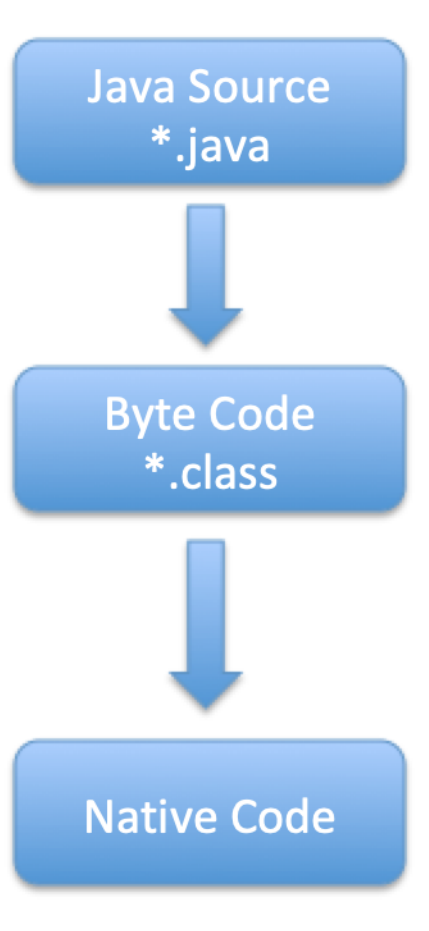
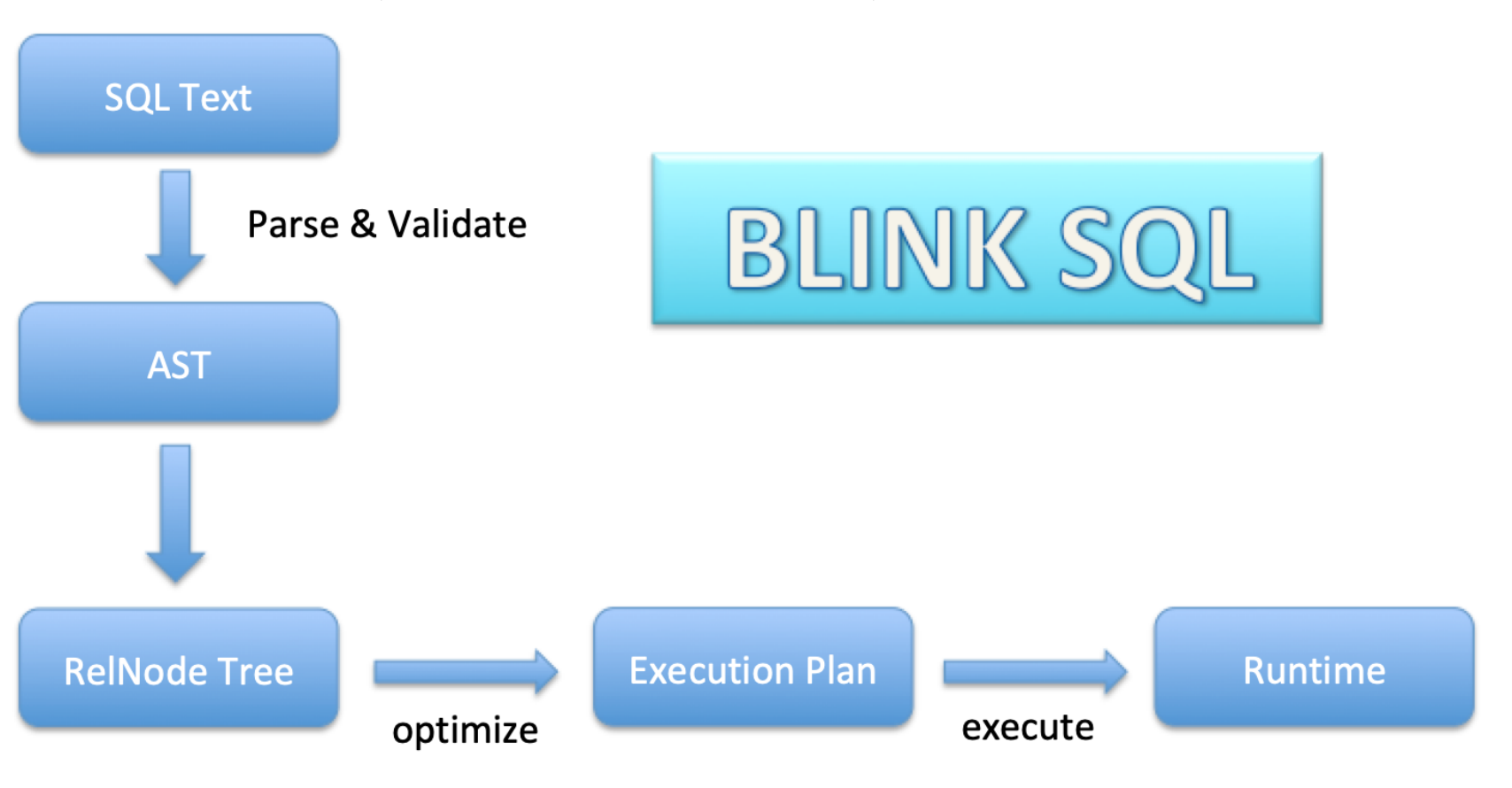
年度盛事双十一落幕, 又到了总结的时刻. 乍一看文章标题有些唬人, 高效计算引擎指的是以 Blink 为代表的新生代计算引擎, 而代码自动生成技术并不是某种 AI 应用, 只是术语 Code Generation 的惯例翻译.
这里试图先抛开代码实现细节, 从另一个角度去思考代码自动生成对于计算引擎效率的意义.

什么是代码自动生成?

自动代码生成(Code Generation)在计算机领域是一种广泛使用的技术.
在编译器中, 自动代码生成是指编译器的代码生成器将源代码的某种中间表示(Intermediate Representation, 简称 IR) 转换为可由机器执行的形式（如机器码）的过程[1.].
比如 Java Source Code -> Byte Code -> Native Machine Code 的过程中, 生成 Machine Code 这个阶段就应用了 Code Generation.



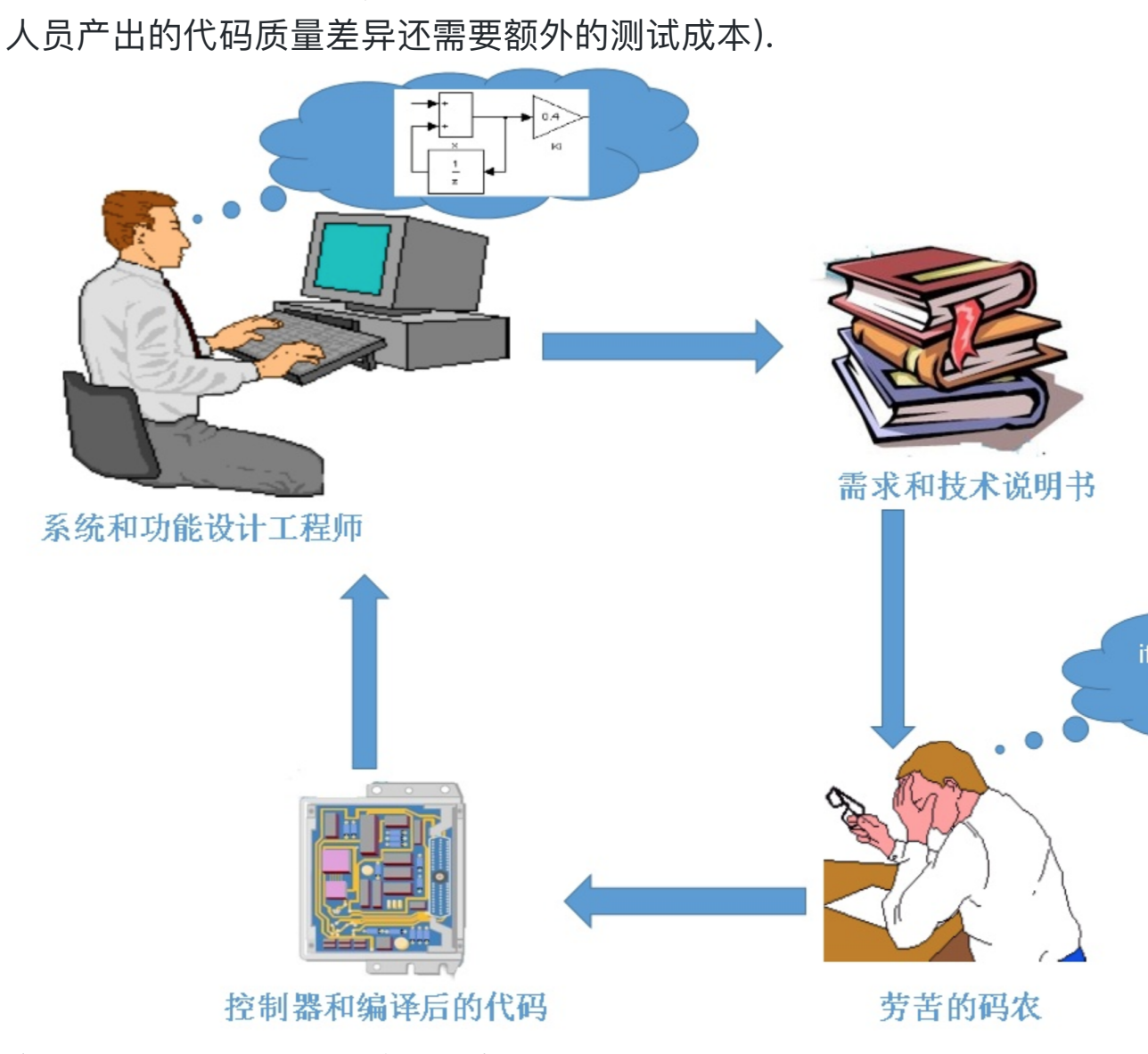
又比如 Java Object-Relational Mapping(ORM) 应用, 简单的配置就能实现常用的如 CRUD操作.
具体到 BLINK 计算引擎上, 对于一个 BLINK SQL 用户作业, 从提交到最终执行大体上会经过如下主要过程:



其中生成 Execution Plan 这步就大量应用了 Code Generation, 而后交付给 Blink Runtime 调度执行. SQL 本身作为一种成熟的 DSL, 能够精确的描述关系操作的语义, 用户使用 SQL 开发一个作业描述的主要是数据操作逻辑, 并没有直接编写执行代码, 从这个角度来看, 整个 SQL 引擎就是一个大的 Code Generator, 能够将用户提交的 DSL 转化成 IR 并最终变成可执行的 Machine Code. 简单来说, 用 Blink SQL 开发运行用户作业, 其实就是在使用 Blink 引擎的代码自动生成技术.
此处留一个问题: 同样的计算逻辑, 用户完全可以使用 Blink 更底层的 API 来开发实现, 为何要使用 Blink SQL?

为什么需要代码自动生成?

在谈 Blink 之前, 先来看一个来源于知乎上的例子[2.]--汽车行业中的 Code Generation 应用.
车辆上有很多控制软件，比如发动机控制软件/自动变速箱控制软件/车身控制软件...， 最初都是手写的 C/汇编代码, 但这些软件代码的开发有很大的重复性, 需要花费巨大的人力和时间成本(不同的开发人员产出的代码质量差异还需要额外的测试成本).



应用 Code Generation 前的开发模式

随着技术的发展, 出现了基于模型设计的工具软件, 能够自动生成代码, 汽车行业就逐渐从外包开发模式转到了以自动代码生成为主的模式

应用 Code Generation 后的开发模式

模式改变后好处是巨大的:
1. 节约了闭环设计中无数的时间和人力成本
2. 系统和功能设计工程师可以独立完成软件的输出，避免技术文档描述不准确或者歧义而导致的从功能到代码的错误, 软件整体的质量更稳定
3. 软件功能修改后可以快速自动生成代码而不需要经过复杂的流程由开发工程师来做修改, 调试方便

```
在谈到 Blink 计算引擎前为何要引述汽车行业的例子呢? 相信通过上述介绍, 大家对引入代码自动生成技术对汽车控制软件的效率提升已有了直观的了解.
回到 Blink 计算引擎, 我们在前文解释什么是自动代码生成时提到了一个问题: 为何要使用 Blink SQL? 对照汽车行业, 使用 Blink SQL 的用户就像汽车行业的系统功能设计工程师, 需要设计实现所处领域的数据处理功
```

Blink SQL 基于 ANSI-SQL, 有成熟、标准的语义(相当于提供了一种标准化的设计语言), Blink SQL 按照标准的语义交付内置计算函数, 保证其正确性和运行效率, 这样各领域的用户可以独立完成数据处理作业的开发、调试、修改.

这些都是理论上对用户的好处, 举一个典型的需要 CodeGen 提升运行效率的例子:
比如一个简单的算术表达式:

```
SELECT a + 1, ...
```

如果用手写代码来实现, 通常需要实现一个加法函数, 在运行时通过函数调用来获取最终的加和结果, 而 CodeGen 之后就能变成语言原生支持的表达式, 相比之下省去了函数开销(关于函数调用开销的优化 可以参考这篇 Spark Blog).
另外, 在 Blink 内部开发中, 同样也在重复性高的部分引入了代码自动生成技术, 从而减少冗余代码量, 提升开发效率, 控制代码质量.

代码自动生成如何工作?

回到实现层面, 我们再来看看代码自动生成在计算引擎中是如何工作的.
以经典的 WordCount 为例, 首先来看使用 DataSet API 编写的 Example

DataSet 开发 WordCount

```
public class WordCount {

    public static void main(String[] args) throws Exception {

        final ParameterTool params = ParameterTool.fromArgs(args);

        // set up the execution environment
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        // make parameters available in the web interface
        env.getConfig().setGlobalJobParameters(params);

        // get input data
        DataSet<String> text;
        if (params.has("input")) {
            // read the text file from given input path
            text = env.readTextFile(params.get("input"));
        } else {
            // get default test text data
            System.out.println("Executing WordCount example with default input data set.");
            System.out.println("Use --input to specify file input.");
            text = WordCountData.getDefaultTextLineDataSet(env);
        }

        DataSet<Tuple2<String, Integer>> counts =
                // split up the lines in pairs (2-tuples) containing: (word,1)
                text.flatMap(new Tokenizer())
                // group by the tuple field "0" and sum up tuple field "1"
                .groupBy(0)
                .sum(1);

        // emit result
        if (params.has("output")) {
            counts.writeAsCsv(params.get("output"), "\n", " ");
            // execute program
            env.execute("WordCount Example");
        } else {
            System.out.println("Printing result to stdout. Use --output to specify output path.");
            counts.print();
        }

    }

    /**
     * Implements the string tokenizer that splits sentences into words as a user-defined
     * FlatMapFunction. The function takes a line (String) and splits it into
     * multiple pairs in the form of "(word,1)" ({@code Tuple2<String, Integer>}).
     */
    public static final class Tokenizer implements FlatMapFunction<String, Tuple2<String, Integer>> {

        @Override
        public void flatMap(String value, Collector<Tuple2<String, Integer>> out) {
            // normalize and split the line
            String[] tokens = value.toLowerCase().split("\\W+");

            // emit the pairs
            for (String token : tokens) {
                if (token.length() > 0) {
                    out.collect(new Tuple2<String, Integer>(token, 1));
                }
            }
        }
    }
}
```

可以注意到核心逻辑的描述相较于 MapReduce 已变得更直观简洁

```
text.flatMap(new Tokenizer()).groupBy(0).sum(1);
```

针对 String 实现一个 Tokenizer 可以快速的让这个例子正常运行, 一切看起来很顺利.

SQL 开发 WordCount

再看 SQL 的描述(假设这里的 SPLIT 表值函数支持以 word boundary 正则 '\b' 对字符串进行切分):

```
SELECT
    word, COUNT(*) as frequency
FROM inputTable, LATERAL TABLE(SPLIT(line, '\\b')) as T(word)
GROUP BY word
```

处理逻辑完整表达在 SQL 中, 对应的执行计划大致如下:

```
Plan:
HashAggregate(groupBy=[word], select=[v, COUNT(*) AS cnt])
   +- Exchange(distribution=[hash[word]])
      +- Correlate(correlate=[table(SPLIT($cor0.f, '\b'))], select=[word], rowType=[RecordType(VARCHAR(65536) word)], joinType=[INNER])
         +- BoundedDataStreamScan(table=[[inputTable]], fields=[line])
```

而后会通过 Code Generator 生成运行时的执行代码.

对比分析

此时对比两种方式看起来区别并不是很大. 如果需求变更, 因发现输入的文本量巨大, 分词后到聚合统计的数据交互压力较大, 需要做性能优化, 此时对于 DataSet API 的作业, 需要改写 String Tokenizer, 增加预聚合处理, 每次分词不再简单输出 Tuple2(token, 1), 而是计算单行文本中每个 Word 的 Frequency, 输出 Tuple2(token, frequency).

再来看 SQL, 因为 BLINK Optimizer 内置了预聚合处理, 同样的 SQL, 可以得到如下优化后的 Plan:

```
== Optimized Plan ==
HashAggregate(isMerge=[true], groupBy=[v], select=[v, Final_COUNT(count$0) AS cnt])
+- Exchange(distribution=[hash[v]])
   +- LocalHashAggregate(groupBy=[v], select=[v, Partial_COUNT(*) AS count$0])
      +- Correlate(correlate=[table(SPLIT($cor0.f, '\b'))], select=[f0], rowType=[RecordType(VARCHAR(65536) f0)], joinType=[INNER])
         +- BoundedDataStreamScan(table=[[inputTable]], fields=[line])
```

可以看到执行计划中增加了一层预聚合计算(Final_COUNT 对应的是 SUM 计算, Plan 的优化细节可以参考《Blink SQL 核心解密 -- 如何彻底解决数据倾斜难题》)
如果功能需求再次变更, 只需要统计少量特定 Word 的词频, DataSet 编写的作业就需要修改对应的代码, 将目标 Words 编码到分词器中, 而 SQL 则可以用标准的 Where 子句来描述新增的过滤条件, 显然 SQL 能更灵活的应对变更(同时也有更多的优化可能, 比如把特定目标词的过滤条件下推到读数据的节点, 尽早过滤数据).

进一步分析

Blink 的代码自动生成实际上做了更多的优化, 比如能够根据数据类型信息进行优化(SQL 都是强类型), 例如对于 Primitive 的类型, CodeGen 时省去拆箱/装箱操作, 减少对象产生; 对于 String 类型, CodeGen 时不再使用 Java String， 而是直接使用 UTF8 编码的 BinaryString, 减少拷贝和编码开销(更多细节请参考 《Blink SQL 核心解密 —— 高性能二进制存储与计算》); 一些具体数据结构和函数的实现的代码，可以直接嵌入到算子的处理函数之中，节省了大量的函数调用开销;
对于上述 WordCount 例子, 以第一版的 SQL Plan 为例

```
Plan:
HashAggregate(groupBy=[word], select=[v, COUNT(*) AS cnt])
   +- Exchange(distribution=[hash[word]])
      +- Correlate(correlate=[table(SPLIT($cor0.f, '\b'))], select=[word], rowType=[RecordType(VARCHAR(65536) word)], joinType=[INNER])
         +- BoundedDataStreamScan(table=[[inputTable]], fields=[line])
```

在生成的读数据算子(BoundedDataStreamScan)中, 如果数据来源于网络字节流(如 TimeTunnel), 在解析时因使用了 BinaryString, 并不会立即产生一个 Java String 对象, 紧接着的表值函数 SPLIT 操作也会可以使用 Binary 的切分方式, 不产生 String 对象直接 shuffle 到聚合计算节点, 在计算过程中也可以继续保持 Binary 的形式, 累积 COUNT 值, 直到最终输出时需要看到 String 形式的结果, 才产生对应的 String 对象, 整个过程的优化如果需要在 DataSet 作业中加入, 修改量可以想象, 更重要的是, SQL 上的这些内置优化是共享的, 并且在不断的迭代进一步优化中.

Blink 的内部演进

Blink 2.x 版本在 Code Generation 上有了较大的改变, 包括 Expression 的推广使用 / Aggregator 优化 / CodeGen HashFunction ...
举一个 SUM Aggregator 的例子, 先来看之前版本的写法:

```
/** The initial accumulator for Sum aggregate function */
class SumAccumulator[T] extends JTuple2[T, Boolean]

/**
  * Base class for built-in Sum aggregate function
  *
  * @tparam T the type for the aggregation result
  */
abstract class SumAggFunction[T: Numeric] extends AggregateFunction[T, SumAccumulator[T]] {

  private val numeric = implicitly[Numeric[T]]

  override def createAccumulator(): SumAccumulator[T] = {
    val acc = new SumAccumulator[T]()
    acc.f0 = numeric.zero //sum
    acc.f1 = false
    acc
  }

  def accumulate(accumulator: SumAccumulator[T], value: Any): Unit = {
    if (value != null) {
      val v = value.asInstanceOf[T]
      accumulator.f0 = numeric.plus(v, accumulator.f0)
      accumulator.f1 = true
    }
  }

  override def getValue(accumulator: SumAccumulator[T]): T = {
    if (accumulator.f1) {
      accumulator.f0
    } else {
      null.asInstanceOf[T]
    }
  }

  def merge(acc: SumAccumulator[T], its: JIterable[SumAccumulator[T]]): Unit = {
    val iter = its.iterator()
    while (iter.hasNext) {
      val a = iter.next()
      if (a.f1) {
        acc.f0 = numeric.plus(acc.f0, a.f0)
        acc.f1 = true
      }
    }
  }

  def resetAccumulator(acc: SumAccumulator[T]): Unit = {
    acc.f0 = numeric.zero
    acc.f1 = false
  }

  override def getAccumulatorType: DataType = {
    DataTypes.createTupleType(
      classOf[SumAccumulator[T]],
      getValueTypeInfo,
      DataTypes.BOOLEAN)
  }

  def getValueTypeInfo: DataType
}

/**
  * Built-in Byte Sum aggregate function
  */
class ByteSumAggFunction extends SumAggFunction[Byte] {
  override def getValueTypeInfo = DataTypes.BYTE
}

/**
  * Built-in Long Sum aggregate function
  */
class LongSumAggFunction extends SumAggFunction[Long] {
  override def getValueTypeInfo = DataTypes.LONG
}

...
```

对一个具体的 `SUM(a)` 调用, 旧版本框架代码会在运行时的代码路径上多一层具体 Agg function 的函数调用, 比如框架的
`aggregateProcessor.accumulate -> sumAggFunction.accumulate`
而应用 Expression 之后, SumAggFunction 简化成了

```
abstract class SumAggFunction extends DeclarativeAggregateFunction {

  override def inputCount: Int = 1

  private lazy val sum = UnresolvedAggBufferReference("sum", getResultType)

  override def aggBufferAttributes: Seq[UnresolvedAggBufferReference] = Seq(sum)

  override def initialValuesExpressions: Seq[Expression] = Seq(
    /* sum = */ Null(sum.resultType)
  )

  override def accumulateExpressions: Seq[Expression] = Seq(
    /* sum = */ IsNull(operands(0)) ? (sum, IsNull(sum) ? (operands(0), sum + operands(0)))
  )

  override def mergeExpressions: Seq[Expression] = Seq(
    If(IsNull(sum.right), sum.left,
      If(IsNull(sum.left), sum.right, sum.left + sum.right))
  )

  override def getValueExpression: Expression = sum
}
```

CodeGen 时直接将具体的 SumAggregator 生成到 Agg 算子代码中, 这样在运行时就少了一层函数调用.
再看 CodeGen HashFunction 的收益, 在数据处理中, 常常用到 Row 的数据结构, 对 Row 的比较操作会涉及 HashCode 的计算, 之前的版本通过 Row 的类型信息创建 RowComparator, 在计算 Hash 时需要遍历 Row 的 Field, 逐个 Field 按类型获取值然后计算 HashCode, 其中涉及了 Field 值获取和单独的 HashCode 计算开销. 在新版本中, 针对整个 BinaryRow 类型信息, 直接生成 Flatten 的 HashCode
Function, 不仅拿到了 Binary 化的取值开销减小, 在 HashCode 计算上, 也变成了一个方法内的计算过程(在 Row Field 多的情况下尤其明显), 整体上节约了 CPU 开销, 提升了计算效率.

类似的优化细节在新版本的 Blink 中还有很多, 具体细节就不再在这里一一赘述了.

代码自动生成后续会如何发展?

超长问题的处理 基于语义的切分
当前在一些较复杂的 SQL 中可能会出现 CodeGen 的代码超长的问题, 这是所有 Java Code Generator 都会面对的一种情形, Blink 之前的版本和 Spark 当前的处理方式类似, 针对 Code Generation 过程中可能出现超长的地方进行切分处理, 这种方式因为耦合到了具体的 CodeGen 处理代码中, 对整体框架有较大的侵入, Blink 在 2.2.4 版本开始已引入了实验性的按语义切分的方式(Spark 社区也开始有相关的提议), 会继续完善.
Whole-Stage Codegen / 更多算子级别的优化
关注 Spark 的同学可能听说过 Spark 的 Whole-Stage Codegen 优化, Blink SQL 中也做了算子级别的 Code Generation 来减少虚函数调用, 只是目前基于投入产出比的考虑没有全面采用这种方案, 后续会继续权衡架构的调整.
向量化计算优化 vertorization/SIMD

1. Code Generation Wiki)
2. 汽车行业代码自动生成的应用