RFC: Analysis semi-structured data with Pipe Processing Language

Problem Statement

Nowadays, user have two ways to query the data in Elasticsearch. One is Elasticsearch Query DSL, the other is ODFE SQL. The Elasticsearch Query DSL is provided by Elasticsearch which is the domain sepecifiec language. Although Query DSL is powerful, it has a steep learning curve, and was not designed as a human interface to easily create ad hoc queries and explore user data. The ODFE SQL is provided by ODFE to let customer query Elasticsearch by using the familar SQL language. SQL is not only the de facto standard for data and analytics, but also one of the most popular languages among engineers.
As the name indicated SQL is more focus on Structured Query Language. Elasitcsearch provide JSON-based semi-structued data model which is not suppoted by SQL (PartiQL could support it). In DevOps domain, SQL still to complicated and not provide much more freedom to explore with.
Thus, we proposed the PPL - Pipe Processing Langage to target two issue descirbed above. (1) handle semi-structed data (2) unix pipe style syntax. The PPL is a semi-structured query language which no related to the specific database implementation. In the following section, we will first introduce the Pipe Processing Langage. Then, descirbe the reference implementation in Elasticsearch Plugin as an example.

Pipe Processing Language

A PPL query is a read-only requets to process semi-structured data and return results. A PPL query consists of a series of commands that are delimited by pipe (|) characters. The first whitespace-delimited string after each pipe character controls the command used. The remainder of the text for each command is handled in a manner specific to the given command.

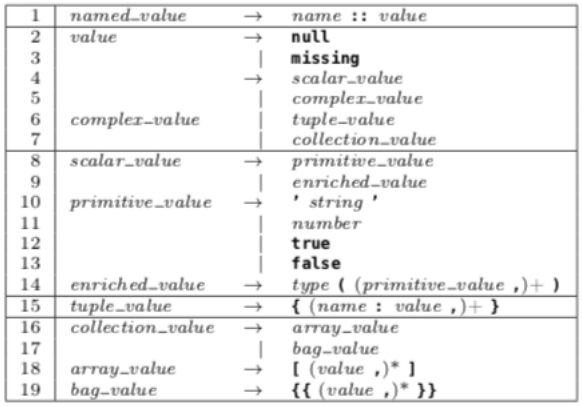
Basic Syntax

```
PPL Statement:  SEARCH command | command1 | command2 ...
```

PPL uses a doc flow model for the search statement. The typical structure of a statement is a composition of commands. The composition is represented by the pipe character (|), giving the statement a very regular form that visually represents the flow of doc from left to right. Each operator accepts a doc set "from the pipe", and additional inputs from the body of the operator, then emits a doc set to the next operator.

Data Model

The PPL data model is borrow from SQL++[1] which is a superset of SQL's relational tables and JSON.



1	*named_value*	→	*name* :: *value*
2	*value*	→	**null**
3		|	**missing**
4		→	*scalar_value*
5		|	*complex_value*
6	*complex_value*	|	*tuple_value*
7		|	*collection_value*
8	*scalar_value*	→	*primitive_value*
9		|	*enriched_value*
10	*primitive_value*	→	' *string* '
11		|	*number*
12		|	**true**
13		|	**false**
14	*enriched_value*	→	*type* **(** (*primitive_value* **,**)+ **)**
15	*tuple_value*	→	**{** (*name* **:** *value* **,**)+ **}**
16	*collection_value*	→	*array_value*
17		|	*bag_value*
18	*array_value*	→	**[** (*value* **,**)* **]**
19	*bag_value*	→	**{{** (*value* **,**)* **}}**

Figure 2: BNF Grammar for SQL++ Values

Binding Tuple

A **binding tuple** denote as <X1:V1,...,Xn:Vn>, where **Xi** is a variable name which ```bind to a PPL value **Vi**.
Then all the operator in PPL is evaluated with a bag/array binding tuples and output a bag/array binding tuples which satisfy the closure property.

> *An important property of operators in general is the closure property. An operator is said to be closed over some given set, if every operation on elements of that given set always results in an(other) element of that same set.*

The following example explain how the PPL command is evaluated with binding tuple. e.g.
There is a existing **access_log** index which has the following data

```
[
    {"ref":"amazon","action":"addtocart","region":"IAD"},
    {"ref":"amazon","action":"addtocart","region":"PDX"},
    {"ref":"amazon","action":"purchase","region":"IAD"},
    {"ref":"amazon","action":"view","region":"IAD"},
    {"ref":"bing","action":"view","region":"IAD"}
]
```

Then the input binding tuple is:

```
<
"access_log":
[
    {"ref":"amazon","action":"addtocart","region":"IAD"},
    {"ref":"amazon","action":"addtocart","region":"PDX"},
    {"ref":"amazon","action":"purchase","region":"IAD"},
    {"ref":"amazon","action":"view","region":"IAD"},
    {"ref":"bing","action":"view","region":"IAD"}
]
>
```

If the PPL command is

```
source=access_log       (1)
| search ref="amazon"   (2)
```

The input binding tuples for command (1) is Bin_of_source which equal to input binding tuple. The output binding tuples denote ad Bout_of_source is

```
[
    <"access_log", {"ref":"amazon","action":"addtocart","region":"IAD"}>,
    <"access_log", {"ref":"amazon","action":"addtocart","region":"PDX"}>,
    <"access_log", {"ref":"amazon","action":"purchase","region":"IAD"}>,
    <"access_log", {"ref":"amazon","action":"view","region":"IAD"}>,
    <"access_log", {"ref":"bing","action":"view","region":"IAD"}>
]
```

The Bin_search=Bout_of_source, the expression ref=”bing“ is evaluated with each binding tuple in Bin_search. In here ref is the syntax sugar of access_log.ref. Then the output binding tuples is

```
[
    <"access_log", {"ref":"bing","action":"view","region":"IAD"}>
]
```

MISSING and NULL

Generic Missing Handling

In general, if any operand evaluates to a **Missing value**, the enclosing operator will return **Missing value** ; if none of operands evaluates to a **Missing value** but there is an operand evaluates to a **Null value**, the enclosing operator will return **Null Value.** However, there are a few exceptions listed in comparison operators and logical operators.

Logical Operator Missing Handling

The following table is the truth table for `AND` and `OR`.

A	B	A AND B	A OR B
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
TRUE	MISSING	MISSING	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
FALSE	MISSING	FALSE	MISSING
NULL	NULL	NULL	NULL
NULL	MISSING	MISSING	NULL
MISSING	MISSING	MISSING	MISSING

The following table demonstrates the results of `NOT` on all possible inputs.

A	NOT A
TRUE	FALSE
FALSE	TRUE
NULL	NULL
MISSING	MISSING

> *Ref: https://asterixdb.apache.org/docs/0.9.3/sqlpp/manual.html*

Command List

Currently, we defined 8 basic commands.

- dedup
- eval
- fields
- rename
- search
- sort
- stats
- where

The detail syntax defintion and example usage in the here.

Proposal Implementation in Elasticsaerch

In this section, we proposla an implmementation of PPL in Elasticsearch. PPL is implemented in Elasticsearch as a plugin and exposed the /_opendistro/_ppl endpoint for query. In the following section, firstly, we will introduce the high level view of the proposed solution, secondly, go deeper how the PPL is implemented in the solution.

Architecture

We demostrate the architecture in execution view and componet view in following sections. The execution view focus on depcit how the request is handling is the system and how the repones is flow back to client. The component view describe the different module in the architecture.

Execution View

The execution view explain how the ppl query go through each component. (1) client send PPL query through endpoint "_opendistor/ppl/". (2) PPL RESTful request handler which running on coordination node receive the request, then call PPLServer to handle the request. (3) PPLServer make the plan call to the Core module to generate the optimized physical plan. (4) PPLServer make the execute call to the Core module to execute the physical plan. (5) During execution stage, the physical plan may call the Elasticsearch TransportSearchService to query/scan the index. (6) The TransportSearchService take care of Elasticsearch query/scan related operation which is transparent to the PPL plugin. (7) The TransportSearchService receive the response of query/scan operation. (8) Then the result return to execution framework in Core module and continue process the response if required. (9) The stream response return to PPLServer. (10) Then finally return to client.

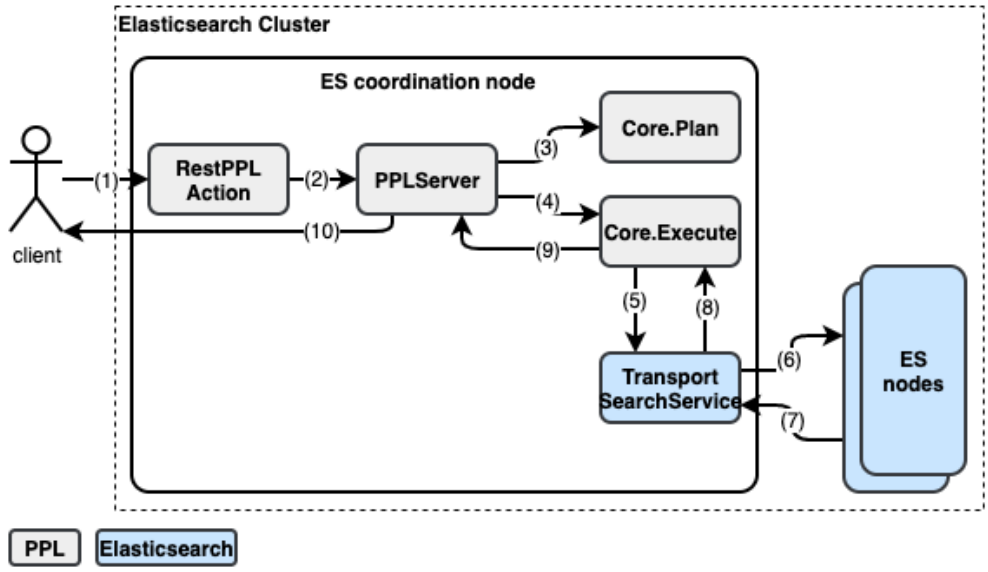


Figure-1 Execution View

Component View

The following component view depict the sub component in the Plugin. In here, we brief introduce the function of each component and will dive deep into each one in the following section.

- Plugin Framework: The plugin framework expose the RESTful endpoint and implement the Elasticsearch plugin framework required interface. Beyond it, it also expose the TransportService API which making it could be integrated with other plugins.
- Management: The resource management and configure management module.
- PPL Server: The PPL Server logical independent of Elasticsearch. The PPL parser is implement in this module.
- Core Engine: The core engine implement the logical plan, physical plan and execution framework. This module is language independent which could be shared by SQL and PPL.
- ESAdapter: The adapter layer between core engine and Elasticsearch.

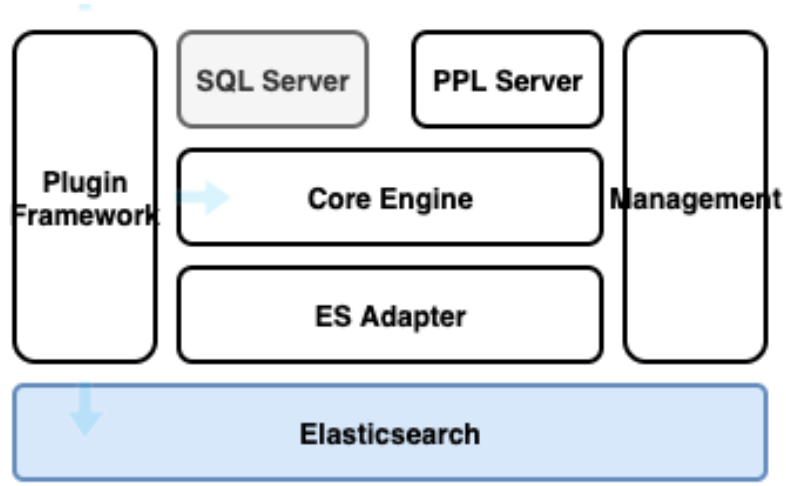
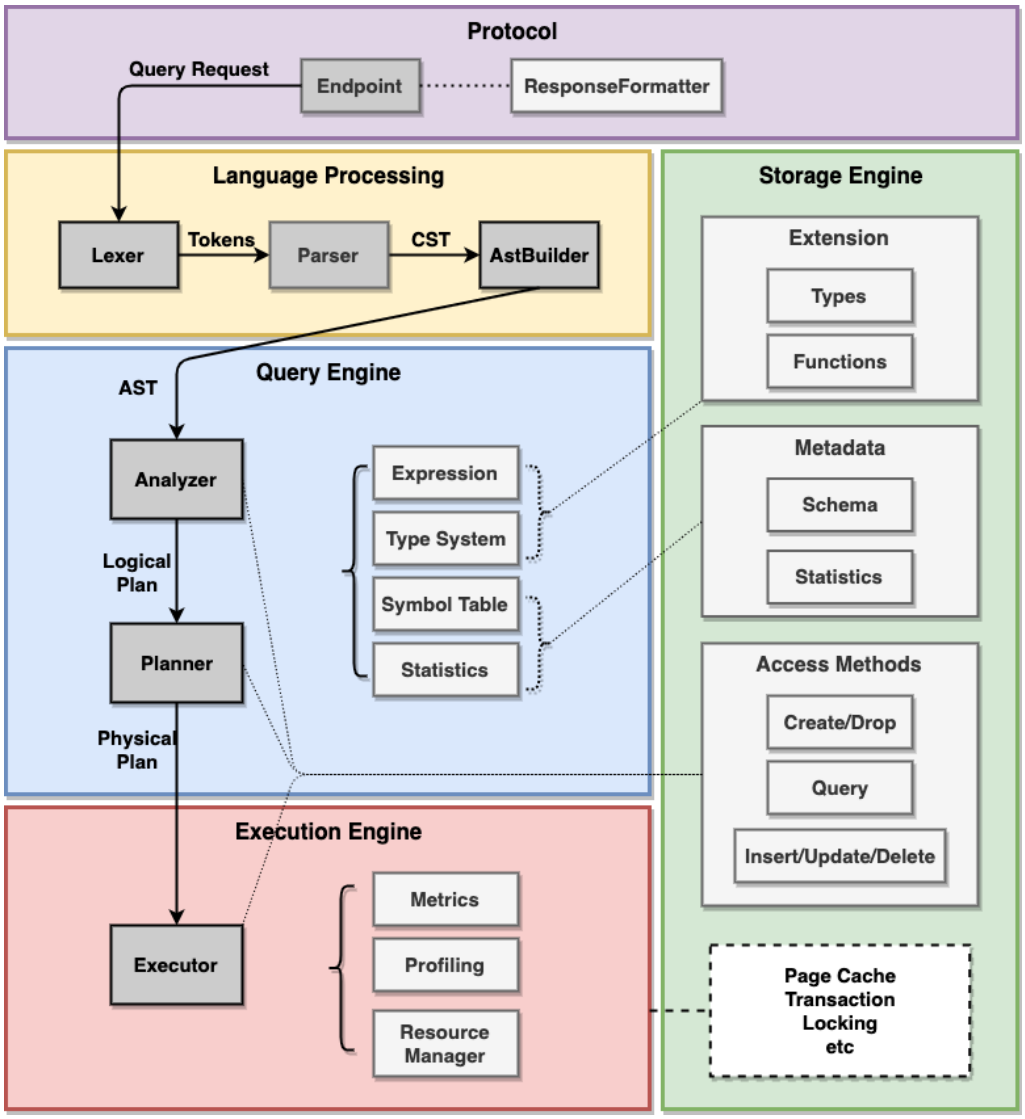


Figure-2 Component View

When we zoom in the component view, we can divide the the archtecture into five major componnets.
In our case, on the high level, a database engine can be thought of as consisting of five components:

1. **Protocol**: Expose endpoint, parse request, specify response formatter, orchestrate the whole processing.
2. **Language Processing**: Served as frontend and parse queries based on language grammar.
3. **Query engine**: Verify, normalize, optimize and plan the query into internal data structure.
4. **Execution engine**: Execute the plan and feed result into response processing pipeline.
5. **Storage engine**: Support the planning and executing of queries.



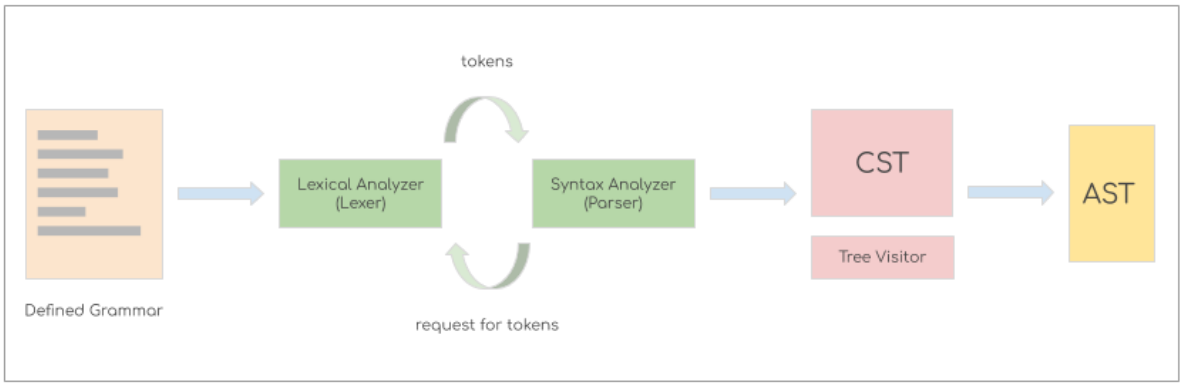
In the following section, we will dive deep two PPL related module and explain how PPL is implemented in there. The other module like store engine is more Elasticsearch related will not be covered.

Language Processing

PPL query parser serves as the syntax recognizer and grammar parser for piped processing language (PPL) component of plugin. In the scope of the entire PPL component, one of the key features in the architecture is the module independence. This parser plays a part only in the front end module, that is to parse the the PPL specification to a set of abstract syntax tree (AST) nodes, so the nodes set enriches the AST with the expected PPL grammar and enables the AST to act as the front end interface for the semantic analysis module and the middle end (logical plan) module.

Tool and parser structure

The ANTLR is a powerful tool to recognize and parse the custom grammar file that specifies a language. ANTLR can generate the parse tree according to the input grammar defined for PPL. Here we define a set of expression nodes that we prepare for the frontend module interface, and did some further processing to build a custom AST that suits the PPL specification well, like guiding the parse tree visitor from the tree nodes parsed from queries, to the expression nodes at the interface, and get the expression nodes all ready without any dependency on the parser module.



ANTLR Parser Structure

The basic structure of the ANTLR parsing tool consists of the grammar files that define the PPL specification, a lexer and a parser that interact with each other by tokens, and finally the AST after processing the tree nodes by modifying the tree visitor. The grammar files are storing the keywords or keyword patterns that might occur in the PPL queries, thus the lexer could recognize them as the tokens and send them to the parser for further processing. Then in the parser, the syntax nodes form the CST. By modifying the parse tree visitor, we can control the visitor to traverse along the designated route, get rid of meaningless components like the commas, parentheses etc. and deal the useful components to the AST nodes.

PPL Grammar

Basically, the grammar files (ANTLR .g4 files) also follow the pattern of the SQL parser. The grammar is divided into the lexical file and syntax file. The lexical file contains the keywords or patterns that highlight the dedicated lexicons in commands and all command components, and the syntax file covers all the PPL syntax formed by the dedicated lexicons. The command grammar is quite straightforward. The following code block is part of the content in the parser file.

```
/** statement */
pplStatement
    : searchCommand (PIPE commands)*
    ;

/** commands */
```

```
commands
    : whereCommand | fieldsCommand | renameCommand | statsCommand | dedupCommand | sor
    ;

searchCommand
    : (SEARCH)? fromClause
    | (SEARCH)? fromClause logicalExpression
    | (SEARCH)? logicalExpression fromClause
    ;

whereCommand
    : WHERE logicalExpression
    ;

fieldsCommand
    : FIELDS (PLUS | MINUS)? wcFieldList
    ;

renameCommand
    : RENAME orignalField=wcFieldExpression AS renamedField=wcFieldExpression
    ;

...
```

Expression in Query Engine

Expression is the core component inside the Query Engine. There are two types of Expression in the core engine. One is Unresolved Expression which is used to build the AST. The other is Resolved Expression which is used in the logical plan and execution plan.

Unresolved Expression

As the name indicated, the unresolved expression is the output of the AST builder which logically represent the literal identifier and function in the command, e.g. the function '`if(error == 200, "OK", "Error")`'. is been represent as the following AST

```
IFExpression
    Expression: condition;  // EqExpression(Identifier(error), Literal(200)
    Expression: trueReturn; // Literal("OK")
    Expression: falseReturn;// Literal("Error")
```

Before introducing the resolved expression, we want highlight the difference between resolved expression and unresolved expression firstly.

- The Unresolved Expression is the logical expression, which only has constructor but without observer method.
- The Unresolved Expression is constructed without semantic check.
- The Unresolved Expression can't been evaluated in the environment.

Resolved Expression

The resolved expression is the core concept in the core engine which is the compiled from unresolved expression in the Analyzer stage. If not specifically pointed out, in the following section we use expression for short.
Each logical plan could include expression. The expression server different purpose in the core engine.

- Expression could be evaluated with BindingTuple. Each logical plan node could denote as f, which f(stream(BindingTuple)) → stream(BindingTuple). The the expression in each command should have the ability to evaluate with BindingTuple.
- Expression should type value. The type check in semantic check is based on each expression has type info, then the type check could be executed during the expression construct time.

Expression Definition

Instead of list the full expression system definition, in here, we list the sample expression definition for demo purpose

```
Expression := literal(var)
Expression := ref(var)
Expression := add(Expression, Expression)
...
```

Then the constructor and observer of the expression systems are

```
Constructors
add: Exp x Exp -> Exp
literal: ExprVal -> Exp
ref: Var -> Exp

observer:
valueOf: Exp x BindingTuple -> ExprVal
typeOf: Exp x BindingTuple -> TypeVal
```

Build Expression from Unresolved Expression

In compile stage, the resolved expression is built from unresolved expression with semantic check. Let's use the example to explain the build process.
The expression builder have two goals,

1. Build the resolved expression from unresolved expression.
2. Do the semantic analysis during the build process.

```
// Indentifier Type Mapping
{
    "a": integer,
    "b": double
}

// (1) Unresolved Expression in AST
uAddExpression = uAdd(uRef("a"), uRef("b"))

// (2) Resolve the uAddExpression recursively
// uAddExpression is the function need to be resolved in the FunctionRepository with s
// The function sinagure is name + argument types
resolve(uAddExpression)

// (3) resolve argument
resolve(uref("a")) -> ref("a")
resolve(uref("b")) -> ref("b")
```

```
// (4) argument type
ref("a").typeOf(typeMapping) -> integer
ref("b").typeOf(typeMapping) -> double

// (5) resolve the uAddExpression in FunctionRepository
FunctionRepository.resolve("add", (integer, double)) -> add(ref("a"), ref("b"))
```

In this section, we explain the procedure to build resolved expression from resolved expression, on the next section we will introduce the semantic analysis procedure.

Semantic Analysis

The semantic analysis include

a. Label checking: index not found, field not found
b. Type checking: wrong field type, wrong number of parameters (if not specified in grammar)
c. Scope checking: function used in wrong place

From dragon book

> *An important part of semantic analysis is type checking. where the compiler checks that each operator has matching operands*

Actually, the major part in build expression from unresolved expression section is doing the semantic analysis. The following section explain each part in detail.

Expression Type Evaluation

When evaluate the expression type, we could dive the expression into three categories.

- literal expression: the type is determined by the value type. e.g. `literal(1).typeOf() → Integer`
- ref expression: the type is determined by resolved the filed in the context, e.g. `ref("a").typeOf({"a", Integer}) → Integer`. Note that, the **semantic label checking** is happened ruing this stage.
- function expression: the type is determined by the procedure's return type. e.g. `add(Integer, Integer) → Integer`.

Resolve Unresolved Function in FunctionRepository

As we list in build expression from unresolved expression section, the unresolved procedure should be resolved in the FunctionRepository with FunctionSignature

FunctionSignature
The function signature is the combination of the function name and a list of parameters type. The function signature distinguish the expression from each other.

Overloading of Function
The function could have different meaning depend on the parameter type. Overloading is programmer friendly polymorphic feature, where a programmer needs to remember only one method name for similar type of operations on different types and number of parameters. The compiler takes care of calling the right function by looking at what type and number of parameters are passed. e.g. the add function could have serval overloading version depend on the parameter.

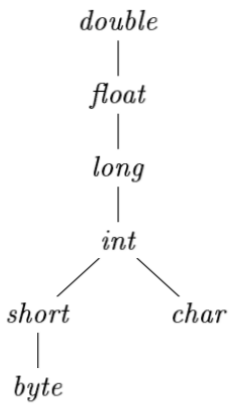
```
Double add(Double, Double)
Float add(Float, Float)
```

```
Long add(Long, Long)
Integer add(Integer, Integer)
```

If the unresolved function has the type exactly match one of the signature, then it could be resolved to one of the overloading version. e.g., The unresolved function has signature: `<“add”, (Double, Double)>` which could be resolved to resolved function has signature `<“add”, (Double, Double)>` directly.
But what if the unresolved expression has parameter which doesn’t exactly match the any of the function signature? The next section will provide one solution.

Widening Primitive Conversion
The widening primitive conversion make the expression signature with widening primitive could accept parameter with narrow primitive.
e.g. The unresolved function has signature `<"add", (Float, Integer)>` which doesn’t have exactly match registered resolved function. resolved directly, Then we need to apply the widening conversion rule and choose the best match resolved function.

- **widening conversion rule**. The widening conversion rules are given by the hierarchy below (From dragon book) which define the compatible type. then the by applying the widening conversion rule, the `<"add", (Float, Integer)>` *is compatible with* `<"add", (Double, Double)>, <"add", (Float, Integer)>`. Then, the next problem is how to choose the best match.
- **compatible distance**. We define the the compatible distance as `SUM(distance(UPi, RFj))`, which
  - TFi is the Type of the i argument in the unresolved function signature, RFi is the Type of the i argument in the registered function signature.
  - The **distance** is then measured by the node distance in the tree. In which
    - distance = 0, when two types are identical.
    - distance = INTEGER.MAX, when two types are not compatible.
    - distance = distanceInTree(type1, type2)
- At the end, we choose the matching registered resolved function with the nearest distance.



(a) Widening conversions

REF: https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html

FunctionRepository

We refer the FunctionRepository many times but didn't touch it yet. The FunctionRepository maintain the predefined the resolved function builder. The unresolved function could be resolved in the function context if possible and get the resolved function.
Each function bundle is identified by the function name in the FunctionRepository. Each function bundle include a bunch of overloaded function builder with different signature. e.g.

```
Double add(Double, Double)
Float add(Float, Float)
Long add(Long, Long)
Integer add(Integer, Integer)
```

Expression Value Evaluation

In execution stage, the expression is evaluated with BindingTuple as the env and produce the ExprVal as the result. Let's take an example to explain the behavior.

```
// BindingTuple
{
    "a": 1,
    "b": 2
}

// Construct Expression by using Expression DSL
addExpression = add(ref("a"), ref("b"))

// Evaluate Expression with BindingTuple.
// The evaluation is executed in application order.
addExpression.valueOf(bindingTuple)
-> (+ ref("a").valueOf(bindingTuple) ref("b").valueOf(bindingTuple))
-> (+ 1 2)
-> 3

// wrap to ExprInterValue
ExprValue = intervalue(3)
```

Null and Missing

In conventional type checking mode, if a symbol value can't been resolved in the env, the exception will been thrown to indicate can not resolve issue.
In Elasticsearch domain, there common case is the mapping is well defined, but some tuple doesn't have the field defined in the mapping. e.g.

```
// mapping
{
  "action": STRING,
  "status": INT,
  "ref": STRING
}

// tuple 1 has all the fields
{
  "action": "add",
  "status": 200,
```

```
  "ref": "www.amazon.com"
}

// but tuple2, missing the ref field
{
  "action": "delete",
  "status": 200,
}
```

For proper handing this case, we defined the **Missing value** as follow,

- if a symbol's type can't been resolved in the type env, the exception will been thrown to indicate the field is not defined.
- if a symbol's value can't been resolved in the value env, it will return special **Missing value**.

Reference

[1] The SQL++ Query Language: Configurable, Unifying and Semi-structured