLE EMPLOYEE\_2 (SSN int, ADDRESS varchar(255), COMPANY\_NAME varchar(255))

QUERY\_TYPE: INSERT

INSERT\_OBJECT: TABLE

TABLE\_NAME: EMPLOYEE

COLUMNS: (SSN NUMBER) (ADDRESS LETTERS) (COMPANY\_NAME LETTERS)



10.

SQL Query: INSERT INTO EMPLOYEE\_2 (SSN,ADDRESS,COMPANY\_NAME) VALUES (6,'123 Main st.', NULL)

QUERY\_TYPE: INSERT

INSERT\_OBJECT: ROW

TABLE\_NAME: EMPLOYEE

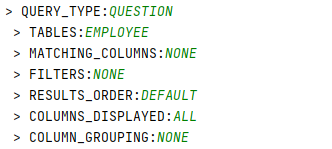
VALUES: (SSN 6) (ADDRESS'123 Main st.') (COMPANY\_NAME NULL)



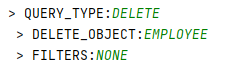
REPL:

One of the main advantages of BSQL is that it provides a REPL unlike normal SQL. The REPL guides the user through prompts to fill out and helps the user learn the language by minimizing the keywords the user must remember. The REPL is adaptive as the prompts align with the input given. The REPL is accessed when no command line arguments for a .bsql file are passed in. A few examples are shown below.











Tests:

If you would like a more complete overview of all of BSQL’s functionality, please refer to the tests that I have included.

I have included a folder called tests. In this folder:

* The *test\_inputs* directory contains test cases to be run by my C++ BSQL interpreter tester (tester.cpp).
* The *test\_results* directory contains the raw output from running each of the test cases.
* The *test\_expected\_output* directorycontains the expected/correct output for each test case.
* The results of all the tests can be found at All\_Result.txt.
* I included a few examples of syntax errors, but not all of them as there are far too many

Test Descriptions: (The most complete test of most functionality is test 4)

1. Aggregates: Tests the aggregate functionality for count, sum, avg, min, and max
2. Basic: Tests the most basic query (SELECT \* from Table)
3. Columns\_displayed: Tests selecting many columns
4. Combination: Tests almost all functionality in one query
5. Delete: Tests deleting tables, rows, and columns
6. Filters: Tests different filter/where conditions along with the Having/Grouping clause
7. Insert: Tests inserting tables, rows, and columns
8. Multiple\_tables: Tests the join functionality
9. Results\_order: Tests ordering of the query result and group by
10. Semantic\_error\_1: Tests operating on unknown table
11. Semantic\_error\_2: Joining non-unique tables
12. Semantic\_error\_3: Joining on the same table
13. Semantic\_error\_4: Aggregates in the Results\_Order clause
14. Semantic\_error\_5: Aggregates in the Column\_Grouping clause
15. Syntax\_error\_1: Missing query\_type
16. Syntax\_error\_2: Didn’t specify correct datatype for column insertion
17. Syntax\_error\_3: Missing ‘.’ Between table and column name
18. Syntax\_error\_4” Use “ instead of ‘

Running the Tests:

g++ -o tester.out -std=c++17 tester.cpp

./tester.out > All\_Result.txt 2>&1

\*The main program is BSQL.cpp, but the tester automatically compiles it and runs it for us.

\*Note it is normal for the compilation and tests to take over a minute.

\*If you would like to run just the program itself and not the tests you can use:

g++ BSQL.cpp global.cpp Scanner.cpp Parser.cpp -std=c++17

./a.out

Error Checking:

The implementation for BSQL I created mostly checks for syntactical errors and a few semantic errors. I was limited in identifying semantic errors, as many of these errors arise after the query has been run by the DBMS. For example, trying to run a select statement on a table that doesn’t exist would result in an error from the DBMS, but my implementation has no details on the actual tables in the database, and therefore, cannot identify the error correctly.

Final Remarks: Now that you have a basic understanding of BSQL, I hope you find this language a useful tool for stepping into the world of query languages. If you have any comments or questions about the language, contact me at kpsokol1@crimson.ua.edu