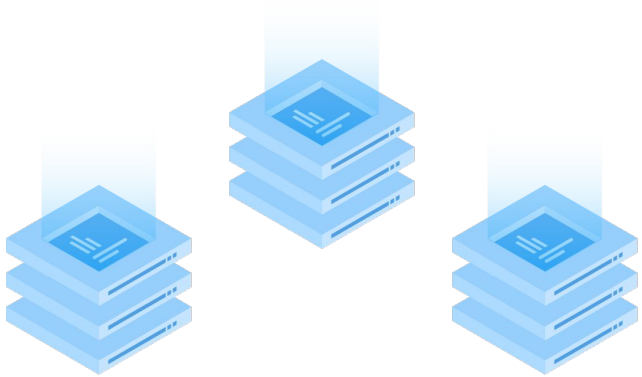
Lesson 4: Execution 性能优化

Presented by Yuanjia Zhang

PingCAP TiDB



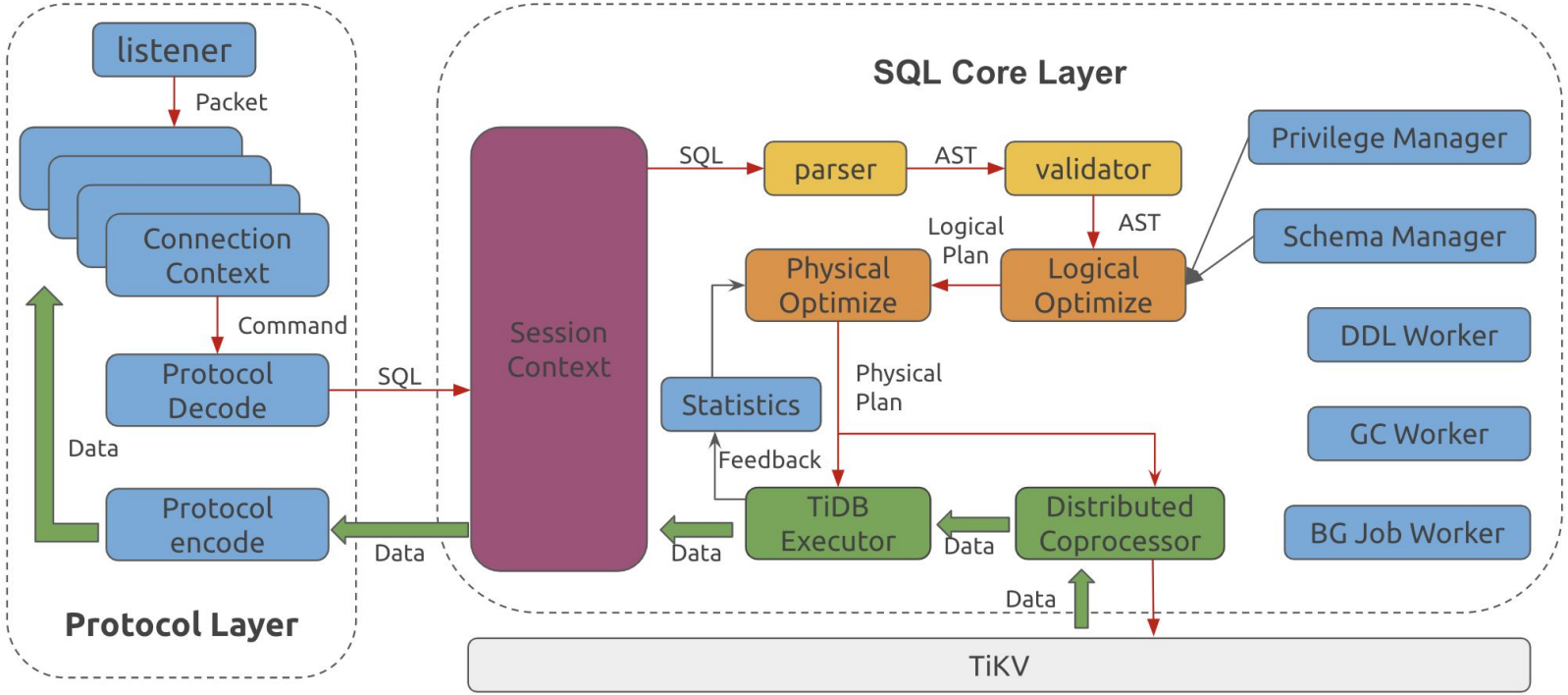
Overview

1. 执行器和表达式框架中的 优化
2. Join 算子中的性能 优化
3. 聚合算子中的性能 优化
4. 窗口算子中的性能 优化
5. 其他性能优化

第一节:执行器和表达式框架介绍

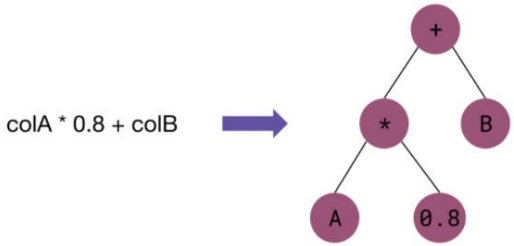
- 执行在 TiDB 中的位置
- 表达式和执行引擎介绍
- 表达式向量化优化

执行器的位置

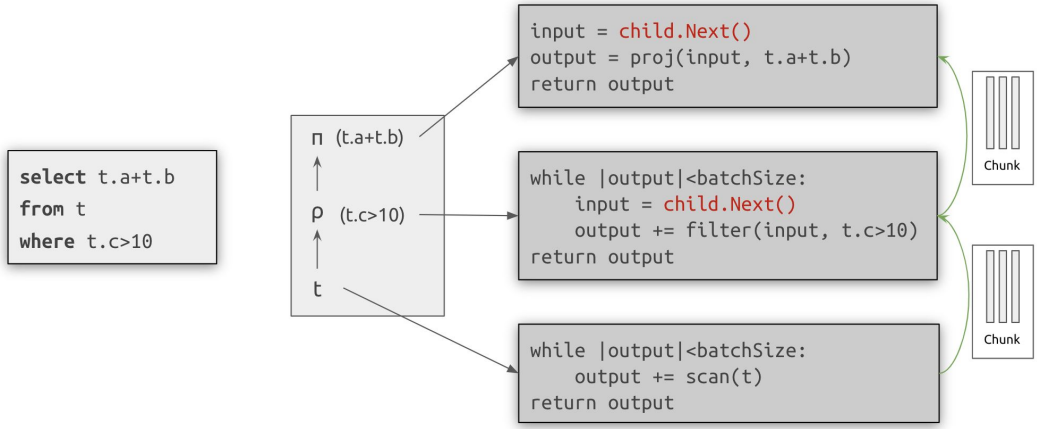

listener
Packet
Connection Context
Command
Protocol Decode
SQL
Data
Protocol encode
Data
Protocol Layer
Session Context
SQL Core Layer
SQL
parser
AST
validator
AST
Logical Plan
Physical Optimize
Logical Optimize
Privilege Manager
Schema Manager
DDL Worker
GC Worker
BG Job Worker
Statistics
Physical Plan
Feedback
TiDB Executor
Data
Distributed Coprocessor
Data
Data
TiKV

执行引擎简介

- Volcano 模型
- 树状结构
- 自底向上执行
- 数据传递单位 chunk
- chunk 在内存中按列存放数据




```
func (f *plus) EvalRow() int {
        a := f.args[0].EvalRow()
        b := f.args[1].EvalRow()
        return a + b
}
```



向量化表达式优化 (1/2)

一次表达式调用的解释开销：

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

解释开销：1 - (8 / (9+30+28+8+7)) = 0.9

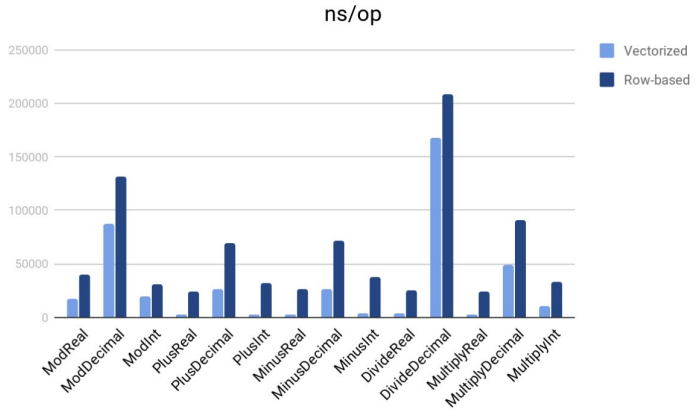
```
func (s *builtinArithmeticMultiplyRealSig) evalReal(row chunk.Row) (float64, bool,
error) {                                                                        9
  a, isNull, err := s.args[0].EvalReal(s.ctx, row)                              27
  if isNull || err != nil {                                                     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
}
```

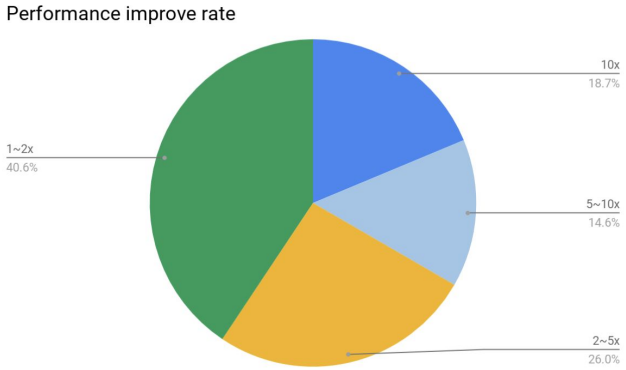
向量化表达式优化 (2/2)

- 为表达式增加列式接口 vecEval
- 各类型/类别的函数签名都有显著提高

```
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]
  }
}
```





PingCAP

PingCAP University

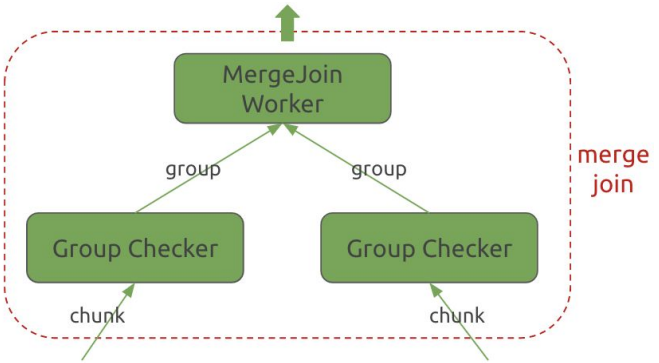
第二节:Join 性能优化

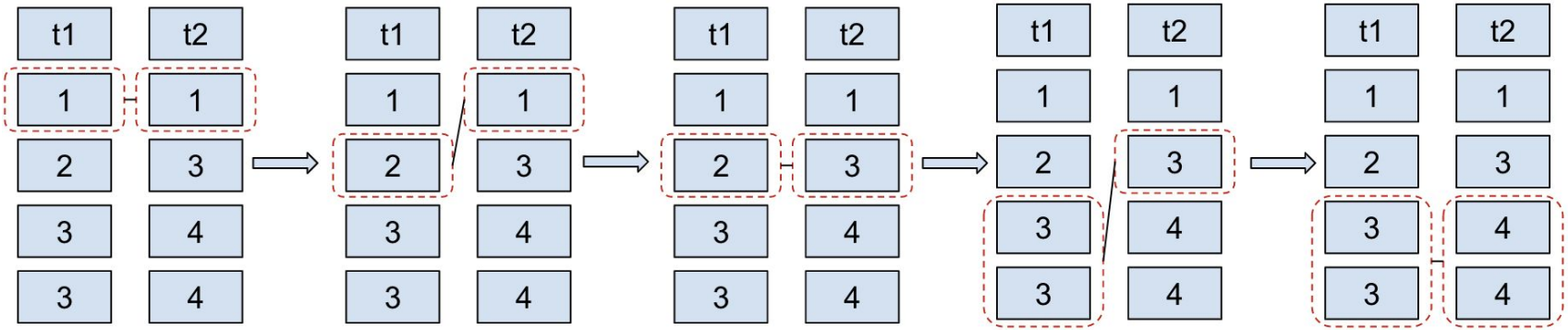
- Merge Join 性能优化
- Hash Join 性能优化
- NestedLoop Apply 性能优化

Merge Join 性能优化 (1/2)

select * from t1, t2 where t1.key = t2.key

- 双指针
- Group 为单位
- 单线程

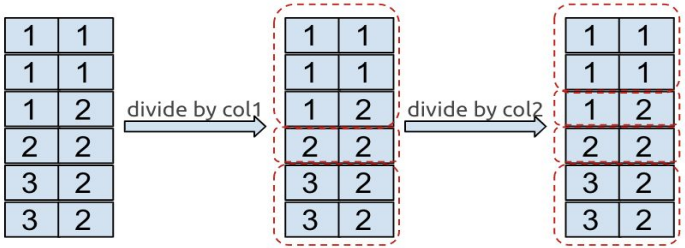




Merge Join 性能优化 (2/2)

向量化分割 group

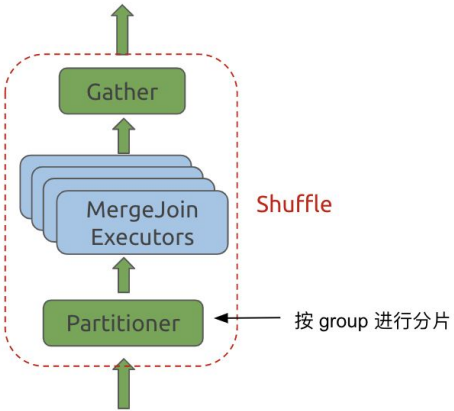
- 数据在内存中按列排放
- 按列依次处理




```
old time/op    new time/op    delta
   1.27s ± 0%     0.34s ± 0%  -72.87%
   338ms ± 0%     172ms ± 0%  -48.96%
   330ms ± 0%      43ms ± 0%  -86.96%
   958ms ± 0%     309ms ± 0%  -67.73%
```

其他优化点

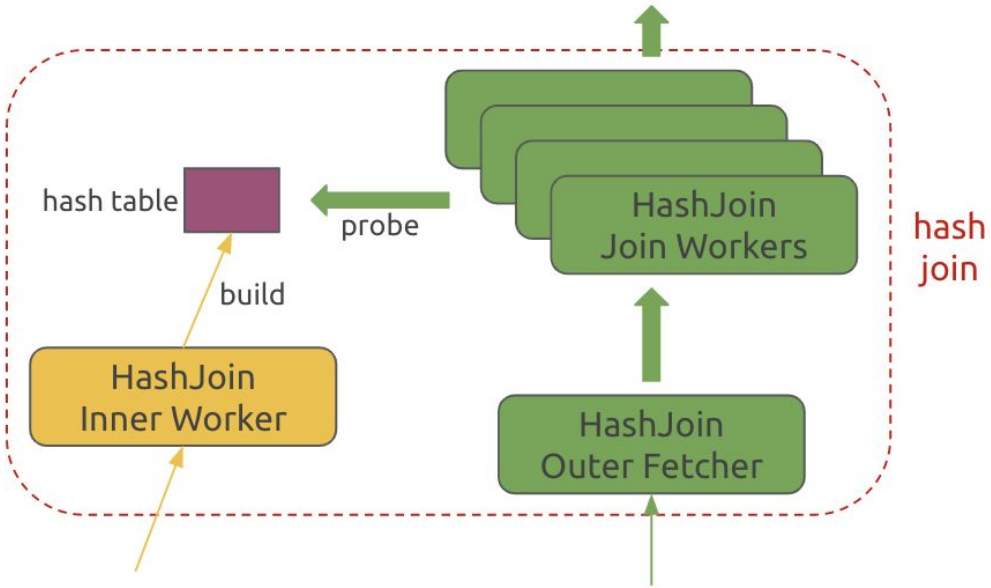
- 使用 shuffle 算子进行并发 [issues/14441]



HashJoin 性能优化 (1/2)

Hash Join

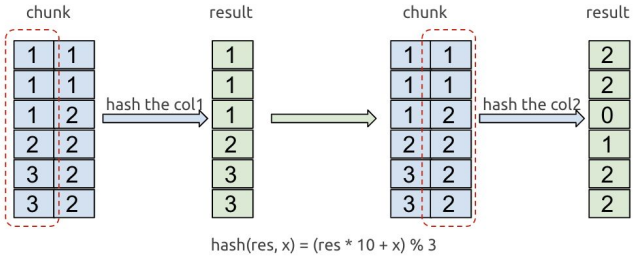
- 两阶段串行
  - build
  - probe
- build 阶段单线程
- probe 阶段并行执行



HashJoin 性能优化 (2/2)

向量化计算 hash key

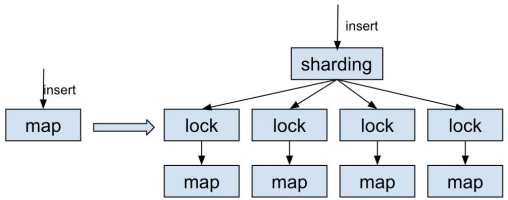
- 数据在内存按列存放
- 一次处理每一列




```
old time/op    new time/op    delta
   810ms ± 1%     806ms ± 1%   -0.54%
   145ms ±16%     135ms ± 3%   -7.23%
   594ms ± 1%     587ms ± 0%   -1.18%
  13.3ms ± 1%    10.8ms ± 0%  -18.85%
  4.17µs ± 4%    4.23µs ± 3%     ~
```

并行 build hash table

- build 阶段并发 issues/16618 WIP
  - sharding + lock




```
concur:1,key:_[0_1])-12    1   1651156792 ns/op
concur:2,key:_[0_1])-12    2   875115518 ns/op
concur:4,key:_[0_1])-12    2   572805819 ns/op
concur:1,key:_[0])-12      6   199510053 ns/op
concur:2,key:_[0])-12      7   149599528 ns/op
concur:4,key:_[0])-12      7   147609291 ns/op
```

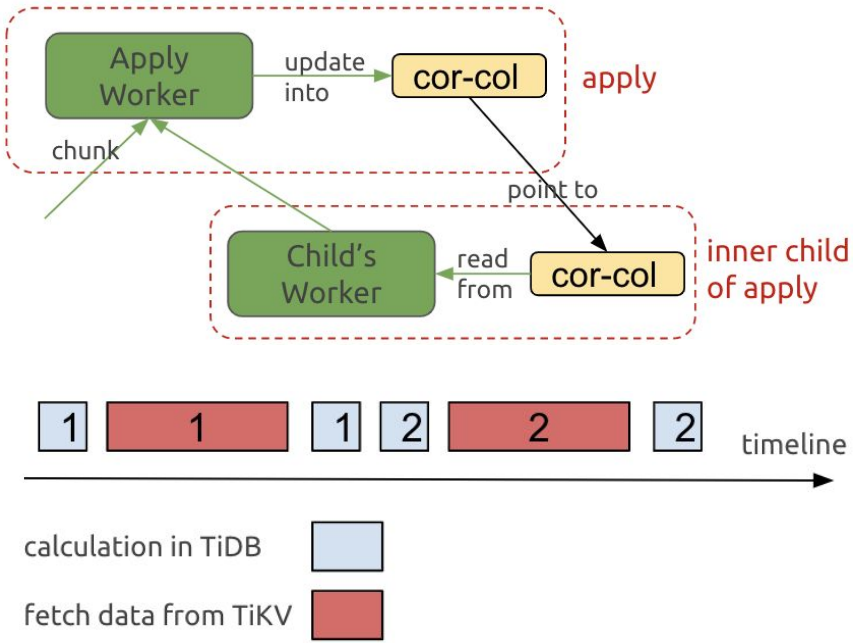
NestedLoop Apply 性能优化 (1/4)

NestedLoopApply 例子：

```
select id from t1 where t1.b > (
        select min(b) from t2
        where t1.a > t2.a
)
```

执行方式特点：

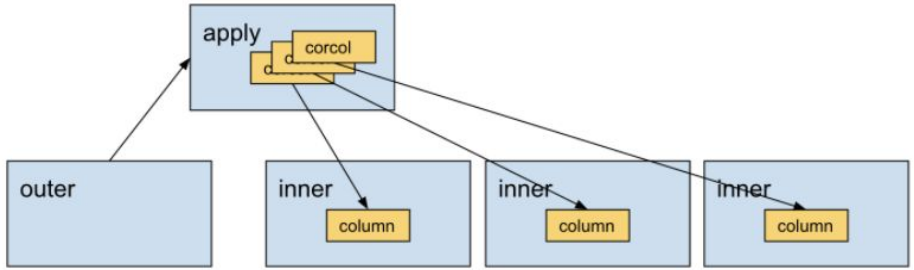
- 把 outer 每一行代入执行
- 串行执行
- 执行方式过于朴素

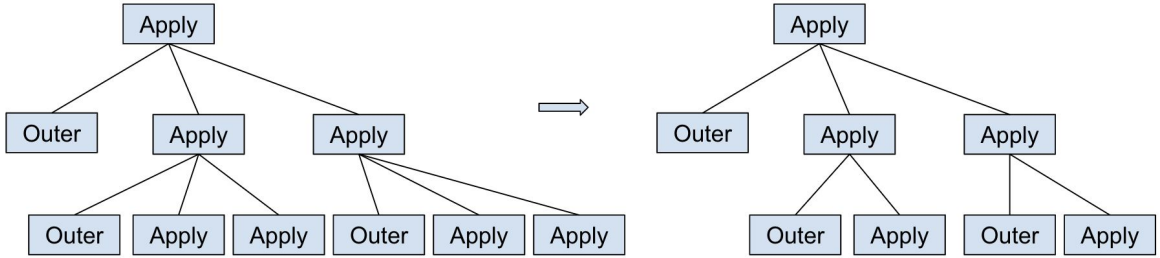


NestedLoop Apply 性能优化 (2/4)

并行化

- 将 inner 端并行起来
- 维护多套 corcol 指针
- 创建多个 child executor
- 避免并行度指数化膨胀



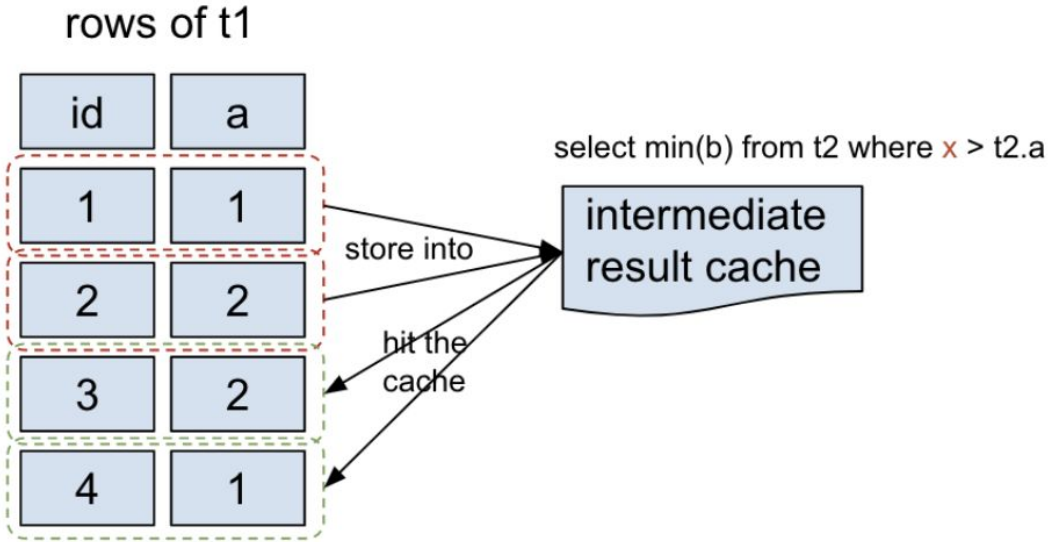


NestedLoop Apply 性能优化 (3/4)

性能优化

- 根据相关列进行临时缓存
- 优化器根据 NDV(number of distinct values) 自动决定是否开启缓存

```
select id from t1 where t1.b > (
        select min(b) from t2
        where t1.a > t2.a
)
```



NestedLoop Apply 性能优化 (4/4)

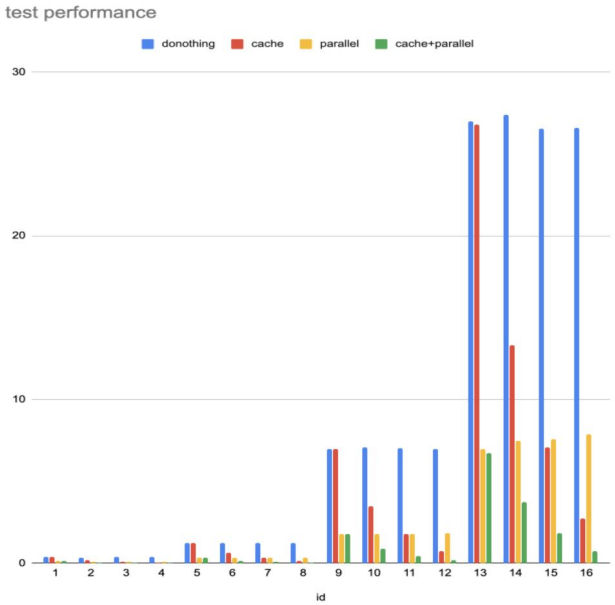
opt: 无任何优化

opt1: Apply Cache

opt2: 并行度 4

opt3: Cache + 并行度 4

用例			测试结果（单位 s）			
id	rows	ndv	opt0	opt1	opt2	opt3
1	1000	1000	0.37	0.38	0.11	0.11
2	1000	500	0.35	0.19	0.10	0.04
3	1000	250	0.36	0.10	0.10	0.02
4	1000	100	0.37	0.04	0.09	0.01
5	2000	2000	1.25	1.24	0.34	0.34
6	2000	1000	1.23	0.64	0.31	0.15
7	2000	500	1.24	0.32	0.31	0.08
8	2000	250	1.25	0.12	0.32	0.04
9	5000	5000	6.96	6.95	1.77	1.76
10	5000	2500	7.07	3.47	1.77	0.88
11	5000	1250	7.02	1.78	1.77	0.44
12	5000	500	6.95	0.72	1.81	0.18
13	10000	10000	26.99	26.77	6.99	6.73
14	10000	5000	27.38	13.32	7.47	3.72
15	10000	2500	26.56	7.05	7.58	1.81
16	10000	1000	26.57	2.71	7.87	0.75



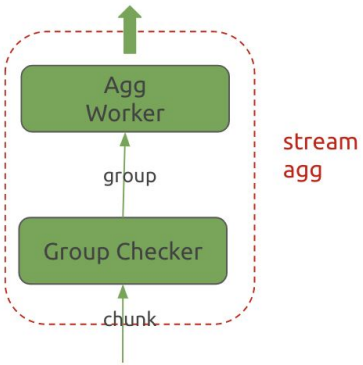
第三节:聚合算子性能优化

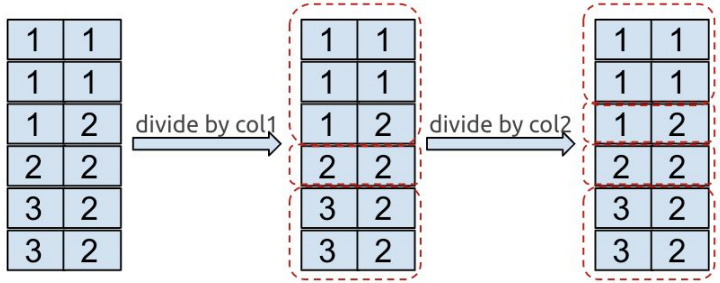
- Stream Aggregate 性能优化
- Hash Aggregate 性能优化

Stream Aggregate 性能优化

Stream Aggregate

- 单线程
- 向量化优化





未来优化：

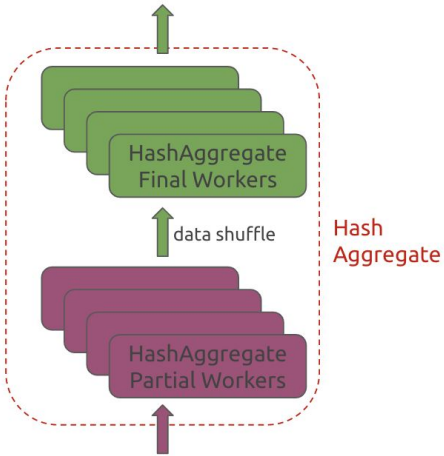
- 并行 Stream Aggregate [issues/9244]

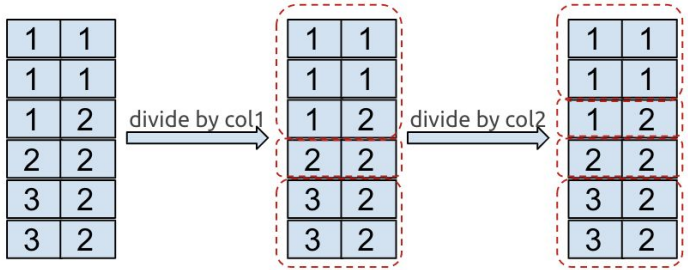
```
old time/op    new time/op    delta
   390ms ± 4%     153ms ± 5%  -60.79%
   401ms ± 3%     159ms ± 8%  -60.33%
   396ms ± 2%     161ms ± 6%  -59.40%
   392ms ± 4%     158ms ±10%  -59.62%
```

Hash Aggregate 性能优化

Hash Agg 执行：

1. 两阶段并行
2. 向量化计算 hash key






```
old time/op    new time/op    delta
  19.9ms ± 3%    14.5ms ± 1%  -27.41%
  5.86ms ±15%    3.21ms ± 9%  -45.13%
  5.78ms ±32%    3.43ms ± 4%  -40.70%
  5.92ms ± 2%    5.08ms ±25%  -14.25%
  6.02ms ± 3%    4.71ms ±10%  -21.85%
  6.78ms ± 7%    6.71ms ±16%     ~
  9.24ms ±13%    9.07ms ±12%     ~
   198ms ± 9%     159ms ±16%  -19.87%
  51.3ms ±19%    17.9ms ±11%  -65.05%
  43.3ms ±10%    15.7ms ± 6%  -63.71%
  38.5ms ± 8%    16.6ms ± 6%  -56.97%
```

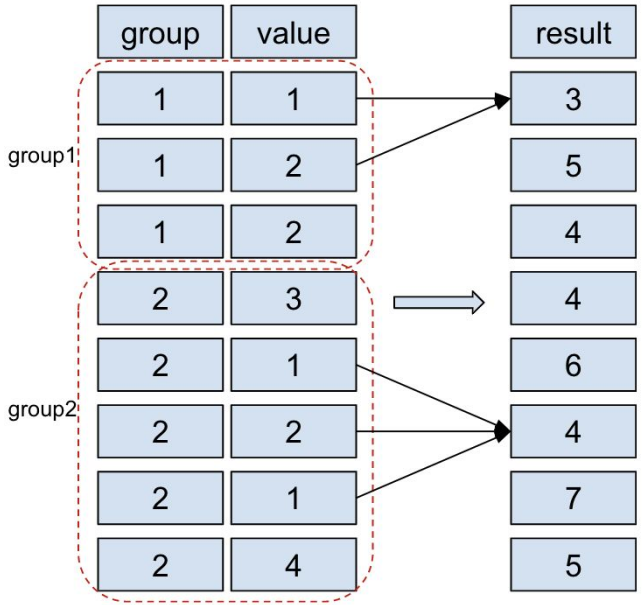
第四节:窗口算子性能优化

- 滑动窗口优化
- Shuffle 并发窗口算子

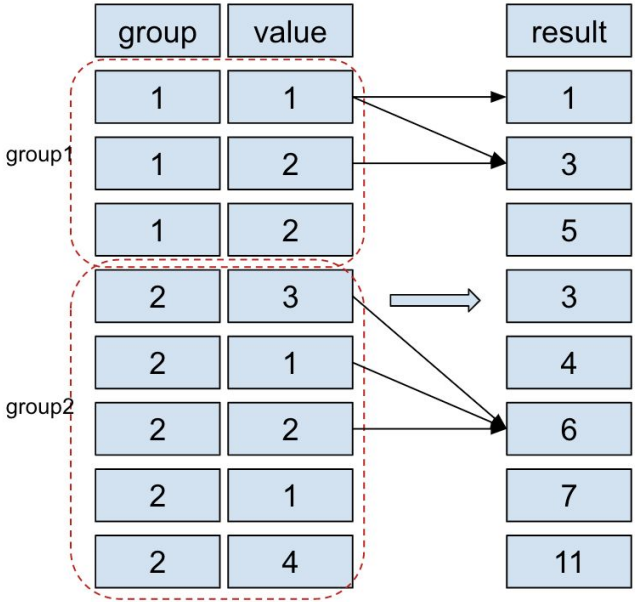
窗口算子性能优化 (1/3)

窗口算子介绍

例 1：相邻两行的和



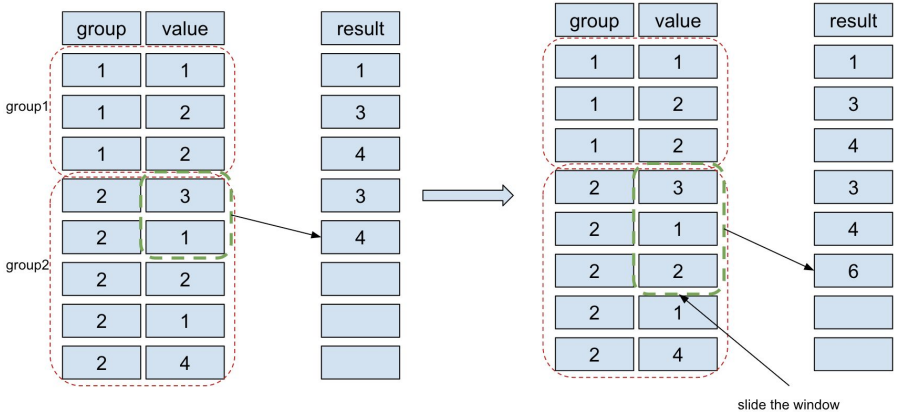
例 2：group 内前序行累积和



窗口算子性能优化 (2/3)

滑动窗口优化：issues/12967 WIP

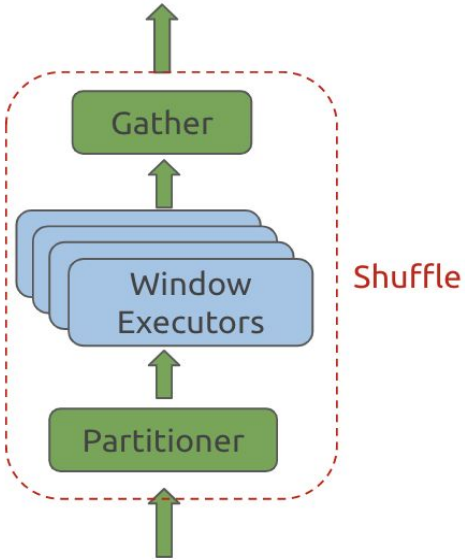
- 传统接口：Update(rowsInGroup []row)
- 滑动窗口接口：Slide(rowsInGroup []row, lastStart, lastEnd, shiftStart, shiftEnd int)




```
21.7ms ± 8%    6.8ms ± 5%  -68.44%
74.4ms ± 3%   12.3ms ± 5%  -83.46%
21.7ms ± 4%    7.0ms ± 2%  -67.85%
```

窗口算子性能优化 (3/3)

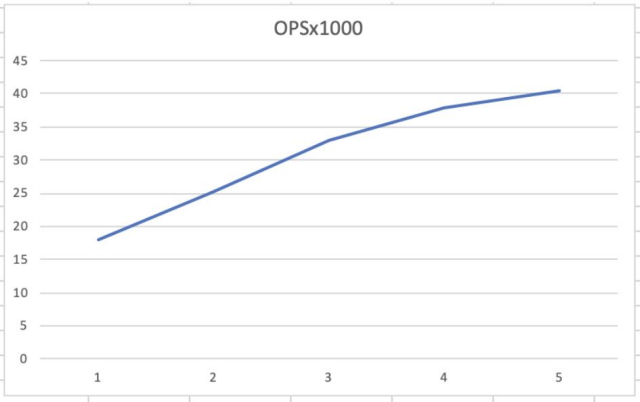
引入 shuffle 算子使 window 并行化




```
explain select sum(a) over(partition by a order by b) from t;
id    count    task    operator info
Projection_7  10000.00        root    Column#6
└─Shuffle_12  10000.00        root    execution info: concurrency:4, data
  └─Window_8  10000.00        root    sum(cast(test.t.a))->Column#6 over(p
    └─Sort_11 10000.00        root    test.t.a:asc, test.t.b:asc
      └─TableReader_10        10000.00        root    data:TableScan_9
        └─TableScan_9 10000.00        cop[tikv]       table:t, range:[-inf
```

Performance increases nearly linearly with number of threads.

bit_xor, 100000 nows, 1000 ndv		
workers	OPSx1000	ns/op
1	17.98486624	55602304
2	25.16221261	39742133
3	32.85727738	30434658
4	37.84142234	26426068
5	40.38335597	24762677



第五节:其他性能优化点

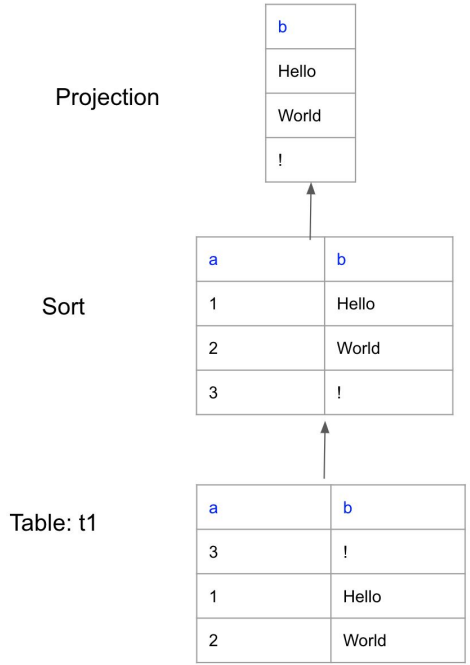
- inline projection
- 内置函数 in 性能优化
- chunk RPC

Inline Projection

例子 : select b from t1 order by a

- 最后只需要 b
- 排序时需要 a, b 都拷贝

==> a 的拷贝多余

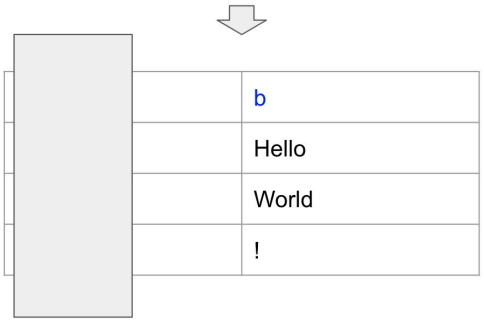


Inline Projection

怎么优化

- 排序时只拷贝 b 到结果
- 相当于把 project 内联(inline) 到了 sort 内

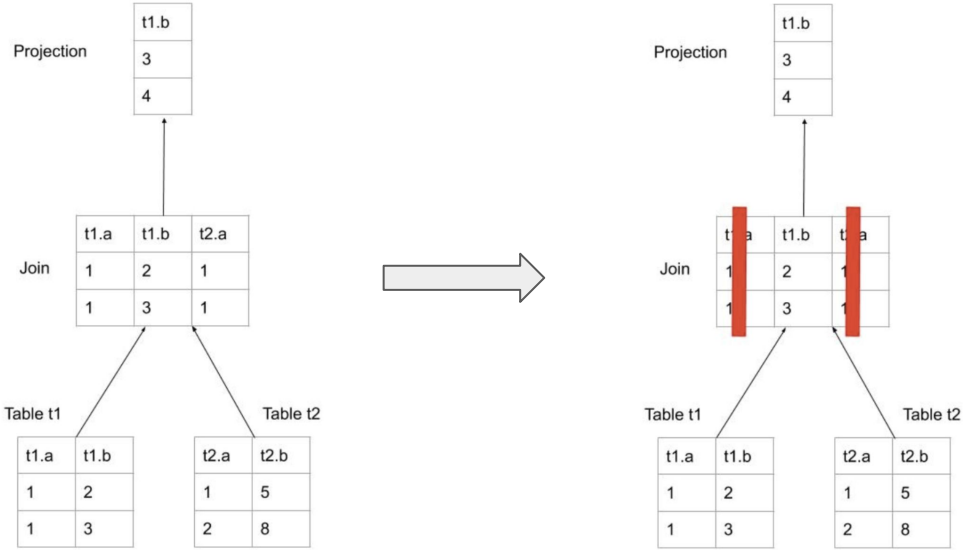
a	b
3	!
1	Hello
2	World



	b
	Hello
	World
	!

Inline Projection

一个 Join 的例子：select t1.b+1 from t1 inner join t2 on t1.a = t2.a



Inline Projection

具体做法：把 projection 信息下推到对应算子

```
func newJoiner(ctx sessionctx.Context, joinType plannercore.JoinType,
      outerIsRight bool, defaultInner []types.Datum, filter []expression.Expression,
-     lhsColTypes, rhsColTypes []*types.FieldType) joiner {
+     lhsColTypes, rhsColTypes []*types.FieldType, childrenUsed [][]bool) joiner {
```

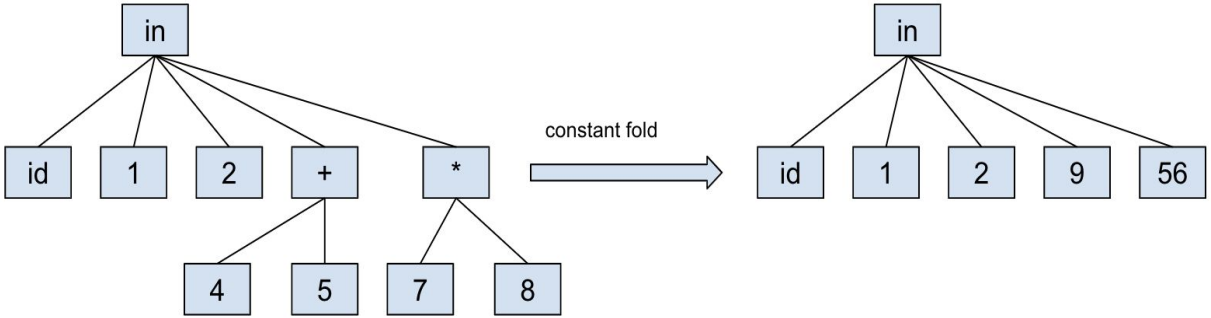
```
func (j *baseJoiner) makeJoinRowToChunk(chk *chunk.Chunk, lhs, rhs chunk.Row) {
      // Call AppendRow() first to increment the virtual rows.
      // Fix: https://github.com/pingcap/tidb/issues/5771
-     chk.AppendRow(lhs)
-     chk.AppendPartialRow(lhs.Len(), rhs)
+     lWide := chk.AppendRowByColIdxs(lhs, j.lUsed)
+     chk.AppendPartialRowByColIdxs(lWide, rhs, j.rUsed)
}
```

```
'InlineProjection:ON-12      13    82913226 ns/op
'InlineProjection:ON-12      14    82013976 ns/op
'InlineProjection:ON-12      14    82275721 ns/op
'InlineProjection:ON-12      14    82140498 ns/op
'InlineProjection:ON-12      14    82043427 ns/op
'InlineProjection:OFF-12     7     161024914 ns/op
'InlineProjection:OFF-12     7     160023113 ns/op
'InlineProjection:OFF-12     7     162132958 ns/op
'InlineProjection:OFF-12     7     160637265 ns/op
'InlineProjection:OFF-12     7     160055525 ns/op
```

Builtin Function 优化

例子：select id from t where id in (1, 2, 4+5, 7*8, ...)

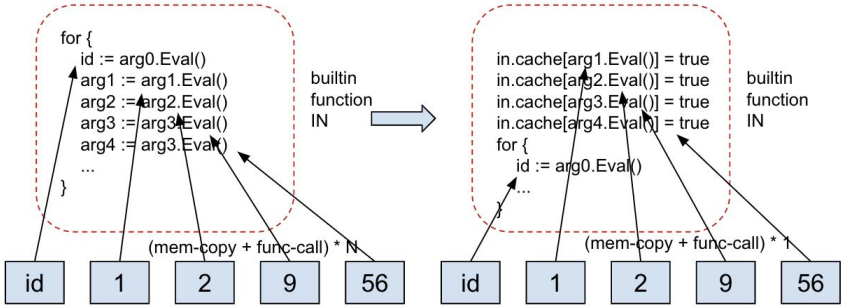
- 常数折叠发生在逻辑优化阶段，对确定性表达式都会进行
- 非确定性函数有 sysdate(), rand(), getVar() 等



Builtin Function 优化

中间结果记忆化优化，例子 select id from t where id in (1, 2, 4+5, 7*8, ...)

- 发生在执行阶段
- 不是每个表达式都有，只有实现了该功能的才具备，如 IN

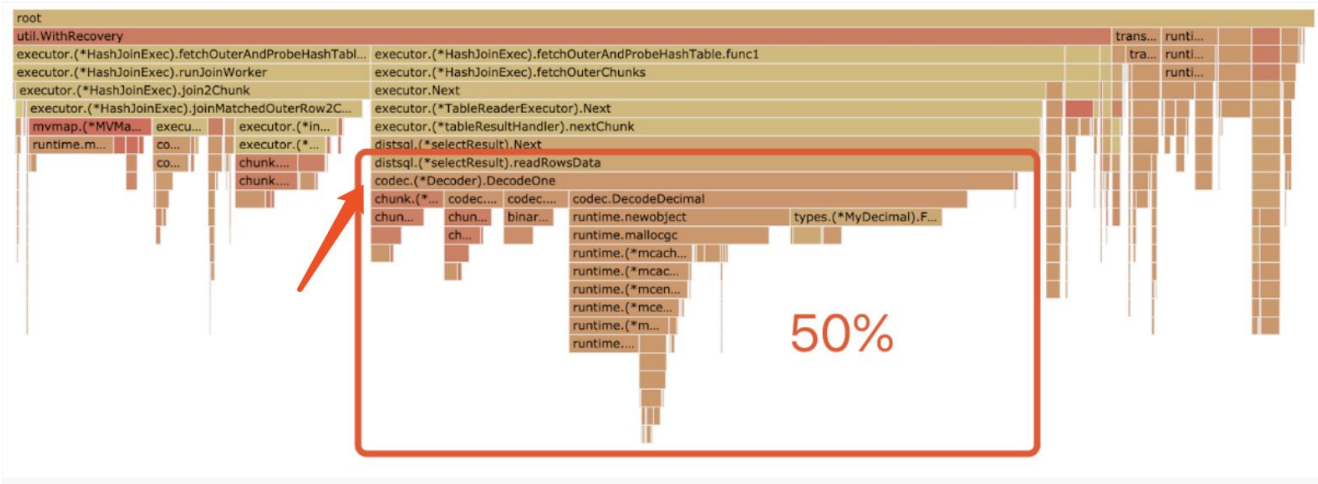



```
const count =  1 , threshold =  1
BenchmarkVectorizedBuiltinOtherFunc/builtinInIntSig-VecBuiltinFunc-12          111224          10272 ns/op          1024 B/op          1 allocs/op
BenchmarkVectorizedBuiltinOtherFunc/builtinInIntSig-NonVecBuiltinFunc-12        62086          18713 ns/op             0 B/op          0 allocs/op

const count =  1 , threshold =  200
BenchmarkVectorizedBuiltinOtherFunc/builtinInIntSig-VecBuiltinFunc-12          52753          22356 ns/op          1024 B/op          1 allocs/op
BenchmarkVectorizedBuiltinOtherFunc/builtinInIntSig-NonVecBuiltinFunc-12       16131          74576 ns/op             0 B/op          0 allocs/op
```

Chunk RPC

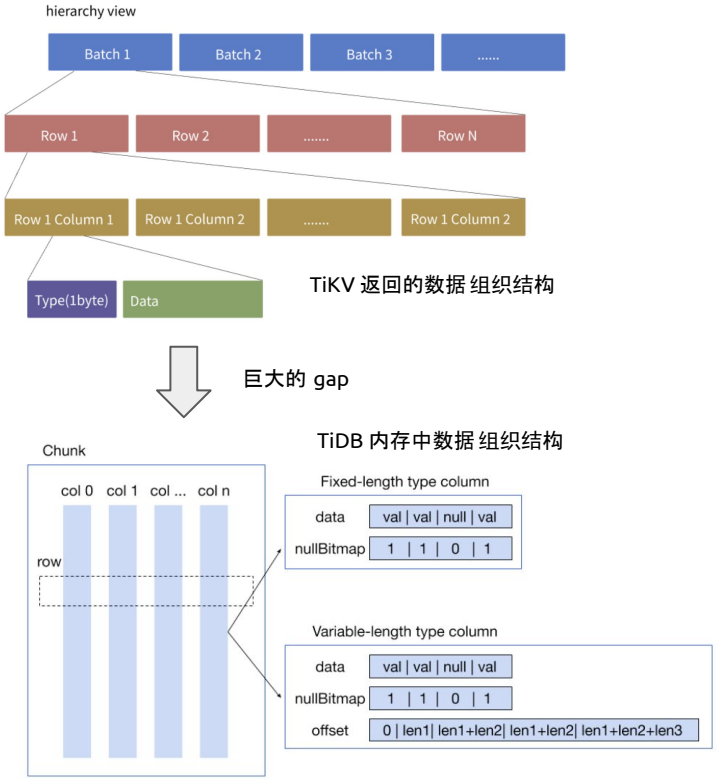
做某测试时发现 Decde TiKV 返回的数据 CPU 开销占比 50%



Chunk RPC

为什么会有如此大的开销？

1. 巨大函数调用开销 (NRows * NCols)
2. 某些类型解析复杂（如 Decimal）

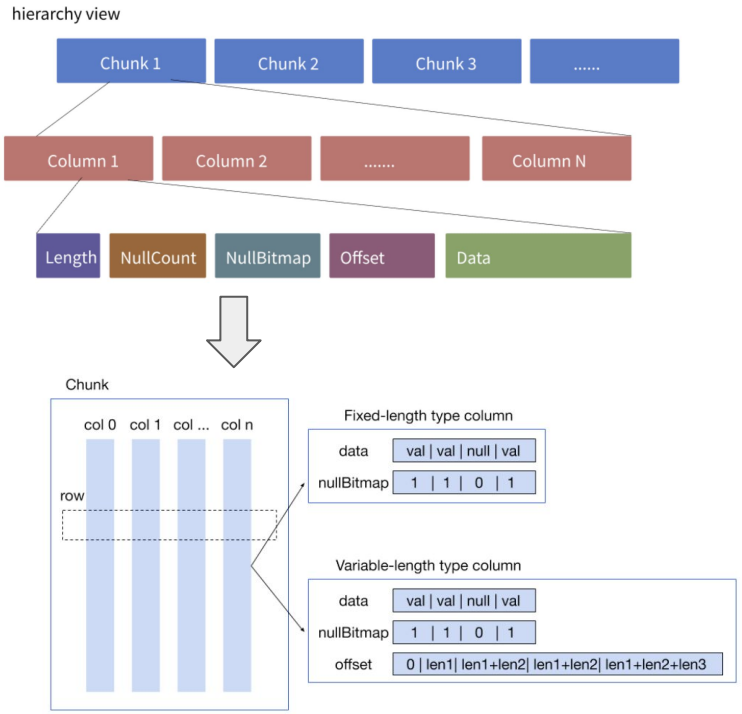


Chunk RPC

如何解决？

- 返回数据行转列（一列类型相同），避免编码过多函数调用
- 复杂的编码工作在 TiKV 端提前弄好（计算下推）

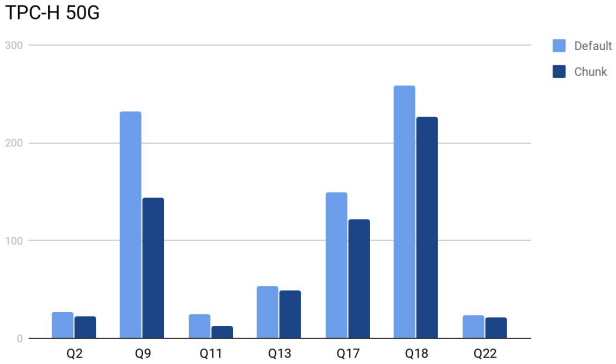
说白了就是让 TiDB 拿到的数据可以直接用，尽量减少编码工作。



Chunk RPC

TPC-H 测试结果：

- 大部分 query 都有收益
- 数据交换量大的 query 尤其明显



	Default (s)	Chunk (s)	
Q2	26.60562034	22.81698	-14.24
Q9	232.6177125	144.19972	-38.01
Q11	24.73787997	12.69548	-48.68
Q13	53.94764497	48.56367	-9.98
Q17	148.9505735	122.07989	-18.04
Q18	259.0387741	226.94387	-12.39
Q22	23.21108807	21.26832	-8.37

PingCAP

课程作业

课程作业

分值：600

题目描述：

目前，聚合函数的内存追踪有待完善；

认领一个聚合函数，并根据 issue 内描述完成他的内存追踪 issues/19369。

课程作业

分值：3000

题目描述：

认领一个还未进行滑动窗口优化的聚合函数，并优化他 issues/12967

课程作业

分值：3000

题目描述：

认领一个还未 inline projection 优化的算子，并优化他 issues/14428

课程作业

分值：3000

题目描述：

认领一个还未被实现的聚合函数，并实现他 issues/7623

课程作业

分值：600

题目描述：

认领一个表达式函数的问题，并修复 issues/11223

课程作业

分值：6600

题目描述：

使用已有的 Shuffle 算子，实现并行化 Stream Aggregate issues/9244

课程作业

分值：6600

题目描述：

使用已有的 Shuffle 算子，实现并行化 Merge Join issues/14441

课程作业

分值：600

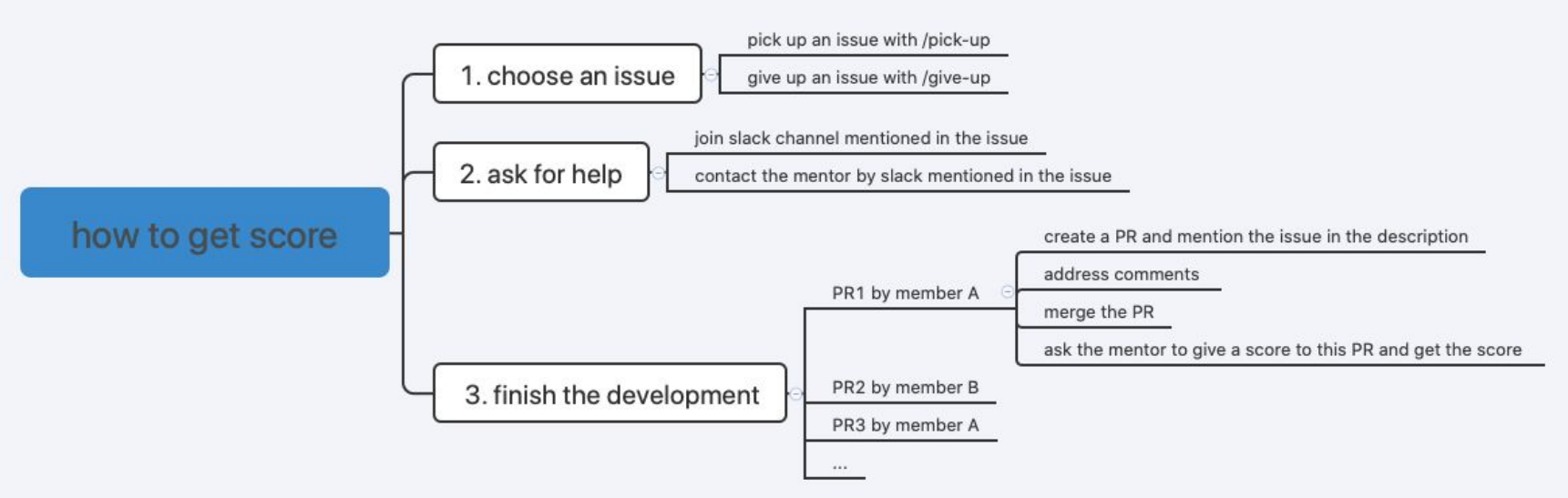
题目描述：

选取任意一个执行器或表达式相关的 bug issue 并修复它

help-wanted issues of execution

help-wanted issues of expressions

作业认领方式


how to get score
1. choose an issue
pick up an issue with /pick-up
give up an issue with /give-up
2. ask for help
join slack channel mentioned in the issue
contact the mentor by slack mentioned in the issue
3. finish the development
PR1 by member A
create a PR and mention the issue in the description
address comments
merge the PR
ask the mentor to give a score to this PR and get the score
PR2 by member B
PR3 by member A
...

作业认领相关命令

/pick-up

- 作用：issue 评论中回复认领 issue，如果是多人协作完成，派一个代表 pick 即可，对外只是标记这个任务已经有人在处理了。pick-up 完毕后，该 issue 会自动打上 picked 标签
- 权限：anyone
- 认领后：七天无动态认为该同学无法完成该任务，将自动 give-up

/give-up

- 作用：issue 评论中回复放弃当前认领的任务，give up 完毕后，该 issue 的 picked 标签会被移除
- 权限：当前挑战者

关联 PR 和 issue，PR 描述中按照以下方式之一关联 issue

- Issue Number: close #xxx
- Issue Number: #xxx

课程答疑与学习反馈

扫描左侧二维码填写报名信息，加入课程学习交流群，课程讲师在线答疑，学习效果 up up！

更多课程

想要了解更多关于 TiDB 运维、部署以及 TiDB 内核原理相关课程，可以扫描左侧二维码，或直接进入 http: //university.pingcap.com 查看

Thank you!