facebook

Incremental vs Rewrite From Scratch
Biased guide to a web-scale DBMS

Mark Callaghan
Database Engineering, Facebook
April 13, 2015

The choice, in theory

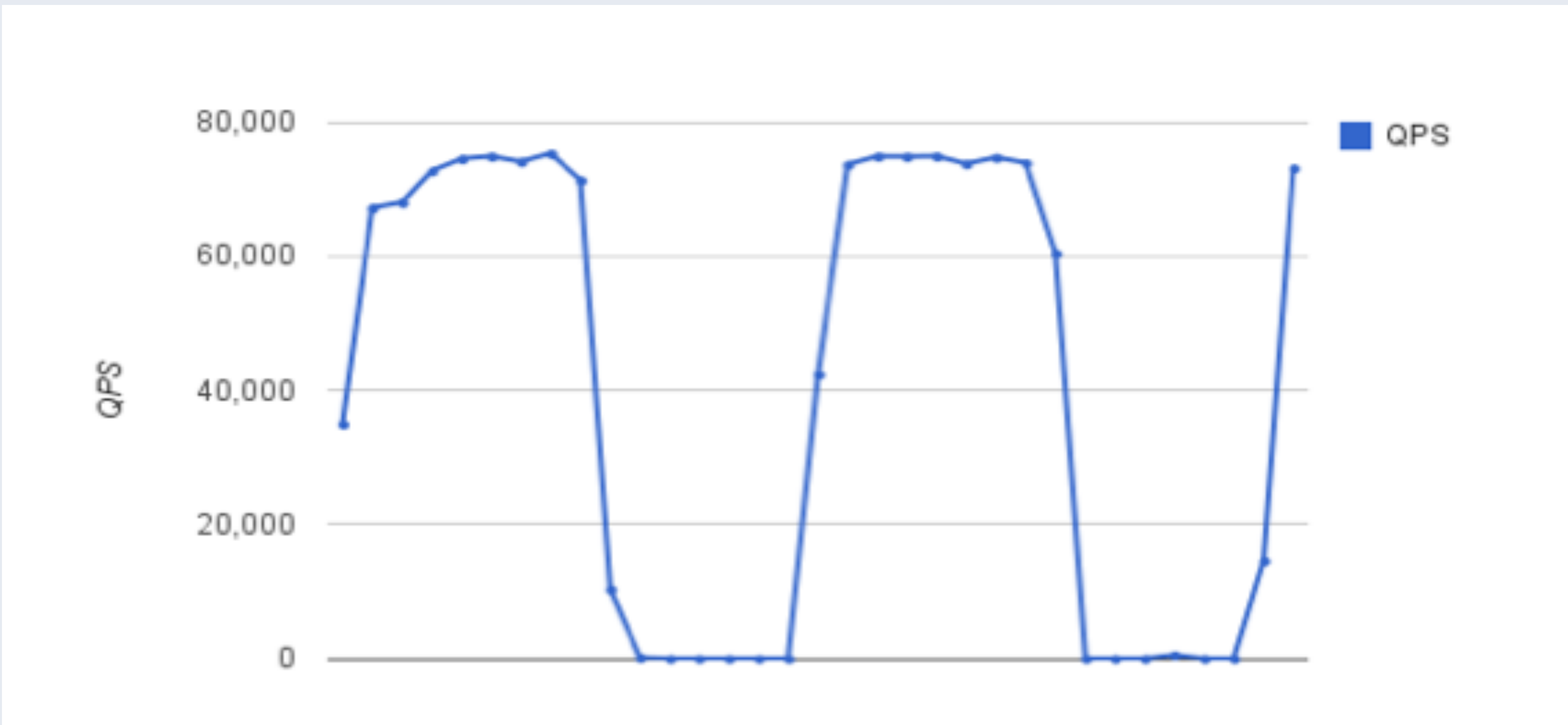
- What is the goal?
  - Make things better
- How do we do that?
  - Incremental improvements vs rewrite from scratch
- Who is making these choices?
  - Vendors, researchers and open-source power users

Define better

- Manageability
  - Fixed size team operates a growing tier
- Quality of Service
  - Response time and availability
- Efficiency
  - Performance per resource

Peak performance is overrated

Don’t forget about predictable and efficient performance



Benchmarking vs benchmarketing

Success requires both

- Benchmarketing – goal is to impress
- Benchmarking – goal is to explain and inform

Benchmark accuracy declines as the number of systems tested increases

How to win on IO-bound benchmarks

Tricks

- Load in key order
- Only use 10% of device
- Don’t run tests for a long time
- Fixed number of user threads

Avoid these overheads

- B-Tree index fragmentation
- LSM compaction
- SSD erase block cleaning
- Convoys on stalls

Rewrite is harder than you think

More fun

- Core server

Less fun

- Client libraries
- Testing
- Documentation
- Expertise
- Utilities
- Monitoring
- Glue

The choice, in practice

- MySQL -> F1/Spanner
  - Motivation: no more failover
- Closed-source -> OceanBase
  - Motivation: open-source
- Social Graph OLTP
  - Motivation: storage efficiency

MySQL over 10 years

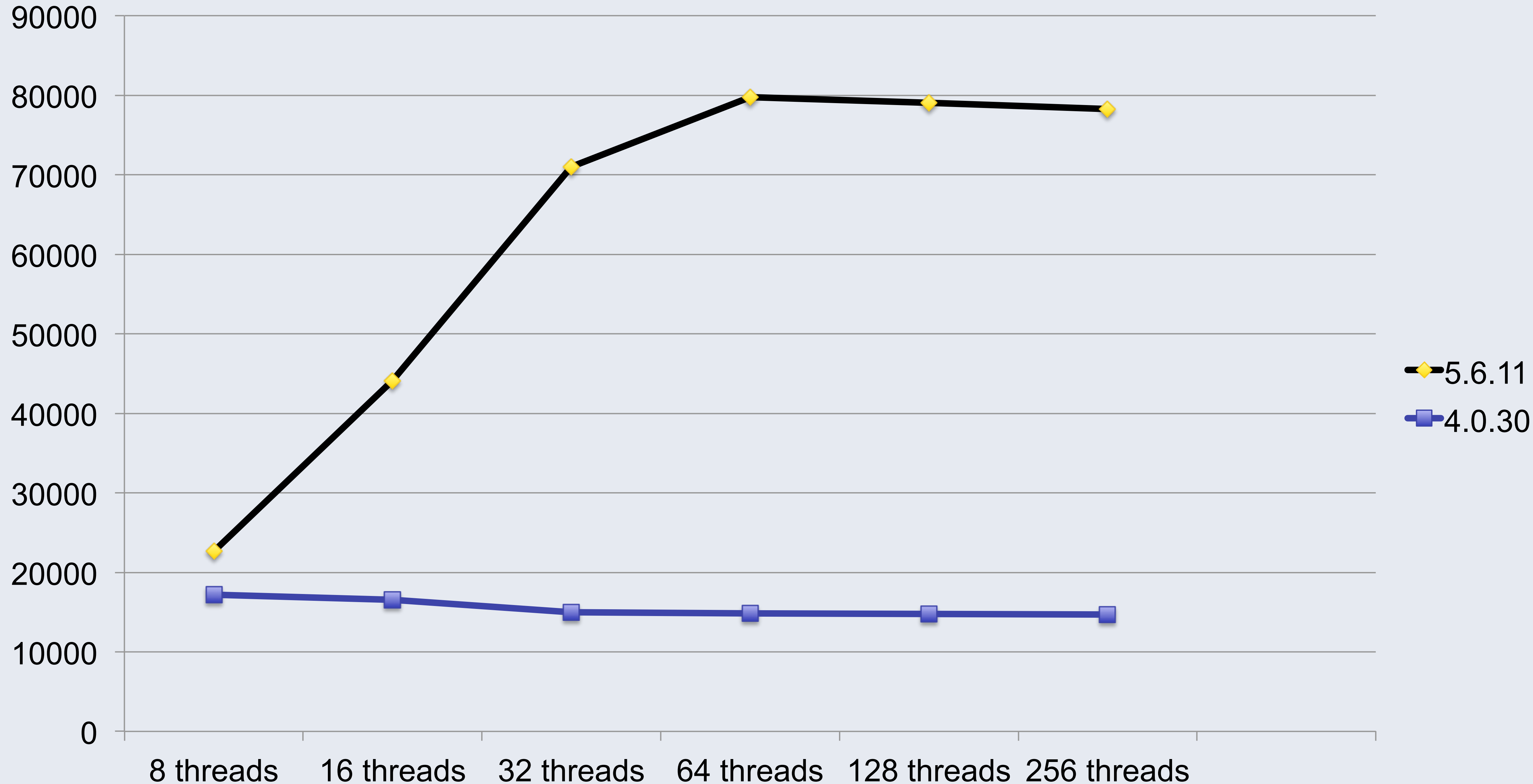
2005

- Typical server
  - 4 cores, 16G RAM, 1000 IOPs
- Many problems in MySQL
- Peak QPS < 20k

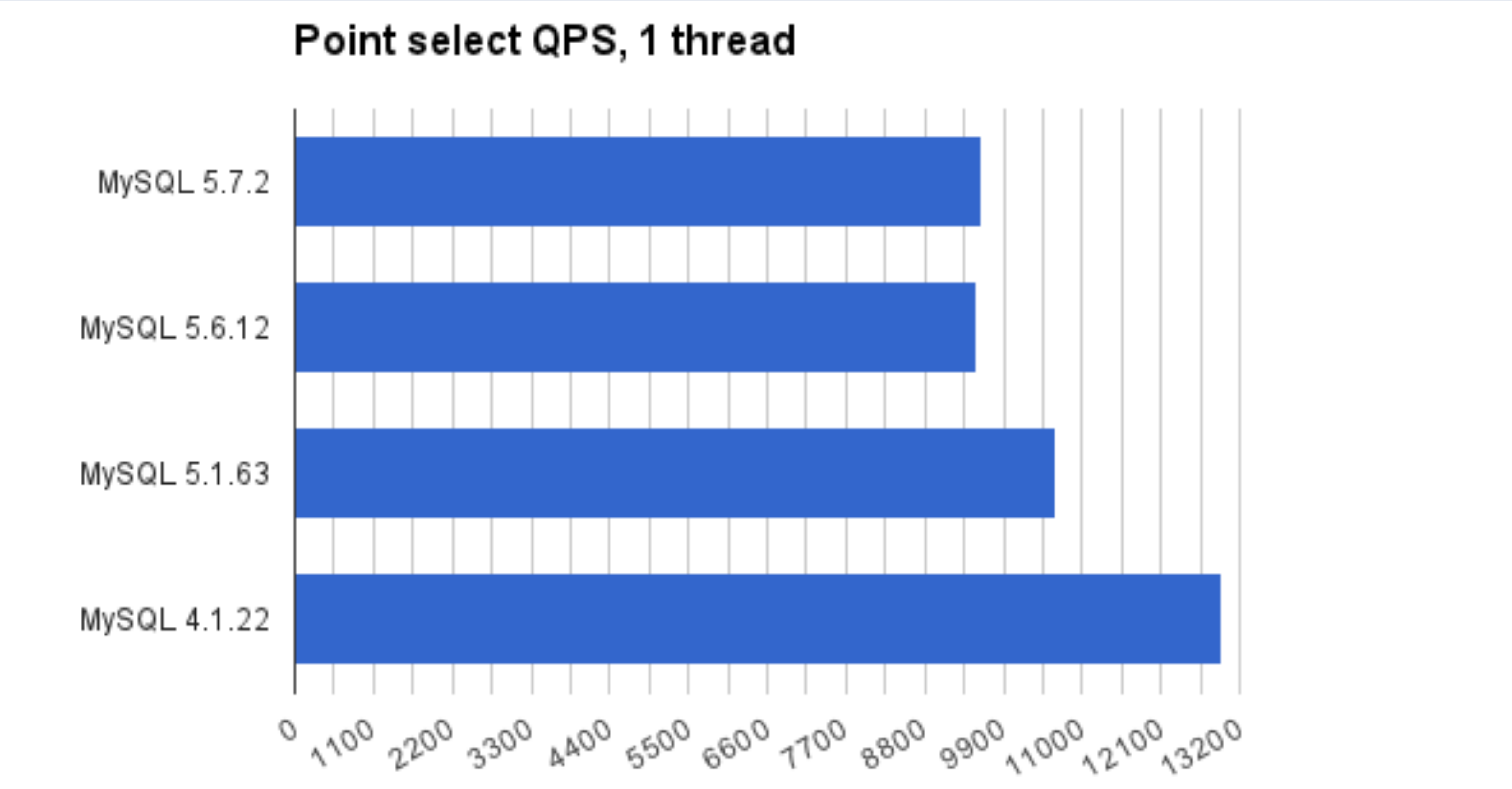
2015

- Typical server
  - Many-core, 144G RAM, 10k IOPs
- Many problems fixed
- 1M QPS on benchmarks

MySQL queries/second on modern HW



Peak QPS for 1 thread, regressions!



MySQL @ Facebook today

Workloads

- Social graph OLTP
- Messaging
- Other OLTP
- Analytics on read-write data

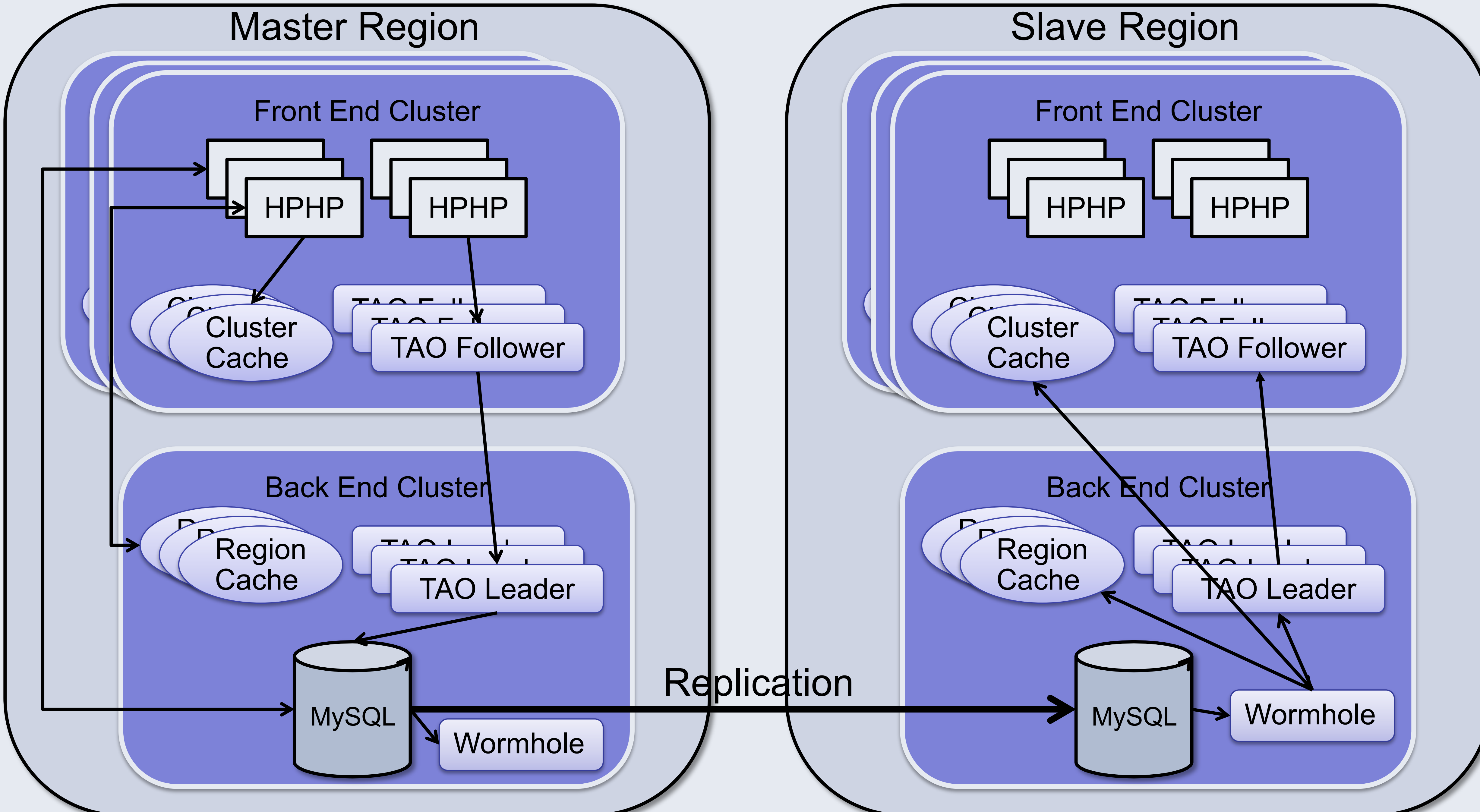
Peak per-second rates

- 175M QPS
- 12B rows read
- 65M rows updated
- 140M pages read
- 45M pages written
- 5ms response time

Social graph OLTP workload

- Dominated (50%) by short range scans
- Indexes: covering and clustered
- Joins are infrequent
- Online schema change is frequent
- Common transaction is 3 or 4 statements
- For more see http://github.com/facebook/linkbench
- Long-running queries rare except for backup, ETL

Social graph OLTP deployment



Messaging at Facebook

Requirements

- User facing
- Many nines availability
  - Auto failover in < 30 seconds
- Response time
  - Single digit milliseconds
  - Low variation

What we get from MySQL

- High availability
  - Fast & automated failover
  - 2-safe commit log
- Reliable performance
  - 70k updates/second/instance

Incremental work adds up

Engineering

- OLTP compression
- Online defragmentation
- 2-safe commit log
- Fix stalls

Operations

- MPS
- Online schema change
- Fast and automated failover
- Backup

Work in progress

- Document datatype
- RocksDB storage engine
- Faster failover
- Incremental backup
- Storage efficiency
- Self-service: sharding, resource limits

Document datatype

- Efficient for developers
  - Less schema, not schema-less
  - Enhance SQL for document semantics
- Efficient for the network
  - Only return requested attributes
- Efficient for storage
  - Avoid storing attribute names in each document
  - Avoid rewriting huge document after a small change

RocksDB for OLTP

- Derived from LevelDB
  - Goal is server-quality log structured merge tree (LSM tree)
  - 2X better compression than InnoDB, ½ the bytes written rate
- Added many features
  - Compaction - multi-threaded, size-tiered
  - Utilities
  - Merge operator
  - Column families

Storage efficiency

- Read, Write & Space Amplification
- Device + Workload
- Tiered Storage

Amplification factors

- Read amplification
  - physical-reads / query
- Write amplification
  - writes-to-storage / ingest
- Space amplification
  - sizeof(database) / sizeof(my-data)

Cannot be great at all three:

- B-Tree – read
- LSM – write & space

B-Tree vs LSM

Workload	Engine
Short range queries	B-Tree
Point queries, keys exist	B-Tree
Point queries, keys don’t exist	LSM
Compression	LSM
Bytes written	LSM
Blind writes	LSM
Other	B-Tree

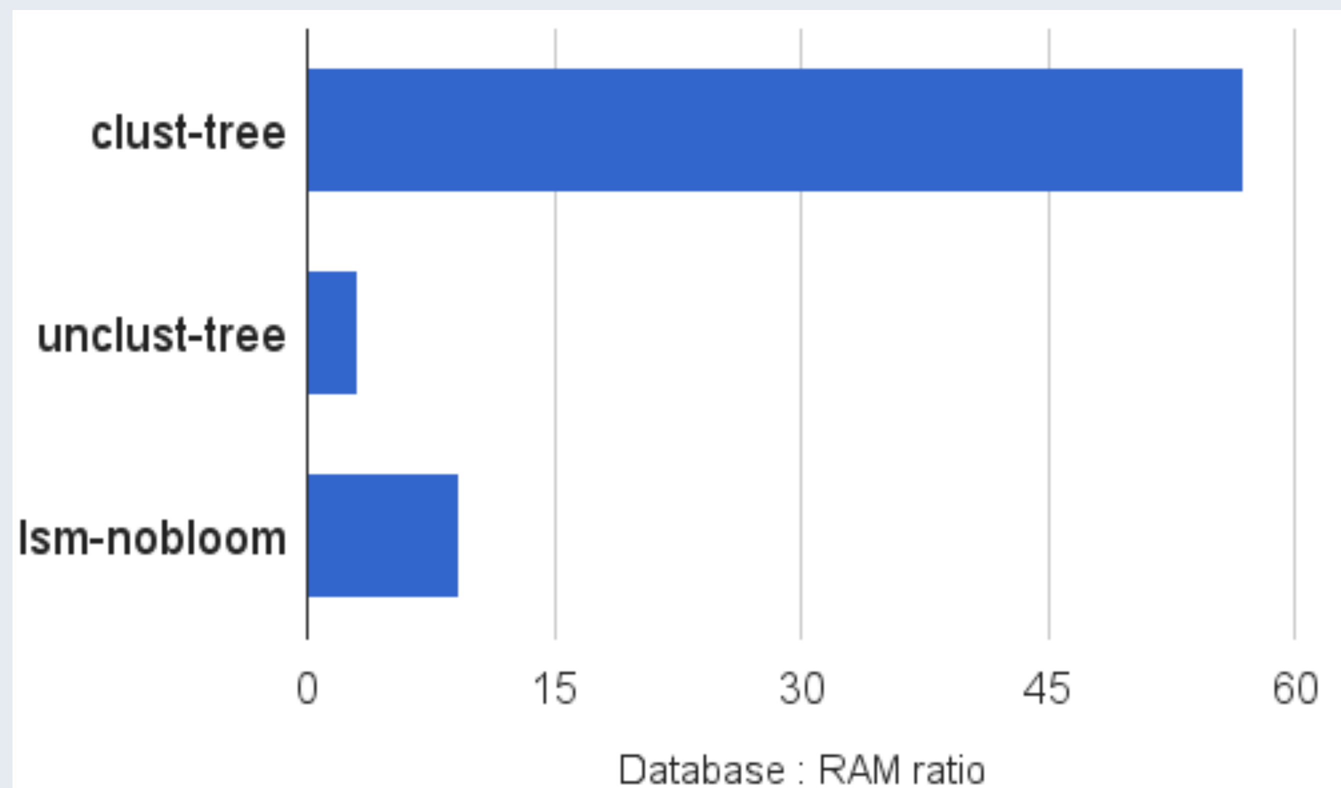
Problems:

- Match algorithm to workload
- But workload might need different algorithm per index

Which algorithm has the largest ratio of database : RAM for point queries with at most 1 disk read/query?

- Clustered tree – cache needs pointer per block
- Unclustered tree – cache needs pointer per row
- LSM – all data above max level, block index for max level in RAM

Assume:
- 16kb pages
- 16 byte key
- 128 byte row



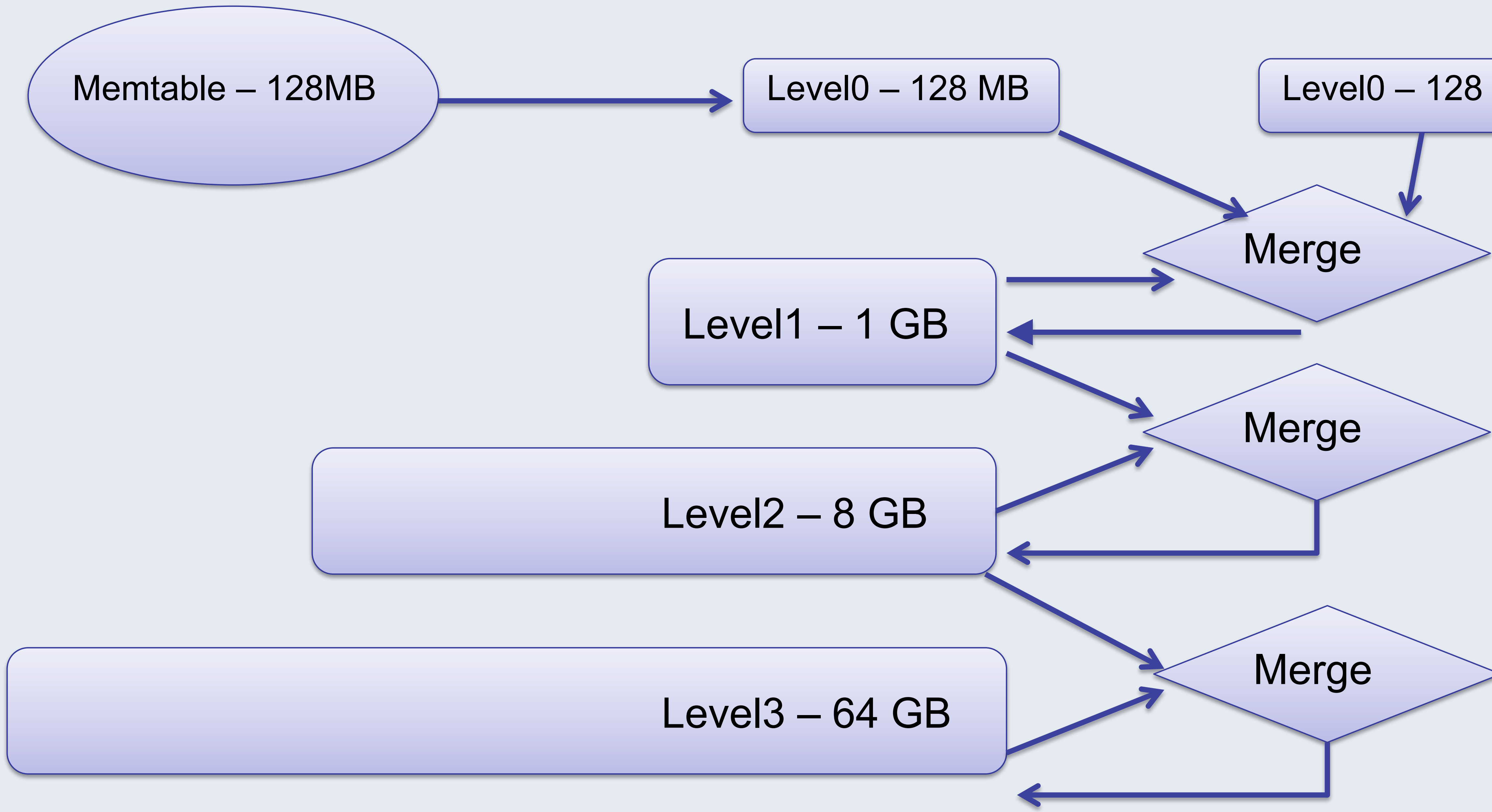
Match workload to device

Read	Write	Device
Frequent	Frequent	Higher endurance SSD
Frequent	Infrequent	Lower endurance SSD
Infrequent	Frequent	Disk
Infrequent	Infrequent	Disk

Problems:

- Determine the access pattern per index
- Handle access patterns that change

IO pattern for RocksDB



LSM IO patterns

Levels	Reads	Writes	Device
Upper/smaller	None	Streaming	Disk
Lower/larger	Streaming	Streaming	Disk
Lower/larger	Streaming, Random	Streaming	SSD or disk+cache

What am I buying?

Device	Capacity / Cost	R-MB/s / Cost	W-MB/s / Cost	R-IOPs / Cost
Disk	1.000	300.00	150.00	280
LE-SSD	0.179	98.21	0.63	16071
HE-SSD	0.042	112.50	6.74	18750

Numbers from retail prices and public specifications for 3 best in class devices

- All devices are 1 TB
- Disk is 2 SATA disks with RAID-10
- Cost for disk is 1, cost for other devices is relative to disk
- Values normalized by device cost
- W-MB/s is the rate that can be sustained for 3 years, not peak

Tiered storage deployments

Flashcache for MySQL

- HE-SSD & disk
- SSD
  - Write-back cache and read cache
  - Logical backup skips cache

Readcache for RocksDB

- Disk for database files
- SSD
  - Read cache for user threads
  - Compaction reads skip cache

No-readcache for RocksDB

- LE-SSD for larger levels
- Disk for smaller levels

Planning for performance

- Workload per-index
  - MB/s blind-write, point-read QPS, range-read QPS
- Algorithm changes workload cost
  - B-Tree for range-read heavy, LSM for write-heavy
- Hardware has physical constraints
  - Size, write-endurance, IOPs

One size doesn’t fit all, need multiple algorithms and multiple storage devices in one solution.

Final advice

- Rewrite or incremental?
  - Either but with quality, it is all about the constant factors
- Find your must have feature
  - Mine is storage efficiency
- Spend a few years near production deployments

Thank you and keep in touch

- facebook.com/MySQLatFacebook
- RocksDB.org
- SmallDatum.blogspot.com
- twitter.com/MarkCallaghan

facebook

How to ask questions

More useful

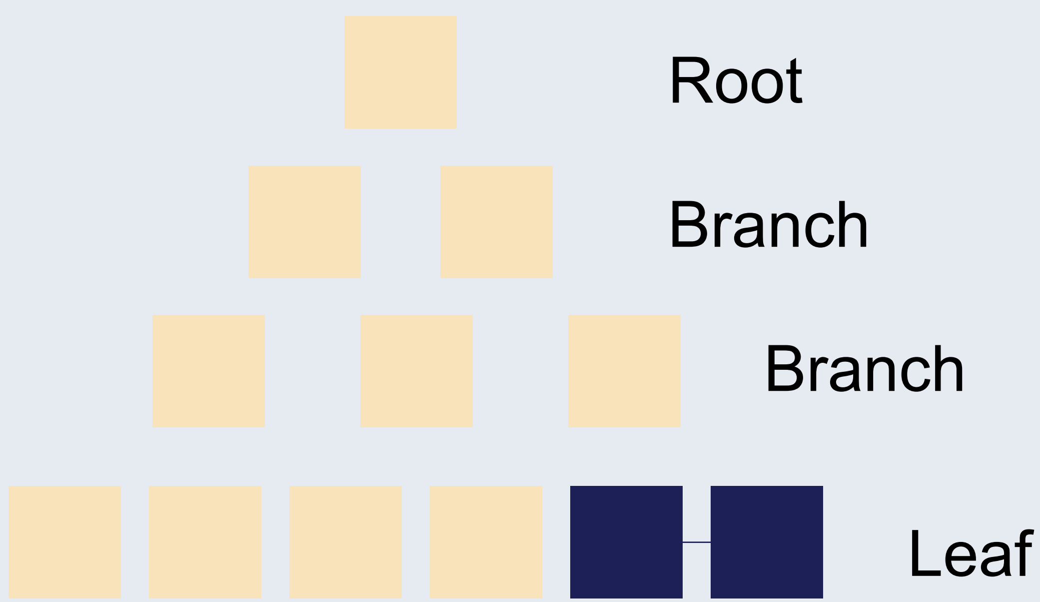
- Why do you use X?
- What would it take for you to migrate from X?

Less useful

- Have you considered using Y?
- Why don’t you use Y?
- You should be using Y!

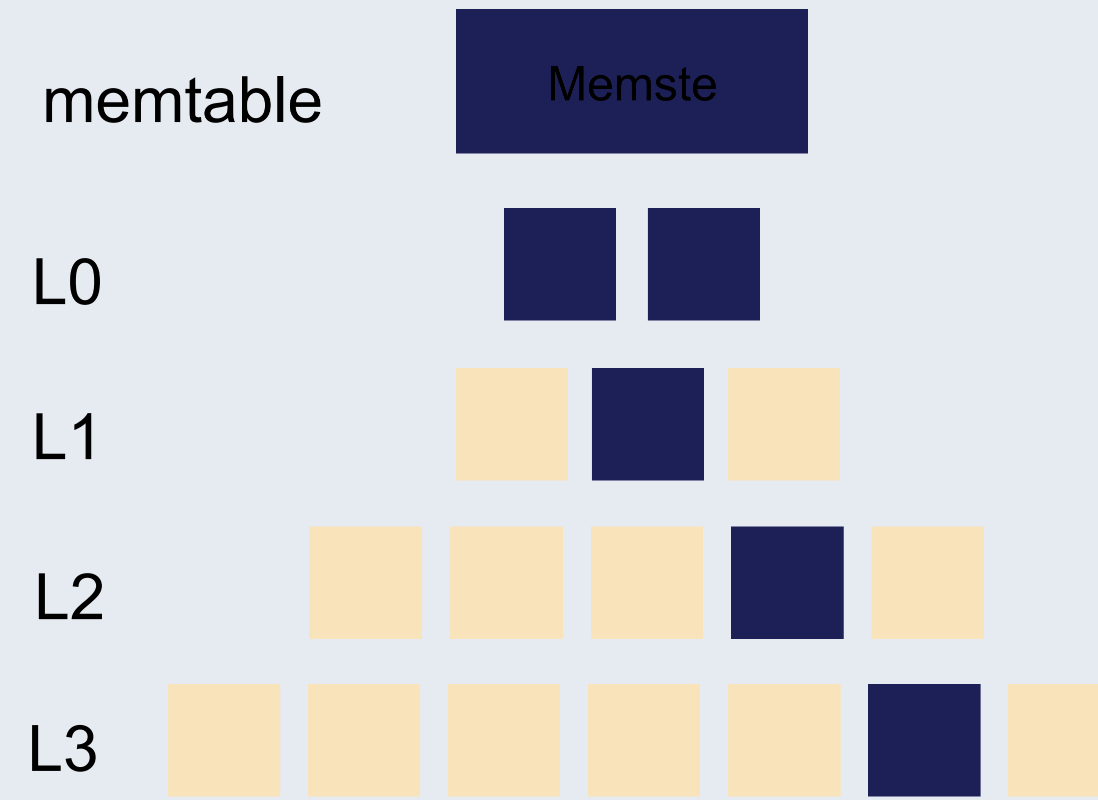
Read Penalty in RocksDB

InnoDB



Typical query is from root to leaf and reads one or two leaf nodes. Branch nodes are in cache.

RocksDB



Check all L0 files, then one file from each level. Largest level is not in cache.

PMP – world's best stall debugger

```
echo "set pagination 0" > /tmp/pmpgdb

echo "thread apply all bt" >> /tmp/pmpgdb

mpid=$( pidof mysqld )

t=$( date +'%y%m%d_%H%M%S' )

gdb --command /tmp/pmpgdb  --batch -p $mpid | grep -v 'New Thread' > f.$t

cat f.$t | awk 'BEGIN { s = ""; }  /Thread/ { print s; s = ""; } /^\#/
  { x=index($2, "0x"); if (x == 1) { n=$4 } else { n=$2 }; if (s != "" ) { s = s
  "," n} else { s = n } } END { print s }' -  | sort | uniq -c | sort -r -n -k 1,1
  > h.$t
```