Hitting a 10x Performance for Expression Evaluation with Vectorized Execution and Community's Help）

前言

TiDB 是一款定位于 HTAP 的数据库产品，需要具备快速处理“大查询”（完成该查询需要处理大量数据）的能力。自创立起，TiDB 的执行引擎参照 Volcano 模型实现，它为每个关系代数运算符抽象出统一的数据访问接口 open-next-close，并以行为单位进行迭代和数据处理，简单且扩展性强。但是当处理“大查询”时，Volcano 模型解释开销很大，并且不能充分利用现代 CPU 的硬件特性（CPU 缓存，分支预测，流水线执行等），于是我们借鉴 Monet/X100 的思路，对 TiDB 进行了向量化的改进。（Andy Pavlo 老师课程中关于 Query Engine 的这一章节，对各种执行模型及表达式求值都有非常棒的讲解，感兴趣的同学可以作为补充阅读）

从 2017 年末开始，TiDB 走上了向量化执行引擎的道路，主要做了下面三个优化，取得了不错的性能提升：

- 一是将数据在 TiDB 内存中的布局，从行式改成了列式存储，见 PR/4856
- 二是将 Volcano 按行迭代的模型改成了按批（1024 行为一批）迭代，见 PR/5178
- 三是对部分算子的执行进行了一定程度的向量化优化，见 PR/5184

完成优化后的 TiDB 2.0 版本和 TiDB 1.0 相比，大部分 TPC-H Query 的性能都有了数量级的提升，详细的性能对比测试可参考 TiDB TPC-H 50G Performance Test Report。

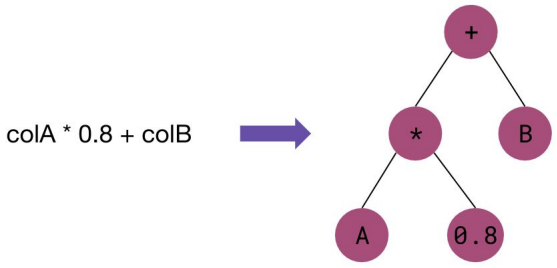
随着时间的推移，TiDB 先后发布了 2.1、3.0 版本，向量化执行引擎运行的越来越稳定。在正在开发的 TiDB 4.0 中，我们决定继续对一些 SQL 算子进行深度的向量化优化，以满足用户对 TiDB 4.0 性能的要求。表达式的向量化优化就是其中工程最为浩大的一个子项目，因为表达式包含各种内置函数（TiDB 内有 500+ 个），我们需要为全部的内置函数进行向量化重构。在我们和社区同学的努力下，大多数内置函数都取得了非常显著的性能提升，部分提升甚至有一到两个数量级，使得整体来看表达式计算的性能有了大幅提高。

在这篇文章中，我们将描述整个表达式向量化优化的过程，并结合 TiDB 源码向大家解释这些性能提升来自哪，以及我们怎么利用代码生成技术和社区的力量帮助我们快速完成内置函数向量化重构的。

TiDB 表达式计算介绍

我们以表达式 `colA * 0.8 + colB` 为例来介绍 TiDB 目前怎么完成表达式计算。

为计算这个表达式，TiDB 会按照算术运算符和他们的运算优先级将它解析成为一棵表达式计算树，树中每个节点代表一种算术运算符，树的叶子节点表示数据源，要么是常量（如 `0.8`），要么是表中某个字段（如 `colA`）。节点之间的父子关系表示了计算的依赖关系：子节点的计算结果是父节点的输入数据。



树中每个节点的求值逻辑都可抽象为如下所示的计算接口：

```
type Node interface {
      evalReal(row Row) (val float64, isNull bool)
}
```

以 `*`、`0.8` 和 `col` 这三个节点为例，他们都会实现上面的接口，每个类型有自己的计算逻辑，计算的伪代码大致如下：

```
func (node *multiplyRealNode) evalReal(row Row) (float64, bool) {
    v1, null1 := node.leftChild.evalReal(row)
    v2, null2 := node.rightChild.evalReal(row)
    return v1 * v2, null1 || null2
}

func (node *constantNode) evalReal(row Row) (float64, bool) {
    return node.someConstantValue, node.isNull  // 0.8 in this case
}

func (node *columnNode) evalReal(row Row) (float64, bool) {
    return row.GetReal(node.colIdx)
}
```

行式调用的解释开销有多大

上小节的表达式计算实现方式非常简单，和 Volcano 计算模型类似，都是以行为单位迭代。在处理少量行数的数据时候，没有什么问题，但是如果参与计算的数据行数很多，会有不小的性能开销。为了解释以行为单位进行迭代的解释开销有多大，我们以 TiDB 中 `builtinArithmeticMultiplyRealSig` 函数实现为例，来量化地进行分析。

该函数用来实现两个浮点数乘法，下面是该函数的代码，右边的数字是对应行经过汇编后得到的汇编指令数，这里只包含了在正常情况下会经过的代码行，错误处理相关的逻辑我们忽略：

```
func (s *builtinArithmeticMultiplyRealSig) evalReal(row chunk.Row) (float64, bool,
error) {                                                                          9
  a, isNull, err := s.args[0].EvalReal(s.ctx, row)                                27
  if isNull || err != nil {                                                       3
     return 0, isNull, err
  }
  b, isNull, err := s.args[1].EvalReal(s.ctx, row)                                25
  if isNull || err != nil {                                                       3
     return 0, isNull, err
  }
  result := a * b                                                                 2
  if math.IsInf(result, 0) {                                                      6
     return 0, true, types.ErrOverflow.GenWithStackByArgs(...)
  }
  return result, false, nil                                                       7
}
```

我们简单分析一下：

- 检查栈和处理函数调用的指令，共 9 个；
- 获取第一个孩子数据的指令，共 27 + 3 = 30 个；
- 获取第二个孩子数据的指令，共 25 + 3 = 28 个；
- 实际做乘法运算和检查错误的指令，共 2 + 6 = 8 个；
- 最后是处理函数返回的指令，共 7 个；

简单计算一下，每次做乘法运算时，只有 8 / (9 + 30 + 28 + 8 + 7) 约等于 1/10 的指令在做“真正”的乘法运算，其他的指令都可以被视作解释开销。（实际上，当我们将此函数向量化后，其性能确实提高了接近 10 倍，可见 PR/12543）

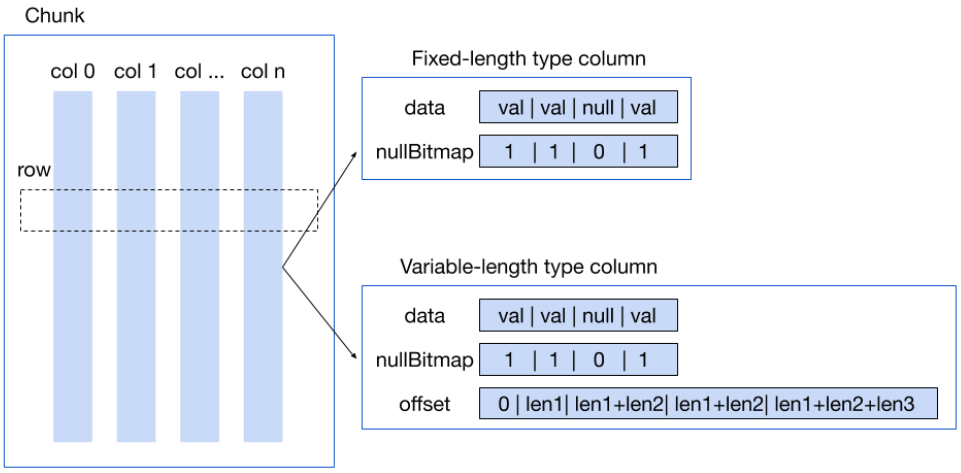
批处理降低解释开销

和 SQL 算子的向量化优化类似，我们可以通过每次处理一批数据，返回一批结果的方式来降低表达式计算过程中的解释开销。假设这一批数据有 1024 行，因为每次函数调用后都会处理 1024 行和返回 1024 行数据，优化后在函数调用上的解释开销就被均摊为了原来的 1/1024。

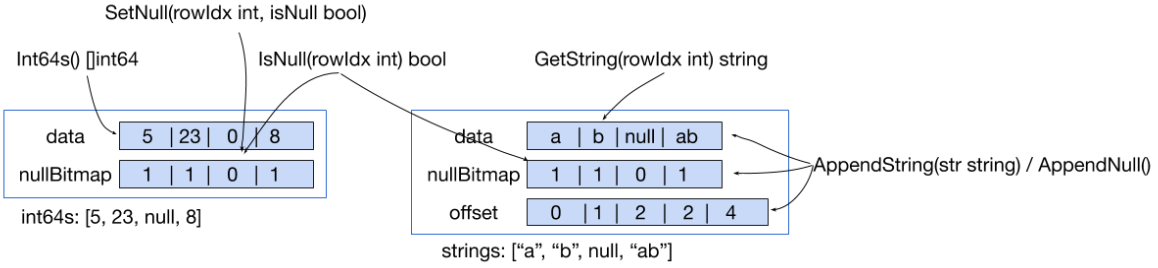
因此，我们可以为表达式求值框架增加类似下面这种批处理接口：

```
type Node interface {
  batchEvalReal(rows []Row) (vals []float64, isNull []bool)
}
```

实际情况下，考虑到编程语言方面的性能优化，我们最终的接口和上面的实例有区别。要理解最终接口的设计，需要先了解 TiDB 中用于存放数据的内存数据结构 `Chunk`。
Chunk 在向量化改造的早期引入（2017 年末），它由多个 `Column` 组成，根据数据类型是定长还是变长，`Column` 有两种数据组织方式，如下图：



不管是定长还是变长，`Column` 当中的数据都被连续存放在内存中（也就是 `Column.data` 数组中），此外变长类型会多用一个 `Column.offset` 数据来记录数据的位移情况。
下面的图演示了我们这次为 Chunk 引入的新的向量化访问接口：



对于定长类型，比如 int64，在 `Int64s() []int64` 中，利用 Go unsafe 包直接把 `Column.data` 转换成 `[]int64` 返回，让外部直接操作数组，以达到最快的访问速度。
对于变长类型，比如 string，只能用 `GetString(rowIdx int) string` 获取对应行号的数据，同时只能以追加的方式进行更新。

根据 Chunk 的内部结构以及 Golang 的语言特性，我们对表达式的求值接口进行了三点优化：

1. 直接传入 `*Chunk`，而不是传入 `[]Row`。这样能避免创建大量 `Row` 对象，减轻 Golang GC 的压力，提升性能；
2. 直接通过 `Column` 访问，而不是通过 `Row` 访问数据（`Column.data`）。这样能减少函数调用次数，从而减少访问数据需要的 CPU 指令，从而降低解释开销，提高了数据的访问速度；
3. 把用于存放数据的 Column 放在参数中传入，而不是直接返回数据数组 []float64 和 []bool。这样可以提高内存复用率，并且能够减轻 Golang GC 开销；

最终我们表达式向量化计算的接口如下：

```
type VecNode interface {
  vecEvalReal(input *Chunk, result *Column)
}
```

向量化的进行计算

接下来演示怎么实现上面的向量化接口，以 `multiplyRealNode` 为例：

```
func (node *multiplyRealNode) vecEvalReal(input *Chunk, result *Column) {
  buf := pool.AllocColumnBuffer(TypeReal, input.NumRows())
  defer pool.ReleaseColumnBuffer(buf)
  node.leftChild.vecEvalReal(input, result)
  node.rightChild.vecEvalReal(input, buf)

  f64s1 := result.Float64s()
  f64s2 := buf.Float64s()
  result.MergeNulls(buf)
  for i := range i64s1 {
     if result.IsNull(i) {
        continue
     }
     i64s1[i] *= i64s2[i]
  }
}
```

对它做个简单说明：

1. 前两行从缓存池申请 ColumnBuffer 来缓存右边孩子的数据，左边孩子的数据就直接存放在 result 指向的内存中。
2. Columns.MergeNulls(cols…) 合并多列的 Null 标记，在这里的作用相当于 result.nulls[i] = result.nulls[i] || buf.nulls[i]。Column 内部用 bitmap 维护 Null 标记，当此函数被调用时，Column 会用位运算将 Null 进行合并。
3. 接下来的循环直接操作左右孩子的数据进行计算。
4. 执行过程中这个函数分别调用左右孩子的接口，拿到他们的数据。

上面的执行方式，除了通过批量处理降低了解释开销外，还有两点对现代 CPU 有利的改进：

1. 数据访问方式都是集中式的访问一段连续的内存，有利于 CPU Cache；
2. 计算工作大部分都落在了一个循环内，且非常简单，有利于 CPU Branch Prediction 和 Instruction Pipelining；

向量化计算性能对比

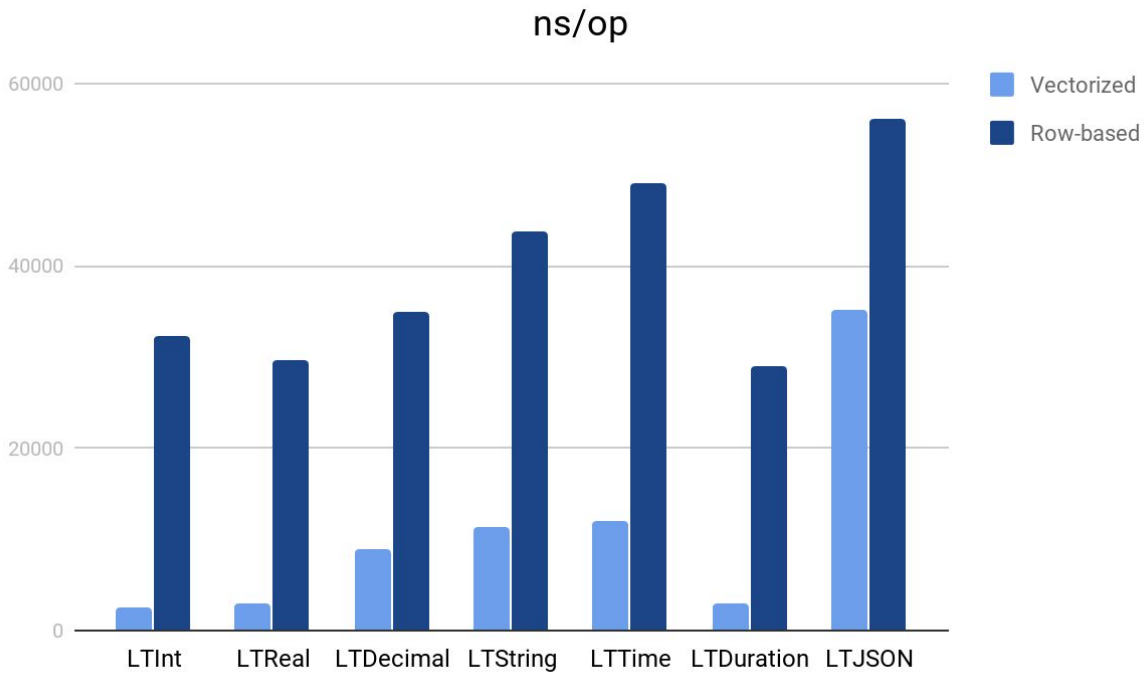
接下来我们直接用 TiDB 向量化前后的代码，来做性能对比。

下面是用 TiDB 源码进行的测试，在相同的数据下（1024 行，两列浮点数），对表达式 col0 * 0.8 + col1 进行压测的结果：

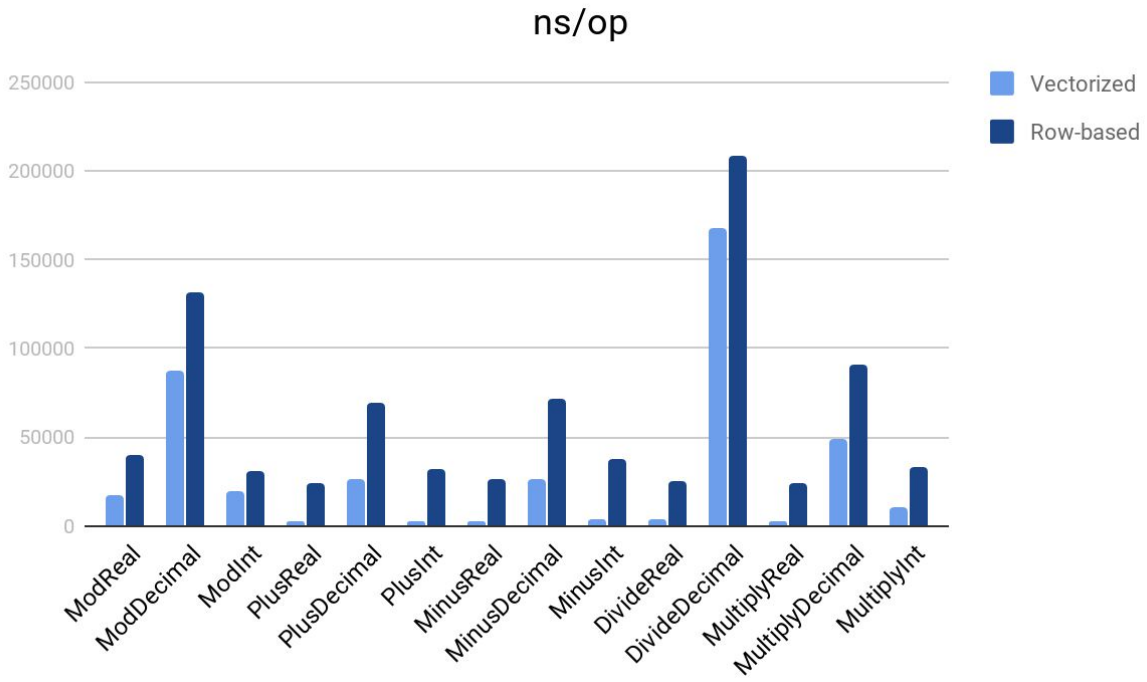
```
BenchmarkVec-12           152166              7056 ns/op                0
B/op          0 allocs/op
BenchmarkRow-12            28944             38461 ns/op                0
B/op          0 allocs/op
```

可以看到向量化速度是行式的 5 倍。

下面是各个类型的 LT(LessThan) 函数向量化前后的性能对比图：

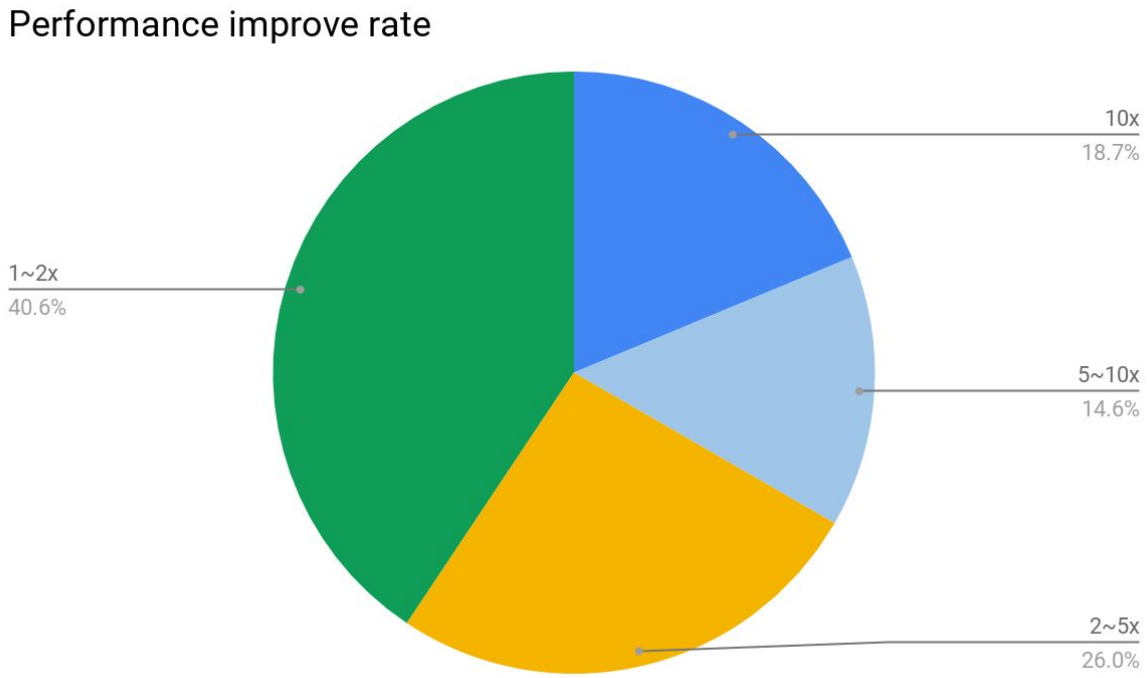


下面是一些算数类型函数向量化前后的性能对比：



很明显的可以看到这些内置函数被向量化后，性能提升显著。

我们跑了超过 300 个已经向量化的函数测试，下面是这些函数的速度提升占比，可以发现超过一半的函数性能有了明显提升（2x），18.7% 的函数速度变为原来的 10x：



利用模板生成向量化代码

在对内置函数进行向量化改造中，我们发现很多函数是相似的，如 LT(<)/GT(>)/LE(<=) 等函数，他们大多数逻辑相似，只是操作符不同，我们利用模板来生成这些函数的代码。
因为 Go 目前不支持泛型和宏定义，我们使用了 Go template 包来生成代码。
我们按照 Go 模板的语法，将待生成的函数抽象成模板，如下面 LT/GT 等比较函数的模板：

```
var builtinCompareVecTpl = template.Must(template.New("").Parse(`
func (b *builtin{{ .compare.CompareName }}{{ .type.TypeName }}Sig) vecEvalInt(input
*chunk.Chunk, result *chunk.Column) error {
        n := input.NumRows()
        buf0, err := b.bufAllocator.get(types.ET{{ .type.ETName }}, n)
        if err != nil {
                return err
        }
        if err := b.args[0].VecEval{{ .type.TypeName }}(b.ctx, input, buf0); err != nil {
                return err
        }
    ...
        for i := 0; i < n; i++ {
{{ if eq .type.ETName "Json" }}
                val := json.CompareBinary(buf0.GetJSON(i), buf1.GetJSON(i))
{{ else if eq .type.ETName "Real" }}
                val := types.CompareFloat64(arg0[i], arg1[i])
{{ else if eq .type.ETName "String" }}
                val := types.CompareString(buf0.GetString(i), buf1.GetString(i))
         ...

         if val {{ .compare.Operator }} 0 {
                        i64s[i] = 1
                } else {
                        i64s[i] = 0
                }
        }
        return nil
}
```

模板中会根据不同的数据类型和操作符，生成对应的代码，完整的代码可以看 PR/12875
这份模板被放在 expression/generator 包下，然后通过运行这份模板文件内的 main() 函数，生成对应的内置函数代码 xx_vec_generated.go 到 expression 包下。

发动社区帮我们进行内置函数向量化重构

除了使用模板，我们还发动社区力量，来帮我们进行内置函数向量化重构，人多力量大。

在活动初期，我们发现合并 PR 时很容易产生冲突，于是我们用脚本，将所有的内置函数向量化接口生成并预留在代码内，让 Contributor 直接去填写（重构）感兴趣的内置函数：

```
func (b *builtinCastStringAsIntSig) vecEvalInt(input *chunk.Chunk, result *chunk.Column)
error {
  return errors.Errorf("not implemented")
}

func (b *builtinCastStringAsDurationSig) vectorized() bool {
  return false
}
```

预留函数接口可以避免合并代码时，不同 PR 修改同一文件带来的冲突。

然后我们编写了一个测试框架，使得 Contributor 在完成重构后，仅仅通过几行简单的配置，便能对该内置函数进行正确性测试和性能测试：

```
var vecBuiltinArithmeticCases = map[string][]vecExprBenchCase{
  ast.Div: {
     {retEvalType: types.ETReal, childrenTypes: []types.EvalType{types.ETReal,
types.ETReal}},
  },
}

func (s *testEvaluatorSuite) TestVectorizedBuiltinArithmeticFunc(c *C) {
  testVectorizedBuiltinFunc(c, vecBuiltinArithmeticCases)
}

func BenchmarkVectorizedBuiltinArithmeticFunc(b *testing.B) {
  benchmarkVectorizedBuiltinFunc(b, vecBuiltinArithmeticCases)
}
```

如上图，用户只需要向 vecBuiltinArithemeticCases 里面增加自己内置函数相关的参数，然后运行下面两个事先生成的函数，便能分别进行正确性和性能测试：

```
> GO111MODULE=on go test -check.f TestVectorizedBuiltinArithmeticFunc
PASS: builtin_arithmetic_vec_test.go:30:
testEvaluatorSuite.TestVectorizedBuiltinArithmeticFunc 0.002s
OK: 1 passed
PASS
ok      github.com/pingcap/tidb/expression      0.636s

> go test -v -benchmem -run=BenchmarkVectorizedBuiltinArithmeticFunc
-bench=BenchmarkVectorizedBuiltinArithmeticFunc
BenchmarkVectorizedBuiltinArithmeticFunc/builtinArithmeticDivideRealSig-VecBuiltinFunc-1
2                 424515              2808 ns/op               0 B/op          0
allocs/op
BenchmarkVectorizedBuiltinArithmeticFunc/builtinArithmeticDivideRealSig-NonVecBuiltinFun
c-12               47856             25272 ns/op               0 B/op          0
allocs/op
PASS
ok      github.com/pingcap/tidb/expression      4.116s
```

正确性测试和性能测试都是直接生成随机数据，把向量化实现和行式实现进行对比完成的。
上面两个操作把 Contributor 参加内置函数向量化重构的门槛降到了最低，我们的活动发起不到两个月内（9/16 ~ 11/11），就完成了 358 个函数的向量化重构，其中绝大多数的贡献来自于社区。
这段时间内社区用户一共给我们提了 256 个 PR，其中有 33 个 Contributor 是通过此活动第一次为我们贡献代码。
通过这次活动诞生了 9 个 Active Contributor（一年内提超过 8 个 PR），2 个 Reviewer（一年内提超过 30 个 PR）。

总结

通过引入向量化执行，我们将表达式计算的性能进一步提高，目前在我们的 Master 分支上，向量化执行都是默认打开的，一旦某个表达式所包含的所有内置函数都已经支持向量化，那这个表达式就会以向量化的方式执行。
另外需要说明的是，上面的所有性能测试，数据都是在内存中的，如果数据在磁盘上，那么将数据读入可能会有比较大的开销，这会让向量化的收益显得没那么大，但是总的来说，向量化已经明显的将我们表达式计算的性能提高。
另外，在进行表达式向量化的过程中，我们还发现不少的可以用向量化思路提升性能的地方，比如：

1. 在 HashJoin 中分别以向量化的方式计算 inner 端(PR/12076) 和 outer 端(PR/12669) 的 HashKey，在特定的场景下分别有 7% 和 18% 的性能提升。这些都是由社区同学 sduzh 独立完成的，非常感谢他。
2. 在 HashAggregation 算子中，以向量化的方式对数据进行编码，在本地测试中有 20%~60% 的收益，具体见 PR/12729；
3. 在 StreamAggregation 算子中，用向量化的方式将数据划分为 group，在本地测试中有 60% 左右的收益，具体见 PR/12903

我们会在后续的博客中，继续介绍我们怎么利用向量化思路优化执行引擎的其他部分，敬请期待。
最后，我们的内置函数向量化重构活动还在继续，欢迎大家加入我们并为我们贡献代码。