Reika: A DSL for Time Series Database

Pinglei Guo
UC Santa Cruz
piguo@ucsc.edu

Chujiao Hou
UC Santa Cruz
chou8@ucsc.edu

ABSTRACT

Reika is a DSL for time series database. It is composed of two part, the general language part and the query language part. By using a DSL, it abstracts the differcence between different time series databases and enable database agnostic optimization. The language is not limited to query, it can also be used for synthetic data generation and graphing, which form a full data exploration pipeline. Reika is a static typing language which can be run in an interpreter or compiled to other language. We also provided an execution engine for database operations. With the emerging of various new time series databases, we hope Reika could allow people analysising data stored in heterogeneous data sources.

1. INTRODUCTION

Time series are data that contains time, in traditional literatures, they are described as ordered sequence of events with fixed intervals, however this is no longer the case. In our opinion, the key feature of time series data is immutability. 'what happened, happened and couldn't have happened any other way' [1]. Though most existing open source time series databases still didn't realize it.

The rise of time series databases (TSDB) is mainly caused by hardware and software monitoring in big companies. In FaceBook, hundreds of thousands of machines are generating trillions of metrics every second [13] The high throughput and unique patterns making traditonal RDBMS and modern NoSQL insufficient and overkill. Thus many time series databases emerged [4] [6]. [2] However they have their own exclusive API and Query Languages with different level of features, which not only create a steep learning curve but also making it hard to use heterogeneous data sources and apply optimization.

In traditional RDBMS field we have SQL to rule them all, from large commercial database vendors like Oracle to small open source embedded database like SQLite, programmers has no burden using most of them once they master any of them. And they all provide same level of functionality, i.e. join, UDF, aggregation function. While for time series databases, some have SQL like syntax, some use JSON, some use URL like syntax, some does not support any aggregation or filter. It's impossible to let all the open source and commercial time series databases to change their API to a unified standard (at least in foreseeable future). However we can add an extra layer to make their difference transparent to user. Which is already the case in the big data field, for instance Facebook built Presto [7] to run query using SQL in heterogeneous data source with reasonable response time. Thus we decided to build a Query Language for Time Series Database called Reika and run on our query engine to support various existing time series databases

We organize this paper as following. In section 2, we compare several popular databases' query language to summarize the characteristics and present a general model of TSDBs' query language as our design goal. In section 3, we present the syntax and semantics of Reika. In section 4, we discuss implementation detail and choice we made. In section 5, we show the staging results. In section 6 and 7, we summarize related work and future work.

2. BACKGROUND AND GOALS

2.1 Existing Query Languages

Graphite [2] is the most used RRD based time series database. Its query language has rich features, it allows you to specify how to graph the time series and return an image instead of raw data, which you can easily tell from its name. However, most advanced queries need to be specified using a syntax like built-in functions, the overall syntax is like a URL when you are filtering products on a shopping website. Following is some example from Graphite's documentation [3]

- target=server.web1.load&height=800&width=600 would return a 800x600 image with a single series named as server.web1.load

[1] `https://en.wikiquote.org/wiki/The_Matrix_Reloaded`

[2] A full list can be found on `https://github.com/xephonhq/awesome-time-series-database`

[3] `http://graphite.readthedocs.io/en/latest/render_api.html`

- target=summarize(derivative(app.numUsers),"1min") would use sum result within the 1 min window, so the return the graph has less points and more representative.

Note when filtering time series, specify series name is the only way, however, using the dot notation would create a illusion of hierarchy, which is misleading and very inflexible. This is addressed by OpenTSDB [6] where they use tag to allow multifaceted filtering.

OpenTSDB is first distributed time series database, its author feels RRD based time series databases can no longer meet their requirement for scaling and rich query need, and they can use external tools to do graphing. To be honest, Graphite's chart does not fit modern design, and most people no longer using Graphite's graph feature and instead using a tool called Grafana [1]

OpenTSDB's default query language is really like Graphite, simple URL with filtering options, except it now allow filtering on tags, so you no longer crumble everything in metric name. It also has a JSON API to allow multiply queries in the same API call, which allow more advanced features like specifying aggregator and config. An example is listed below.

```
{
    "start": 1356998400,
    "end": 1356998460,
    "queries": [
        {
            "aggregator": "sum",
            "metric": "sys.cpu.0",
            "rate": "true",
            "filters": [
                {
                   "type":"wildcard",
                   "tagk":"host",
                   "filter":"*",
                   "groupBy":true
                },
                {
                   "type":"literal_or",
                   "tagk":"dc",
                   "filter":"lga|lga1|lga2",
                   "groupBy":false
                },
            ]
        },
        {
            "aggregator": "sum",
            "tsuids": [
                "000001000002000042",
                "000001000002000043"
            ]
        }
    ]
}
```

Using JSON, one could actually present something like AST, which is done by Heroic [3] already, their JSON API is actually the AST of their Query Language. Like $key = "hello kitty" and host = foo.example.com is equal to the following JSON.

```
{
  "range": {"type": "relative", "unit": "
     HOURS", "value": 2},
  "filter": ["and", ["\$key", "hello kitty"]
     , ["=", "host", "foo.example.com"]],
  "groupBy": ["site"]
}
```

There are also time series database like InfluxDB [4] which follows the syntax of SQL exactly because it wants to be a columnar database instead of just a time series database. it has enterprise services and a whole stack called TICK[4] including another DSL for stream processing.

```
// Get views batch data
var views = batch
    |query('SELECT␣sum(value)␣FROM␣"pages"."autogen".vi
        .period(1h)
        .every(1h)
        .groupBy(time(1m), *)
        .fill(0)
```

The author of Prometheus [8] also did a comparison on the query language of Graphite, Prometheus and InflxuDB [9] few months ago. It is quite objective considering Prometheus is no longer just an open source project but a company called robustperception. However he did not seems to have any intent of standardizing the query language for time series databases and satisfied with current diverge in the community, which the authors of this paper can't agree on, we believe unifying the query interface is the trend, take the big data eco system for example, Hadoop comes with the MapReduce model after Google's paper, however, even though the model is not complex, tools like Pig, Hive comes out with SQL support to attract more users. Spark has the great RDD concept, and they decided to created SparkQL for better optimization [10]. We believe what happened to them would also be the case for time series database as the community and the market becomes more mature.

2.2 Design Goals

The original design goal of Reika is to build something similar to InfluxDB's InfluxQL, which is basically a SQL port to time series database, with more advanced data type and operations on date and time. However, we also want to do data loading and graphing in our language, which requires our language to be more general purpose language. After seeing LINQ [11] and various ORM for RDBMS, we believe SQL like syntax is handy in a shell like environment but not that handy when it involves complex application logics. Thus we decided to split Reika into two parts, the general programming language part, which is like Java without class, and the query language part, which is like SQL, but using the language directly instead of concatenating string in a manner similar to LINQ. The language will run on a currently under construction query engine called tsdb-proxy[5]. We hope tsdb-proxy itself can be ported to

[4]`https://www.influxdata.com/open-source/`
[5]`https://github.com/xephonhq/tsdb-proxy-java`

several other languages, which would allow us to compile Reika to other language and use tsdb-proxy as a runtime library. Thus benefits from the optimization in many existing programming languages.

Currently, only the general programming language part is implemented, so in the following paper we would mainly talk about the implemented part and skip the query language part even though it is our original motivation, most technique we used on the general part could also be applied to the the query language part.

3. SYNTAX AND SEMANTICS

We use a syntax similar to Java[6]. The reason we use an imperative language as base is we think it is more likely to be accepted by the majority of programmers. Functional paradigm is popular now but it is hard to change the tide in a short time. We want Reika to be easy to get started with, so we try to reduce the learning curve as much as possible. The detailed syntax is omitted due to the length of the paper. [7]

However, Reika does not have class and doesn't even allow user to define functions, user can only use built in functions, which seems to be one step back from traditional SQL. But we do this based on the following concern and assumptions

- Reika can't be optimized as most general purpose programming languages, it's hard to develop some engines that is fast and optimized to the limit like JVM. But by limiting the freedom of user, we can push the optimization down to the query execution engine, and ease the burden of interpreter and/or translator.

- Reika is not built for writing a full fledge application, you can't write a website using Reika and you shouldn't. However if we provide many features like Java, Python, it would mislead users to write Reika program like they are writing Java, Python. By limiting the features, we force users to be aware of they are using a DSL for time series database, no a real general purpose language.

- A lot of advanced features should be built into the query execution engine, which is written in Java/Golang etc. Providing an standard API in those mature language is less error prone than providing user defined functions in Reika, where the tool chain for large scale development and debugging is blank.

Under these assumptions, Reika is more like SQL, which is initially built for people who don't know how to program. One may say Reika is a syntax sugar for SQL, this is true when data loading and graphing is not considered. It's more like a toolbox when all is considered, where languages like MATLAB leads the market.

For our semantics, it has little difference with Java, we support operator overloading for most types, having to use separated functions for concatenate string is a pain, type inference is also supported as Reika is a static typing language, one could declare variable using the var keyword and its type would be inference d from right hands side.

We split statement and expression. Statement is defined as no return value (evaluation result) and depends on side effect, like variable declaration,assignment etc. Expression is not actually not pure expression because when calling our built in function, there are actually tons of side effects in the query execution engine, like binary operation between primitive types. Expression when evaluated, would return a result, in an environment like REPL, 1 + 1 would result in 2 printed in the next line.

A program is defined as multiple statements, note since we don't allow user to define function, we don't have the concept scope, everything is in a global scope, built-in functions are black box to users, there is no need to worry about have conflict identifier between user's code and the built-in functions because the built-in functions are not written in Reika, they are written in Java to gain better performance.

Following is a simple example of Reika, its string has Unicode support thanks to Java

```
/*
* An example of Reika
*/
int a = 1; // var declaration
a = −1; // var assign
int b = abs(a); // pure function
print(b == a); // side effect
print("你好world"); // Unicode
```

4. IMPLEMENTATION

4.1 Lexer and Parser

We use ANTLR4 [12] for generating parser, and build AST from parse tree. Choosing ANTLR is the first right choice (and might be the only) we made when implementing Reika, we didn't craft our own lexer and parser and using ANTLR for the following reasons:

- It is the de facto choice, mature and used by a lot of famous open source softwares like Derby, Spark, Hive etc.

- ANTLR has runtime in various languages including Java/Golang, our own implementation requires a full rewrite if we want to change language.

- The Definitive ANTLR 4 Reference [12] is a good book. I bought it after seeing the review from the author of Yacc++ commented on amazon 'Writing good, maintainable, correct parsers shouldn't be hard, with this book and ANTLR in your toolkit, it isn't.' [8]

However, it's not all good about ANTLR. ANTLR's runtime library has different quality, Reika was intended to be written in Golang but end up implemented in Java because the Golang runtime can't run when using visitor pattern,

[6]actually more similar to Golang

[7]`https://github.com/xephonhq/tsdb-proxy-java/blob/master/ql/src/main/java/io/xephon/proxy/ql/parser/Reika.g4`

[8]`https://www.amazon.com/gp/customer-reviews/R3N86Q851MHR5M`

the related code is commented out with a TODO mark [9], so we run away from Golang and used the most mature Java runtime. Though in fact, we found Java is more productive because of its rich libraries and full OOP support, the advantage of Golang is small when you don't need features like goroutine and small binary.

4.2 AST Builder

A core part of Reika is the AST Builder, ReikaASTBuilder.java in the parser package (around 500 lines), which does the following

- Generate abstract syntax tree from parse tree
- Symbol checking
- Type checking
- Optimization

ReikaASTBuilder extends BaseVisitor generated by ANTLR, by overriding the default implementation for different node (context) in parse tree, we generate the AST tree while traversing the parse tree in a DFS style. The listener pattern could also work, but requires a stack which makes the code not that clear as the visitor one.

AST Nodes are defined in the ast package, both Expression and Statement extends Node. The visitor return Node for all the visit* methods, where * is replaced by context name like Program, Statement, Variable, Add etc. When the exact type is needed, we use Java's instanceof method to check all possible types. It would be much cleaner if we use a language with pattern matching like Scala, however, their learning curve would stall the project, so we sticked with Java even though our code is relatively verbose.

4.3 Symbol and Type Checking

Due the fact we don't allow user defined functions, we don't need to consider scope, which is a difficult part in symbol checking, only one global symbol table is needed. We have three types of symbol errors

- Undefined variable, i.e. use $x + 1$ without defining x
- Duplicate definition, i.e. $int\ x = 1; double\ x = 2.0;$
- Mixed use of variable and function

Built in functions are filled into the SymbolTable when the ASTBuilder first started, the SymbolTable is defined in the checker package along with corresponding exception classes like DuplicateDeclarationException, UndefinedIdentifierException etc.

Since Reika is a static typing language, type check is done at compile time instead of runtime, the ASTBuilder do the type check, so the interpreter and translator simply throw runtime exception and blame it on the ASTBuilder when there is an type error when running the program. We have three types of type errors

- incompatible assign, i.e. $string\ a = 123;$
- operation between different types, i.e. $"elder" + 1;$
- unsupported operation, i.e. $"elder" - "1s";$

[9] `https://github.com/antlr/antlr4/blob/master/runtime/Go/antlr/tree.go`

Incompatible assign is separated from operation between different types because their recovery mechanism is different. Unsupported operation is common because even though it's possible to give operations like subtraction between strings an semantics, it's likely it would cause more problem that it solves because there is no common agreement on the semantics.

When we meet an error, we throw an exception with symbol information filled for generating human and machine readable information for REPL and Editor. But that's not the end of it, we try to recover from current error to reduce future errors. For symbol errors, if we could inference the type from right hand side, we just put it into the symbol using that type. If there are duplicate definition, we replace the old one with the new one. For type errors, ANY_TYPE is our best friend, it can work with all the other types and itself. These recovery methods can make the symbol table and AST tree seems funny, but their sole purpose is to reduce future error messages. The recovered AST tree should not be passed to the next step, though in some cases, it might execute without any error, the result might may not be the intention of the programmer.

4.4 Test

Reika has around 400 lines code for test, the development style is test driven (TDD), this is especially useful when the team have to work on different part the project at same time. It's hard to find some bug during code review, while test simply has two states, pass and fail. We found several subtle bugs after we modified the ASTBuilder for the REPL, and we may never found them if we don't have test.

The test is written in JUnit, our coverage is 69%, and is integrated with Travis CI, so every pushed commit would trigger new CI build and notify team members if the test is broken.

4.5 REPL

Read Eval Print Loop (REPL) is very handy when it comes to learn a new language, and it also kinds of like the shell which most databases have. Our REPL is implemented using JLine3, which is the only readline like library in Java, jvm based projects like Scale, Spark, Hive all use JLine3. It has poor documentation and very complex example. Once again, by limiting our features, we managed to ship it in time. The REPL will show the syntax and semantic errors, and it won't exit until you type the :exit command.

However, the REPL introduced more challenge to the ASTBuilder, the recovery mechanism should be disabled or tracked. Otherwise a failed line and make the following wrong line correct. i.e. $b = 1;$ and $b + 1;$, without recovery, both line would got symbol error. However, if we have recovery, the second line would break the REPL with runtime exception instead of showing symbol error because during the recovery, the symbol b is put into the symbol table, but it is not executed, so the actual store does not have b. When the ASTBuilder do the checking for the second line, it found *b* in symbol table pass the AST to the interpreter, the interpreter try to lookup *b* and didn't find it thus throwing the error. In order to solve this, we add logging to our recovery, so before traverse the new parse tree, we first revert all the recovery we have logged. And based on this, we introduced a :revert command to our REPL, so you can do CTRL+Z like your are in an editor or Photoshop. Though some external side effects can't be reverted, like printed value.

4.6 Editor Plugin

We wrote the editor plugin for Visual Studio Code. The original idea is to write a plugin for its sibling, its browser port, but there is too little information about it. Currently, only highlight and basic completion is impletmented. VS-Code use textmate syntax (tmLanguage), which use regular expression and some extra notations like begin and end to compensate. It would be great if we could translate our ANTLR grammar to tmLanguage, and this idea itself worth another project.

5. RESULT

The code is available on Github as a subproject for tsdb-proxy. [10] Java8 and Gradle is required to build from source and run it. If you have docker installed, you can also use docker to play with the REPL irk. Screen shots can be found on Github and slides. They are omitted following the dry (don't repeat yourself) pattern.

6. RELATED WORK

Besides all the time series databases mentioned in section 2, Juttle [5] is pretty close to the original idea and part of current direction. It use a pipe like style to chain different data source and display layer together. It's sad we only found it when we are finalizing the project and the developer of Juttle have given it up.

7. FUTURE WORK

Only the general purpose language part is partially implemented, the query language part already have specification for syntax and semantics but is still volatile, we are considering add more features for support stream processing in the future.

The query execution engine is another project from the author but is written in Golang while Reika is written in Java, current workaround is to have an IR to communicate between two programs, the ideal solution is the author can fix the Golang runtime for ANTLR visitor mode. The Golang and Java version would co-exist and share grammar files and code template, code generation is needed due to the poor generic support in both languages. (Golang does not have plan to support generic, Java's generic is far less efficient than C++'s template with things like type erasure). C++ support is also possible in the future since there are people in the community planning to build libtsdb, a JDBC like library for C++ and TSDB.

The editor support is still not smart enough, it can not handle auto completion when only a small fraction of the code is wrong. And it can't format the code, which is an essential feature for team collaboration. It is also possible to support the language server protocol [11] to make it works with Eclipse, Emacs etc.

[10] `https://github.com/xephonhq/tsdb-proxy-java`

8. CONCLUSIONS

Reika aims to provide a unified interface for query and processing data from various time series databases. It tastes like a general purpose programming language but works as a query language. We hope it could become a mature open source project and be an innovation for the time series database community and market.

9. ACKNOWLEDGMENTS

The authors like to thank Zheyuan Chen, Ce Gao and Prof. Peter Alvaro for their previous work and instruction on Xephon-B, from which the TSDB-Proxy project derived. We would also like to thank our instructor Prof. Cormac Flanagan and teaching assistant Dustin in the programming language course, the course project is the driving force for us to hit the initial milestones for Reika.

10. REFERENCES

[1] Grafana. `https://grafana.com/`.

[2] Graphite. `https://graphiteapp.org/`.

[3] Heroic. `https://github.com/spotify/heroic`.

[4] Influxdb. `https://github.com/influxdata/influxdb`.

[5] Juttle. `http://juttle.github.io/`.

[6] Opentsdb. `http://opentsdb.net`.

[7] Presto. `https://prestodb.io/`.

[8] Prometheus. `https://github.com/prometheus/prometheus`.

[9] Translating between monitoring languages. `https://www.robustperception.io/translating-between-monitoring-languages`.

[10] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[11] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the. net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706. ACM, 2006.

[12] Terence Parr. *The definitive ANTLR 4 reference.* Pragmatic Bookshelf, 2013.

[13] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: a fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, 2015.

[11] `https://github.com/Microsoft/language-server-protocol`