MySQL for the Web

Some experience of hacking/extending MySQL

汪 源
网易.杭州研究院.副院长
@网易汪源

SACC2012

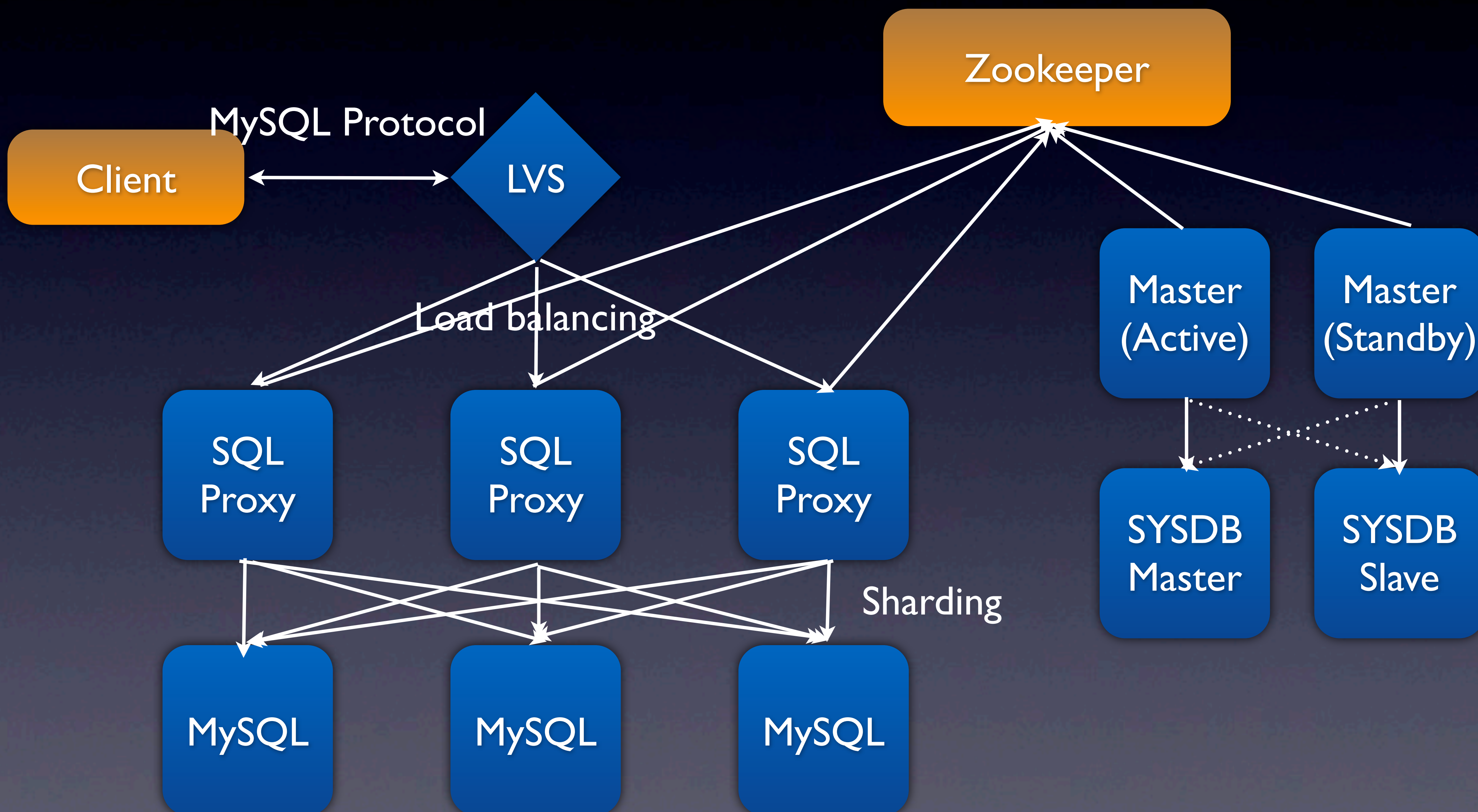
MySQL Projects@NetEase

- Distributed RDBMS: based on MySQL
- Customized storage engine: Transactional or Non-transactional
- Open source MySQL branch: InnoSQL

Outline

- Scale out MySQL
- Consistent Memcached integration
- Layered approach for storage engine design
- Scalable RW lock and intention lock
- Dynamic schema
- Tailoring row level cache
- Flash cache in InnoSQL

Architecture



Sharding

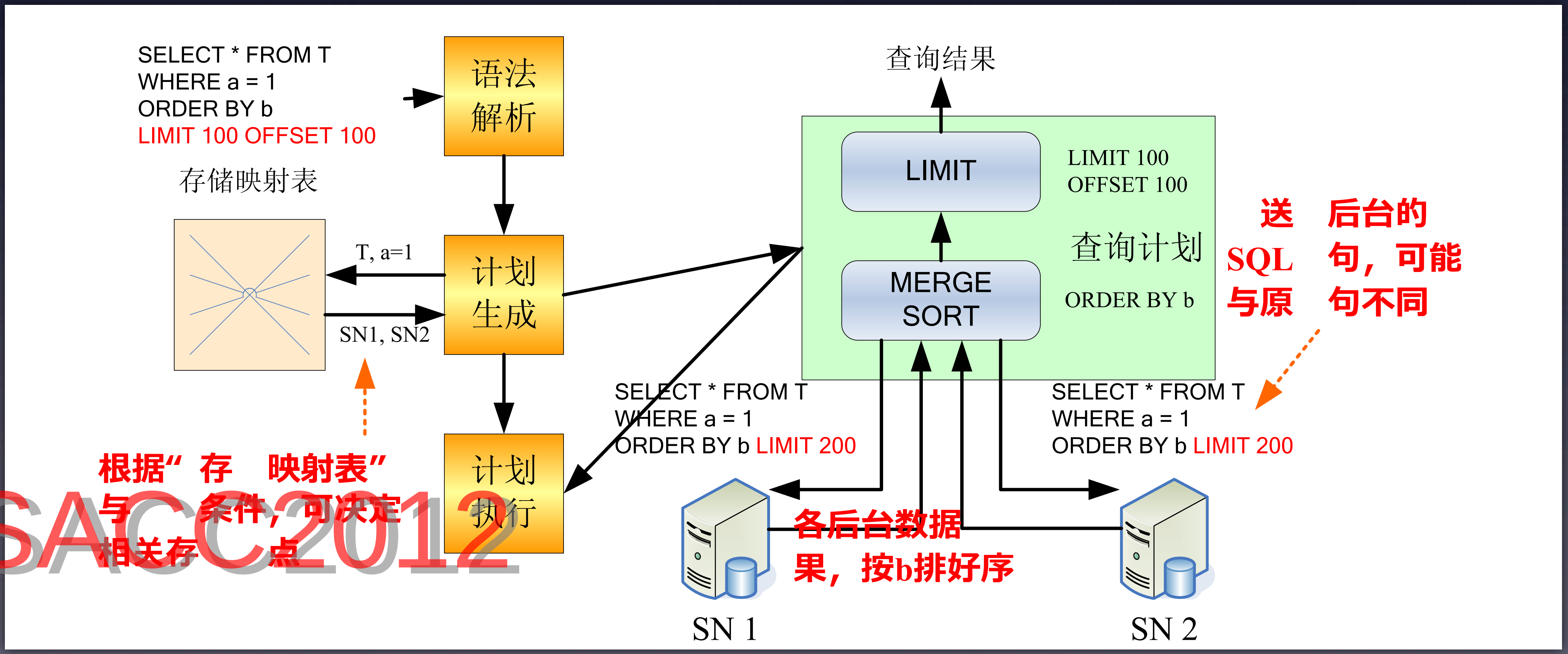
- Sharding methods
  - Based on one or multiple columns
  - Hashing or UDF
  - Mapping cached on every SQL proxy server
- Policy
  - Several table can use same sharding policy
    - FK reference is common
  - EQUI-JOIN on sharding column on tables belongs to the same policy becomes local join.

Scalability

- Almost online, sub minute of downtime
- Based on MySQL replication and query rewriting on SQL proxy
- Scale out procedure
  - Make two slaves, A1 and A2, for MySQL server A
  - Delete unneeded data on A1 and A2
  - Waiting for A1 and A2 to catch up
  - Block access to A
  - Waiting for A1 and A2 to catch up
  - Start rewriting on SQL proxies, adding some conditions to filter out unneeded data on A1 and A2
  - Switch to A1 and A2
  - Delete unneeded data on A1 and A2 again
  - Stop rewriting on SQL proxies

Distributed queries

- Support: distributed GROUPBY/AGG/HAVING, ORDER BY, LIMIT/OFFSET, EQUI-JOIN
- Not support: subquery



Distributed transactions(1)

- We use 2PC in production for many years without complaints
- Distributed transactions are rare in execution(3%) but common in code(30%)
- SQL proxy server as coordinator
  - Logging 2PC decisions
- If SQL proxy server fails and can not come back soon (say, after 10 minutes), Master will rollback XA transactions blindly
- Limited support for XA transactions of MySQL
  - Rollback PREPARED transactions after client disconnect or safe shutdown
    - We fix it
  - Missing PREPARED transactions in binlog after crash
    - Not fixed

Distributed transactions(2)

- Can not get consistent global snapshot
  - Scenario
    - T1: read MySQL server 1
    - T2: update MySQL server 1 and MySQL server 2, PREPARE and COMMIT
    - T1: read MySQL server 2
  - However, nobody complains
- Future plan
  - Put XA logs in high available shared storage
  - consistent global snapshot is hard
    - 2PL is not acceptable

High availability

- SYSDB
  - Based on replication or DRBD and Linux-HA
- Master
  - Master is stateless, all state is in SYSDB (cache state in memory)
  - Two masters compete on Zookeeper lease
  - New master read all state from SYSDB before going to service
- SQL proxy server
  - LVS load balancing
- MySQL
  - Same as SYSDB
- Issues
  - XA logs on SQL proxy server is not HA
  - DRBD and Linux-HA is overkill but replication is not safe.

Other interesting findings

- Read replica is useless
- Transactions are more important than expected
  - Although distributed transactions and queries are not always ACID

Outline

- Scale out MySQL
- Consistent Memcached integration
- Layered approach for storage engine design
- Scalable RW lock and intention lock
- Dynamic schema
- Tailoring row level cache
- Flash cache in InnoSQL

Typical usage of Memcached

- Query
  - Search Memcached(GET)
  - If miss
    - Fetch from DBMS
    - Put into Memcached(SET/ADD)
- Update
  - Method 1
    - Remove from Memcached(DELETE)
    - Update database
  - Method 2
    - Update database
    - UpdateMemcached(SET)

Inconsistency is easy

- Suppose there is an object O(k:v1), not in Memcached initially. And suppose there are transactions T1(get O) and T2(update O) acting as follows:
  - T1: searches Memcached for O, miss
  - T1: reads O from DBMS, got(k:v1)
  - T2: deletes O from DBMS, miss
  - T1: puts O(k:v1) into Memcached
  - T2: updates O in DBMS to (k:v2)
- DBMS left with O(k:v2) and Memcached with O(k:v1), inconsistency

Seems better

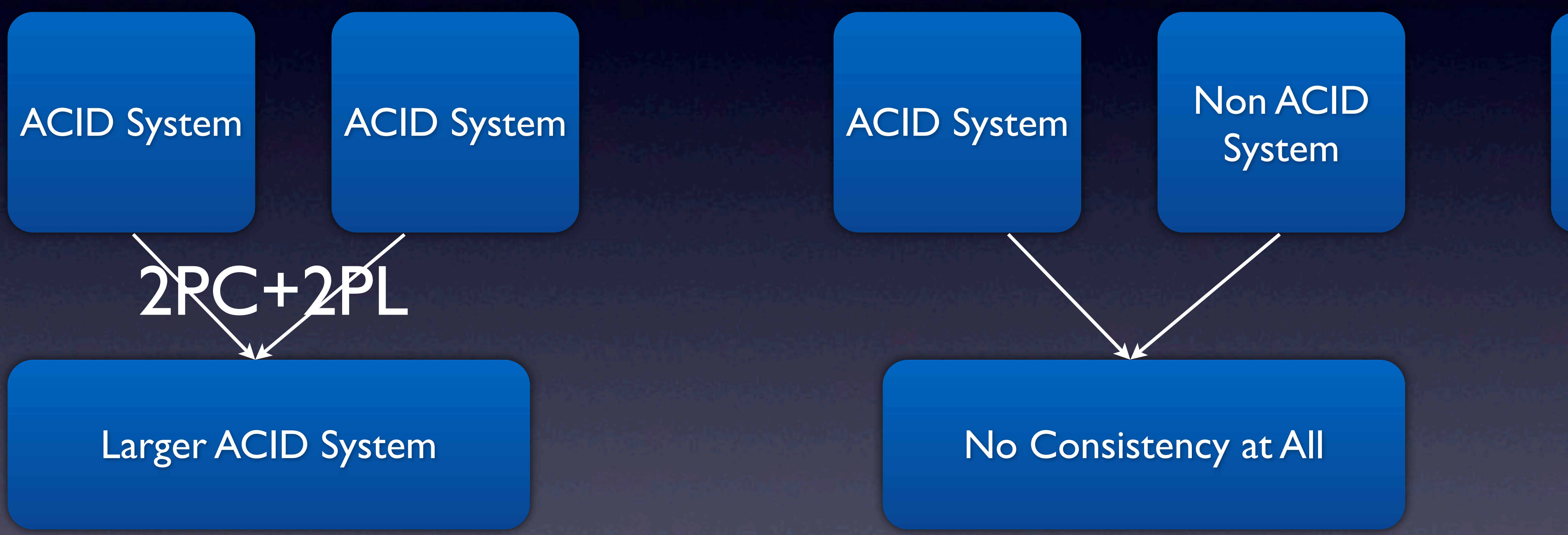
- SET in update, ADD in query
- T2 follows T1
  - T1: searches Memcached for O, miss
  - T1: reads O(k:v1) from DBMS and ADD into Memcached
  - T2: updates DBMS to (k:v2)
  - T2: SET Memcached to (k:v2)
- T1 follows T2
  - T1: searches Memcached for O, miss. Read O(k:v1) from DBMS
  - T2: updates DBMS to (k:v2)
  - T2: SET Memcached to (k:v2)
  - T1: tries to ADD O(k:v1) to Memcached, skip for already exist
- DBMS and Memcached converge to (k:v2) in both scenarios.

Inconsistency still possible

- Inconsistency scenarios
  - Updated item (k:v2) of T2 is replaced, then T1's ADD(k:v1) will succeed
  - If T2 commits after updating Memcached, then:
    - If T2 aborts with failure, DBMS rollback but not Memcached
  - If T2 commits before updating Memcached, then:
    - If failure before updating Memcached, Memcached will not get update
    - Two concurrent update transactions leads to inconsistency
      - T2: updates DBMS to (k:v2) and commit
      - T3: updates DBMS to (k:v3) and commit
      - T3: SET Memcached to (k:v3)
      - T2: SET Memcached to (k:v2)
- Anyway, SET in update, ADD in query, update Memcached after update DBMS but before commit is much safer.

Is strict consistency possible?

Traditional wisdom



A consistent protocol

- Query
  - GET from Memcached, return if hit a normal row.
  - ADD LOCK row with value 0 into Memcached and GETS the version
  - Read DBMS
  - CAS the LOCK to the record if GETS got a LOCK row with value 0
- Update
  - Set a lock, repeating
    - GETS from Memcached
    - If miss, ADD LOCK row with value 1 into Memcached, end repeating if succeed
    - If got a normal row, CAS to LOCK row with value 1, end repeating if succeed
    - If got a LOCK row, CAS to LOCK row with value old value +1, end repeating if succeed
  - Update DBMS and commit
  - DECR the lock row

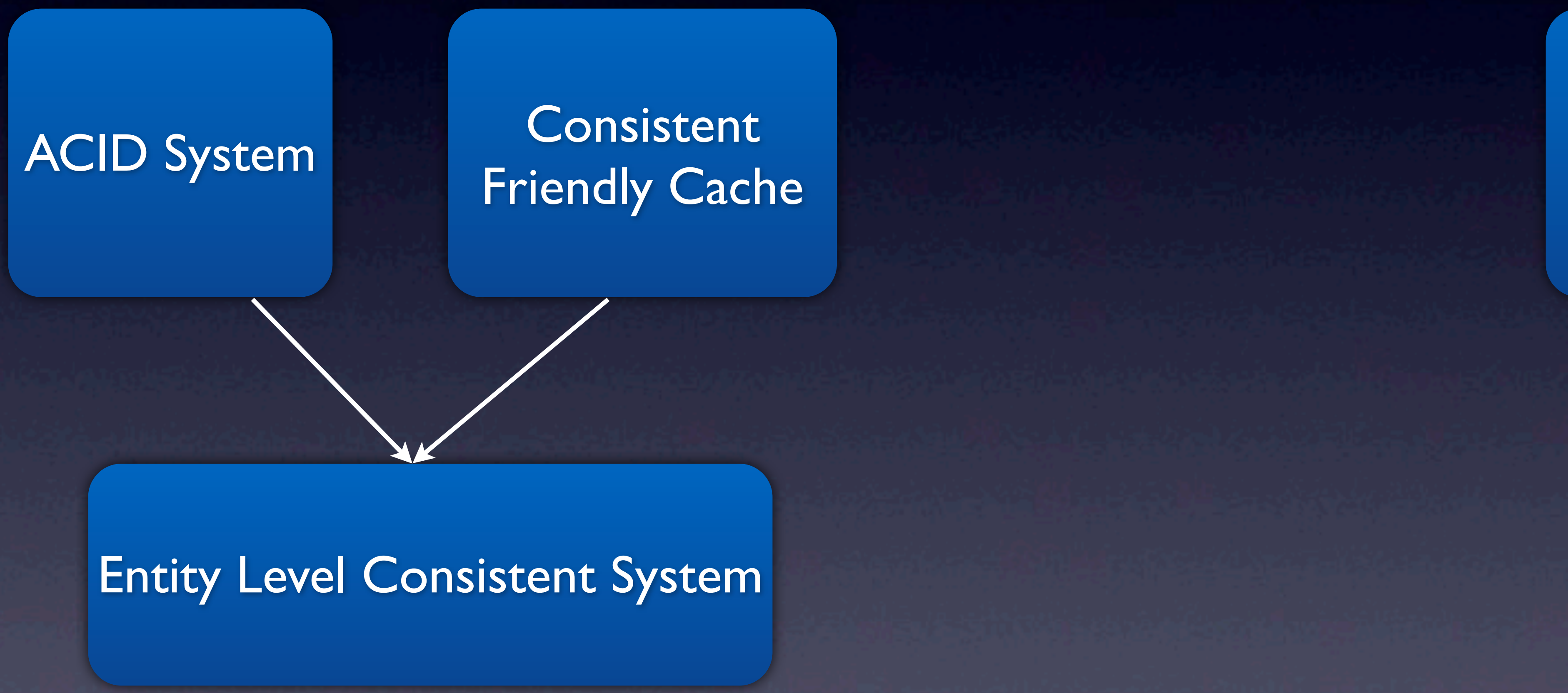
Protocol cont.

- How consistency is guaranteed
  - LOCK row is a hint that someone is updating the DBMS, value of the LOCK row is the number of clients that are updating
  - Failsafe
    - Query fails, nothing happens
    - Update fails, remaining LOCK count will prevent loading into Memcached, some performance lost but no consistency
      - LOCK row can have a modest expiration time
- Entity level consistency only

Consistent friendly cache

- Operation requirements
  - GET
  - GETS
  - ADD
  - CAS
  - DECR:
- Other requirements
  - Version number can not go back even after crash
  - LOCK rows can not be swapped out
  - Can not restart Memcached too fast, must wait for existing update complete

New wisdom



ACID System
Consistent Friendly Cache
Entity Level Consistent System

Outline

- Scale out MySQL
- Consistent Memcached integration
- Layered approach for storage engine design
- Scalable RW lock and intention lock
- Dynamic schema
- Tailoring row level cache
- Flash cache in InnoSQL

Traditional monolithic design

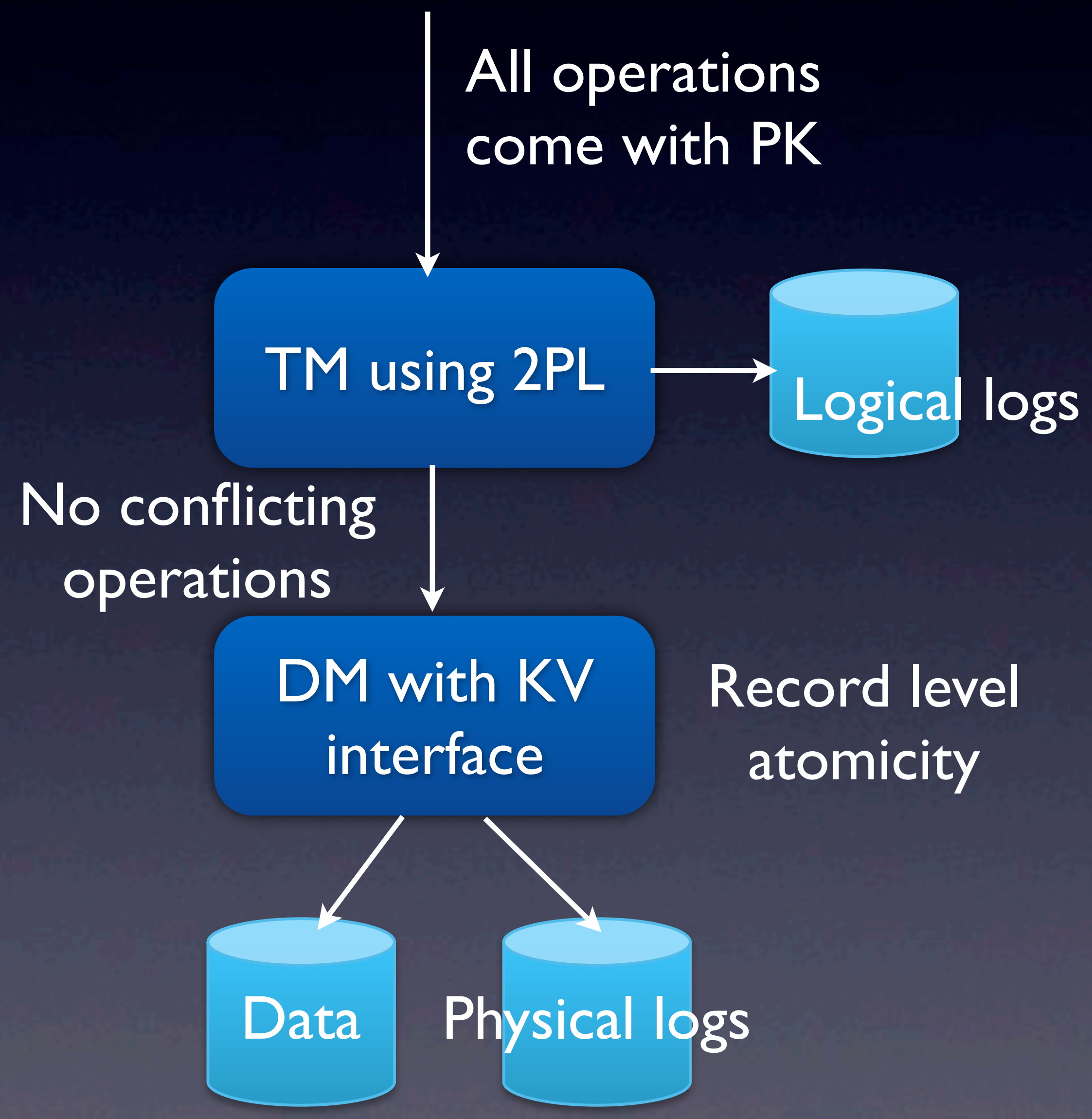
- Transaction manager(TM) and data manager(DM) are tightly coupled
  - Locking is based on physical RID
  - Locking is done in data management
    - Holding the latch and trylock
    - If fails, unlatch, lock and recheck
  - Versioning info is embedded in physical records and pages
  - Logging is done while holding latch and recovery in based on physical info (pageLSN)

Why decouple TM and DM

- For flexibility
  - Transactional and non-transactional in the same storage engine
  - Dynamic shifting between transactional and non-transactional
- For performance
  - Want multi versioning but don't want versioning info overhead
  - Versioning info should be only in memory for active transactions
- For a new way to database design
  - Scalable transaction processing on HBase (HBase as DM)?

How to decouple?
Simple but bad way

- No secondary index scan
- No snapshot isolation



The TNT way

- Multi-versioning. However version info only in memory
  - Record level version info for heap record
  - Page level version for index. Low overhead but achieves coverage index scan most of the time(same as InnoDB)
  - No version info for BLOB, use Copy-on-write
- Operations
  - INSERT goes to DM directly but put version info in memory
  - UPDATE/DELETE goes to memory
- Most recent record version in memory, but older versions in version pool
- Purge committed modification to DM periodically

Benefits of the TNT way

- Can do secondary index scan
- Can do coverage index scan
- Minimal multi-versioning overhead
  - Page level version info for index
  - No version info in DM
- Low memory consumption
  - Only most recent version must be kept in memory
  - INSERT goes to DM directly, only version info in memory
  - Unmodified index has no new versions

Compared to Falcon

- Falcon
  - Recognized to be the future of MySQL storage engine but failed
  - Designed by famous database guru, Jim Starkey, father of InterBase(the first RDBMS supporting multi-versioning)
  - Multi-versions in memory and single version on disk, same as TNT
- Major problems of Falcon
  - Every unmerged transaction has his own modified index, so read has to merge lots of tiny indice
  - Can not do coverage index scan
  - New records go to memory, so memory comsuption is higher than TNT

Preliminary performance result

New wisdom

- Efficient general transaction processing can be built upon an entity level consistent DM with little overhead
  - With secondary index scan
  - With coverage index scan
  - With snapshot isolation

Outline

- Scale out MySQL
- Consistent Memcached integration
- Layered approach for storage engine design
- Scalable RW lock and intention lock
- Dynamic schema
- Tailoring row level cache
- Flash cache in InnoSQL

Locks in storage engine

- RWLock
  - Catalog lock: lock the catalog to search for the table structure used in query
  - Table definition lock: lock the table definition to prevent DDLs
  - Row lock: lock the rows touched by the transaction
  - Page latch: lock the pages touched by the transaction
- Table level intention lock
  - Common multi-level lock strategy in DBMS
    - IS: Typical SELECTs
    - IX: Typical INSERT/UPDATE/DELETEs
    - SIX/S/X: Lock upgrade for big transactions

No scalability bottleneck?

- Row lock and page latch: There are many of them, so conflicts will not be too often
- Catalog lock, table definition lock and intention lock: They don't conflict for typical transaction processing, so no contentions at all
- Wrong. Those don't conflict logically could be scalability bottleneck in practice

Scalability of reader lock

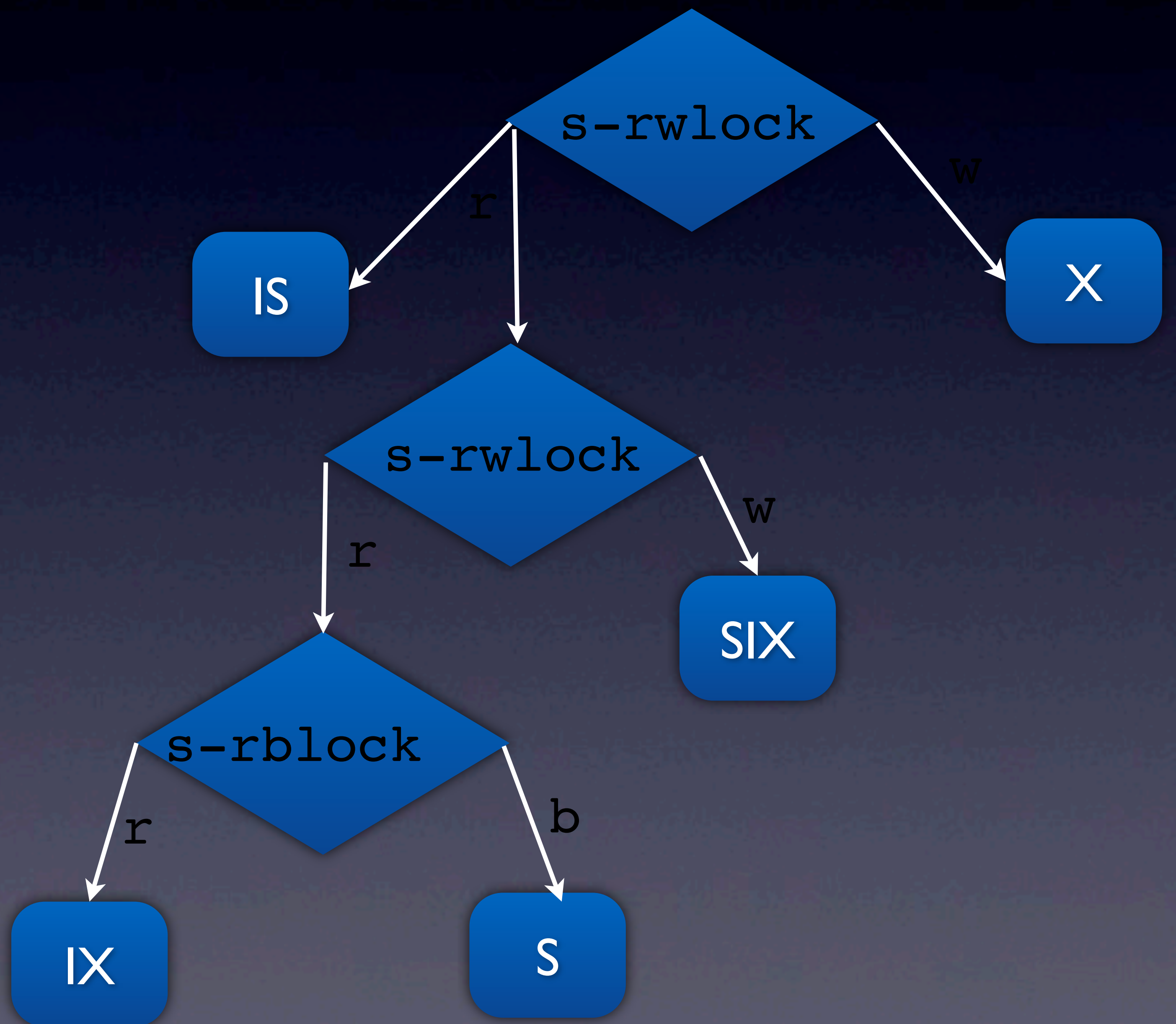
- Intel Xeon E5645*2, 12 Cores, 24 Threads
- Optimized RWLock with single CAS instruction to acquire reader lock
- Throughput drops sharply from single thread to multi-thread
  - Due to frequent cache invalidation

Scalable Rwlock

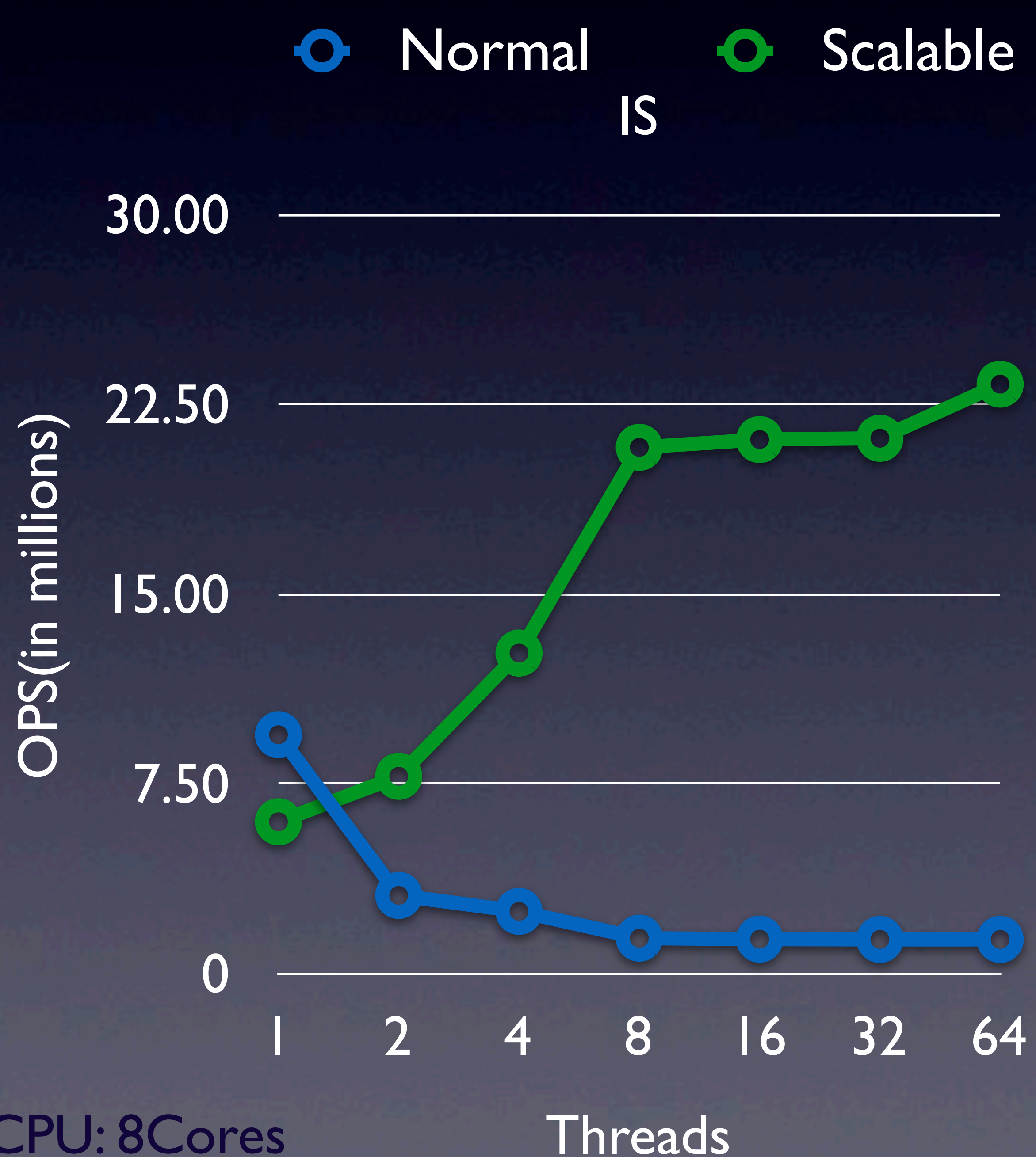
- Scalable Rwlock := A collection or normal RWLocks
  - Collection size is number of CPU cores in typical setting
  - Each normal RWLock is in his own cache line
- rdlock
  - Got corresponding normal RWLock
  - Acquire reader lock for that lock
- wrlock
  - Acquire all writer locks in the same order
- Scalable and fast for rdlock, slow and not scalable for wrlock

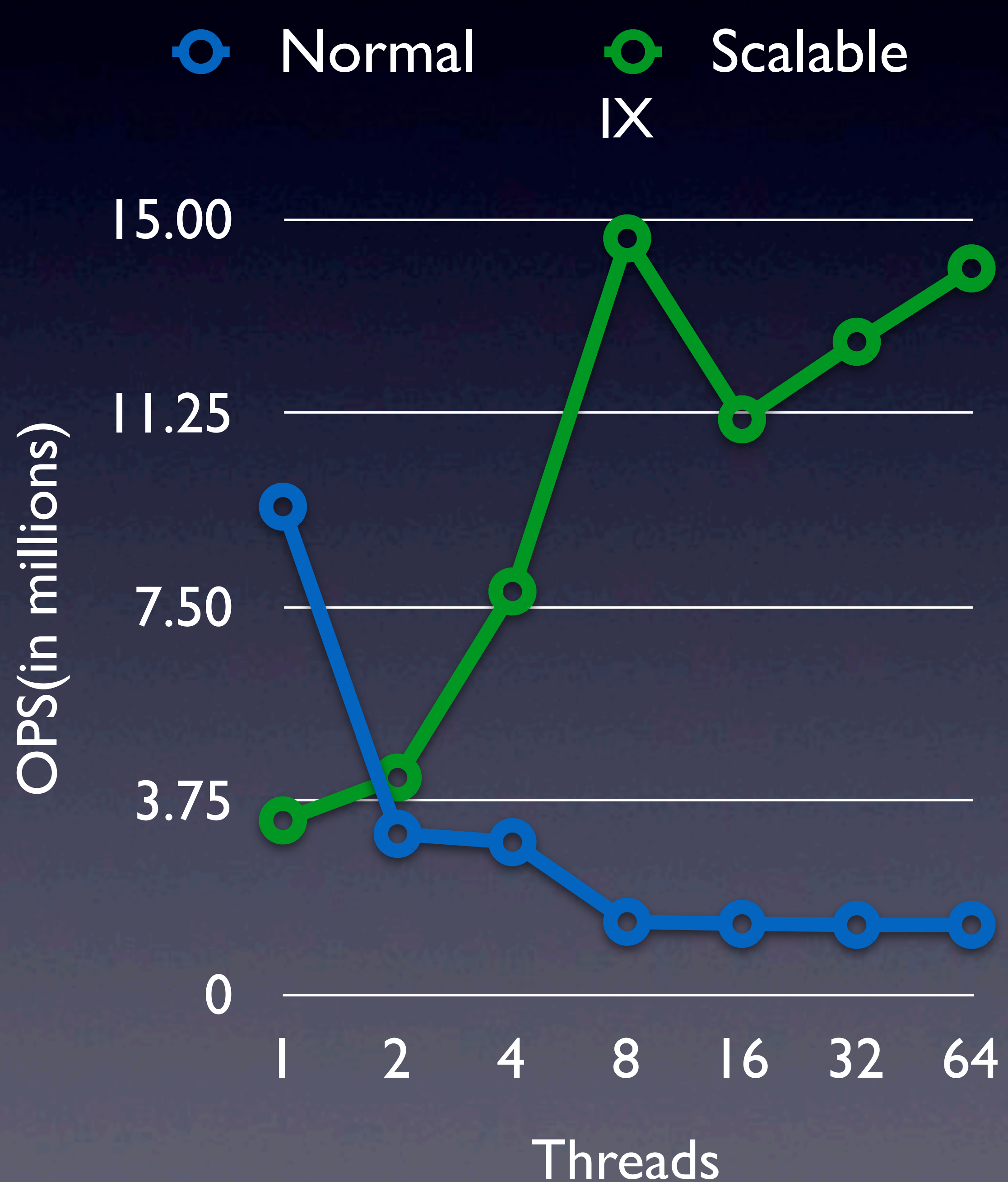
Scalable intention lock

- A hierarchy of scalable RWLock and RedBlackLock
- Scalable RedBlackLock
  - Red: Lock corresponding lock in red
  - Black: Lock all locks in black
- Scalable and fastest for IS, scalable and fast for IX, slow and not scalable for others



Benchmark result





CPU: 8Cores

Outline

- Scale out MySQL
- Consistent Memcached integration
- Layered approach for storage engine design
- Scalable RW lock and intention lock
- Dynamic schema
- Tailoring row level cache
- Flash cache in InnoSQL

Why dynamic schema?

- InnoDB becomes readonly during ADD/ REMOVE columns
- Online schema change can be done using replication or trigger, however you might have to wait for hours or even days
- NoSQL is cool. They have no schema and all headaches are gone
- Are RDBMS doomed? No!

How to do

- Quite simple
  - Modify metadata only when ADD/REMOVE columns
  - Valid column number in every record
  - Missing column is filled automatically using default value and removed column is skipped
- Why others didn't do this? Hard to understand

Best of the two worlds?

	Flexible schame	Consistency enforcement
Traditional RDBMS	No	Yes
NoSQL	Yes(Schemaless)	No
TNT/NTSE	Yes(Has schema, on the fly modification)	Yes

Outline

- Scale out MySQL
- Consistent Memcached integration
- Layered approach for storage engine design
- Scalable RW lock and intention lock
- Dynamic schema
- Tailoring row level cache
- Flash cache in InnoSQL

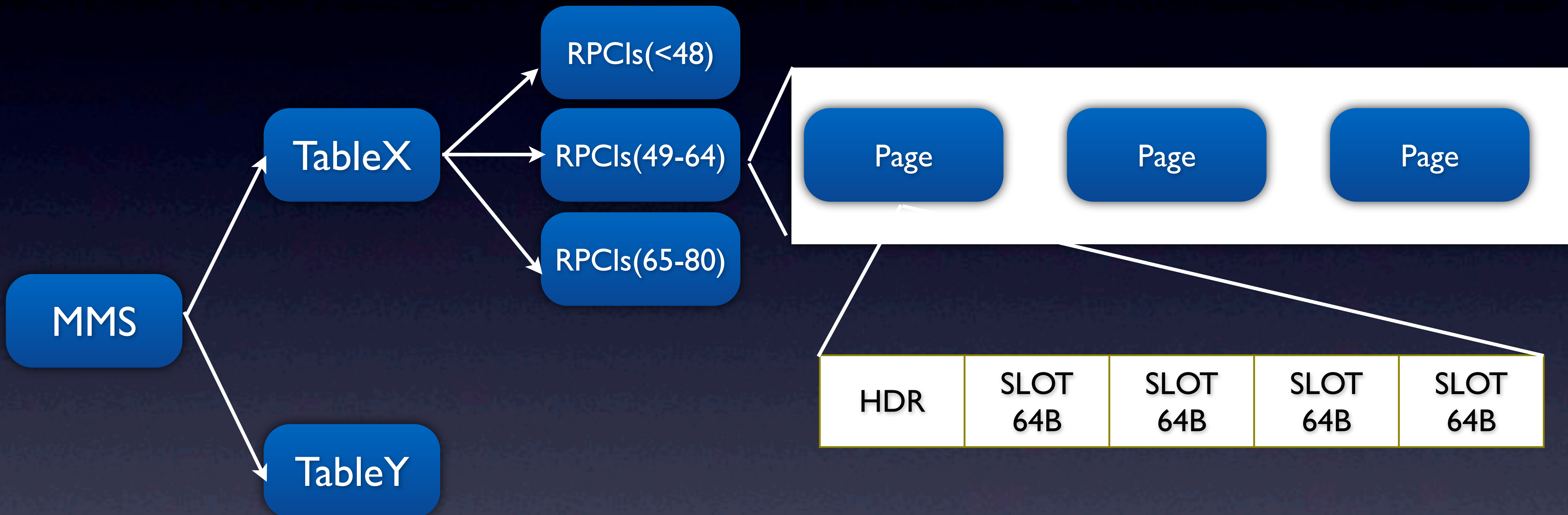
Why row cache?

- No well known database have row cache
- We have Memcached
- So why integrated row cache?
  - Consistency: Data consistency in Memcached can not be guaranteed in general
    - We only achieve entity level consistency with great effort
  - Productivity: Many codes for Memcached and error prone
  - Performance
    - No network round-trips for manipulating two datasets

Challenges

- Various object size
- Very small object size(10s-100s in Byte)
- Frequent updates
- Competes with page cache

For various object size(1)



- Memory management is nothing special, just a slab allocator.

For various object size(2)

- Replacement policy
  - Local row level replacement in same RPCIs
  - Global page level replacement
    - Minimal heap of FPage(access frequency of page). We can not use LRU list here.
    - FPage = (access frequency of hotest row in page + access frequency of coldest row in page)/2
    - access frequency of row = 1/(now() - atime)
- How to choose between these two replacement policy?
  - A background thread do page level replacement periodically and do row replacement in all other situations
  - No good measurement to justify the choice

For small object size

- Compact row level LRU
  - Standard way: Doubly linked list, 16 bytes per row is a huge overhead
  - NTSE's way: 2 bytes local LRU in page + minimal heap of page based on atime of coldest row in page => near 2 bytes global LRU
- Compact RID->MMSRecord mapping
  - Compact linear hash, 16 bytes/row

For frequent updates

- Can do writeback on updates is a huge advantage over Memcached
- However, a major problem: Lots of IO for flushing dirty records
- First try: Make random IO to sequential IO by sorting dirty records
  - This helps in small scale but not enough when row cache is 10s of GBs
- Second try: Dump dirty records to log if their corresponding pages are not in page cache
  - This solves the problem when row cache is 10s of GBs
- We don't know what will happen when row cache is > 100GB
- what goes around comes around(出来混总是要还的)

Coexistence with page cache

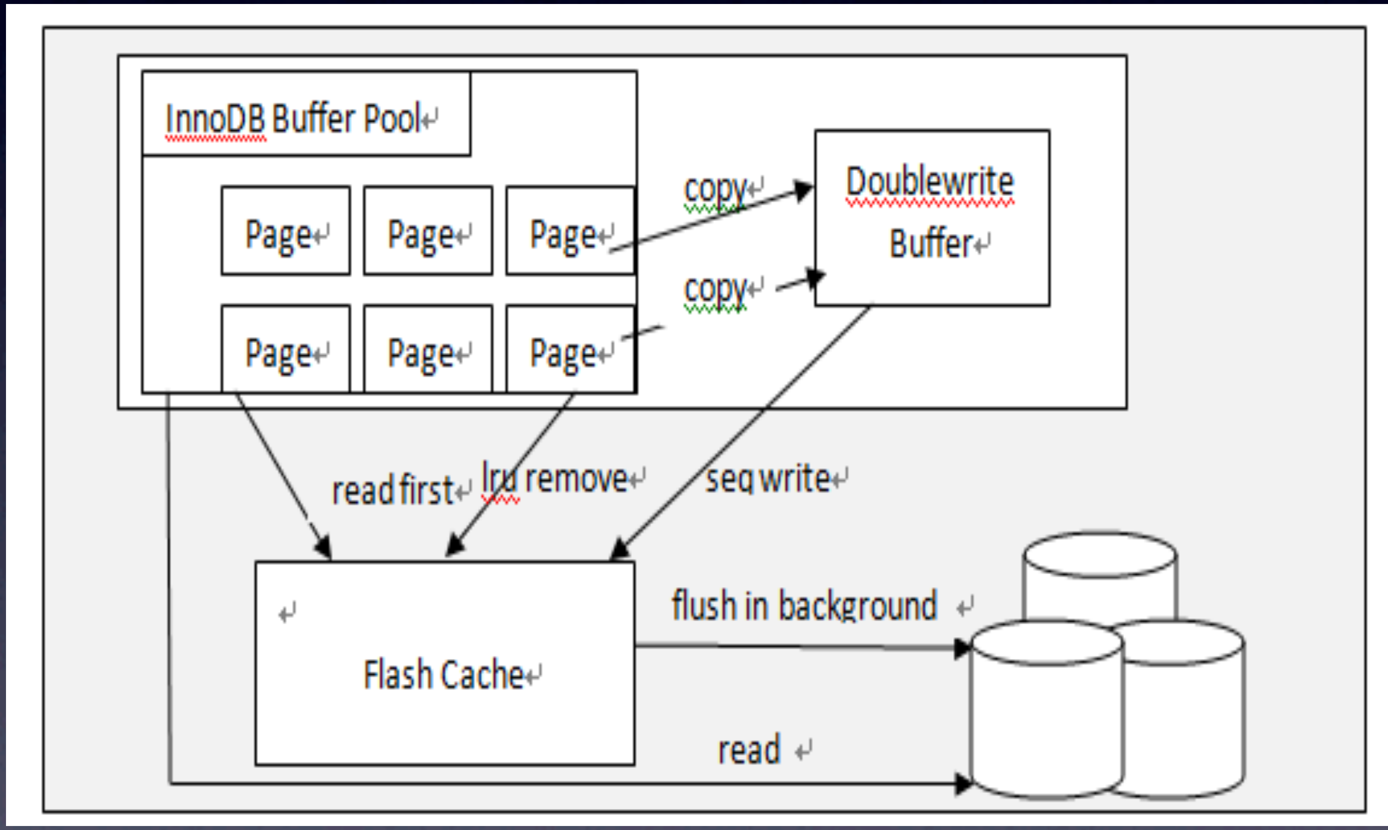
- A hard problem for DBA: How much memory for row cache and how much for page cache?
- InnoDB's DBA is happy: Just throw all memory to InnoDB's page buffer
- NTSE's way
  - Size of row cache + page cache is fixed
  - Size of row cache or page cache can be changed online
  - Lots of statistics for DBA to guess a good balance between row cache and page cache
  - Rule of thumb: 80% to row cache

Outline

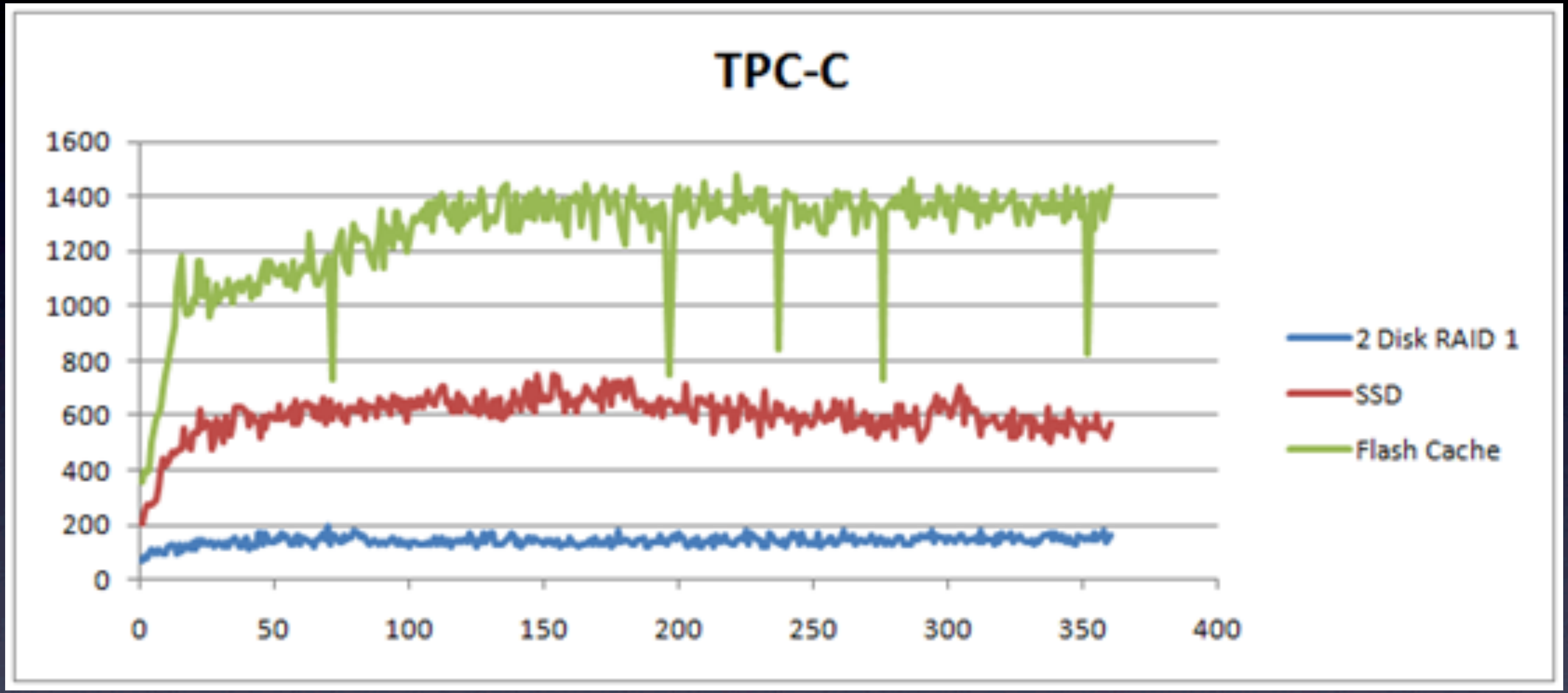
- Scale out MySQL
- Consistent Memcached integration
- Layered approach for storage engine design
- Scalable RW lock and intention lock
- Dynamic schema
- Tailoring row level cache
- Flash cache in InnoSQL

Architecture

- SSD as a (much) bigger doublewrite buffer
- read cache/write back
- no random write to SSD



Performance



- Even higher than on SSD
- And higher than Facebook's flashcache(not shown)

General vs. Specialized

- Why InnoSQL's FC is much more effective than general system level solutions?
  - Half write IOPS
    - No need to update original pages
  - No realtime mapping index update
    - space_id and page_offset at the header
  - No double caching in memory and SSD, caching clean pages after swapped out from memory