Scaling DB Access for 100s of billions of queries per day

Hera - Kenneth Kang & Petrica Voicu - June 2019 QCon New York
TLDR - Hera

Open Source
High Efficiency
Reliable Access to data stores

Out Of DB Connections?

• Scaling
  • Multiplexing
  • Sharding
• Resiliency
• Manageability
• Our switch to Go Lang
About Petrica

• Works @ PayPal
  • developing the database proxy.
  • database backed publish-subscriber messaging system
  • PayPal Here backend

• Golang contributor

• Previously @
  • Worked on distributed systems, embedded system, desktop apps
About Kenneth

- Works @ PayPal
  - paypal.de
  - PayPal Verisign Two Factor Authentication
  - Coded shared memory for sharding
  - Migrated to Go (hera)

- Previously @
  - Visualization of Genomic Metabolism, Artificial Chemistry, and EPICS Monitoring
About PayPal - 2018 facts

100+ Currencies
200+ Markets
267M Active Customer Accounts

9.9B Payments Transactions
$578B Total Payments Volume
Up 27% YoY*
Total Payment Volume

2,700 Applications
4,500 Engineers
17,000 Releases
PayPal OLTP Environment scale

- 2000+ Database Instances
- 3000+ Database Nodes
- 74 PB Total Storage
- 200,000+ Application Servers
- 10+ Availability Zones
- 300+ Schemas
- 1+ Billion Calls/hour
- 100+ Billion Calls/day
Key Take-aways

- Massive scale
- Complexity
- Growth
Typical Database Access by Application
Typical Database Access by Application

![Diagram showing the process of a microservice making database requests and receiving responses with delays indicated.](image-url)
Typical Database Access by Application

- Request
- Lease
- Connect
- Request
- Request
- Request
- Release
- Response

Connection is pre-established, cached and re-used for each request
Configure the pool cache large enough for availability.
From Few to Thousands Microservices

Configure the pool cache large enough for availability

Add more service when scaling up
From Few to Thousands Microservices

Configure the pool cache large enough for availability

Add more service when scaling up

Microservice

Microservice

Microservice
From Few to Thousands Microservices

Configure the pool cache large enough for availability

Add more service when scaling up
From Few to Thousands Microservices

- Configure the pool cache large enough for availability
- Add more service when scaling up
- Hit a connection limit
Solutions?

- Lower the connection pool size in each microservice node
- Improve microservice code to use the connection more efficiently
- Buy better database hardware
While servicing requests, the microservices use only some DB connections.
While servicing requests, the microservices use only some DB connections.

In some node all the connections can be temporary busy.
While servicing requests, the microservices use only some DB connections. In some node all the connections can be temporary busy. Client requests will incur latency though the database has enough capacity.
Solutions?

Scale with shared pool !!!
Shared connection pool

Request

Microservice

Shared pool

Request

Microservice
Typical service request

Service request → SQL request → SQL request → SQL request → SQL request → Service response

DB connection

Database connection is allocated but mostly idle
Serving other requests when client seems idle?
Multiplexing

Services perceive that they have a database connection.
Multiplexing

Services perceive that they have a database connection
Hera - Highly Efficient Reliable Access to data stores

Shared connection pool multiplexer
HERA
Microservice-First
DB proxy

Principles and constraints

- Correctness
- Resiliency, -ilities, performance
- Minimal configuration
- Thin client
- Minimal change to wire protocol
Let’s talk about other features!
Scale

Features

• Multiplexing
• Read write split
• Sharding
Read-Write Split

Each database is exchanging lock information

Oracle RAC - Real Application Cluster

©2019 PayPal Inc. Confidential and proprietary.
Read-Write Split

Separate write queries and direct to one node. Reduces traffic between DB nodes.

Oracle RAC - Real Application Cluster

Hera Node
Sharding

• Client vs Server
• Finding the Shard
• Sample Query Run
• Convert Legacy App
• Move Scuttle Bin’s Data
Sharding: Client vs Server

C++ Client

C++ Lib

Database (shard 0)

Database (shard 1)

Database (shard 2)
Sharding: Same Logic in Java Client
Sharding: Duplicate Logic in Go Client

- C++ Client
- Java Client
- Go Client

- C++ Lib
- Java Lib
- Go Lib

- Database (shard 0)
- Database (shard 1)
- Database (shard 2)
Sharding: Copy Logic in Python
Sharding: Node.js Client Too
Sharding: Keep Logic in Server

- Worker
- Hera Multiplexer
- C++ Client
  - C++ Lib
  - Hera Java Lib
  - Database (shard 0)
- Java Client
  - Hera Java Lib
  - Database (shard 1)
- Go Client
  - Hera Go Lib
  - Database (shard 2)
- Python Client
  - Python Lib
  - Node.js Client
  - Node.js Lib
Sharding: Implementation

1. Shard Key: account_id=2000

Diagram:
- Shard Key
- Hash function (shard key value) %K
- Scuttle bin to logical shard mapping
- Logical to physical shard mapping
- DB Shard 0
- DB Shard N
Sharding: Implementation

1. Shard Key: account_id=2000

**Sharding: Implementation**

1. **Shard Key:** account_id=2000

2. **Murmur3Hash(2000)%1024=280**

3. **Select** shard_id **from** hera_shard_map **where** scuttle_id = 280;

   shard_id

   ---------

   1
Sharding: Implementation

1. Shard Key: account_id=2000
2. Murmur3Hash(2000)%1024=280
3. Select shard_id from hera_shard_map where scuttle_id = 280;
   shard_id
   ------
   1
4. TWO_TASK_1 = LOAN_SH1
   TWO_TASK_1='tcp(loan-sh1:3306)/loan'
Sharding: Converting Application Query

- Shard Key: account_id

- select * from loan, appfile where loan.id = ? and appfile.loan_id = loan.id
Sharding Compatible Query

- Shard Key: account_id

- select * from loan, appfile where loan.id = ? and appfile.loan_id = loan.id and loan.account_id = ? and appfile.account_id = loan.account_id
Sharding: Hera JDBC SQL Rewrite

- Shard Key: account_id

- `select * from loan, appfile where loan.id = :loan_id and appfile.loan_id = loan.id and loan.account_id = :account_id and appfile.account_id = loan.account_id`

- Hera can now pick out the shard key name and value
Sharding: Legacy App Conversion

- All queries directed to shard 0
- Logs queries that don’t bind to shard key

Shard Key

Hash function (shard key value) % K

Scuttle bin to logical shard mapping

Shard 0

Logical to physical shard mapping

DB Shard 0

... DB Shard N
**Sharding: Legacy App Conversion**

- Whitelist sends one value to a specific shard
- Limits risk of failures to 1 value
- Hera uses Shard 1, but same DB
- Fast rollback on errors
- Repeat for larger sets
Sharding: Physical Whitelist

- Validates permissions and data copy
• If successful, do a physical data move next

Sharding: Logical Move for One Scuttle Bin

- Whitelist
- Hash function (shard key value)
- Scuttle bin to logical shard mapping
- Logical to physical shard mapping
- DB Shard 0
- DB Shard N
Sharding: Moving Scuttle Bin

- Start data copy

- Typically, tables are partitioned by scuttle bin
Sharding: Moving Scuttle Bin

- Start data copy
- Typically, tables are partitioned by scuttle bin
- Block writes

Scuttle bin to logical shard mapping

Logical to physical shard mapping

Hash function (shard key value) % K
Sharding: Moving Scuttle Bin

- Start data copy
- Typically, tables are partitioned by scuttle bin
- Block writes
- Data fully copied

Hash function (shard key value) %K

Scuttle bin to logical shard mapping

Logical to physical shard mapping

DB Shard 0

DB Shard N
Sharding: Moving Scuttle Bin

- Start data copy
- Typically, tables are partitioned by scuttle bin
- Block writes
- Data fully copied
- Update map
- Use scuttle bin in new location
Sharding: Take Away

- Queries must bind shard key & value
- Manage risk converting legacy code
- Controlled data redistribution
Resiliency Principles

- Deal with exceptional cases
  - Surge in load
  - Slow SQL requests
  - Issues with the DB
Resiliency

Principles

- Deal with exceptional cases
  - Surge in load
  - Slow SQL requests
  - Issues with the DB
- Limit or avoid impact
- Self heal
Resiliency

Features

- Bouncer
- Surge Queue
- Slow Query Eviction
- Read Replica
- Transparent Application
- Failover
Resiliency: Oversubscription - Bouncer

Service 1

Service 2

Service 3

Service 4

Hera Node

Service 4 connection is tcp accepted and then closed without SSL.
Resiliency: Oversubscription - Surge Queue

Service 1

Service 2

Service 3

Hera Node

Service 3 can make a request. It waits--queued
Resiliency: Oversubscription - Surge Queue

Generally, 1ms later, Service 3 gets run.
Resiliency: Oversubscription - Surge Queue

Waiting more than 1s gives an error to Service 3.
Resiliency: Slow Query Eviction

When Hera is overloaded

Service 1

Service 2

Service 3

1s Slow Query

Hera Node
Resiliency: Slow Query Eviction

Hundred of queries can run once slow query is removed.
Monitoring Usage

Service 1

Service 2

Service 3

Hera Node

Free DB Connections (acpt)

Each Hera Node

Time
Resiliency: Eviction - Actual event

Each Hera Node

Happened few years back, when automatic eviction went unnoticed.
Evicting expensive query makes space for many typical queries.
Read Replicas

R1 & R2: different data replication lag
Transparent Application Failover - Read Replica Retry

- Build failover on server side
- Also failover when query is slow
Manageability

• Maintenance without customer impact
• Recommend clients recycle often
• Oracle RAC maintenance
Recycle Connections - Node Going Away

Service 2

Service 3

TCP Load Balancer

Hera Node

Hera Node

Hera Node

Database
Recycle Connections for Manageability

TCP Load Balancer

Service 2

Service 3

Hera Node

Hera Node

Hera Node
Recycle Connections - Node Returns

Service 2

Service 3

TCP Load Balancer

Hera Node

Hera Node

Hera Node
DB Maintenance

- Preparing Oracle RAC node 3 maintenance
- DBAs remove node 3 from Oracle configs
- Insert into hera_maint (inst_id, status, status_time, module, machine) values (3, ‘F’, [unix epoch]+2, [hera pool name], [host])
DB Maintenance

• After [unix epoch] +2

Take Aways
• Keep controls near those who need it
• Avoid being unnecessarily involved
Transition to Go
Transition to Go

Previous version

- Written in C++, asynchronous code
- Scalable
- Efficient
- Complex
Transition to Go

Previous version

- Written in C++, asynchronous code
- Scalable
- Efficient
- Complex

New changes

- New functional requirements
- New non functional requirements (efficiency)
# Transition to Go

## Previous version
- Written in C++, asynchronous code
- Scalable
- Efficient
- Complex

## New changes
- New functional requirements
- New non functional requirements (efficiency)

## How about writing in Go?
- Go programming is synchronous
- Scale efficiently out of the box
- Does it meet our requirements?
- Is stop-the-world garbage collection time an issue?

©2019 PayPal Inc. Confidential and proprietary.
Transition to Go

Going with Go

Proof-of-concept

1st Release: No Sharding

Full feature with C++ dependencies

Full feature all Go
Mission Accomplished

- Running in production for 1 year
- Latency and CPU met the requirements
- No clients change

- Had some issues and learnings
func routine() {
    time.Sleep(time.Second * 10)
}

func main() {
    routines := 10000
    wg.Add(routines)
    for i := 0; i < routines; i++ {
        go func() {
            routine(i)
            wg.Done()
        }()
    }
    wg.Wait()
}

Learning Example

Spawn 10000 go-routines sleep for 10 seconds to simulate processing

Running on my 2 cores test machine

How many OS threads I expect at runtime?
func routine() {
    time.Sleep(time.Second * 10)
}

func main() {
    routines := 10000
    wg.Add(routines)
    for i := 0; i < routines; i++ {
        go func() {
            routine(i)
            wg.Done()
        }()
    }
    wg.Wait()
}

Spawn 10000 go-routines sleep for 10 seconds to simulate processing

Running on my 2 cores test machine

$> ./mytest &
[1] 12345
$> ps -L -C mytest -no-headers | wc -l
6
$>
func routine() {
    //time.Sleep(time.Second * 10)
syscall.Select(0, nil, nil, nil, &syscall.Timeval{Sec: 10})
}
func main() {
    routines := 10000
    wg.Add(routines)
    for i := 0; i < routines; i++ {
        go func() {
            routine(i)
            wg.Done()
        }()
    }
    wg.Wait()
func routine() {
    //time.Sleep(time.Second * 10)
    syscall.Select(0, nil, nil, nil, &syscall.Timeval{Sec: 10})
}

func main() {
    routines := 10000
    wg.Add(routines)
    for i := 0; i < routines; i++ {
        go func() {
            routine(i)
            wg.Done()
        }()
    }
    wg.Wait()
}
Learning Example

from "runtime/debug" func SetMaxThreads()

... “A Go program creates a new thread only when a goroutine is ready to run but all the existing threads are blocked in system calls, cgo calls, or are locked to other goroutines due to use of runtime.LockOSThread”
Key Take Away

- System calls lock the OS thread. Ex: os.File read
- When using C/C++ library, be aware of the systems calls (mutexes, file reads, socket reads, sleep)
- Golang locks, socket reads, sleep are not locking the OS thread
- In your load tests, verify the number of OS threads
Open source

github.com/paypal/hera
✓ Code, documentation, examples
✓ Standard clients
  • Java: JDBC driver
  • Golang: database/sql driver

Google group heradatalink
Acknowledgements

Kamlakar Mukundan Liana Sid Yaping Shuping Stephanie Varun Qing Chun Dwain Paresh Srini Dheeraj Pradeep Pramod Samrat Akash Ashwin and countless others
Next Steps

Need more database connections?

Hera can help scale
• Multiplexing
• Sharding
• Resiliency
• Manageability

Open Source
github.com/paypal/hera
Google Groups - heradatalink

Our Future
• Scaling Tools
• More Clients