High Availability for Database Systems in Cloud Computing Environments

Ashraf Aboulnaga
University of Waterloo

University of
Waterloo
Database Group

Acknowledgments

- University of Waterloo
  - Prof. Kenneth Salem
  - Umar Farooq Minhas
  - Rui Liu (post-doctoral fellow)

- University of British Columbia
  - Prof. Andrew Warfield
  - Shriram Rajagopalan
  - Brendan Cully

Database Systems in the Cloud

- Using cloud technologies plus SQL database systems to build a scalable highly available database service in the cloud
- Better deployment of database systems in the cloud
- Better support for database systems by cloud technologies

Why Database Systems?

- Databases are important!
- A narrow interface to the user (SQL)
- Transactions offer well defined semantics for data access and update
- Well defined internal structures (e.g., buffer pool) and query execution operators (e.g., hash joins)
- Accurate performance models for query execution time and resource consumption

Outline

- Introduction
- **RemusDB:** Database high availability using virtualization
  Umar Farooq Minhas, Shriram Rajagopalan, Brendan Cully, Ashraf Aboulnaga, Kenneth Salem, and Andrew Warfield. "RemusDB: Transparent High Availability for Database Systems," In *Proceedings of the VLDB Endowment (PVLDB)*, 2011.
  (Best Paper Award)
- **DBECS:** Database high availability (and scalability) using eventually consistent cloud storage
- Conclusion

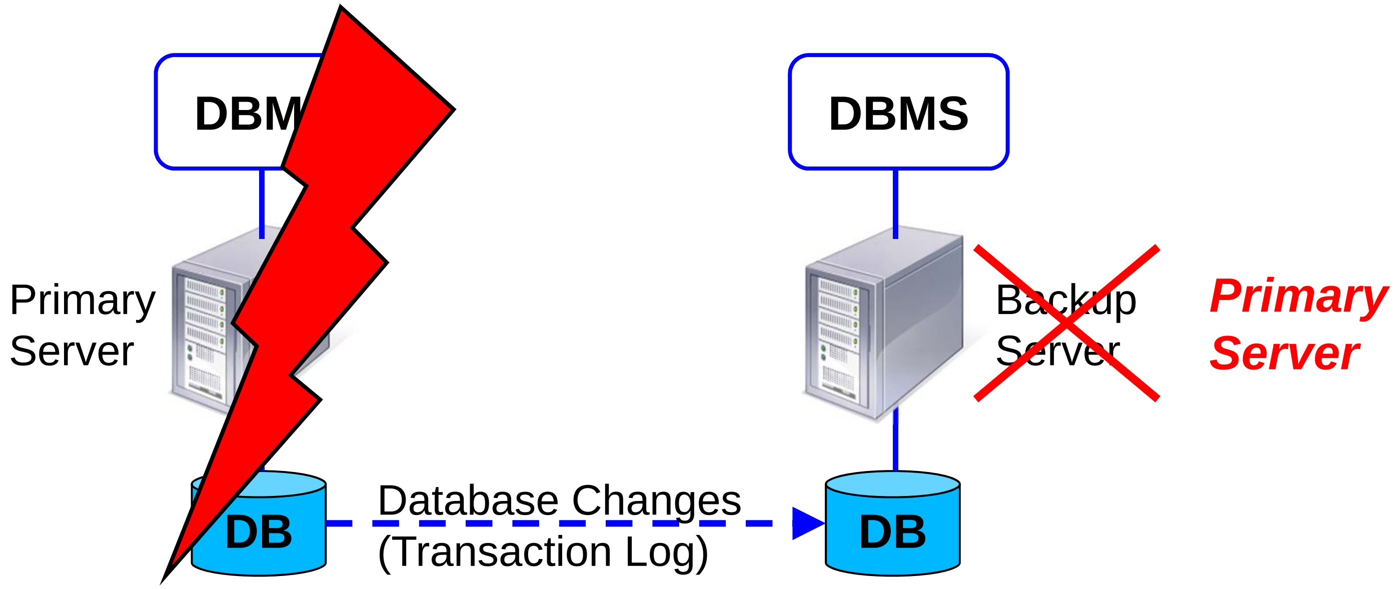
High Availability

- A database system is ***highly available*** if it remains accessible to its users in the face of hardware failures

- ***High availability (HA)*** is becoming a requirement for almost all database applications, not just mission critical ones

- Key issues:
  - Maintaining database consistency in the face of failure
  - Minimizing the impact on performance during normal operation and after a failure
  - Reducing the complexity and administrative overhead of HA

Active/Standby Replication

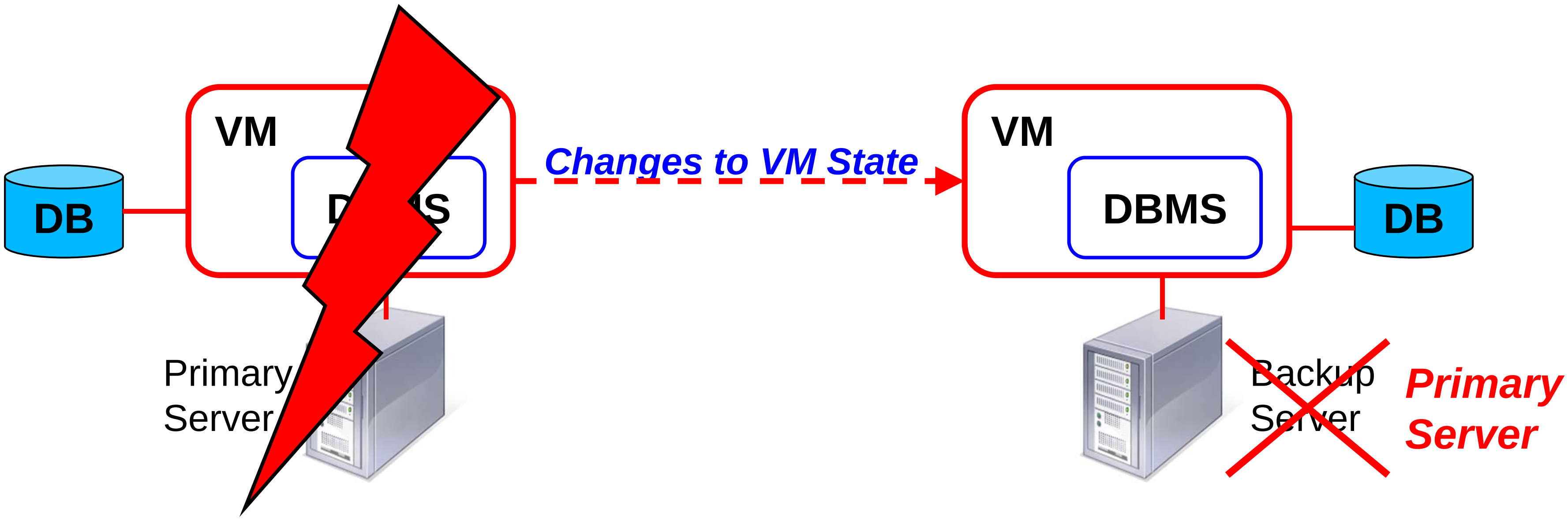


- A copy of the database is stored on two servers, a ***primary*** and a ***backup***
- Primary server ***(active)*** accepts user requests and performs database updates
- Changes to database propagated to backup server ***(standby)*** by ***propagating the transaction log***
- Upon failure, backup server takes over as primary

Active/Standby Replication

- Active/standby replication is complex to implement in the DBMS, and complex to administer
  - Propagating the transaction log
  - Atomic handover from primary to backup on failure
  - Redirecting client requests to backup after failure
  - Minimizing effect on performance (e.g., warming the buffer pool of the backup)
- ***Our approach:*** Implement active/standby replication at the virtual machine layer
  - Push the complexity out of the DBMS
  - ***High availability as a service:*** Any DBMS can be made highly available with little or no code changes
  - Low performance overhead

Transparent HA for DBMS



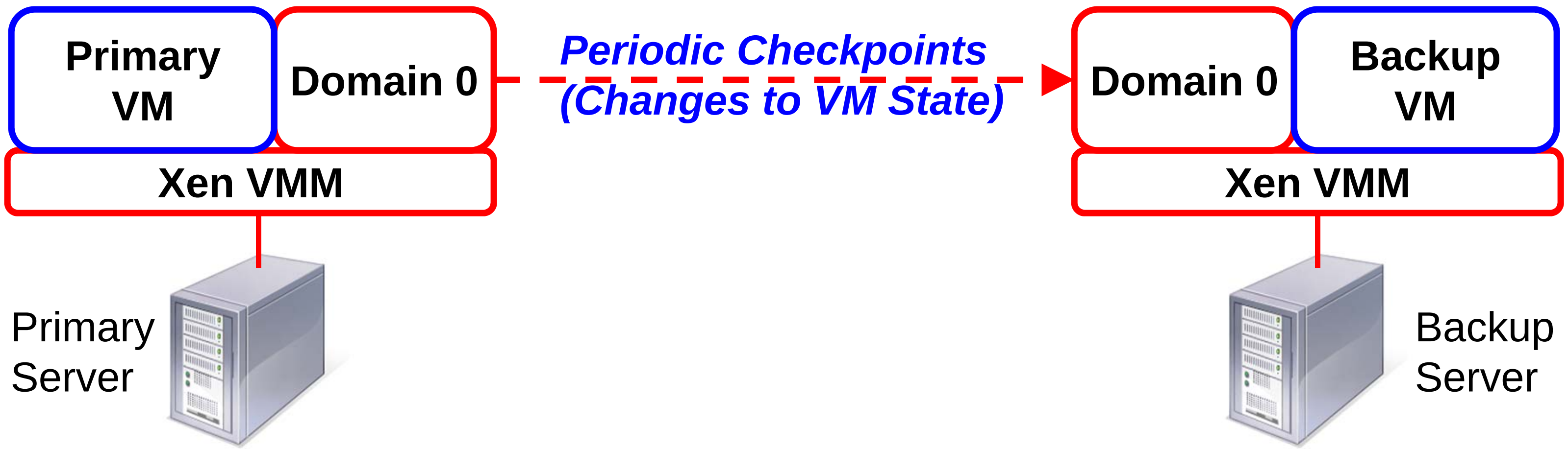
- ***RemusDB:*** efficient and transparent active/standby high availability for DBMS implemented in the virtualization layer
  - Propagates all changes in VM state from primary to backup
  - High availability with no code changes to the DBMS
  - Completely transparent failover from primary to backup
  - Failover to a warmed up backup server

Remus and VM Checkpointing

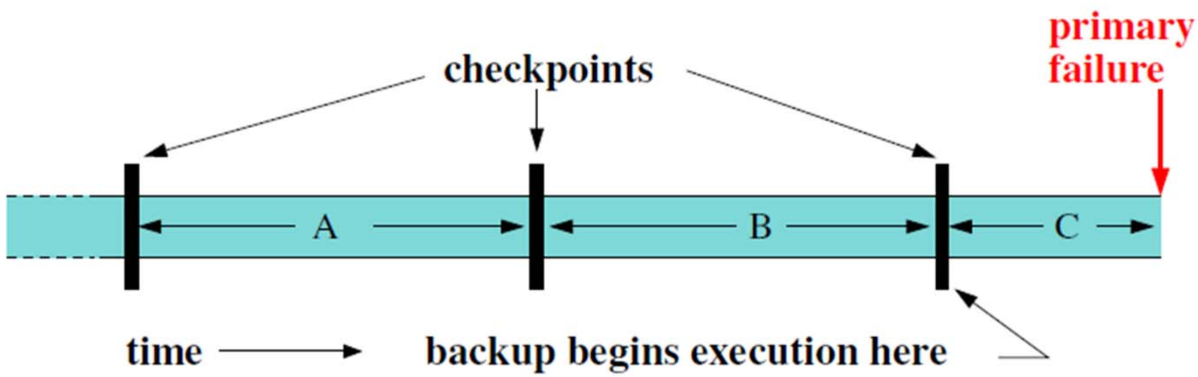
- RemusDB is based on ***Remus***, a high availability solution that is now part of the ***Xen*** virtual machine monitor
- Remus maintains a replica of a running VM on a separate physical machine
- Periodically replicates state changes from the primary VM to the backup VM using ***whole machine checkpointing***
  - Checkpointing is based on extensions to live VM migration
- Provides transparent failover with only seconds of downtime

Remus Checkpoints

- Remus divides time into ***epochs*** (~25ms)
- Performs a ***checkpoint*** at the end of each epoch
  1. Suspend primary VM
  2. Copy all state changes to a buffer in ***Domain 0***
  3. Resume primary VM
  4. Send asynchronous message to backup containing state changes
  5. Backup VM applies state changes

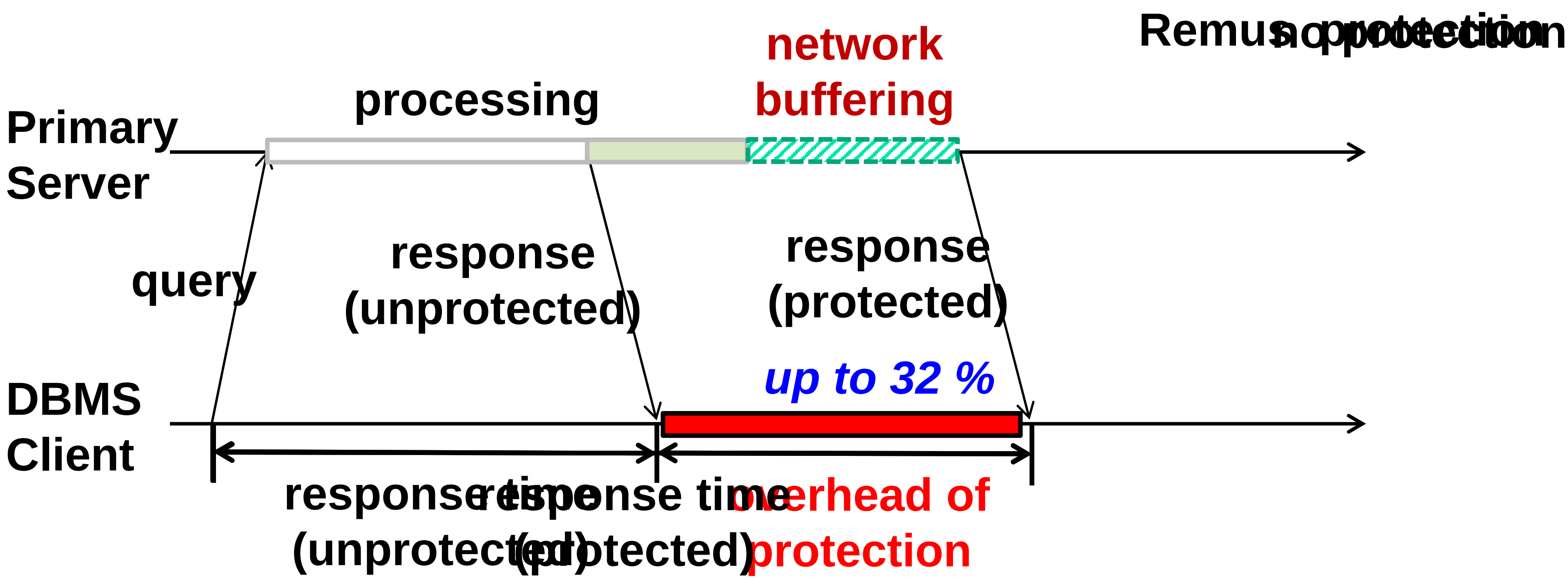


Remus Checkpoints



- After a failure, the backup resumes execution from the ***latest checkpoint***
  - Any work done by the primary during epoch C will be lost **(unsafe)**
- Remus provides a consistent view of execution to clients
  - Any network packets sent during an epoch are ***buffered*** until the next checkpoint
  - Guarantees that a client will see results only if they are based on ***safe*** execution
  - Same principle is also applied to disk writes

Remus and DB Workloads



- **RemusDB** implements optimizations to reduce the overhead of protection for database workloads
  - Incurs $\leq$ ***3% overhead*** and recovers from failures in $\leq$ ***3 seconds***

RemusDB

- Remus optimized for protecting database workloads
- ***Memory optimizations***
  - Database workloads tend to modify a lot of memory in each epoch (buffer pool, working memory for queries, etc.)
  - ***Reduce checkpointing overhead***
    - Asynchronous checkpoint compression ← ***Send less data***
    - Disk read tracking
    - Memory deprotection

    (Disk read tracking, Memory deprotection: ***Protect less memory***)
- ***Network optimization***
  - Some database workloads are sensitive to the network latency added by buffering network packets
  - ***Exploit semantics of database transactions to avoid buffering***
    - Commit protection

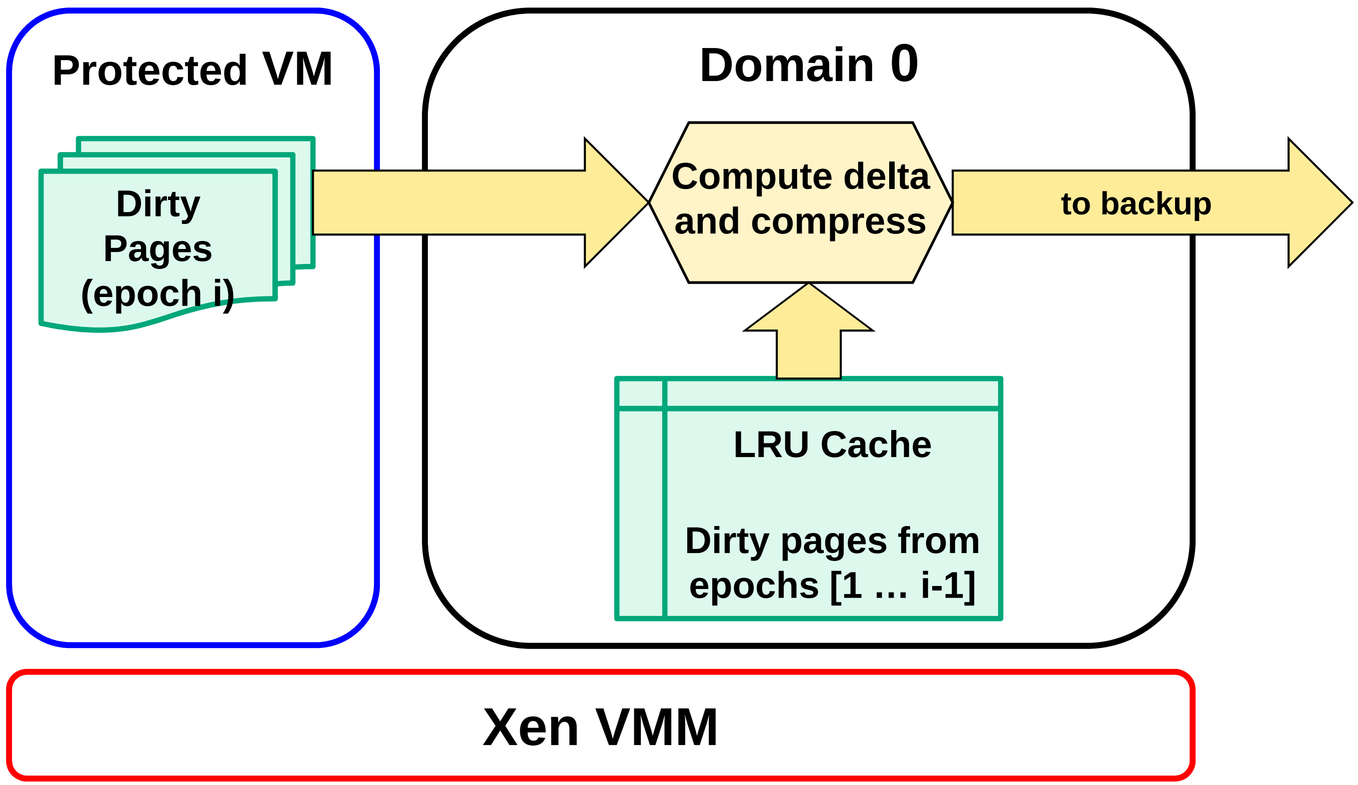
Async Checkpoint Compression

- Database workloads typically involve a large set of frequently changing pages of memory (e.g., ***buffer pool pages***)
  - Results in a large amount of replication traffic

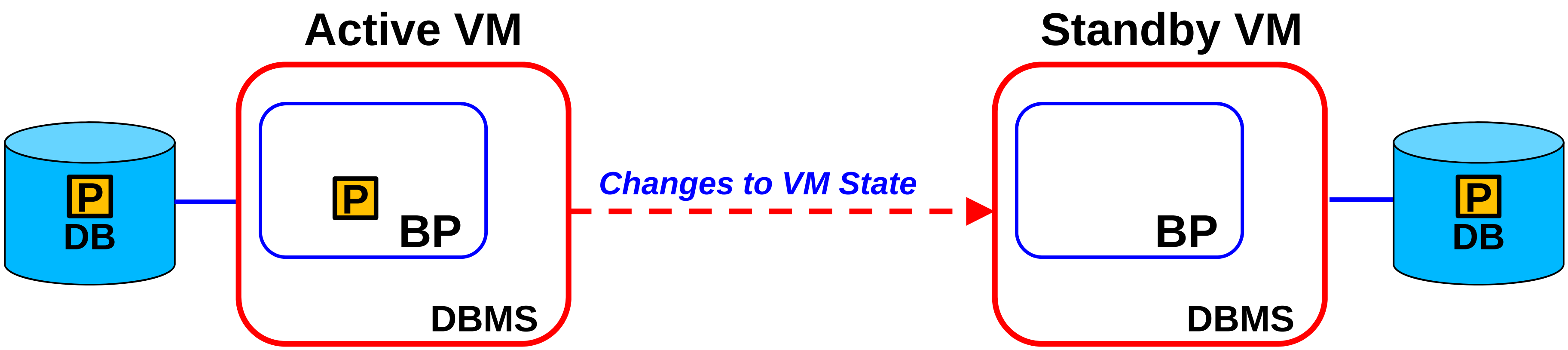
- The DBMS often changes only a small part of the pages
  - Data that is replicated contains redundancy

- ***Reduce replication traffic by only sending the changes to the memory pages (and send them compressed)***

Async Checkpoint Compression



Disk Read Tracking



- DBMS loads pages from disk into its buffer pool (BP)
  - Clean to DBMS, dirty to Remus
- Remus synchronizes dirty BP pages in every checkpoint
- ***Synchronization of clean BP pages is unnecessary***
  - Can be read from disk at the backup

Disk Read Tracking

- Track the memory pages into which disk reads are placed
- ***Do not mark these pages as dirty*** until they are actually modified
- Add an ***annotation to the replication stream*** indicating the disk sectors to read to reconstruct these pages

Memory Deprotection

- A mechanism that we implemented but did not find useful!
- Allow the DBMS to declare regions of its memory as ***deprotected*** (i.e., not replicated in checkpoints)
  - Hot memory regions such as buffer pool descriptors
  - Memory regions that can easily be reconstructed such as working memory for query processing operators
- After a failure, a ***recovery handler*** at the backup would reconstruct or drop the deprotetced memory regions
  - Memory deprotection is not transparent to the DBMS

Memory Deprotection

- Memory deprotection not useful for our workloads because:
  - Disk read tracking (which is transparent) gets us the same benefit for the buffer pool
  - CPU overhead of tracking deprotected pages is high so the benefit that we get from deprotection is low
  - Benefit does not justify the complex non-transparent interafce

- May be useful for other applications and workloads

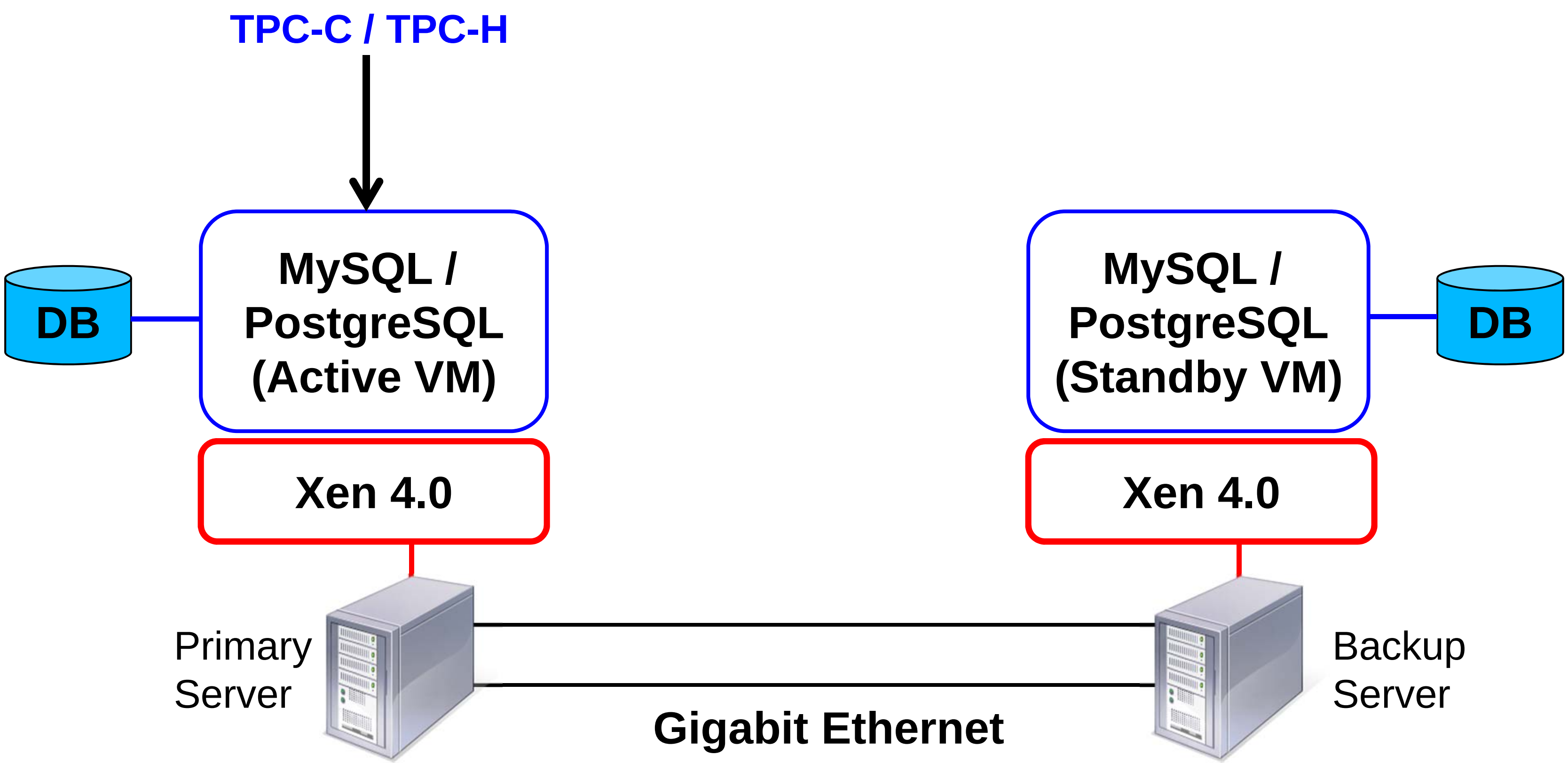
Network Optimization

- Remus buffers every outgoing network packet
  - Ensures clients never see results of unsafe execution
  - But increases round trip latency by 2-3 orders of magnitude
  - Largest source of overhead for many database workloads
  - ***Unnecessarily conservative for database systems***
- Database systems provide transactions with clear consistency and durability semantics
  - Remus's TCP-level per-checkpoint transactions are redundant
- Provide an interface to allow a DBMS to decide which packets are ***protected*** (i.e., buffered until the next checkpoint) and which are unprotected
  - Implemented as a new **setsockopt()** option in Linux

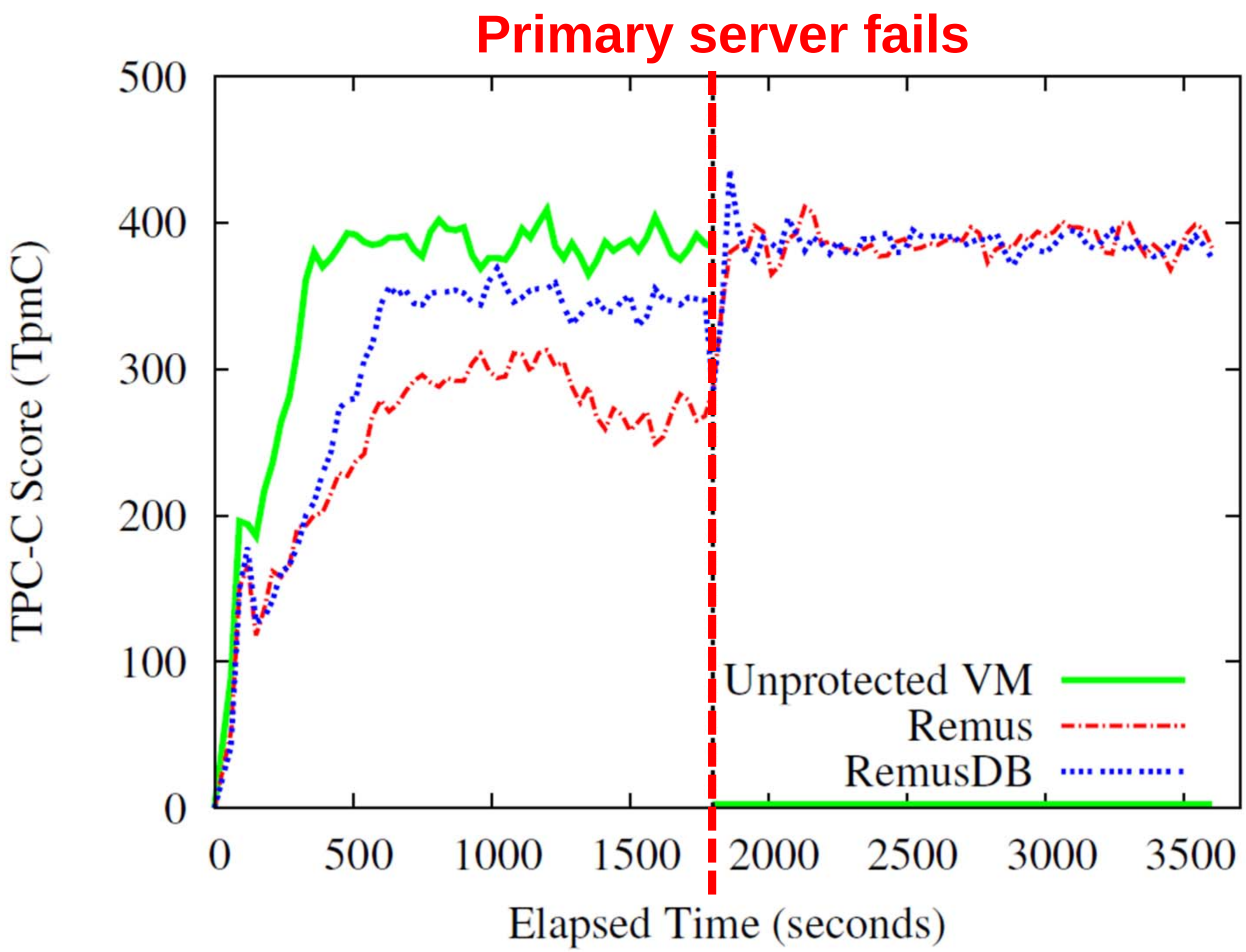
Commit Protection

- ***Commit Protection***
  - DBMS only protects transaction control packets (BEGIN TRANSACTION, COMMIT, ABORT)
  - Other packets are unprotected
- After failover, a ***recovery handler*** runs in the DBMS at the backup
  - Aborts all in-flight transaction where the client connection was in unprotected mode
- Not transparent to the DBMS
  - Requires minor modifications to the client connection layer
  - 103 LoC for PostgreSQL, 85 LoC for MySQL
- Transaction safety is guaranteed

Experimental Setup

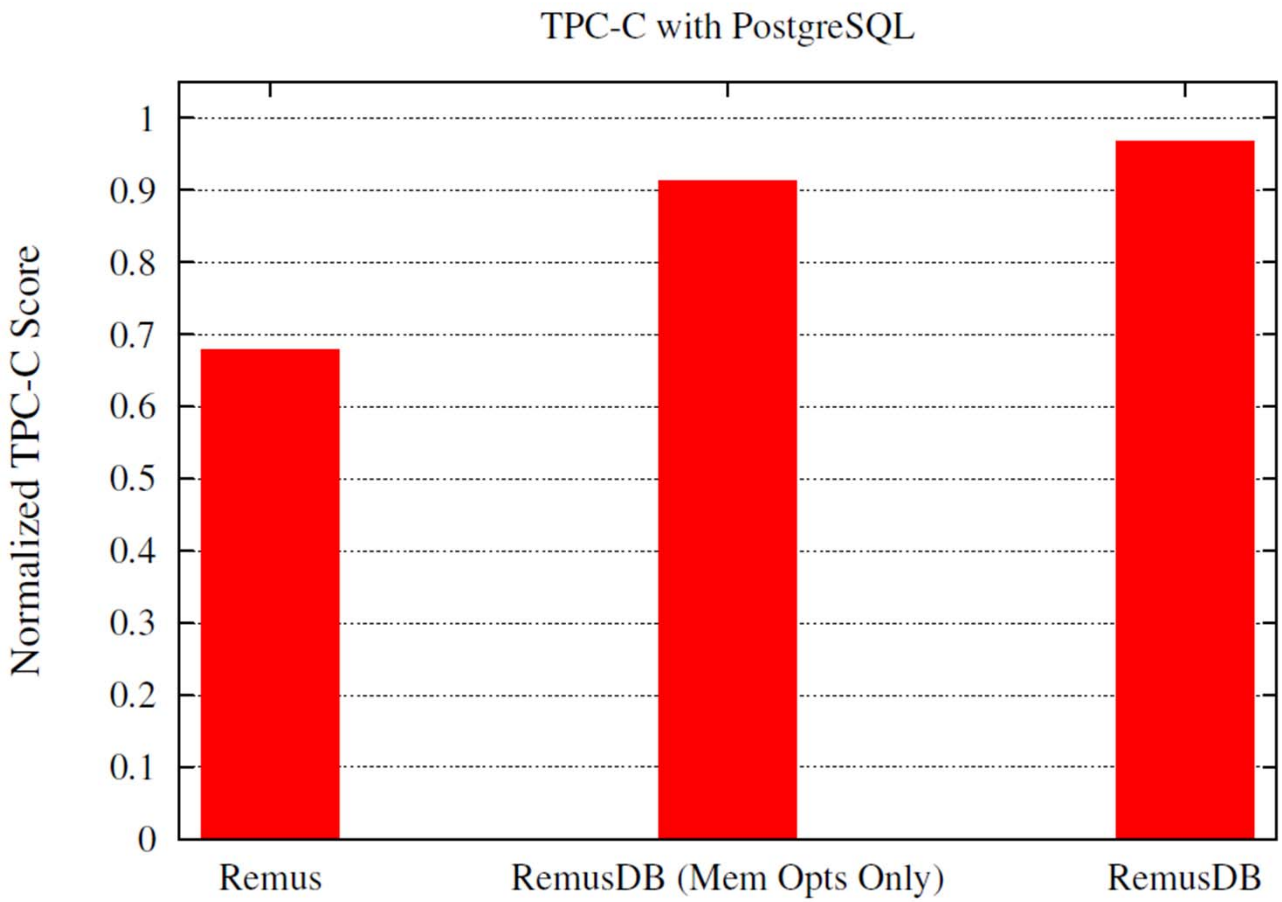


Failover



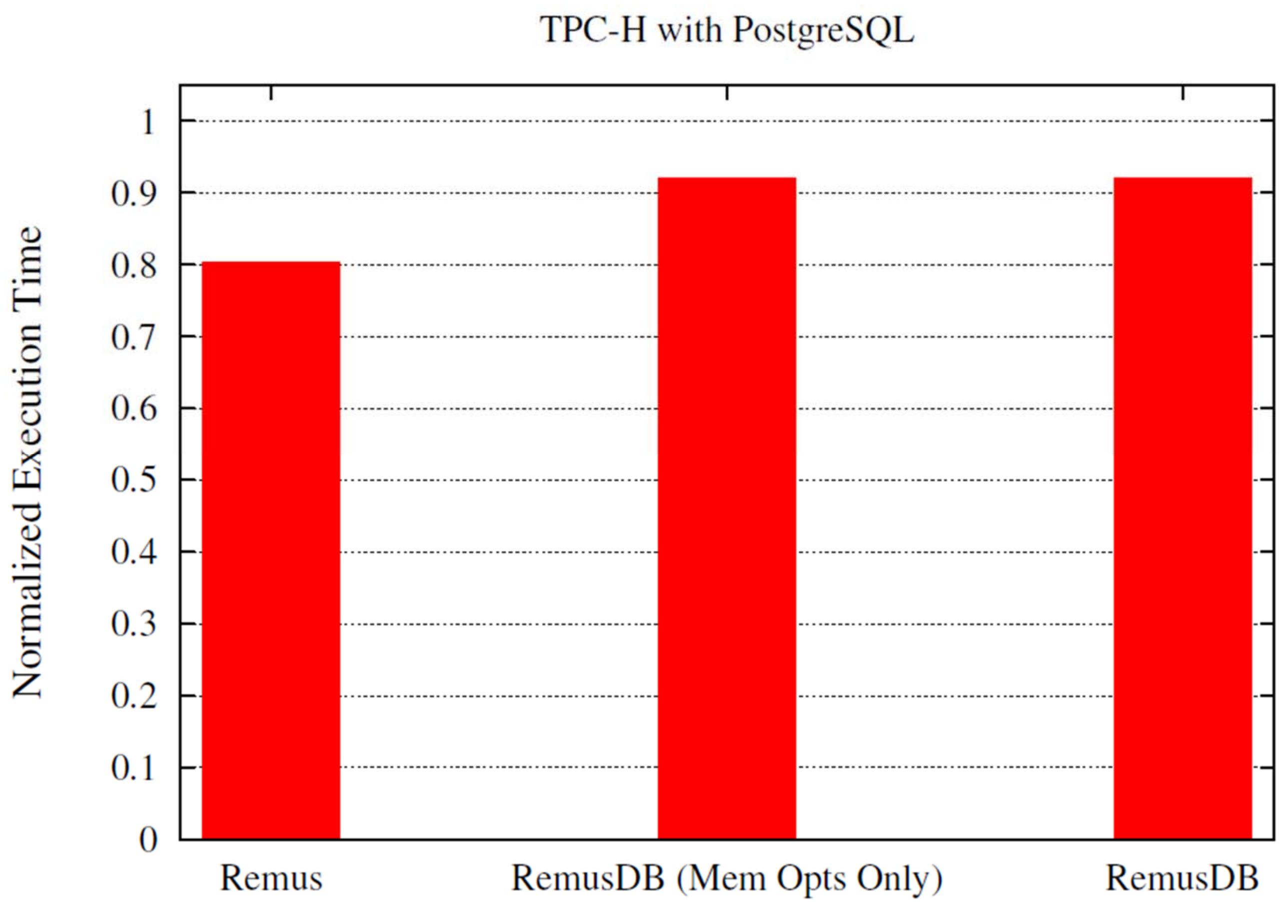
TPC-C on MySQL

Normal Operation



TPC-C on PostgreSQL

Normal Operation



TPC-H on PostgreSQL

Benefits of RemusDB

- High availability for any DBMS with no code changes
  - Or with very little code changes if we use commit protection
  - “High availability as a service”
- Automatic and fully transparent failover to a warmed up system
- Next steps
  - Reprotection after a failure
  - One server as the backup for multiple primary servers
  - Administration of RemusDB failover

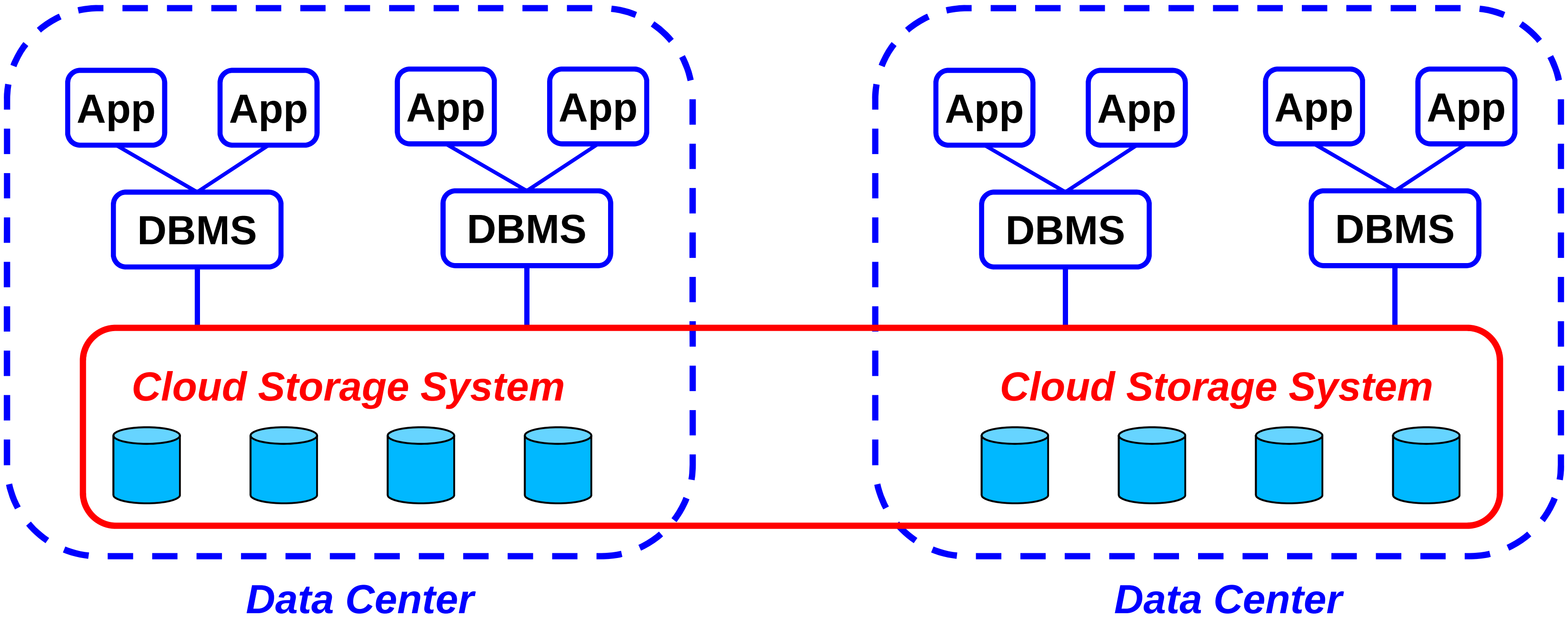
Outline

- Introduction
- **RemusDB:** Database high availability using virtualization
- **DBECS:** Database high availability (and scalability) using eventually consistent cloud storage
  (Under submission)
- Conclusion

Cloud Storage

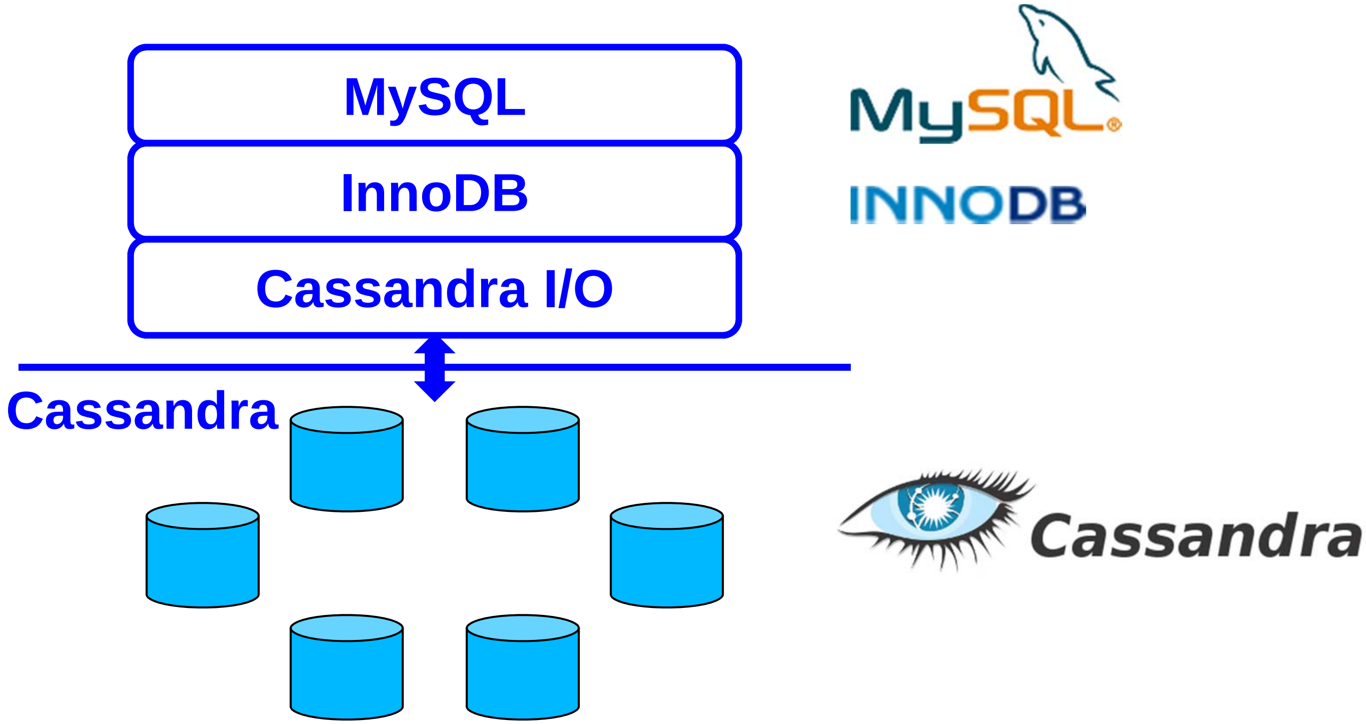
- Many cloud storage systems
  - Amazon S3
  - HBase
  - Cassandra
  - …and more
- Scalable, distributed, fault tolerant
- Support simple read and write operations
  - write(key, value)
  - value = read(key)
- Atomicity only for single-row operations
  - No atomic multi-row reads or writes
  - Interface much simpler than SQL ***("NoSQL")***

Databases Over Cloud Storage



- ***Goal: A scalable, elastic, highly available, multi-tenant database service that supports SQL and ACID transactions***
  - Cloud storage system provides scalability, elasticity, and availability. DBMS provides SQL and ACID transactions.

DBECS



- ***DBECS: Databases on Eventually Consistent Stores***
- Can replace MySQL with another DBMS
- Need Cassandra since we want eventual consistency

Why Cassandra?

- Relaxing consistency reduces the ***write latency*** of Cassandra and makes it ***partition tolerant***

- Cassandra stores semi-structured rows that belong to column families
  - Rows are accessed by a ***key***
  - Rows are replicated and distributed by hashing keys

- Multi-master replication for each row
  - Enables Cassandra to run in multiple data centers
  - Also gives us partition tolerance
  - DBECS leverages this for ***disaster tolerance***

Why Cassandra?

- Client controls the consistency vs. latency trade-off for each read and write operation
  - write(1)/read(1) – fast but not necessarily consistent
  - write(ALL)/read(ALL) – consistent but may be slow
  - ***We posit that database systems can control this trade-off quite well***

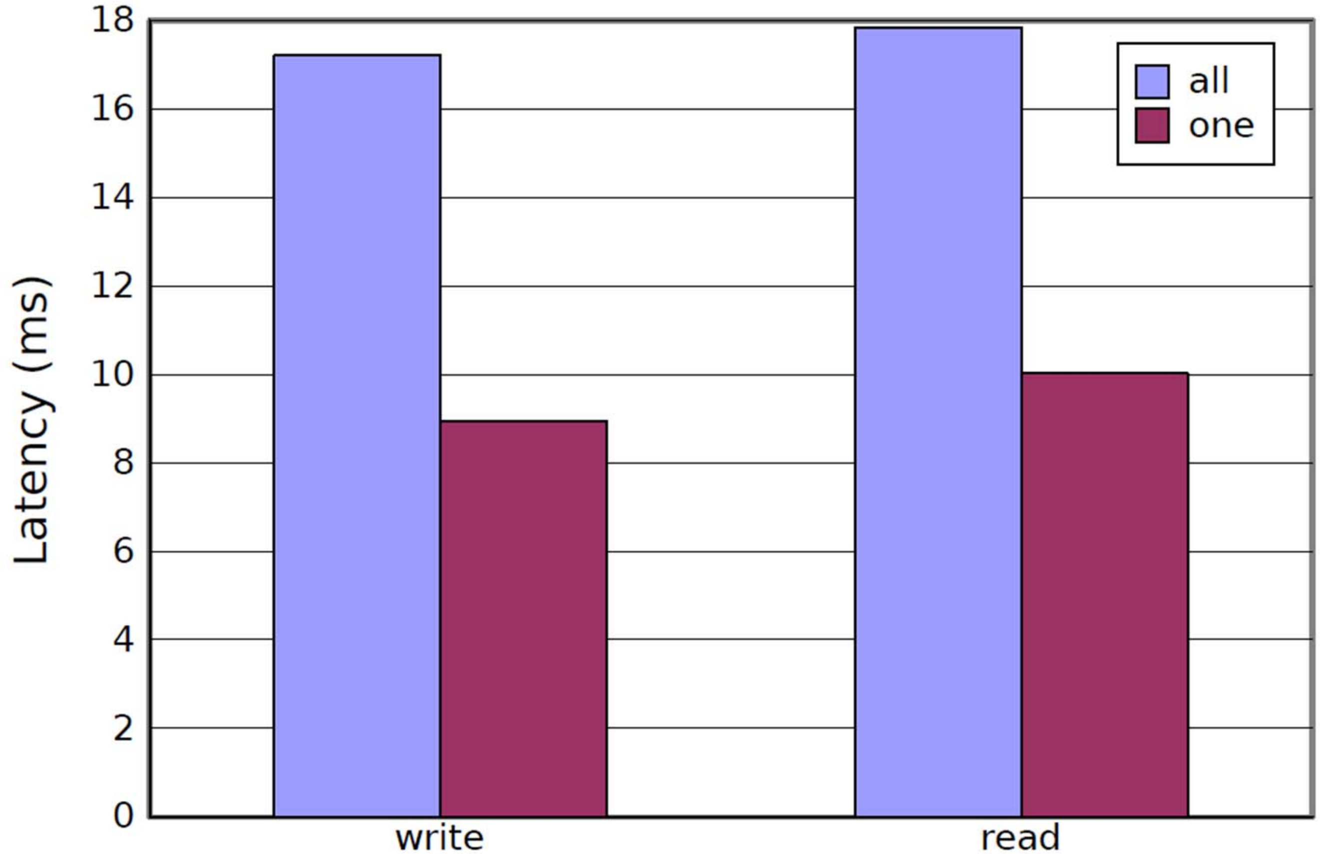
- Client decides the serialization order of updates
  - Important for consistency in DBECS

- Scalable, elastic, highly available
  - Like many other cloud storage systems!

Consistency vs. Latency

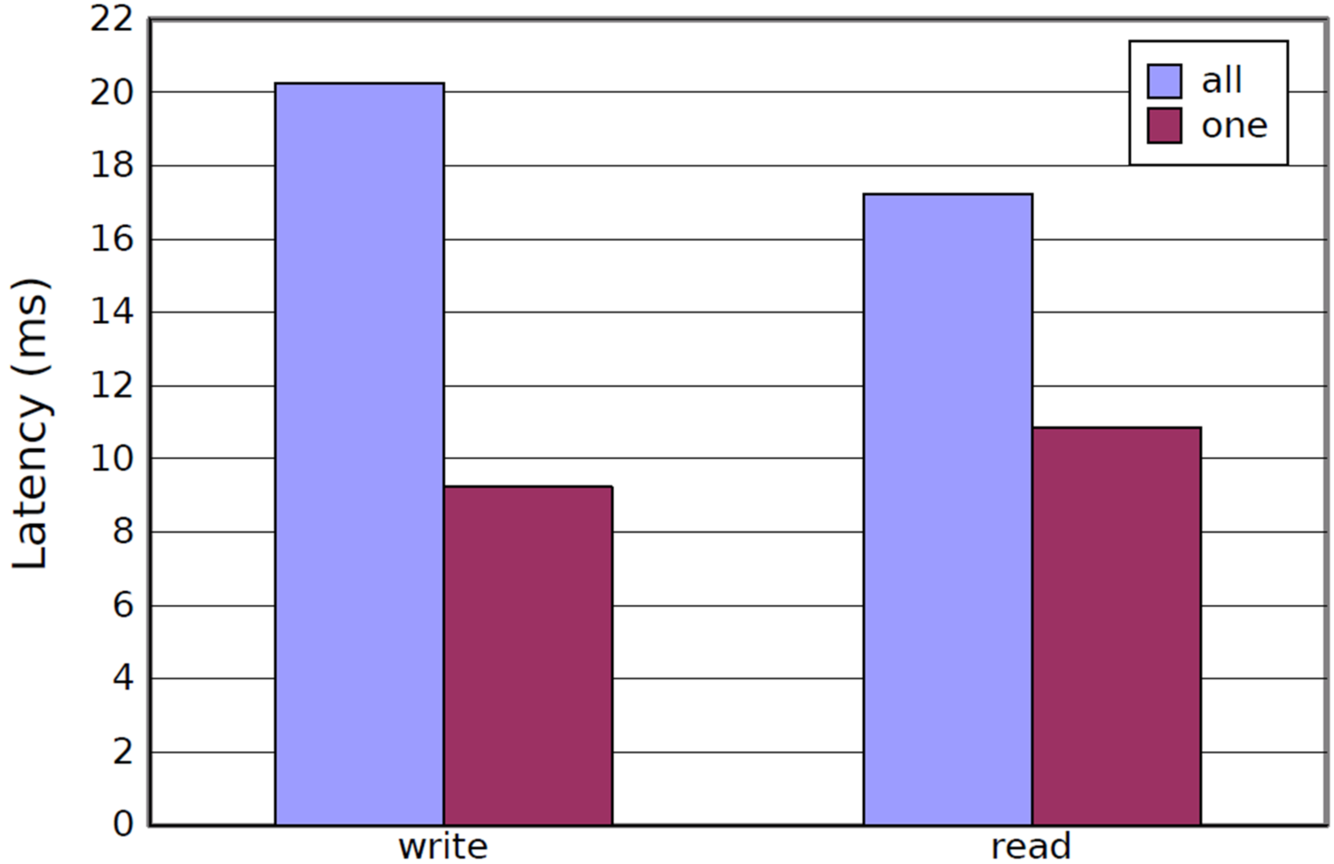
- value = read(1, key, column)
  - Send read request to all replicas of the row (based on key)
  - Return first response received to client
  - Returns quickly but may return stale data
- value = read(ALL, key, column)
  - Send read request to all replicas of the row (based on key)
  - Wait until all replicas respond and return ***latest version*** to client
  - Consistent but as slow as the slowest replica
- write(1) vs. write(ALL)
  - Send write request to all replicas
  - ***Client provides a timestamp for each write***
- Other consistency levels are supported

Consistency vs. Latency



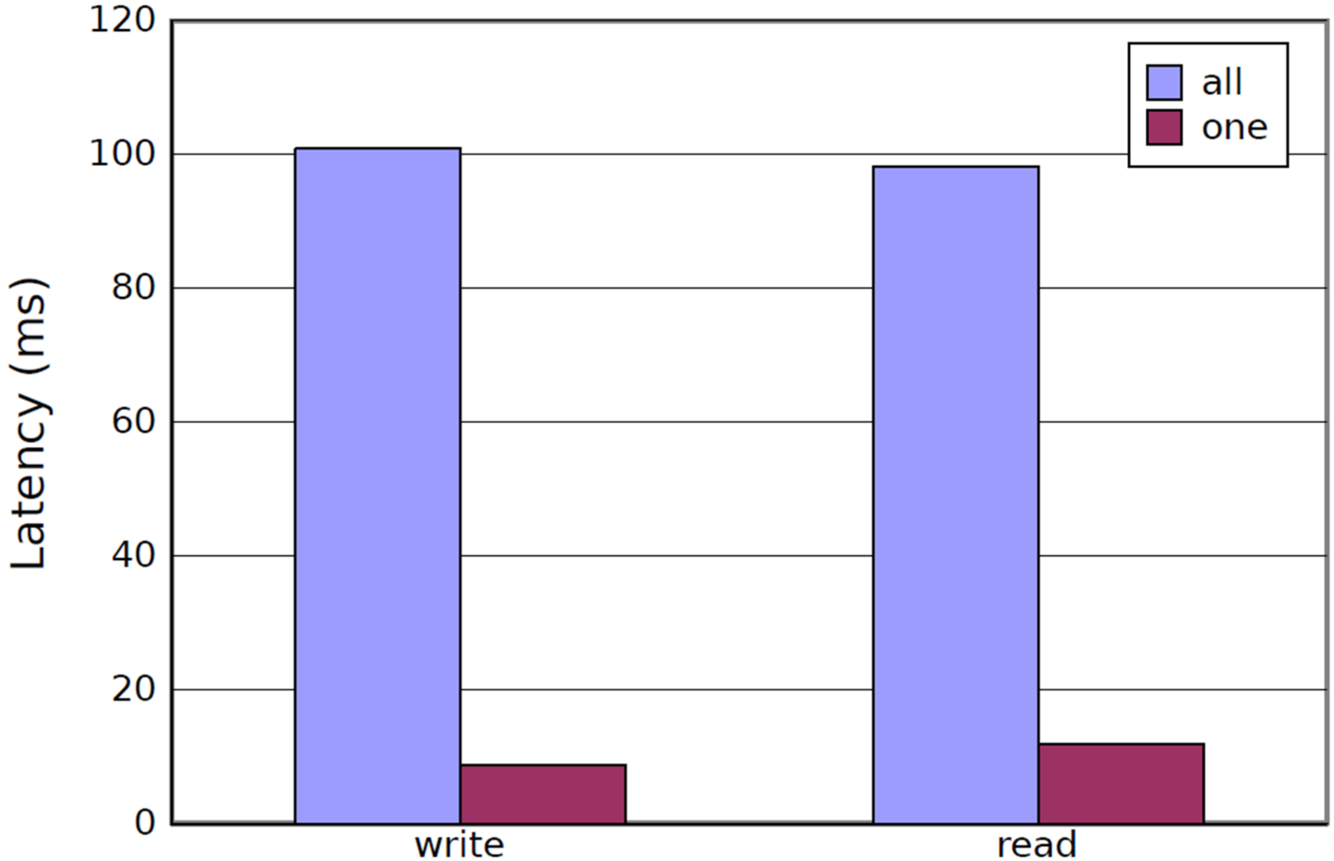
Experiment on Amazon EC2 – Yahoo! Cloud Serving Benchmark (YCSB) – 4 Cassandra Nodes Same EC2 Availability Zone

Consistency vs. Latency



Two EC2 Availability Zones
Same EC2 Geographic Region

Consistency vs. Latency



Two EC2 Regions
(US East and US West)

Databases Over Cassandra

- Make Cassandra ***look like a disk*** to the DBMS tenants
  - Databases stored in one column (in one column family)
  - key = DBMS id + disk block id
  - value = contents of disk block
- Cassandra I/O layer maps DBMS reads and writes to Cassandra reads and writes
  - Which consistency level to use?
  - write(1)/read(1): Fast but may return stale data and provides no durability guarantees. ***Not good for a DBMS.***
  - write(ALL)/read(1): Returns no stale data and guarantees durability but writes are slow.

Goal of DBECS

- ***Achieve the performance of write(1)/read(1) while maintaining consistency, durability, and availability***
- Optimistic I/O
  - Use write(1)/read(1) and detect stale data
- Client-controlled synchronization
  - Make database updates safe in the face of failures

Optimistic I/O

- ***Key observation: with write(1)/read(1), most reads will not return stale data***
  - Single writer for each database block
  - Reads unlikely to come soon after writes because of DBMS buffer pool
  - Cassandra sends writes to all replicas
  - Network topology means that first replica to acknowledge a write will likely be the first to acknowledge a read
- So use write(1)/read(1), detect stale data, and recover from it

Optimistic I/O

- Detecting stale data
  - Cassandra I/O stores a version number with each database block and remembers the current version of each block
  - Checks the version number returned by read(1) against the current version number
- Recovering from stale data
  - Use read(ALL)
  - Retry the read(1)
    - When read(1) detects stale data, Cassandra brings the stale replicas up to date ***(read repair)***
- We only store version information about recently accessed database blocks
  - For the rest, use read(ALL)

Dealing with Failures

- With write(1), data is not safe
- With read(ALL), will block if one replica is down
- Naive solution: use write(ALL)/read(QUORUM)
- ***Key observation: Transaction semantics tell us when writes must be safe and when the DBMS can tolerate unsafe writes***
  - Write Ahead Logging tells us when data needs to be safe (write log before data and flush log on commit)
  - Database systems explicitly synchronize data at the necessary points, for example, by using fsync()
  - Need an fsync() for Cassandra
  - Can abort transactions if unsafe writes are lost

Client-Controlled Sync

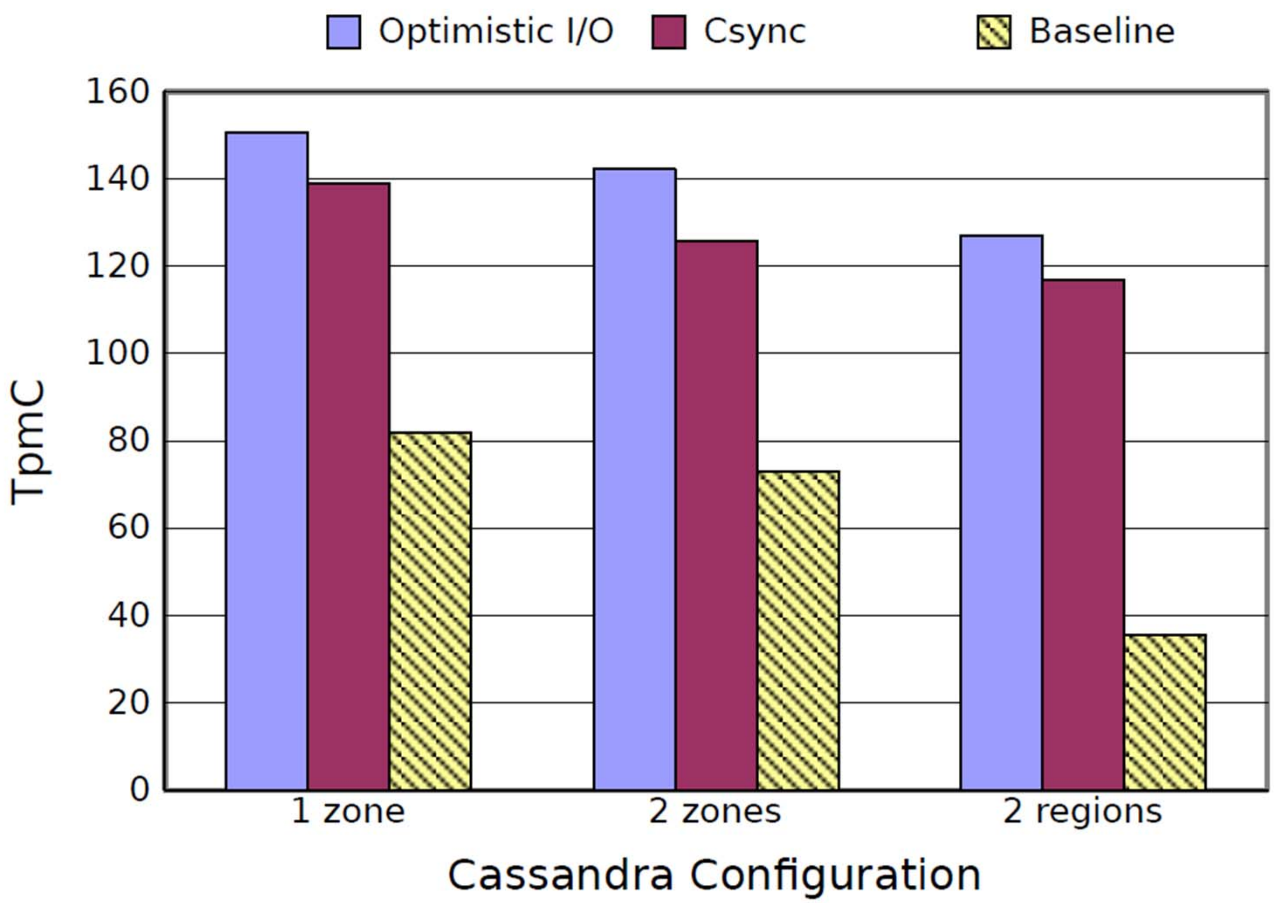
- Added new type of write in Cassandra: ***write(CSYNC)***
  - Like write(1), but stores the key of the written row in a ***sync_pending*** list
  - Cassandra client can issue a ***CSync()*** call to synchronize all rows in the sync_pending list

- To protect against failure, Cassandra I/O
  - Uses write(CSYNC) instead of write(1)
  - Calls CSync() whenever the DBMS calls fsync()
    - ***Data or log pages are made safe only when needed***
    - Time between write() and CSync() enables latency hiding
  - Uses read(QUORUM)

Examples of Failures

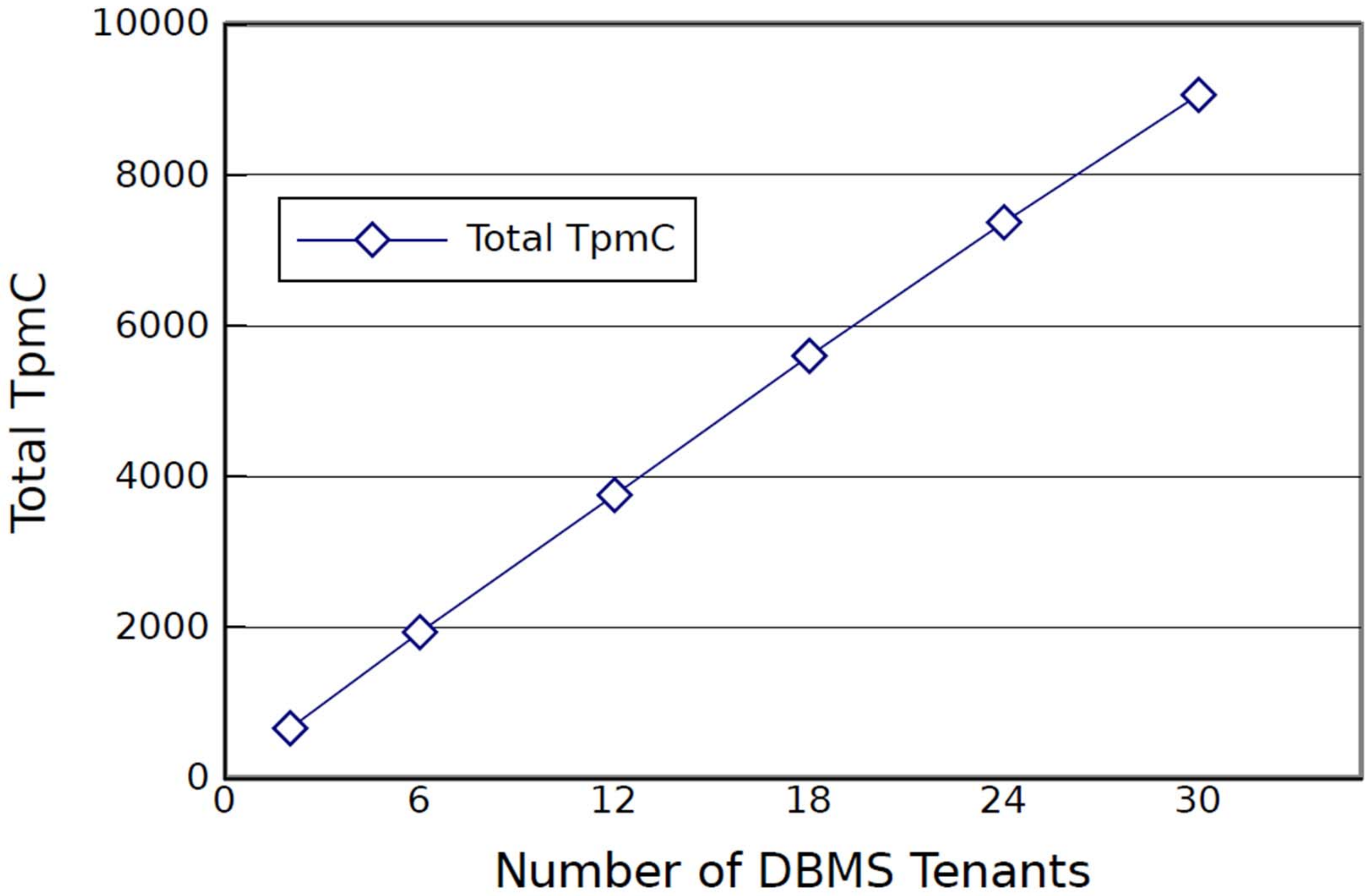
- Loss of a Cassandra node
  - Handled by Cassandra
  - Completely transparent to DBMS
- Loss of the primary data center ***(Disaster Recovery)***
  - Cassandra needs to be running in multiple data centers
  - Restart the DBMS in a backup data center
  - Log-based recovery brings the database up to date in a transactionally consistent way

Throughput



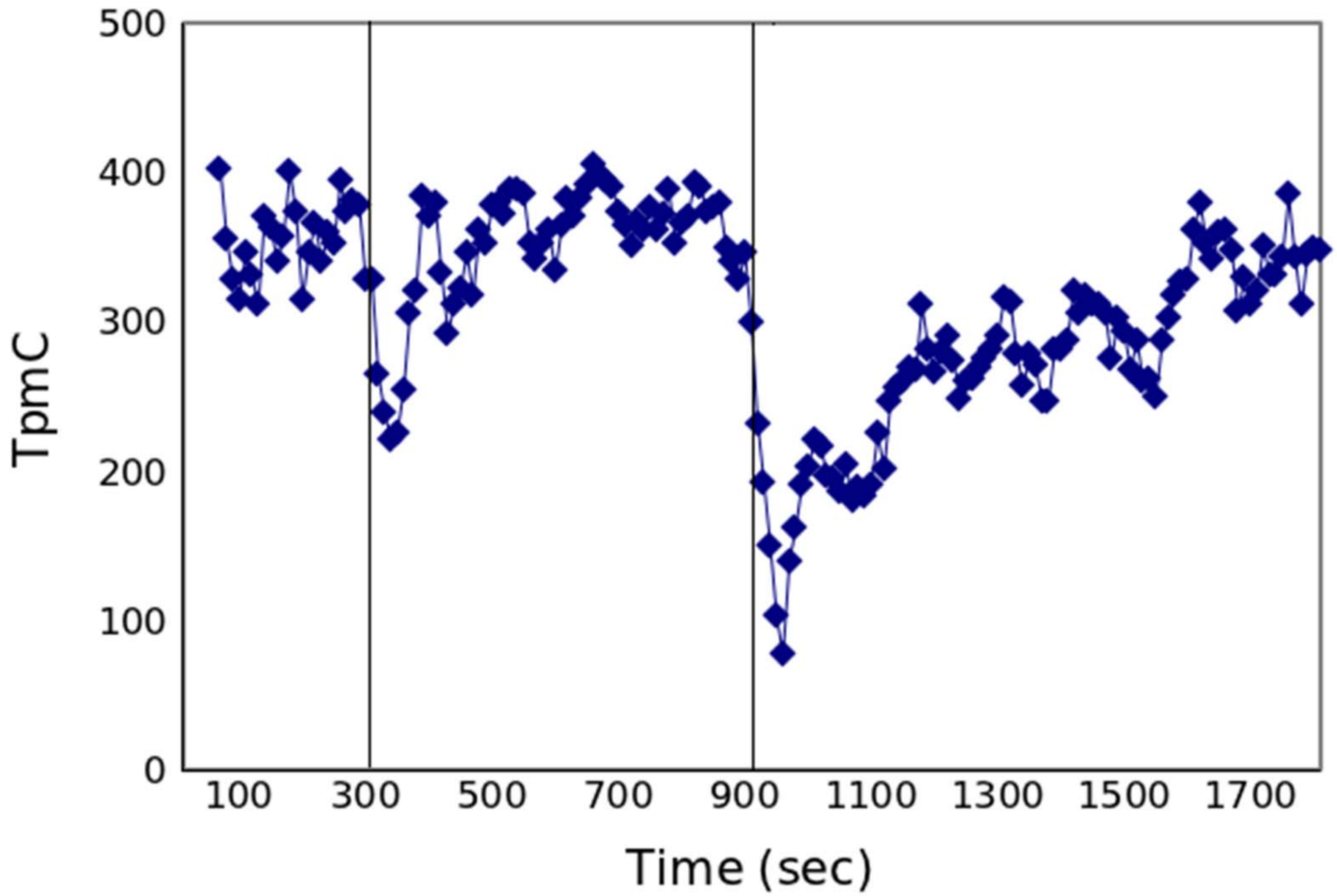
MySQL and Cassandra in Amazon EC2
TPC-C Workload – 6 Cassandra Nodes

Scalability



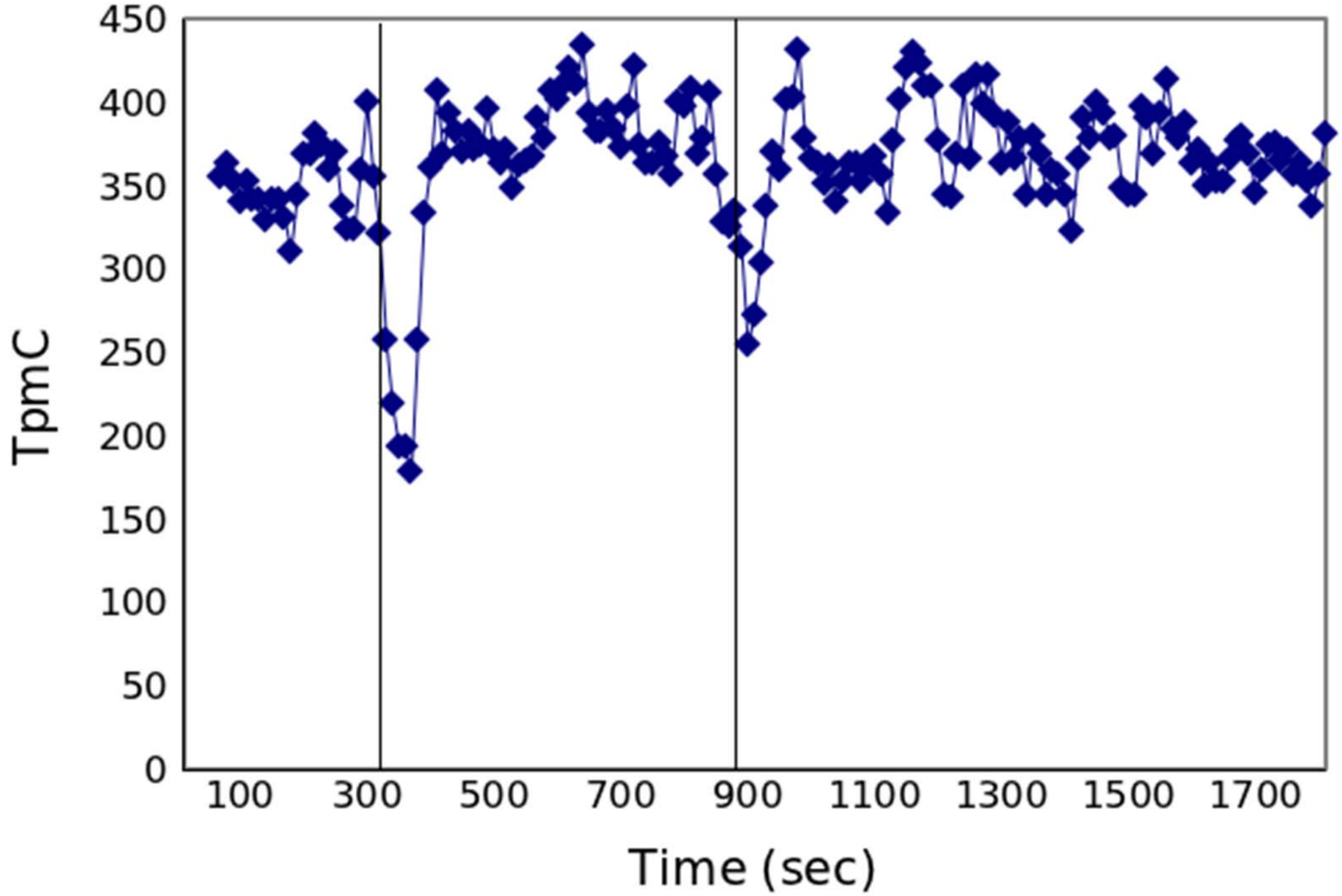
Adding DBMS Tenants and Proportionally Increasing the Number of Cassandra Nodes

High Availability



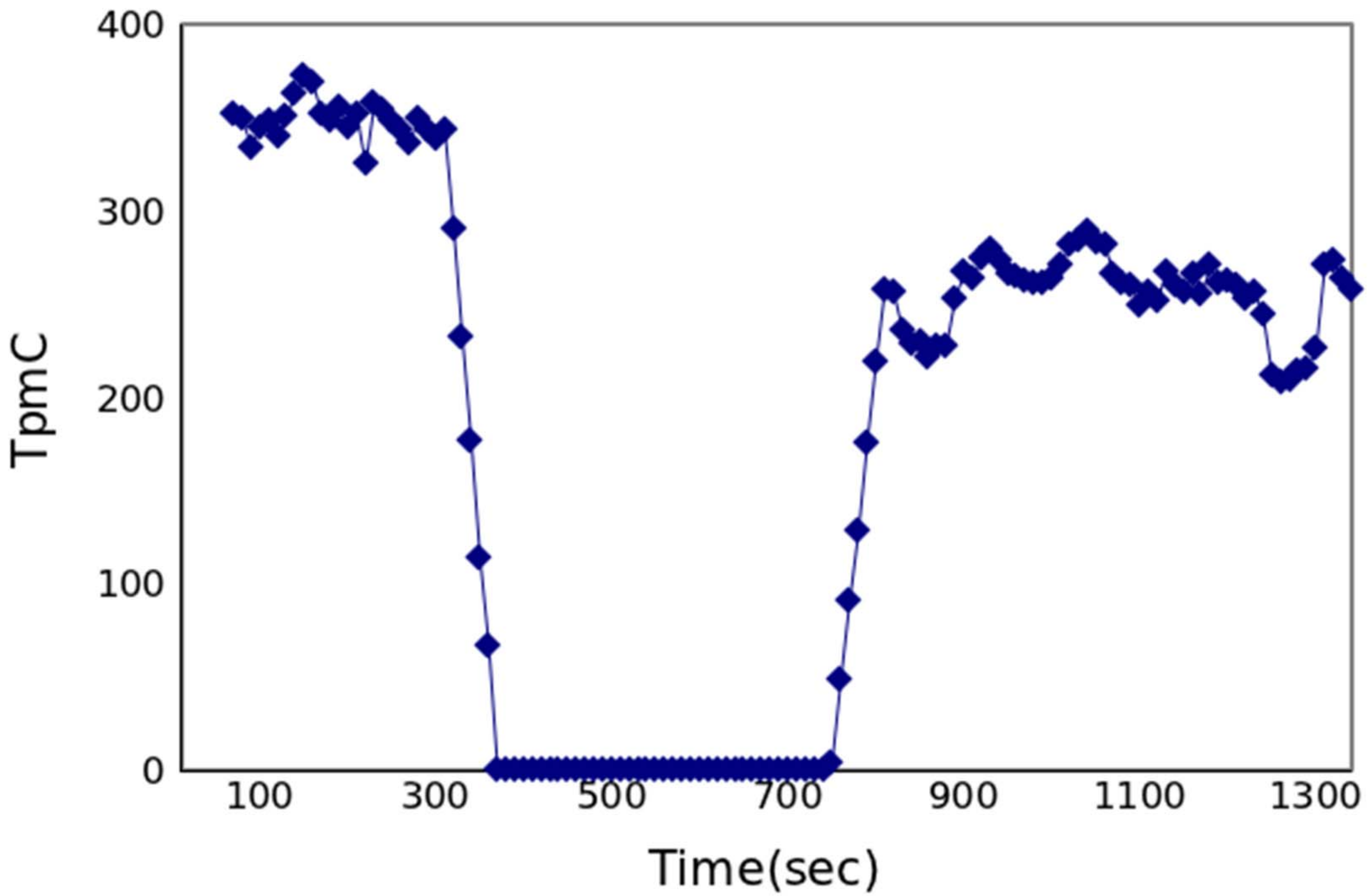
Failure of Cassandra Node in Primary Data Center

High Availability



Failure of Cassandra Node in Secondary DC

Disaster Recovery



Loss of Primary Data Center

Benefits of DBECS

- Scalable and elastic storage capacity and bandwidth
- Scalable in the number of database tenants
- Highly available and disaster tolerant storage tier
- SQL and ACID transactions for the tenants
- ***An interesting point in the spectrum of answers to the question: "Can consistency scale?"***

- Missing from DBECS (next steps)
  - Always-up hosted DBMS tenants
    - DBECS enables DBMS tenant to use standard log-based recovery, but tenant incurs significant down time
  - Scaling of individual hosted DBMS tenants

Conclusion

- High availability (and scalability) for database systems can be provided by the cloud infrastructure
- Taking advantage of the well-known characteristics of database systems can greatly enhance the solutions
- **RemusDB:** Efficient and transparent database high availability in the virtualization layer
- **DBECS:** Scalability and high availability for database clients built on the Cassandra cloud storage system