Go Practices in TiDB

姚维

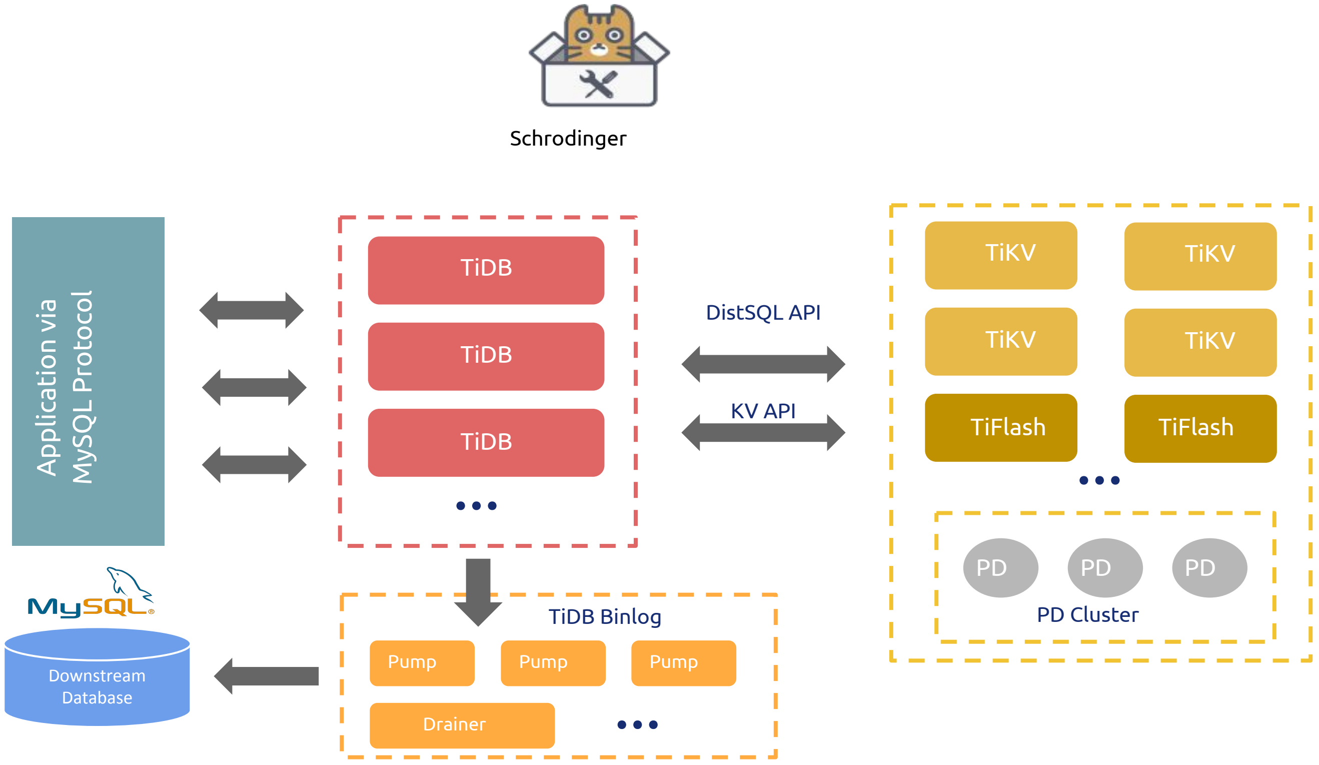
PingCAP
wink@pingcap.com

探探 Gopher China 2019

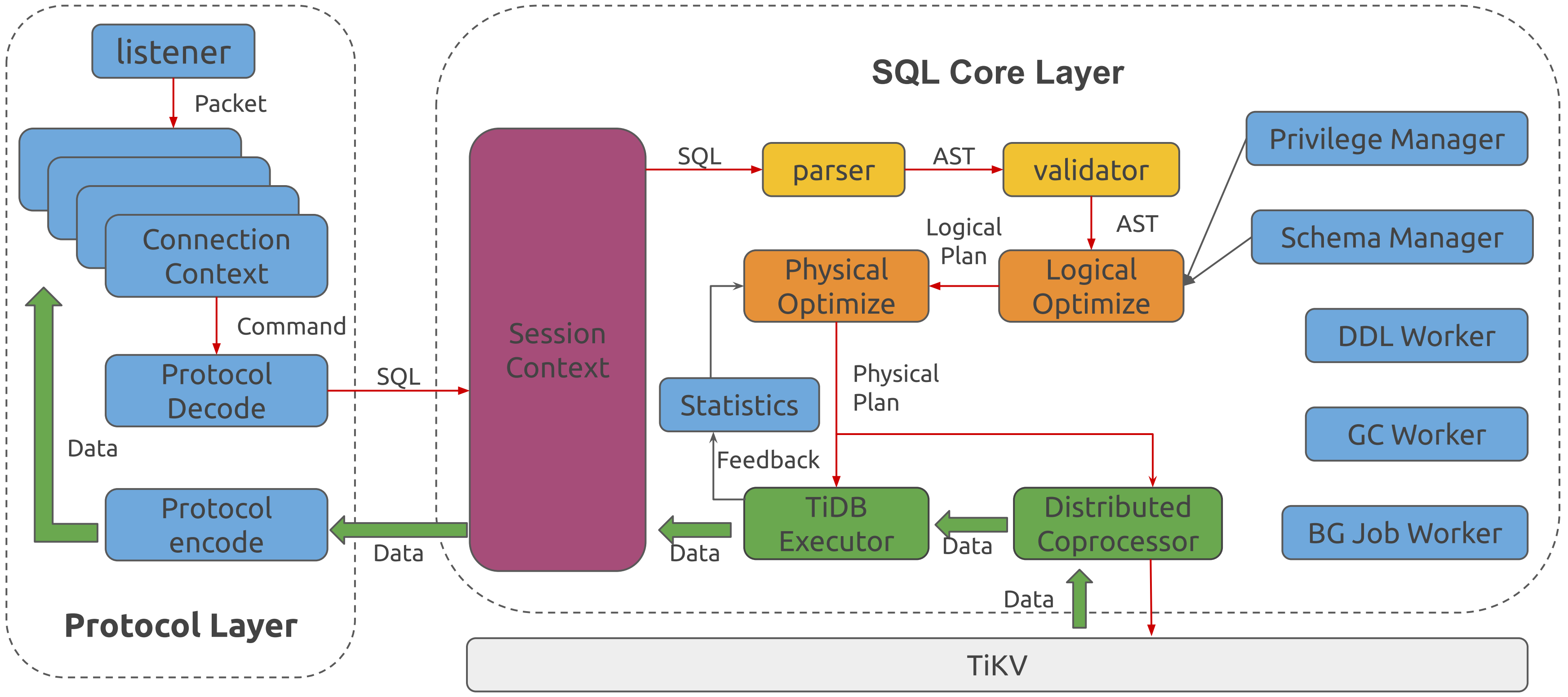
Agenda

- How to build a stable database
    - Schrodinger-test platform
    - Failpoint injection
    - Goroutine-leak detection
- Optimization
    - Chunk vs interface{}
    - Vectorized execution

TiDB Overview



TiDB SQL Layer



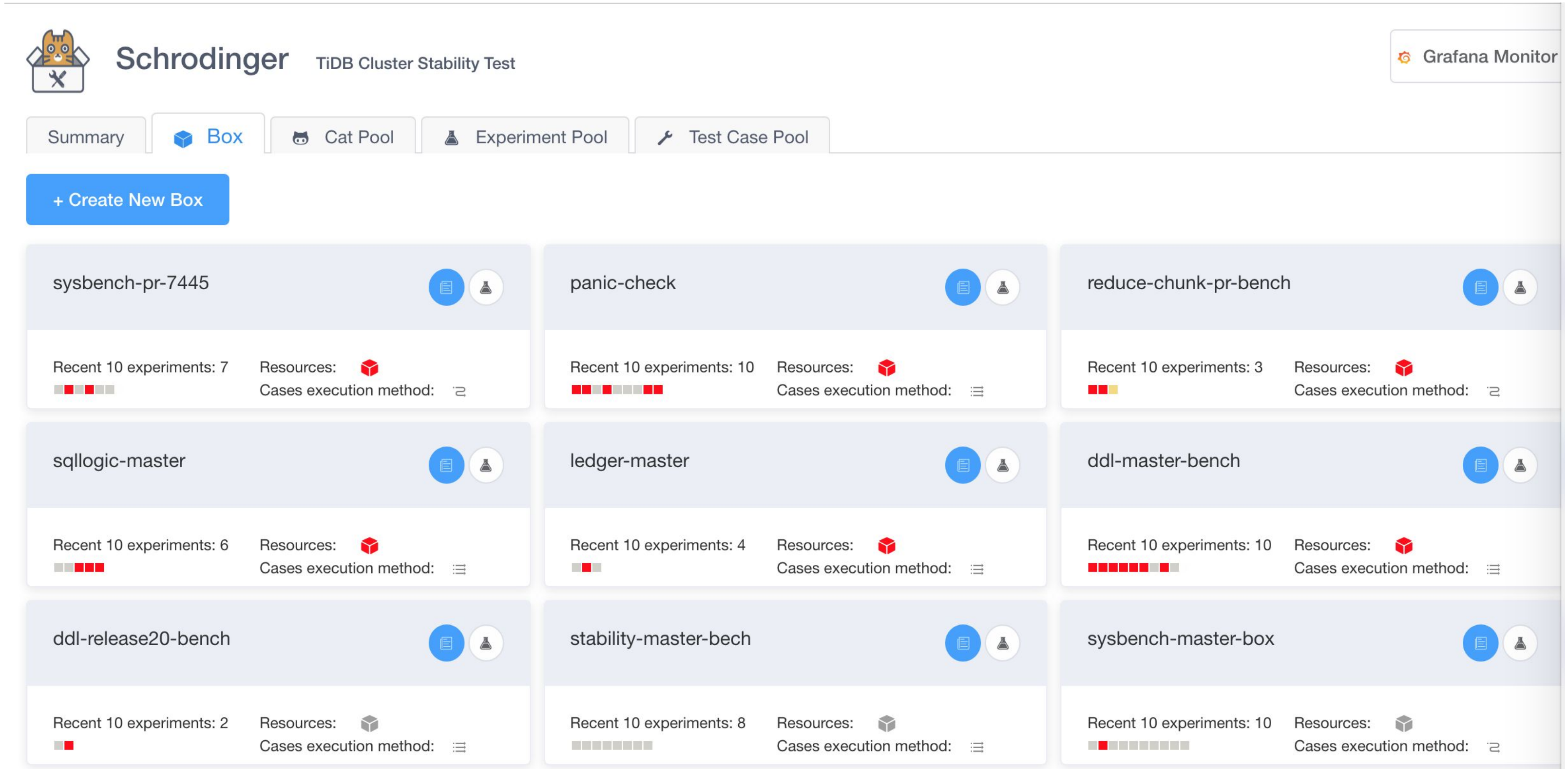
Distributed system testing

- Errors can happen anywhere, any time
- Hardware
  - disk error
  - network error
  - CPU
  - clock
- Software
  - file system
  - network & protocol
  - library
- We need to simulate everything

What is Schrodinger(1/2)

- Define a cluster with configuration, we call it Cat
- Prepare some test cases, like money transfer
- Decide which faults we should inject, we call it Nemesis
- Put the Cat, test cases, nemesises into a Box
- The Schrodinger will help us start the cluster, run the tests, inject faults
- If something is wrong, the Cat is dead, and the Schrodinger will give us a report

What is Schrodinger(2/2)



Schrodinger TiDB Cluster Stability Test

Grafana Monitor

Summary | Box | Cat Pool | Experiment Pool | Test Case Pool

+ Create New Box

sysbench-pr-7445
Recent 10 experiments: 7
Resources:
Cases execution method:

panic-check
Recent 10 experiments: 10
Resources:
Cases execution method:

reduce-chunk-pr-bench
Recent 10 experiments: 3
Resources:
Cases execution method:

sqllogic-master
Recent 10 experiments: 6
Resources:
Cases execution method:

ledger-master
Recent 10 experiments: 4
Resources:
Cases execution method:

ddl-master-bench
Recent 10 experiments: 10
Resources:
Cases execution method:

ddl-release20-bench
Recent 10 experiments: 2
Resources:
Cases execution method:

stability-master-bech
Recent 10 experiments: 8
Resources:
Cases execution method:

sysbench-master-box
Recent 10 experiments: 10
Resources:
Cases execution method:

Failpoint injection

- Failpoints are used to add code points where errors may be injected
- Why we need failpoints?
    - Some errors are hard to reproduce. Like network, disk errors
    - Easier to cover corner cases

```
func someFunc() string {
        // gofail: var SomeFuncString string
        // // this is called when the failpoint is triggered
        // return SomeFuncString
        return "default"
}
```

About gofail

- An implementation of FreeBSD failpoints for Golang.
  - https://www.freebsd.org/cgi/man.cgi?query=fail
- Define failpoints by comments
  - `gofail enable` converts comments to code
  - `gofail disable` converts code to comments

Gofail in TiDB

https://github.com/pingcap/tidb/blob/master/store/tikv/txn.go#L223

```
func (txn *tikvTxn) Commit(ctx context.Context) error {
            // gofail: var mockCommitError bool
            // if mockCommitError && kv.IsMockCommitErrorEnable() {
            //  kv.MockCommitErrorDisable()
            //          return errors.New("mock commit error")
            // }
            ....
}
```

The generated gofail code

```
$ gofail enable store/tikv/txn.go
```

```
func (txn *tikvTxn) Commit(ctx context.Context) error {
        if vmockCommitError, __fpErr := __fp_mockCommitError.Acquire(); __fpErr == nil { defer __fp_mockCommitError.Release();
mockCommitError, __fpTypeOK := vmockCommitError.(bool); if !__fpTypeOK { goto __badTypemockCommitError}
                if mockCommitError && kv.IsMockCommitErrorEnable() {
                 kv.MockCommitErrorDisable()
                        return errors.New("mock commit error")
                }; __badTypemockCommitError: __fp_mockCommitError.BadType(vmockCommitError, "bool"); };


        ....
}
```

Why we need a new failpoint

- Generated code is not readable
- Concurrent testing will use the same failpoint
- No tools to enable and disable the failpoint automatically
- Code in comments can’t be analyzed by static analysis tools

https://github.com/pingcap/failpoint

The ideal form of failpoint (Used in TiKV)

- pingcap/fail-rs

```
fail_point!("transport_on_send_store", |sid| if let Some(sid) = sid {
    let sid: u64 = sid.parse().unwrap();
    if sid == store_id {
        self.raft_client.wl().addrs.remove(&store_id);
    }
})
```

- What difficulties have we encountered?
  - No macro support in Golang
  - No compiler plugin support in Golang
  - It’s not elegant to use go build tags (`go build --tags="enable-failpoint"`)

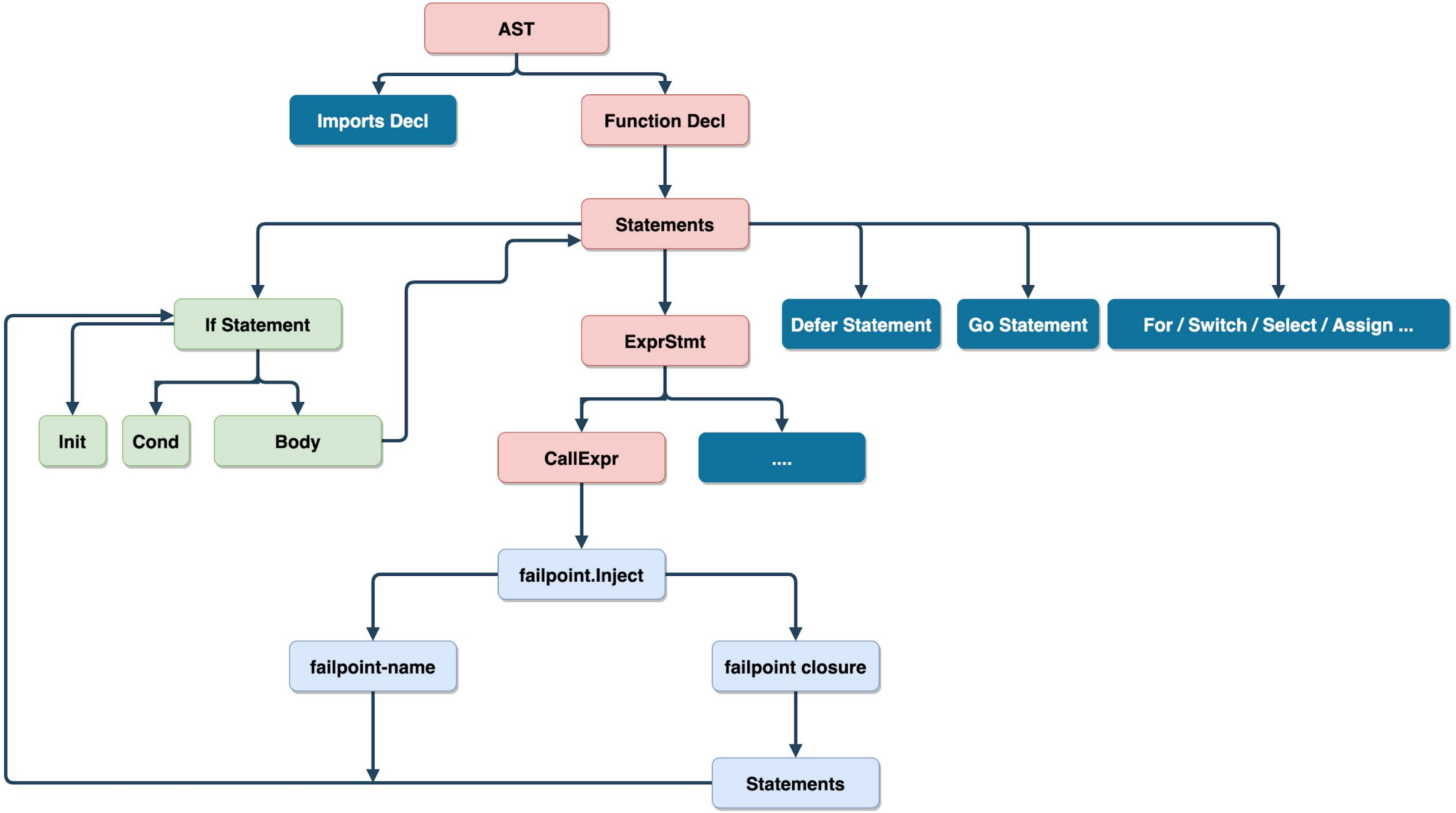
Implementation in the new failpoint

- Define a group of marker functions
- Parse imports and prune a source file which does not import `failpoint`
- Traverse AST to find marker function calls
- Marker function call will be rewritten with an *IF* statement, which calls `failpoint.Eval` to determine whether a failpoint is active and executes failpoint code if the failpoint is enabled

```
var outVar = "declare in outer scope"
failpoint.Inject("failpoint-name", func(val failpoint.Value) {
        return errors.Errorf("mock failpoint error")
})
```

AST Rewrite

```
var outVar = "declare in outer scope"
if ok, val := failpoint.Eval("failpoint-name"); ok {
        return errors.Errorf("mock failpoint error")
}
```


AST
Imports Decl
Function Decl
Statements
If Statement
ExprStmt
Defer Statement
Go Statement
For / Switch / Select / Assign ...
Init
Cond
Body
CallExpr
....
failpoint.Inject
failpoint-name
failpoint closure
Statements

Marker functions in the new failpoint

Just an empty function:

- `func Inject(fpname string, fpblock func(val Value)) {}`
- `func InjectContext(fpname string, ctx context.Context, fpblock func(val Value)) {}`
- `func Break(label ...string) {}`
- `func Goto(label string) {}`
- `func Continue(label ...string) {}`
- `func Fallthrough() {}`
- `func Label(label string) {}`

Concurrent failpoint

You can control a failpoint by failpoint.WithHook

```
func (s *dmlSuite) TestCRUDParallel() {
    sctx := failpoint.WithHook(context.Backgroud(), func(ctx context.Context, fpname string) bool {
        return ctx.Value(fpname) != nil // Determine by ctx key
    })
    insertFailpoints = map[string]struct{} {
        "insert-record-fp": {},
        "insert-index-fp": {},
        "on-duplicate-fp": {},
    }
    ictx := failpoint.WithHook(context.Backgroud(), func(ctx context.Context, fpname string) bool {
        _, found := insertFailpoints[fpname] // Only enables some faipoints
        return found
    })

    // ... other dml parallel test cases
    s.RunParallel(buildSelectTests(sctx))
    s.RunParallel(buildInsertTests(ictx)
}
```

All markers

```go
failpoint.Label("outer")
for i := 0; i < 100; i++ {
    inner:
        for j := 0; j < 1000; j++ {
            switch rand.Intn(j) + i {
            case j / 5:
                failpoint.Break()
            case j / 7:
                failpoint.Continue("outer")
            case j / 9:
                failpoint.Fallthrough()
            case j / 10:
                failpoint.Goto("outer")
            default:
                failpoint.Inject("failpoint-name", func(val failpoint.Value) {
                    fmt.Println("unit-test", val.(int))
                    if val == j/11 {
                        failpoint.Break("inner")
                    } else {
                        failpoint.Goto("outer")
                    }
                })
            }
        }
}
```

```go
outer:
    for i := 0; i < 100; i++ {
    inner:
        for j := 0; j < 1000; j++ {
            switch rand.Intn(j) + i {
            case j / 5:
                break
            case j / 7:
                continue outer
            case j / 9:
                fallthrough
            case j / 10:
                goto outer
            default:
                if ok, val := failpoint.Eval("failpoint-name"); ok {
                    fmt.Println("unit-test", val.(int))
                    if val == j/11 {
                        break inner
                    } else {
                        goto outer
                    }
                }
            }
        }
    }
```

Let us talk about goroutine leak

What is goroutine leak?

```
func main() {
	go func() {
		// Just invalid the deadlock detection.
		for {
			time.Sleep(1 * time.Second)
		}
	}()

	done := make(chan bool)

	leakCh := make(chan string, 1)
	go func() {
		// This goroutine is leaked.
		for {
			recv, more := <-leakCh
			if !more {
				break
			}
			fmt.Printf("recv: %v", recv)
		}
		done <- true
	}()

	// We forget to close the channel.
	// close(leakCh)
	<-done
}
```

Detect the goroutine leak in UT

- runtime.Stack(buf, true) to find out all running goroutines
- Before the unit test runs, remembers all running goroutines
- After the unit test is finished, if there are any new goroutines, that are leaked goroutines

```
func TestT(t *testing.T) {
        testleak.BeforeTest()
        TestingT(t)
        testleak.AfterTestT(t)()
}
```

Chunk - Effective row format in Go

Row format in TiDB

```
CREATE TABLE `t` (
  `a` int(11) DEFAULT NULL,
  `b` varchar(10) DEFAULT NULL,
  `c` decimal(10,5) DEFAULT NULL,
  `d` timestamp NULL DEFAULT NULL
)
```

$\Rightarrow$

a (int)	b (varchar)	c (decimal)	d (timestamp)
1	"a"	1.0	1555516235
2	"b"	1.2	1555514235
3	"c"	5.1	1555518235

Row format in TiDB

In old days:

```
// Datum is a data box holds different kind of data.
// It has better performance and is easier to use than `interface{}`.
type Datum struct {
        k           byte            // datum kind.
        collation   uint8           // collation can hold uint8 values.
        decimal     uint16          // decimal can hold uint16 values.
        length      uint32          // length can hold uint32 values.
        i           int64           // i can hold int64 uint64 float64 values.
        b           []byte          // b can hold string or []byte values.
        x           interface{}     // x hold all other types.
}
```

[]Datum ⟶

a (int)	b (varchar)	c (decimal)	d (timestamp)
1	"a"	1.0	1555516235
2	"b"	1.2	1555514235
3	"c"	5.1	1555518235

Row format in TiDB

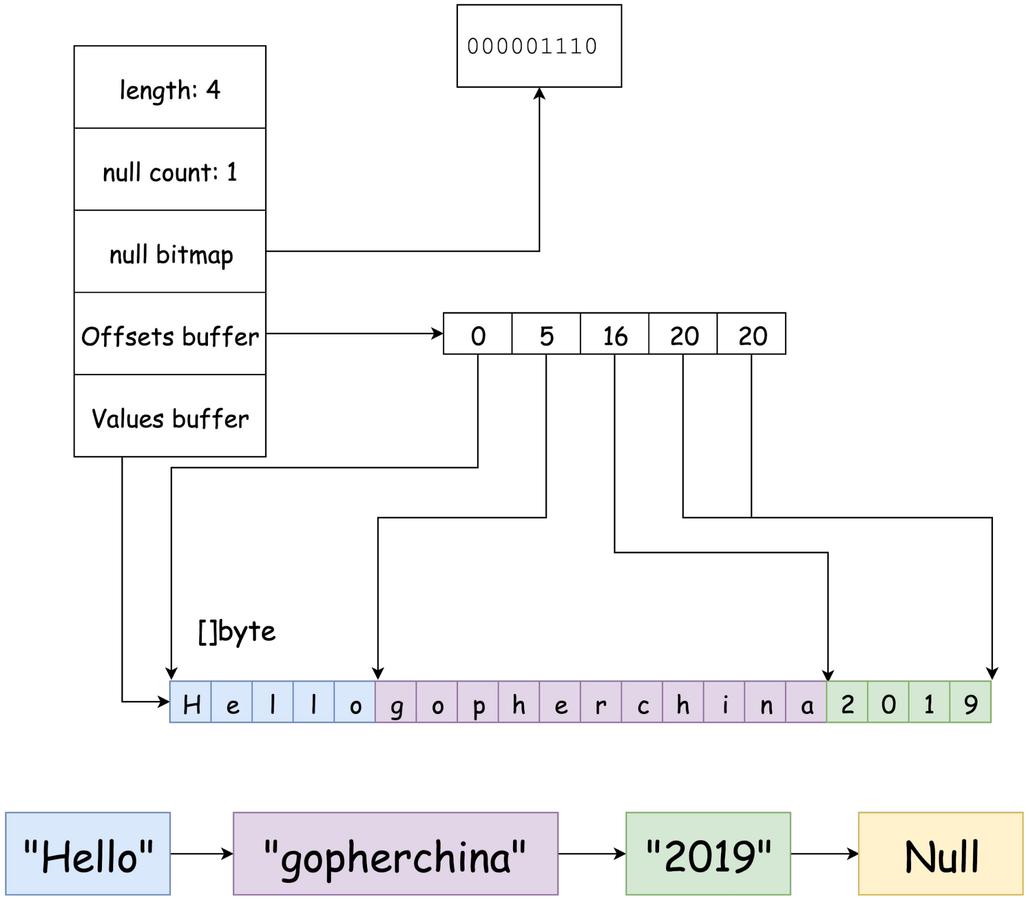
- **What is the disadvantages of Datum?**
    - Use unnecessary memory in every column.
    - Must use type assertion to get complex types:

```
func (d *Datum) GetMysqlDecimal() *MyDecimal {
        return d.x.(*MyDecimal)
}
```

    - Non-effective to do vectorizable serial computation
- So how to optimize it?

Apache Arrow

- Binary data format
- Array lengths
- Null count
- Null bitmaps
- Offsets buffer
- Values Array
- More details please see the doc

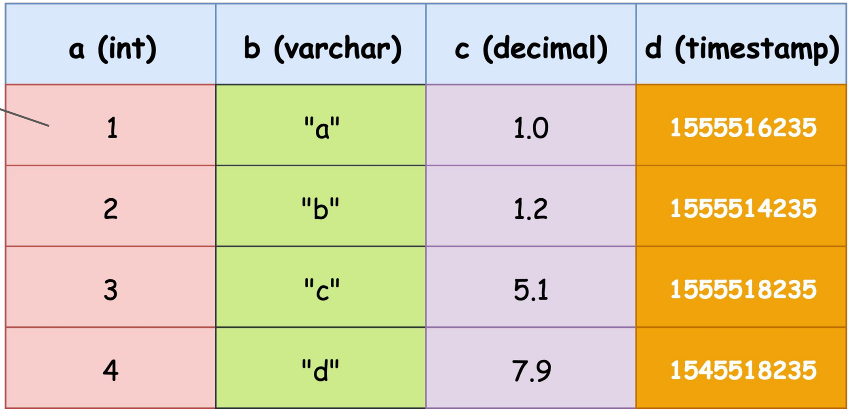


Chunk

- Columnar layout
- Fixed length type can eliminate the offsets buffer.

```
type Chunk struct {
        columns []*column
}

type column struct {
        length          int
        nullCount       int
        nullBitmap      []byte
        offsets         []int32
        data            []byte
        elemBuf         []byte
}
```



a (int)	b (varchar)	c (decimal)	d (timestamp)
1	"a"	1.0	1555516235
2	"b"	1.2	1555514235
3	"c"	5.1	1555518235
4	"d"	7.9	1545518235

Chunk

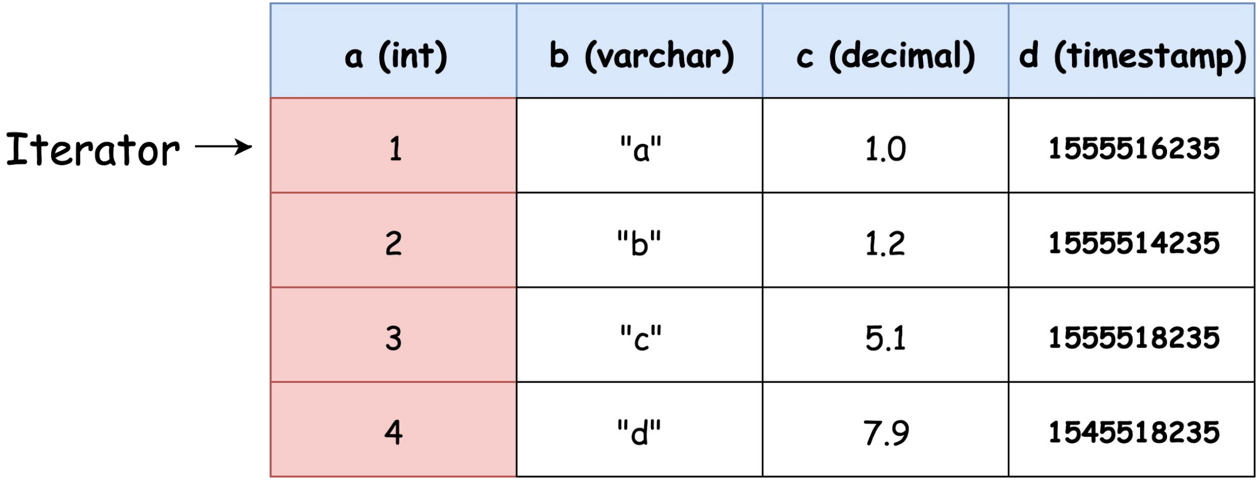
- **Use unsafe.pointers to get complex types:**

```
func (r Row) GetMyDecimal(colIdx int) *types.MyDecimal {
        col := r.c.columns[colIdx]
        return (*types.MyDecimal)(unsafe.Pointer(&col.data[r.idx*types.MyDecimalStructSize]))
}

func (r Row) GetUint64(colIdx int) uint64 {
        col := r.c.columns[colIdx]
        return *(*uint64)(unsafe.Pointer(&col.data[r.idx*8]))
}
```

Chunk

- Less CPU cache miss
- Vectorized Execute expressions:



a (int)	b (varchar)	c (decimal)	d (timestamp)
1	"a"	1.0	1555516235
2	"b"	1.2	1555514235
3	"c"	5.1	1555518235
4	"d"	7.9	1545518235

Chunk

- **Vectorized Execute expressions:**

```
func VectorizedExecute(ctx Context, exprs []Expression, iterator *Iterator4Chunk, output *Chunk) error {
        for colID, expr := range exprs {
                evalOneColumn(ctx, expr, iterator, output, colID)
        }
        return nil
}


func evalOneColumn(ctx Context, expr Expression, iterator *Iterator4Chunk, output *Chunk, colID int) (err error) {
        switch fieldType, evalType := expr.GetType(), expr.GetType().EvalType(); evalType {
        case types.ETInt:
                for row := iterator.Begin(); err == nil && row != iterator.End(); row = iterator.Next() {
                        err = executeToInt(ctx, expr, fieldType, row, output, colID)
                }

        ....
}
```

Lessons learned

- Make things right, then make things faster
- Premature optimization is the root of all evil
  - Interface{} → Datum → Chunk
- Suspect any abnormal things, and find the root reason
- More test types:
  - random generated testing
  - compatibility testing
  - concurrent testing
  - large-scale cluster testing
  - stability testing

Thanks!

Gopherchina 2019

该二维码7天内(5月2日前)有效，重新进入将更新