Siddhi Query Optimization with JIT Code Generation

Introduction

In this article we will discuss about applying 'Just In Time(JIT)' code generation for Siddhi queries to optimize them. Siddhi is a Streaming SQL and it can process events in a streaming manner, detect complex patterns, and notify them in real time. Siddhi uses a pipeline to do this whole process and we can use JIT code generation in places of this pipeline where computationally heavy operations may happen. We observed that event filtering process and pattern detecting process of Siddhi as places where JIT code generation can be applied. This article will discuss how we can use JIT code generation to optimize Siddhi event filtering process with the help of 'ASM' byte-code generating library.

Before going into details about how JIT code generation can be applied to optimize Siddhi filter. Let's have a basic idea about JIT code generation, ASM library and Siddhi Filter processor.

Applies to

WSO2 SP 4.0 and above

Contents

JIT Code Generation

In java, compiler translates a program into a byte-code. Then java virtual machine(JVM) executes that byte-code. Inside the JVM there is a special feature called JIT compiler. JIT compiler identifies sections of the byte-code which are frequently get executed by JVM. So JIT compiler has a threshold number of executions to identify those sections. If a section passes this threshold JIT compiler will optimize the byte-code of that section. So if we can manually generate a byte-code similar to the byte-code generated by the JIT compiler. We can obtain an optimization since we are executing a more optimized code on JVM from the beginning. So if we can identify sections in our code that might be optimized by the JIT compiler and create the JIT code for those sections and execute them on JVM without waiting till JIT compiler

generating them. We can obtain a performance gain. This is the whole idea of JIT code generation. Instead of waiting till JIT compiler to generate the optimized byte-code, we generate it and take the advantage.

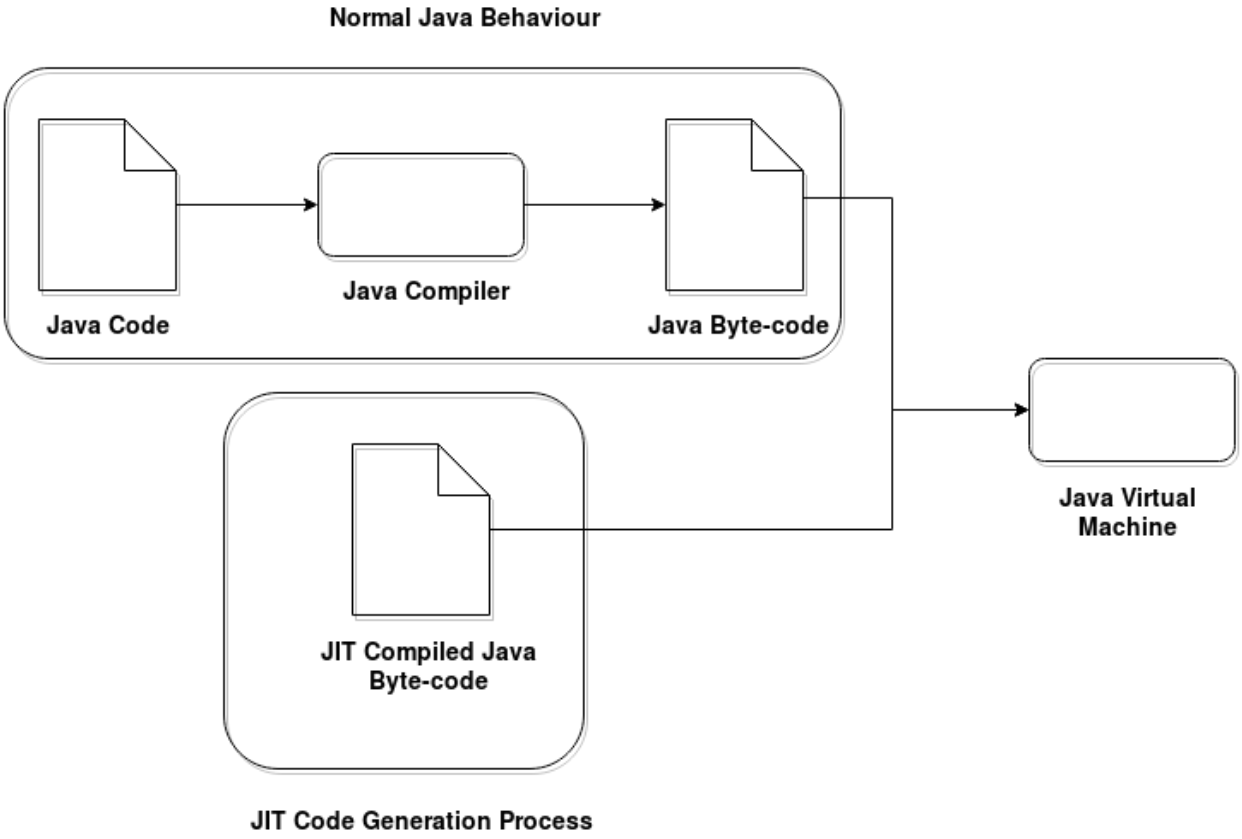


Figure - 1

Since Siddhi is implemented using JAVA we can apply JIT code generation to frequently executing code sections in Siddhi. Hence Siddhi uses a pipeline to handle queries, there are places in this pipeline where computationally heavy tasks happen. Every time when a Siddhi query is executing this pipeline gets executed. If we can apply JIT code to places where heavy computational tasks happen, we can optimize this pipeline. As a result of it Siddhi queries will be handled faster. So we are creating a more optimized byte-code for Siddhi query pipeline without waiting till JIT compiler to do that.

ASM Library.

To create JIT code we need some sort of a tool to manipulate byte-code. We have to create byte-code that we need to execute on the JVM. For that we can use

ASM which is a byte-code manipulating library. We can create the byte-code we need to execute on JVM using ASM.

ASM allows users to create classes in the form of byte-code using ASM features. These classes can contain methods, fields, constructors, etc. ASM provides byte-code that is being generated using ASM in a form of a byte array. There are 3 main features in ASM that is needed if someone needs to generate byte-code using ASM. They are,

- ClassWriter
- MethodVisitor
- FieldVisitor

Basically we can create a class in the form of byte-code using ASM. We need a class writer instance for each class we are creating in the form of byte-code. Then we need to add methods for this class. For that we can create an instance of MethodVisitor related to the ClassWriter instance of the class which we need to generate the byte-code. We can add any number of methods to the class using instances of MethodVisitor. If we need to add variables to this class we can use an instance of FieldVisitor related to the ClassWriter instance of the class which we need to generate the byte-code. Simply, using these features of ASM we can create a complete class in the form of byte-code.

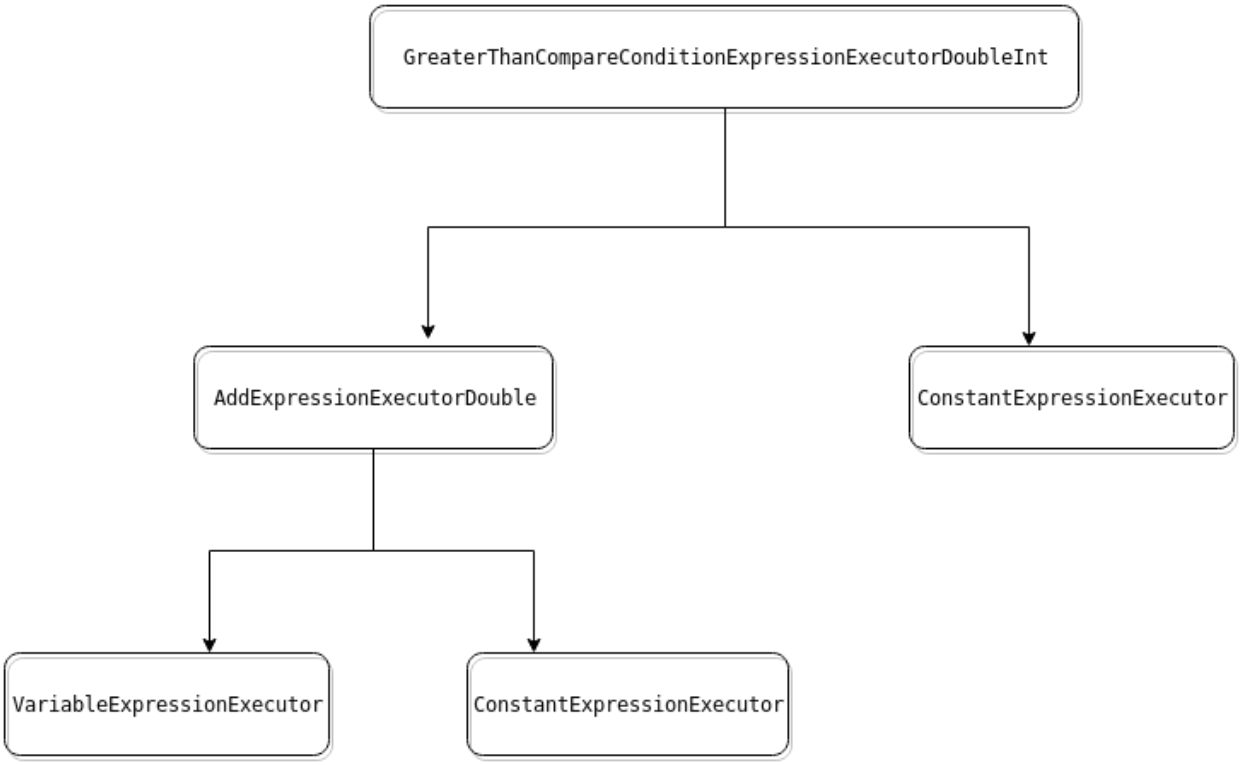
Siddhi Filter Processor

Siddhi uses a tree structure to identify events which are needed to be moved forward in the pipeline. For each event, Siddhi traverses this tree and finds a boolean value to find whether to drop the event or not. For an example, let's consider a tree structure that is built for a sample query.

Sample query: TestAverage + 1 > 46

Corresponding tree structure generated by Siddhi in order to evaluate this filter condition is as follows.

(note: Assume TestAverage as a primitive double.)



So for each event arrival Siddhi will traverse this tree to get a boolean result. Siddhi creates this tree structure using a class called ‘ExpressionExecutor’ . Each ExpressionExecutor will have it’s child node or child nodes. Leaf nodes of this tree would be a VariableExpressionExecutor or a ConstantExpressionExecutor. So Siddhi has logical operators, mathematical operators, compare operators and functions written using ExpressionExecutors. Using these ExpressionExecutors Siddhi will create a tree to match the user given filter condition. Simply, this tree will illustrate a clear picture about the filter condition and it will contain the user given filter logic in an executable manner.

Siddhi will evaluate the filter condition by doing a leftmost traverse. Inside the tree there are some optimizations that have done in order to skip unnecessary operations. For an example if current node represents logic for ‘and’ operation then, if it’s left subtree returns false Siddhi will not traverse the right subtree and move into the parent node. These kinds of optimizations are currently existing in Siddhi. The main reason for using a tree to evaluate an user given filter condition is that Siddhi cannot evaluate a user given filter condition directly. Because Siddhi cannot guarantee that the user will give correct filter condition without errors. So Siddhi needs an iterative manner to evaluate the user given filter condition and to check the errors in the user given filter condition. So, this tree structure is a must for Siddhi.

When we consider this tree from a JIT code generation perspective this is a perfect place to generate JIT code. Mainly this tree is frequently used because for every event arrival this tree needs to be executed. Also we can clearly identify that using a tree to evaluate a boolean expression is not the optimum way. So currently, byte-code related to filtering process of Siddhi is not the best optimize byte-code we can run on the JVM. If we can generate a more optimized byte-code for this section, Siddhi pipeline will run more faster as filtering process consumes considerable amount of computational task in normal Siddhi query evaluation process.

Application of JIT code generation for Siddhi Filter Processor.

Simply we can reduce this tree structure used by Siddhi to filter events to a single ‘if’ condition at byte-code level. Siddhi creates a tree after checking errors in the user given filter condition. Though the user given filter condition can have errors, tree generated by Siddhi is a perfect error less logic. We cannot use the user given filter condition directly inside an ‘if’ statement. But we can reduce the tree generated by Siddhi to an ‘if’ condition at byte-code level. So at the runtime we are executing a byte-code for an ‘if’ condition not for a tree. This will increase the performance of Siddhi queries.

The main reason why we can obtain a performance gain is that we reduce the number of jump operations done from one method frame to another. Simply jumping from one method frame to another is a costly operation at byte-code level. Because when jumping to another method frame, JVM has to store operand stack and local variable array of current method frame and reload them when execution return to that frame. So if we are doing computationally heavy operations, operand stack and local variable array will be large. So storing and restoring large chunks of data consumes more time.

So traversing a tree will cause jump operations at byte-code level. Simply when Siddhi goes from one node to another in the tree, jumping from one frame to another will occur. First, we need to reduce these jumping operations. We need to put all the logic into a single method frame instead of using multiple method frames. To do this we need to have the logic we need to put into a single method frame before creating that method frame. As we have the tree generated by Siddhi for filter condition, we can use that tree to take the filter logic and put it into a single method frame.

So if we can create an ExpressionExecutor that contains a method which has less number of jump operations using byte-code, our goal of reducing the tree to a single ‘if’ condition can be occurred. Inside the ExpressionExecutor which we are creating using the byte-code, a method created using byte-code will give a boolean result according to the event. This method will use its local variable array and operand

stack to minimize jump operations. Basically, we add the byte-code instructions into this method which will represent the logic given by the user in the filter condition. At the run time these instructions will get executed and the final boolean result will contain in the operand stack.

We will create a JAVA class similar to below one using byte-code. Let's name the class we are creating using byte-code as JITCompiledExpressionExecutor.

```
public class JITCompiledExpressionExecutor implements ExpressionExecutor {
        private ArrayList<ExpressionExecutor> unhandledExpressionExecutors;

        JITCompiledExpressionExecutor
        (ArrayList<ExpressionExecutor>unhandledExpressionExecutors) {
                this.unhandledExpressionExecutors = unhandledExpressionExecutors;
        }

        Object execute(ComplexEvent complexEvent) {
                return "boolean result obtained after evaluating the logic created using JIT
                code generation";
        }

        public Attribute.Type getReturnType() {
                return Attribute.Type.BOOL;
        }

        public ExpressionExecutor cloneExecutor(String key) {
                For (int i = 0 ; i < unhandledExpressionExecutors.size(); i++) {
                        unhandledExpressionExecutors.get(i). cloneExecutor(key);
                }
                return this;
        }
}
```

So the method we are creating with minimum number of jump operations is represented as 'execute' which is the overridden method from the interface ExpressionExecutor. This method will consume an event as its parameter and will give a result whether this event needs to be proceed or not.

Also Siddhi has extensions and users can create extensions for Siddhi as well. So it's not practical to generate byte-code for Siddhi extensions as well. So what we can

do is that keep respective ExpressionExecutor for the extension and execute it in runtime and put the result of that ExpressionExecutor into the byte-code. As this is an operation where jump operation happens from one frame to another, this will hinder our main goal of reducing the number of jump operations. But we have to do this and it will not cause a huge problem because most of the time filter queries will not have extensions inside them. So while creating the byte-code we can keep track of ExpressionExecutors, those are not handled by byte-code and assign those to a local variable of the class JITCompiledExpressionExecutor. To assign these ExpressionExecutors we can use the constructor of JITCompiledExpressionExecutor. At runtime JITCompiledExpressionExecutor will consume those ExpressionExecutors.

Other than execute method we have two other methods inside the ExpressionExecutor which we created using byte-code. We need these two methods in order to unbreak the normal flow of Siddhi. Also we are using constructor of the class generated using byte-code to set the local variable that contains Expressionexecutors for those we haven't prepared the byte-code. We need to have clone method because Siddhi, clone processes from current process by giving a key. Basically cloning process of Siddhi will provide an ExpressionExecutor after running its clone method . Simply Siddhi will update the current tree from the key and will return the tree. This is the process of cloning. So what we need to do inside the clone method of the class we are creating using byte-code is to just run the clone methods of the ExpressionExecutors for those we are not generating byte-code and give an instance of the byte-code class created using byte-code. When we consider about getReturnType method, Siddhi normally uses it to identify the order of connecting ExpreesionExecutors. Simply root node of the tree must be an ExpressionExecutor which will return a boolean result as finally we need a boolean result. As our class created using byte-code gives a boolean result getReturnType method will give BOOL as the type.

When we consider about creating the class using byte code we need to utilize the properties of ASM. Basically we need a ClassWriter instance to create the class and four instances of MethodVisitor to create class constructor, method execute, method clone and method getReturnType. Also we need an instance of FieldVisitor in order to create a field inside the class. Then we need to add instructions to the execute method while traversing the tree. After fully traversing the tree we can completely build the logic of the filter condition inside the execute method. After completely creating the class we can obtain the class in a form of a byte array using ASM. So we can use a class loader and create a class from this byte array. Then we can use JAVA reflection to create an instance from that class by providing an Arraylist of ExpressionExecutors as the constructor parameter. As this class is an ExpressionExecutor we can cast this instance into an ExpressionExecutor and execute the method 'execute'. This execute method will

provide the boolean result to drop the event or not. But the speciality is that this execute method is a more optimized one as a result of JIT code generation.

Conclusion

JIT code generation can be applied to Siddhi in order to optimize queries. It is possible to generate JIT code for frequently executing code segments of Siddhi at deployment time and gain performance optimizations at runtime. Siddhi handles queries using a pipeline. For each event arrival Siddhi uses the query pipeline. JIT code generation can be applied to sections of query pipeline. Siddhi filter and Siddhi pattern sections can be replaced using JIT code generation. JIT code generation for Siddhi filter has implemented by replacing its tree structure by a single ‘if’ condition at byte-code level.

References

- Java virtual machine specification.
- ASM byte-code generating library.
- IBM Knowledge Center.
- Siddhi Query Guide.