Design and Implementation of HSQL: A SQL-like language for Data Analysis in Distributed Systems

Anurag Singh Bhadauria[1]
Computer Science and Engineering
R. V. College of Engineering
Bengaluru, India

Atreya Bain[2]
Computer Science and Engineering
R. V. College of Engineering
Bengaluru, India

Prof. Jyoti Shetty[3]
Computer Science and Engineering
R. V. College of Engineering
Bengaluru, India

Dr. Shobha G.[4]
Computer Science and Engineering
R. V. College of Engineering
Bengaluru, India

Arjuna Chala[5]
Sr. Director, Innovative Technologies
LexisNexis Risk Solutions
Atlanta, U.S.A

Jeremy Clements[6]
Software Engineer III
LexisNexis Risk Solutions
Atlanta, U.S.A

***Abstract*—In today's modern world, we're experiencing a substantial increase in the use of data in various fields, and this has necessitated the use of distributed systems to consume and process Big Data. Machine learning tends to benefit from the usage of Big Data, and the models generated from such techniques tend to be more effective. However, there is a steep learning curve to getting used to handling Big Data, as traditional data management tools fail to perform well. Distributed systems have become popular, where the task of data processing is split amongst various nodes in clusters. SQL, is a popular database management language popular to data scientists. It is often given second class support, where SQL can be embedded into a primary language of use (e.g. SQL in Scala for Spark), which allows for using SQL but one still needs to know the primary language of the platform (Scala, as per the example, or ECL in HPCC Systems). It may also be present as a supported language. In either case, using useful tooling such as Visualizing data and creating and using machine learning models become difficult, as the user needs to fall back to the primary language of the system. In the proposed work, a new SQL-like language, HSQL, an open source distributed systems solution, was developed for allowing new users to get used to its distributed architecture and the ECL language, with which it primarily operates with (which was chosen as a target). Additionally, a program that could translate HSQL-based programs to ECL for use was made. HSQL was made to be completely inter-compatible with ECL programs, and it was able to provide a compact and easy to comprehend SQL-like syntax for performing general data analysis, creation of Machine learning models and visualizations while allowing a modular structure to such programs.**

***Keywords*—ANTLR4; big data; context free grammar; distributed systems; HPCC; Javascript; language; machine learning; Parser; SQL; transpiler**

I. Introduction

Data has become an essential resource in this age of computing, where a lot of the advancements and innovations we see right now, are based upon models which require huge amounts of data to be built. There has been widespread adoption of Machine Learning, and Data Analysis tools have become ever more important, especially with Big Data being increasingly common. SQL has been a widespread language that has been primarily used and well known to data analysts, for data analysis; especially due to the time its been around and its prevalence in relational databases.

Big data, has made it somewhat difficult to continue using traditional tools for data analysis, where standard databases would take too long to process the data. This, has led to a boom in the usage of Distributed Systems, where data is processed with the use of multiple different computing systems (often referred to as nodes) in a cluster. If done effectively, distributed computing is a vastly better option as it is not possible to vertical scaling (using more powerful hardware) cannot keep up with the scale and volume of data that is being generated nowadays. [**?**]

Distributed Systems have become popular, with Hadoop being a well-known option. Hadoop's core technologies are based on a storage section, and a processing part; it offers a huge library of plugins and integrations which allow it to be easily used for a variety of use-cases. Spark, is one such plugin that is known as a unified analytics engine, used as part of Hadoop. The primary languages used here are a mixture of SQL acting as a second-class language and Scala as the primary language of use. [**?**]

This is commonplace, as pure SQL often makes data analysis difficult (Eg. Visualizations and Machine Learning aren't a part of SQL). Here, SQL takes a second-class language approach where it is embedded in a primary language (e.g. in Scala for Spark [**?**], in ECL for HPCC Systems®). There are also places where SQL have first-class support, but here access to valuable tools such as visualizations and working with Machine Learning Models as well as commonplace language features get restricted, which have become commonplace and important since the time SQL was developed.

As such technologies are being applied everywhere, having a steep learning curve for such tools would be rather inconvenient. Hence, as data analysis grows more important, there is a good need for an SQL-like analytics language that has support for querying, visualization and machine learning.

Hence, HPCC Structured Query Language – HSQL was developed, a language that is SQL-like, for focusing on easy-to-learn and simple data analytics. HPCC Systems was chosen as

the target architecture; an open source platform developed by LexisNexis® Risk Solutions, which uses commodity hardware for data-intensive parallel computing. The platform presents an all-in-one integrated data lake solution that is extremely versatile and removed from the MapReduce Architecture of Hadoop that allows for much more versatile programming. HSQL is open-source, easy to write, comprehend and transpiles to ECL for use in HPCC Systems. Many of the typical preprocessing for ECL is abstracted away in HSQL, and the simple SQL-like syntax eases the learning curve related to getting started with HPCC Systems. Additionally, targeted to work with ECL, HSQL compliments ECL very well by providing an abstraction that is easy to use by data scientists who are already familiar with SQL; where data scientists can still take their time to learn the more complex and powerful ECL language for any complex solution they may require. This helps quickly bridg the skill gap to use the same Data Lake to both shape the data (ECL) and perform analytics (HSQL). Here, some key concepts of HPCC Systems will be introduced in order to explain the architecture that HSQL targets, and the features to bee used. Following this, a design for HSQL is shown which presents a concrete syntax and then, an implementation for a compiler that translates the specification to ECL, the language used in HPCC Systems.

II. HPCC Systems and ECL

HPCC Systems (High Performance Computing Clusters), an open source platform developed by LexisNexis® Risk Solutions, has been used to set up a cluster, distribute the data and perform all the operations parallelly for a faster and a more effective computation. HPCC Systems provide high performance, parallel processing and delivery for applications using big data. It is open source, and presented as an all-in-one solution as a data lake and big data processing system that makes it easy and fuss-free to work with [**?**].

The entire platform (Fig. **??**) is divided into separate platforms, each optimized for a specific workload. The first of these platforms is called Thor, a data refinery whose overall purpose is the general processing of massive volumes of raw data of any type. A Thor cluster is similar in its function, execution environment, filesystem, and capabilities to the Google and Hadoop MapReduce platforms [**?**]. The second platform, named as Roxie, functions as a rapid data delivery engine. A Roxie cluster is similar in its function and capabilities to ElasticSearch and Hadoop with HBase and Hive capabilities added.

HPCC Systems, including both Thor and Roxie clusters utilize the ECL programming language for implementing applications, a data-centric declarative language designed specifically for huge data projects using the HPCC Systems platform [**?**] [**?**]. Each of the platforms use the declarations in a way that best aligns with its goals for performance and latency. ECL as a language allows for power ETL operations to be carried out.

Its extreme scalability comes from a design that allows you to leverage every query you create for re-use in subsequent queries as needed [**?**]. ECL, is a powerful language, and due to its expansive and declarative nature, it is well suited for performing data extraction, cleaning, normalizing and aggregating [**?**].

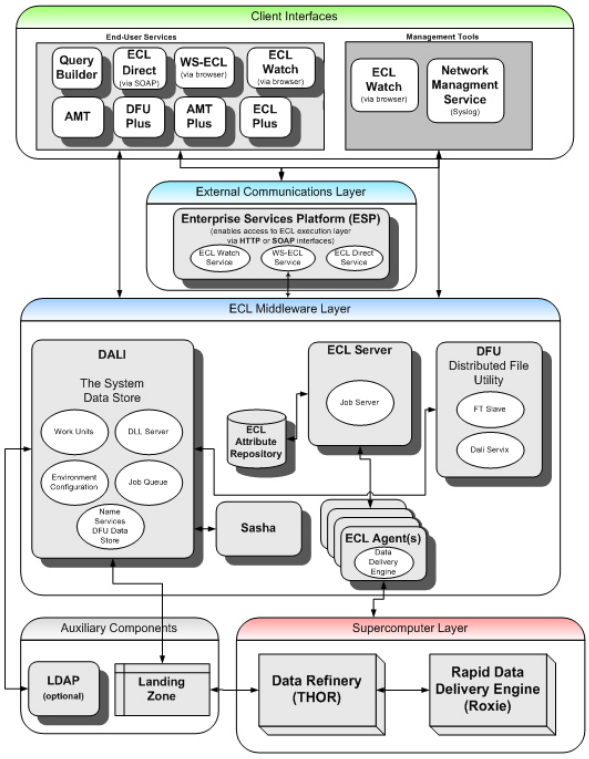

Fig. 1. HPCC Systems Structure - CC/SA.

As HSQL is intended to be for data analysts, ECL was chosen as a target due to its highly optimizing compiler and that machine learning performs exceptionally fast in HPCC Systems, even outperforming similarly configured Hadoop for the first iteration of many configured Machine Learning Algorithms.

HSQL is intended to be used in HPCC Systems, where the primary intention is to complement ECL. The primary purpose is to provide simpler syntax for common data operations, without needing to know ECL but also introducing key concepts of ECL and HPCC Systems along the way.

III. Design and Implementation: HSQL

Building a language such as HSQL, that can be used, would atleast require a basic grammar specification and some way of executing it on a machine. Languages either translate to another language of lesser or similar levels of abstraction. This operation of translation, is done by compilers [**?**]. Software that translate programs to similar levels of abstraction, are specifically termed as Transpilers (e.g. Babel, which is a Javascript to Javascript transpiler). Below sections will first define a syntax then describe the various steps used to translate HSQL to the target language ECL (Fig. **??**). In the following subsections, the HSQL language will be discussed as a language, followed by a brief description of the implementation process.

A. Defining the HSQL Syntax

The first step towards making HSQL, was to define the syntax and a barebones featureset. HSQL is a language designed to be SQL-like, and yet, expose many features of ECL

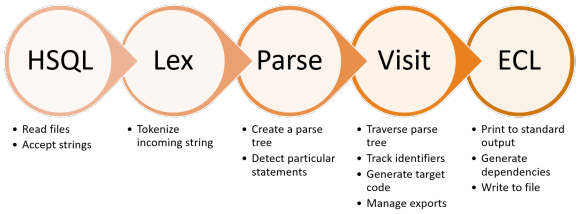

Fig. 2. General Conversion Workflow.

such as modules, layouts (analogous to SQL's `CREATE TABLE`) and actions. The general syntax of statements can be seen as:

```
<identifier_name> = <statement>;
```

Actions have been defined with the following syntax:

```
<action> <identifier> [options];
```

A `<statement>` would be an SQL select statement which uses existing definitions or loads up a dataset from a table.

Listing 1: Simple statement and an action

```
p = SELECT * FROM table1;
OUTPUT p;
```

This syntax was made to keep a lot of similarity to SQL, but to slowly introduce concepts of ECL to a user, such as imports and modules. Actions written above are always run in sequential order. Like SQL, HSQL is also intended to be case-insensitive, including its variables (additionally as ECL is also case-insensitive).

Definitions in HSQL, can be exported, by writing an optional export statement at the end, listing all the identifiers, which shall be exported. HSQL, unlike ECL, presents only two visibility modes, for ease of use and understanding. HSQL definitions convert to *SHARED* or *EXPORT* in ECL.

The language was designed to be completely compatible with ECL, and the transpiler developed, provides optional type checks. (Imports from ECL files do not support type checking, except a type definition file can be added in.) The language also allows an optional export statement at the end, to allow for specific definitions to be exported and available for use as a module.

The general syntax of a program is intended to be ($<,>$ implying mandatory features, $[,]$ implying optional):

Listing 2: General Syntax

```
import <identifier> [as <alias>];
<identifier> = <definition>;
<action> <identifier> <options>;
...
[export <identifier1>[,<identifier2>[,<identifier3
    >,...]];
```

1) Layouts: Layouts can be considered as a structure for a dataset (analogous to a structure definition in C). The syntax for them is similarly designed as:

Listing 3: Creating a sample layout with two columns

```
sampleLayout = CREATE LAYOUT(
    c1 integer,
    c2 string
);
```

`sampleLayout` internally in the target language will be represented by a record, a case of one-to-one translation.

2) Plotting: Plotting is another important part of HSQL, as a way of visualizing data to understand it better. This is done with the use of the `plot` statement.

Listing 4: Plotting a table

```
plot from table1 title 'Optional_title' type Column;
```

On the target language, visualizations are made by a named output, followed by calling the respective call to an appropriate Visualizer bundle function. These two statements are wrapped into a singular statement in HSQL.

3) Machine Learning: The idea behind using Machine Learning in HSQL is to be able to easily create and use Machine Learning models. This is done by using the `train` and `predict` statements.

Listing 5: Making a model

```
model = train from ind,dep method LinearRegression;
```

The making of model requires anywhere from 3-6 statements in ECL, which involves adding a sequential ID, converting from the standard row-based form to the ML bundle compatible cell-based form and then calling the model creation statement on the result (Which is based on the method required). This is represented in one singular statement in HSQL.

Similarly, the predict statement can be used to make predictions from models. This similarly requires some table conversions, and is represented via a singular statement in HSQL.

```
model1_predict = predict model1 from test_ind method
     RegressionForest;
```

B. Language Recognition

As the language and its primary featureset been established, the target implementation was carried out; The general syntax of the language, was written in CFGs (Most languages are expressed as a context free grammar at the syntactic analysis phase [**?**]), so that it can be passed to ANTLR4. ANTLR4 is able to accept a CFG (Context Free Grammar) which does not contain left recursive derivations [**?**] to create lexers, and parsers which use the ALL(*) parsing methodology for generating a parse tree for the given CFG. The lexers and parsers created support a variety of targets languages, including C++, JavaScript, C# and so on. For this work, JavaScript was chosen as it would allow fast development and allow for further integration in web services. The lexer and the parsers created, are able to take in a stream of text, and break them up into tokens, and then construct a parse tree (Fig. **??**) according to the grammar made for the language. ANTLR4 sets up the class structure for each node in the parse tree, which is useful while processing the parse tree [**?**].

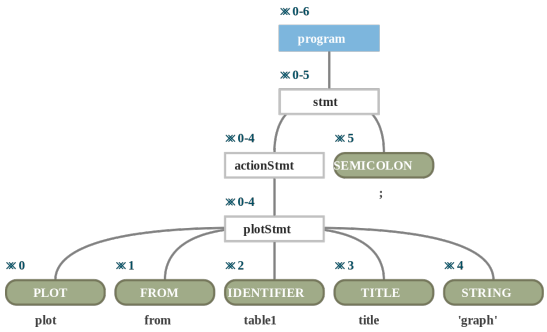

Fig. 3. Parse Tree for a Simple Statement.

The core of the conversion is through ANTLR4 and using String Templates to template generated code. The translation of HSQL to ECL happens in two phases - both in the parsing stage and while processing the parse tree nodes.

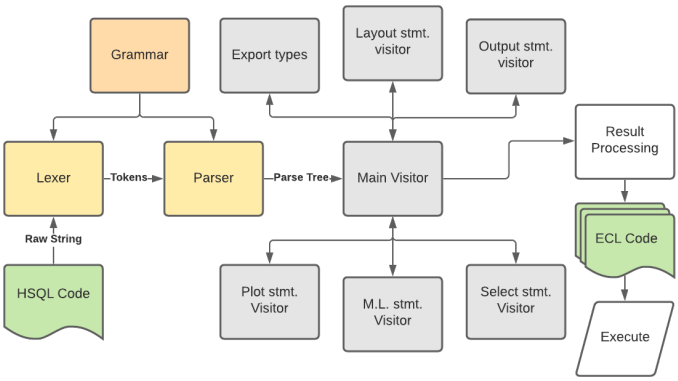

Fig. 4. Basic Architecture for the Program.

The process of walking the parse tree was split into various visitors as per the architecture Fig. **??**, which allows for better code readability and better structure. The visitors (Objects written under the well-known visitor pattern [**?**]), traverse the parse tree, calling other visitors as required, and return the specific translations. The visitors also perform other important functions, and some of the other functions are enumerated as we go along.

1) Symbol Table Management: HSQL uses a flat table of identifiers; it does not have scoping; Identifiers and their types are tracked by the transpiler where available via an object oriented symbol Table [**?**]. ECL, which is the language HSQL converts to, does have the concept of types, and hence this applies to HSQL too. However, this cannot be read easily. This has been couteracted by making type checking optional - HSQL will attempt to obtain and infer types when possible and show issues with it if present, but otherwise will allow the users to continue. As it needs to be compatible with ECL, such type checking is kept optional.

Listing 6: Selecting from a module

```
import module1;
a = select col1 from module1.table1;
```

In the above example **??**, if there exists a `module1.hsql`, then that file is parsed. Similarly, if there exists a `module1.dhsql` file, that is also referred to for understanding what types are exported. In these cases, the compiler can check if `table1` exists, and can raise an error if `table1` or `col1` does not exist. However, if no files are found, then a compiler error is raised. Additionally, if there is only a `module1.ecl`, in that case no checking is done, and all columns are allowed; ie. the types are not known, and can be as per the user's discretion.

Listing 7: dummy.ecl - ECL File

```
export dummy := module
    export Layout1 := RECORD
        INTEGER col;
        STRING25  col2;
    END;
    export Layout2 := RECORD
        INTEGER  col3;
        STRING  col4;
    END;
    export someTable := TABLE(DATASET([{1,'foo'}],
        Layout1));
    export someTable2 := TABLE(DATASET([{2,'bar'}],
        Layout2));
end;
```

Listing 8: dummy.d.hsql - ECL File's Type Definition

```
map export a INTEGER;
map export someTable TABLE ( col INTEGER,col2 STRING
    );
map export someTable2 TABLE (col3 INTEGER,col4
    STRING);
```

In HSQL, ECL files can be imported without any issues, but do not have any form of type checking, but this can be worked around, by adding a type definition file (File that mentions the export types, with extension *.d.hsql*). Imported HSQL files, are parsed in a recursive manner, and their exported types are extracted and used.

2) Dependency Tracking: The primary visitor in the transpiler implemented, tracks all the dependencies and allows for recursively parsing dependent HSQL modules too. Additionally, for every ECL file imported, it looks for a corresponding type definitions file (File with same name but *.d.hsql* extension) to provide the types exported by it. Cyclic imports are prevented by the use of an import list which keeps track of all the imports that have occurred for a given module to require transpilation.

3) Actions Collecting: Collecting actions - All the actions that are mentioned in HSQL, are tracked, as they are translated to definitions for the action. To ensure modules can be executed, the target ECL code contains a *main* function export, that calls on all the actions sequentially. This, allows modules to keep actions that can be executed. This is not usually used in ECL, but is retained in HSQL for maintaining ease of use. An example for this is shown later on.

4) Module Support: Module field visibility is done by an export statement, present at the end of the program. Here, specific identifiers can be marked for being exported.

Listing 9: Source HSQL

```
import source;

-- lets see the marks for each column
markslist = select marks from source.marks ;
-- or even better
```

```
output markslist title 'marksList';

-- lets get all the average marks of each subject
counting = select subid,AVG(marks)  from source.
    marks group by subid order by subid ;
// join this to get the subject names
marksJoined = select * from counting join source.
    subs on counting.subid=source.subs.subid;

output marksJoined title 'avgmarks';
```

Listing 10: Target ECL - Wrapped in a Module

```
IMPORT source ;
export hsqlc:= MODULE
SHARED markslist := TABLE(source.marks,{marks});
SHARED _reservedaction0 := OUTPUT(markslist,,NAMED('
    marksList'));
SHARED counting := SORT(TABLE(source.marks,{subid,
    REAL marks := ave(GROUP,marks)},subid),subid);
SHARED marksJoined := TABLE(JOIN(counting,source.
    subs,LEFT.subid = RIGHT.subid,INNER));
SHARED _reservedaction1 := OUTPUT(marksJoined,,NAMED
    ('avgmarks'));
EXPORT main := FUNCTION return PARALLEL(
    _reservedaction0,_reservedaction1); END;
END;
```

After translating the statements, two tasks need to be executed:

- Arranging the tasks to be executed into a main field that will be exported to be executed
- Wrapping the translated statement into an ECL module.

The main visitor then returns the resultant statements as an array to the rest of the program (which handles the arguments, errors and output). The rest of the module wraps the output of the visitors, and allows access to the types as understood by HSQLC, and warnings and errors that may have been raised by it.

C. Transpiler

The lexer, parser and the visitor are only part of the whole program, which makes up the transpiler HSQLC which is used to translate HSQL to ECL. The whole program wraps up the above components, in a command line UI, and provides:

- Wrappers to read and write files for the visitors as required.
- A general command line user interface for accepting files, arguments, and for presenting errors and warnings neatly.
- Endpoints for interfacing with the transpiler. Various functions for providing a string or a file is provided. These function calls also have Typescript definitions, to allow use inside a TypeScript environment as well.

The transpiler can hence run as a command line tool for transpiling HSQL files to ECL files, or function as a module that can be called on to provide translation, syntax highlighting and other such language features.

IV. Features of the Proposed Language

The HSQL language and the transpiler hence built, had a set of notable features which should prove it easy to use and integrate into existing workflows.

Intercompatibility

HSQL was designed to be fully compatible with the existing structure of HPCC Systems, and is completely interusable with ECL. HSQL modules can be imported from within ECL after translation, and HSQL can make use of existing ECL modules to provide extended functionality or use data from other sources.

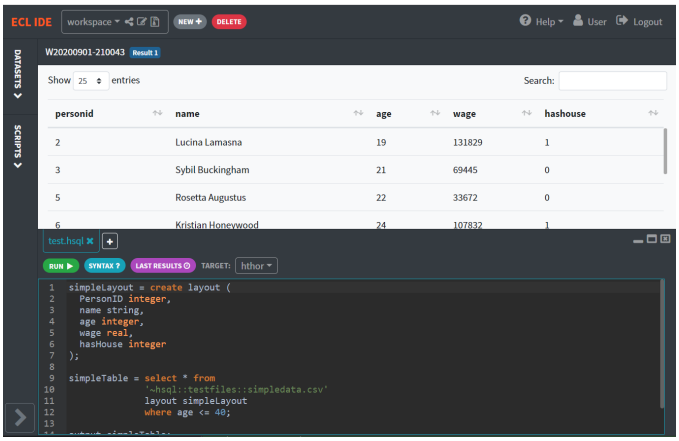

Fig. 5. ECL Cloud IDE, using HSQL.

Integration

As the transpiler(HSQLC) was made using Javascript, it is rather easy to integrate into web solutions, and use for supporting HSQL in a web environment. Using this, it was also integrated into the ECL Cloud IDE Fig. **??**, a web-based solution for running ECL on a HPCC Systems cluster.

HSQLC was also used to create a language server to provide language support for HSQL in popular IDEs. The initial target was VSCode, but as the Language Server Protocol Fig. **??** is well established, it can be easily ported over to other IDEs which support using the LSP [**?**] [**?**].

V. Conclusions

The language HSQL was defined, which lets users write SQL-like queries without worrying about most of the preprocessing required in HPCC Systems while using ECL, and a basic syntax set was produced, for performing filters, joins, sorts, and so on. This, allows for HPCC Systems as a Distributed Systems to be accessed and used easily by a person who is familiar with SQL. The transpiler HSQLC was created to be able to use this language, was able to successfully translate HSQL to ECL, and was able to correctly report syntax and semantic errors, while also allowing HSQL and ECL modules to be used interchangeably. The compiler can also report the data types for the variables used in its program to help with integration into IDEs.

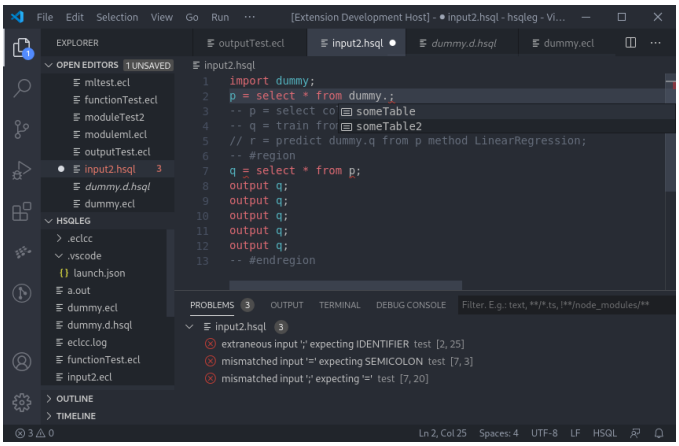

Fig. 6. Language Server - Providing Syntax Highlighting, Completion and Error Highlighting.

Using this, HSQLC was also integrated into a language server, creating a VSCode Extension, which could be used to provide language support in development environments, and also in ECL Cloud IDE, to easily allow writing HSQL in a web-based editor.

Testing was set up for testing the validity of the machine learning models created and basic syntax, which reported predictions within good intervals. Various examples have also been shown to showcase the syntax in HSQL **?? ?? ?? ??**, comparing it to the translated code in ECL. These program snippets compare the languages and show how some of the boilerplate code in ECL is automatically generated by the HSQL compiler. A complete file transpilation is also shown, **??** where a simple Random Forest Regression model is created.

Limitations and Future Work

The current solution focuses on extensibility, usability and simplicity heavily.

HSQL, contains only some basic operations in the current revision, and can be extensively improved by adding in syntax for other ECL features which can still use a SQL-like syntax, which should improve its usability and allow for greater and more extensive usecases.

HSQLC, has been made as a simple command line, and hence, is an additional step require to run a program on HPCC Systems. This can be worked around by automating the process (e.g. by using a task automation toolkit/make utility), or by integrating it with existing systems. The compiler can also be better maintained with the help of static typing [**?**], although this is slightly impeded by ANTLR4 Typescript support still being in the works at the time of writing.

The language server developed for HSQL uses HSQLC to provide language support in IDEs, and performs syntax checking, but cannot perform syntax highlighting [**?**] as of the current specification, and requires the use of IDE-specific extensions (e.g. VSCode requires providing a Textmate Grammar) for syntax highlighting.

REFERENCES

[1] A. Prasad, G. Shobha, N. Deepamala, S. S. Badhya, Y. Yashwanth and S. Rohan, "Machine Learning Techniques to Understand Partial and Implied Data Values for Conversion of Natural Language to SQL Queries on HPCC Systems," 2019 4th International Conference on Computational Systems and Information Technology for Sustainable Solution (CSITSS), 2019, pp. 1-5, doi: 10.1109/CSITSS47250.2019.9031035.

[2] E. Shaikh, I. Mohiuddin, Y. Alufaisan and I. Nahvi, "Apache Spark: A Big Data Processing Engine," 2019 2nd IEEE Middle East and North Africa COMMunications Conference (MENACOMM), 2019, pp. 1-6, doi: 10.1109/MENACOMM46666.2019.8988541.

[3] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). Association for Computing Machinery, New York, NY, USA, 1383–1394. DOI:https://doi.org/10.1145/2723372.2742797

[4] HPCC Systems, "Why HPCC Systems is a superior alternative to Hadoop",[Online] Available: https://hpccsystems.com/about/hpcc-hadoop-comparison/superior-to-hadoop

[5] HPCC Systems, "Introduction to HPCC Systems", 2015, [Online] Available: http://cdn.hpccsystems.com/whitepapers/wp_introduction_HPCC.pdf. [Accessed July 1, 2020]

[6] A.M. Middleton. Handbook of Cloud Computing. Springer, 2010., Handbook of Cloud Computing, "Data-Intensive Technologies for Cloud Computing" [Online] Available: https://www.springer.com/gp/book/9781441965233

[7] HPCC Systems, "ECL Language Reference", 2020, [Online] Available: https://d2wulyp08c6njk.cloudfront.net/releases/CE-Candidate-7.8.24/docs/EN_US/ECLLanguageReference_EN_US-7.8.24-1.pdf . [Accessed July 1, 2020]

[8] Aho, Sethi, Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.

[9] N. Chomsky, "Three models for the description of language," in IRE Transactions on Information Theory, vol. 2, no. 3, pp. 113-124, September 1956, doi: 10.1109/TIT.1956.1056813.

[10] Parr et al., Adaptive LL(*) "Parsing: The Power of Dynamic Analysis", [Online] Available: https://www.antlr.org/papers/allstar-techreport.pdf. [Accessed June 21, 2020]

[11] Danyang Cao and Donghui Bai, "Design and implementation for SQL parser based on ANTLR," 2010 2nd International Conference on Computer Engineering and Technology, 2010, pp. V4-276-V4-279, doi: 10.1109/ICCET.2010.5485593.

[12] Jens Palsberg1 C. and Barry Jay ,"The Essence of Design Patterns", [Online] Available: http://web.cs.ucla.edu/~palsberg/paper/compsac98.pdf. [Accessed June 23, 2020]

[13] J. F. Power and B. A. Malloy, "Symbol table construction and name lookup in ISO C++," Proceedings 37th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS-Pacific 2000, 2000, pp. 57-68, doi: 10.1109/TOOLS.2000.891358.

[14] Matt, Language Server Protocol, May 7 2020, [Online] Available: https://nshipster.com/language-server-protocol/ [Accessed July 20, 2020]

[15] Z. Gao, C. Bird and E. T. Barr, "To Type or Not to Type: Quantifying Detectable Bugs in JavaScript," 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017, pp. 758-769, doi: 10.1109/ICSE.2017.75.

[16] Microsoft, "Language Server Protocol", [Online] Available: https://microsoft.github.io/language-server-protocol/ [Accessed July 15, 2020]

APPENDIX - TRANSLATED EXAMPLES

TABLE I. SELECT STATEMENTS

HSQL	ECL
`weatherSnowy = select Date,SnowDepth from weather where SnowDepth>0;`	`weatherSnowy := TABLE(weather(SnowDepth > 0),{Date,SnowDepth});`
`weather = select * from '~hsql::testfiles::weatherdata.csv' layout common.WeatherDataLayout;`	`weather := TABLE(DATASET('~hsql::testfiles::weatherdata.csv', common.WeatherDataLayout,CSV(HEADING(1))));`
`weatherSnowyNum = select COUNT(SnowDepth) from weather where SnowDepth>0;`	`weatherSnowyNum := COUNT(TABLE(weather(SnowDepth > 0),{SnowDepth}));`
`weatherSnowyTotal = select Sum(SnowDepth) from weather where SnowDepth>0;`	`weatherSnowy := TABLE(weather(SnowDepth > 0 and NewSnow > 0),{Date,SnowDepth,NewSnow});`
`weatherGrp = select Date,SnowDepth from weather group by SnowDepth;`	`weatherGrp := TABLE(weather,{Date,SnowDepth},SnowDepth);`
`weatherjoin = select * from weatherSnowDepth where newSnow>0 join weatherNewSnow on weatherSnowDepth.Date = weatherNewSnow.Date;`	`weatherjoin := TABLE(JOIN(weatherSnowDepth,weatherNewSnow,LEFT.Date=RIGHT.Date,INNER)(newSnow > 0));`

TABLE II. PLOTTING

HSQL	ECL
`plot from weatherSnowyTotal title 'SnowyDays' type bar;`	`_reservedaction4:= OUTPUT(weatherSnowyTotal,NAMED('SnowyDays'));` `_reservedaction5 := Visualizer.MultiD.Bar('SnowyDays');`

TABLE III. TRAINING A ML MODEL

HSQL	ECL

`model1 = train from ind, dep method RegressionForest;`	`ML_Core.ToField(ind,_reserved_ind0);`
	`SHARED _reserved_ind1 := _reserved_ind0;`
	`ML_Core.ToField(dep,_reserved_dep0);`
	`SHARED _reserved_dep1 := _reserved_dep0;`
	`SHARED _reserved_ind10 := PROJECT(_reserved_ind1,TRANSFORM(RECORDOF(LEFT),SELF.id:=COUNTER,SELF:=LEFT));`
	`SHARED _reserved_dep10 := PROJECT(_reserved_dep1,TRANSFORM(RECORDOF(LEFT),SELF.id:=COUNTER,SELF:=LEFT));`
	`SHARED model1 := LearningTrees.RegressionForest().getModel(_reserved_ind10,_reserved_dep10);`

TABLE IV. PREDICTING RESULTS FROM A MODEL

HSQL	ECL
model1_predict = predict model1 from test_ind method RegressionForest;	ML_Core.ToField(test_ind,_reserved_test_ind0); SHARED _reserved_test_ind1 := _reserved_test_ind0; SHARED _reserved_test_ind00 := PROJECT(_reserved_test_ind1, TRANSFORM(RECORDOF(LEFT),SELF.id:=COUNTER,SELF := LEFT)); SHARED model1_predict := LearningTrees.RegressionForest().Predict (model1,_reserved_test_ind00);

TABLE V. CREATING A MODEL AND USING IT

HSQL (ols.hsql)	ECL (ols.ecl)

import commonsimple;
 	
ind = select PersonID,age from commonsimple.simpleTable where PersonID<5;
dep = select PersonID,wage from commonsimple.simpleTable where PersonID<5;
 	
output ind;
output dep;
 	
test = select PersonID,age from commonsimple.simpleTable where PersonID>4;
 	
model = train from ind,dep method LinearRegression;
 	
result = predict model from test;
 	
output result;
	IMPORT ML_Core ;
	IMPORT ML_Core.Types AS Types;
	IMPORT commonsimple ;
	IMPORT LinearRegression ;
	export ols:= MODULE
	SHARED ind := TABLE(commonsimple.simpleTable(PersonID < 5),{
	PersonID,age});
	SHARED dep := TABLE(commonsimple.simpleTable(PersonID < 5),{
	PersonID,wage});
	SHARED _reservedaction0 := OUTPUT(ind);
	SHARED _reservedaction1 := OUTPUT(dep);
	SHARED test := TABLE(commonsimple.simpleTable(PersonID > 4),{
	PersonID,age});
	ML_Core.ToField(ind,_reserved_ind0);
	SHARED _reserved_ind1 := _reserved_ind0;
	ML_Core.ToField(dep,_reserved_dep0);
	SHARED _reserved_dep1 := _reserved_dep0;
	SHARED _reserved_ind10 := PROJECT(_reserved_ind1,TRANSFORM(
	RECORDOF(LEFT),SELF.id:=(COUNTER-1)/MAX(_reserved_ind1,
	_reserved_ind1.number)+1,SELF:=LEFT));
	SHARED _reserved_dep10 := PROJECT(_reserved_dep1,TRANSFORM(
	RECORDOF(LEFT),SELF.id:=(COUNTER-1)/MAX(_reserved_dep1,
	_reserved_dep1.number)+1,SELF:=LEFT));
	SHARED model := LinearRegression.OLS(_reserved_ind10,
	_reserved_dep10).GetModel;
	ML_Core.ToField(test,_reserved_test0);
	SHARED _reserved_test1 := _reserved_test0;
	SHARED _reserved_test00 := PROJECT(_reserved_test1,TRANSFORM(
	RECORDOF(LEFT),SELF.id:=(COUNTER-1)/MAX(_reserved_test1,
	_reserved_test1.number)+1,SELF:=LEFT));
	SHARED result := LinearRegression.OLS().Predict(
	_reserved_test00,model);
	SHARED _reservedaction2 := OUTPUT(result);
	EXPORT main := FUNCTION return SEQUENTIAL(_reservedaction0,
	_reservedaction1,_reservedaction2); END;
	END;