Deep Dive on Amazon Aurora

Steve Abraham, Solutions Architect

September 2016

What is Amazon Aurora?

MySQL-compatible relational database

Performance and **availability** of commercial databases

Simplicity and cost-effectiveness of open-source databases

Aurora customer adoption

Fastest growing service in AWS history

- Business applications
- Web and mobile
- Content management
- E-commerce, retail
- Internet of Things
- Search, advertising
- BI, analytics
- Games, media

ISCS

GE Oil & Gas

Alfresco

Earth Networks
Taking the Pulse of the Planet

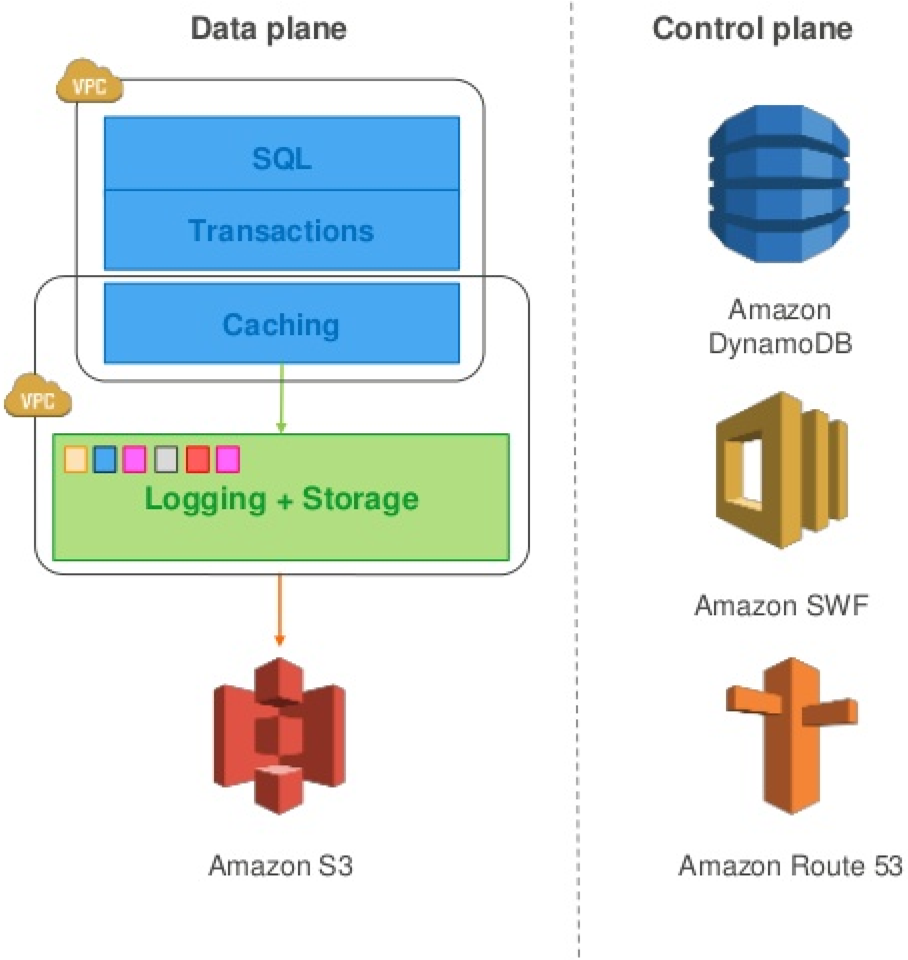
Expedia

A service-oriented architecture applied to databases

1 Moved the logging and storage layer into a multitenant, scale-out database-optimized storage service

2 Integrated with other AWS services like Amazon EC2, Amazon VPC, Amazon DynamoDB, Amazon SWF, and Amazon Route 53 for control plane operations

3 Integrated with Amazon S3 for continuous backup with 99.999999999% durability



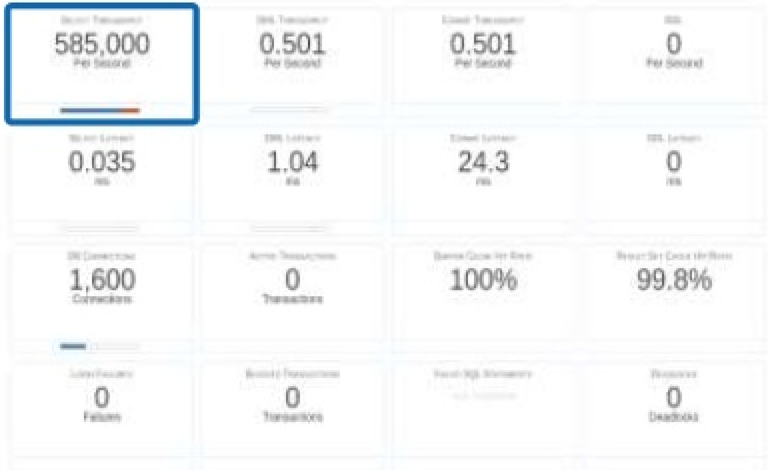
SQL benchmark results

Using MySQL SysBench with Amazon Aurora R3.8XL with 32 cores and 244 GB RAM

WRITE PERFORMANCE

4 client machines with 1,000 connections each

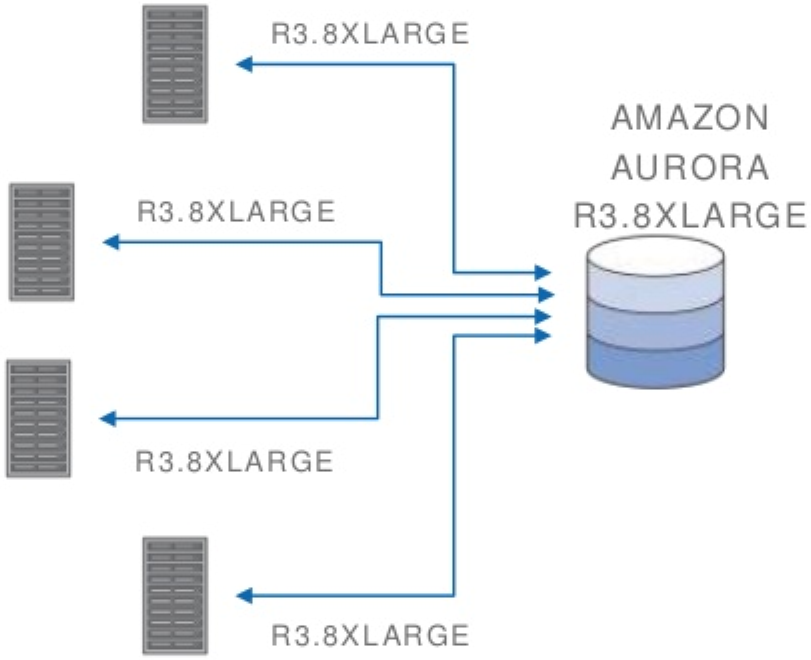
READ PERFORMANCE



Single client machine with 1,600 connections

Reproducing these results

1. Create an Amazon VPC (or use an existing one).
2. Create 4 EC2 R3.8XL client instances to run the SysBench client. All 4 should be in the same Availability Zone (AZ).
3. Enable enhanced networking on your clients.
4. Tune Linux settings (see whitepaper referenced below).
5. Install SysBench version 0.5.
6. Launch a r3.8xlarge Amazon Aurora DB instance in the same VPC and AZ as your clients.
7. Start your benchmark!



https://d0.awsstatic.com/product-marketing/Aurora/RDS_Aurora_Performance_Assessment_Benchmarking_v1-2.pdf

Performance best practices

MySQL and RDBMS practices still apply

- Choose the right tool for the right job (OLAP vs. OLTP vs. NoSQL)
- Create appropriate indexes
- Tune your SQL code, use explain plans, performance schema
- Many more…

Leverage high concurrency

- Aurora throughput increases with number of connections
- Architect your applications to leverage high concurrency in Aurora

Read scaling

- Aurora offers read replicas with virtually no replication lag
- Leverage multiple read replicas to distribute your reads

Performance best practices

Parameter tuning

- No need to migrate your performance-related MySQL parameters to Aurora
- Aurora parameter groups are pre-tuned and already optimal in most cases

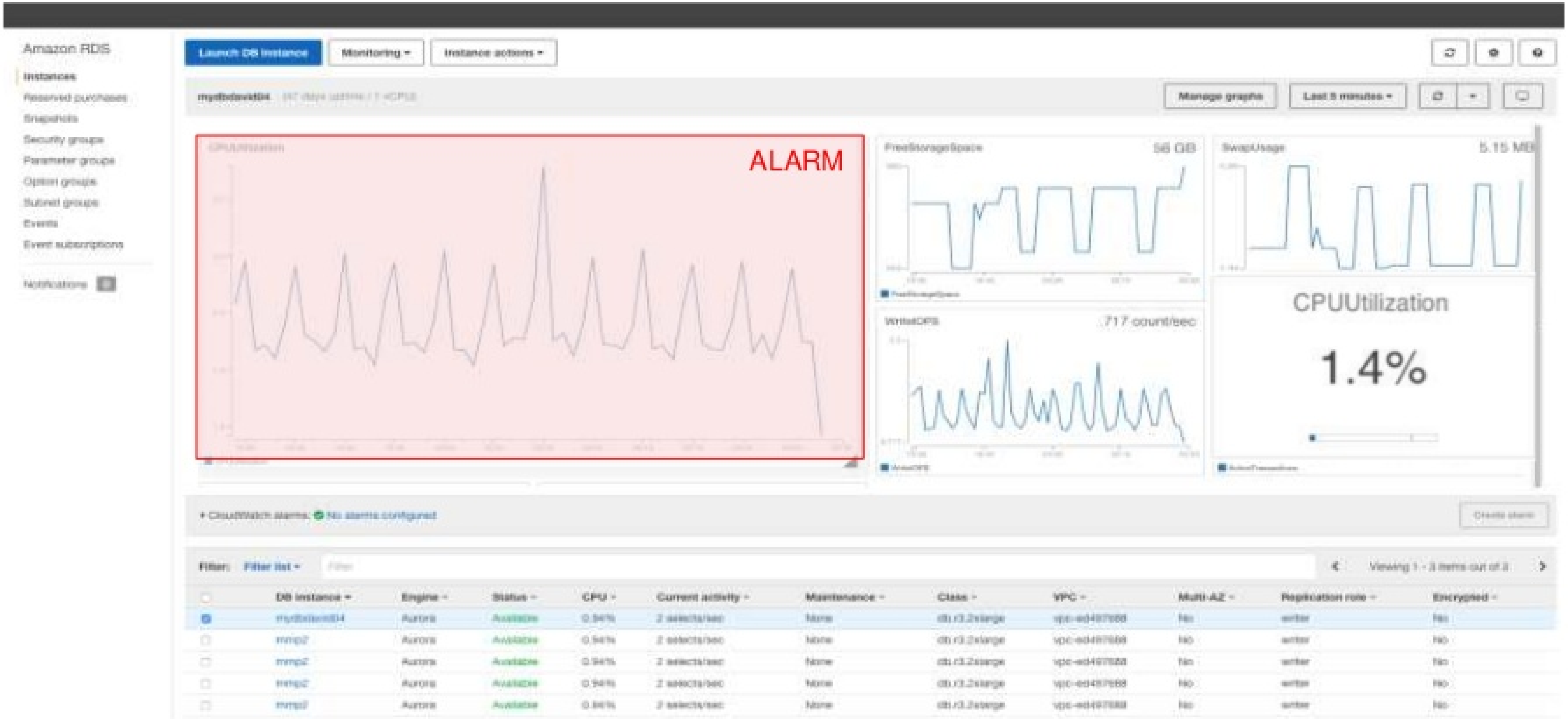
Performance comparison

- Don’t obsess over individual metrics (CPU, IOPS, I/O throughput)
- Focus on what matters—that is, application performance

Other best practices

- Keep query cache on
- Leverage Amazon CloudWatch metrics

Advanced monitoring



50+ system/OS metrics | sorted process list view | 1–60 sec granularity

Alarms on specific metrics | egress to CloudWatch Logs | integration with third-party tools

Important systems and OS metrics

CPU utilization

User
System
Wait
IRQ
Idle
Nice
Steal

Network

Rx per declared *ethn*
Tx per declared *ethn*

Load average

1 min
5 min
15 min

Processes

Sleeping
Running
Total
Stopped
Blocked
Zombie

Process list

Process ID
Process name
VSS
Res
Mem % consumed
CPU % used
CPU time
Parent ID

Memory

Free
Cached
Buffered
Total
Writeback
Inactive
Dirty
Mapped
Slab
Page tables
Huge pages free
Huge pages rsvd
Huge pages surp
Huge pages size
Huge pages total
Swap
Swap free
Swap committed

Device I/O

Read latency
Write latency
Read throughput
Write throughput
Read I/O/sec
Write I/O/sec
Queue depth
Read queue depth
Write queue depth
Free local storage

File system

Used
Total
Used Inodes/%
Max Inodes/%

Important database metrics

- View database level metrics from Aurora and CloudWatch console
- Perform retroactive workload analysis

Select throughput
Select latency

DML throughput
DML latency
Commit throughput
Commit latency

DDL throughput
DDL latency

DB connections
Active connections
Login failures

Buffer cache hit ratio
Resultset cache hit ratio

Deadlocks
Blocked transactions
Failed SQL statements

Replica lag
Replica lag maximum
Replica lag minimum
Free local storage

Beyond benchmarks

If only real-world applications saw benchmark performance

POSSIBLE DISTORTIONS

- Real-world requests contend with each other
- Real-world metadata rarely fits in the data dictionary cache
- Real-world data rarely fits in the buffer cache
- Real-world production databases need to run at high availability

Scaling user connections

Connections	Amazon Aurora	Amazon RDS MySQL 30 K IOPS (single AZ)
50	40,000	10,000
500	71,000	21,000
5,000	110,000	13,000

SysBench OLTP workload

250 tables

UP TO

8x

FASTER

Scaling table count

Number of write operations per second

Tables	Amazon Aurora	MySQL I2.8XL local SSD	MySQL I2.8XL RAM disk	RDS MySQL 30 K IOPS (single AZ)
10	60,000	18,000	22,000	25,000
100	66,000	19,000	24,000	23,000
1,000	64,000	7,000	18,000	8,000
10,000	54,000	4,000	8,000	5,000

SysBench write-only workload

1,000 connections, default settings

UP TO

11x

FASTER

Scaling dataset size

SYSBENCH WRITE-ONLY

DB size	Amazon Aurora	RDS MySQL 30 K IOPS (single AZ)
1 GB	107,000	8,400
10 GB	107,000	2,400
100 GB	101,000	1,500
1 TB	26,000	1,200

UP TO

67x

FASTER

CLOUDHARMONY TPC-C

DB size	Amazon Aurora	RDS MySQL 30K IOPS (single AZ)
80 GB	12,582	585
800 GB	9,406	69

UP TO

136x

FASTER

Running with read replicas

Updates per second	Amazon Aurora	RDS MySQL 30 K IOPS (single AZ)
1,000	2.62 ms	0 s
2,000	3.42 ms	1 s
5,000	3.94 ms	60 s
10,000	5.38 ms	300 s

UP TO

500x

LOWER LAG

SysBench write-only workload
250 tables

How do we achieve these results?

DO LESS WORK

- Do fewer I/Os
- Minimize network packets
- Cache prior results
- Offload the database engine

BE MORE EFFICIENT

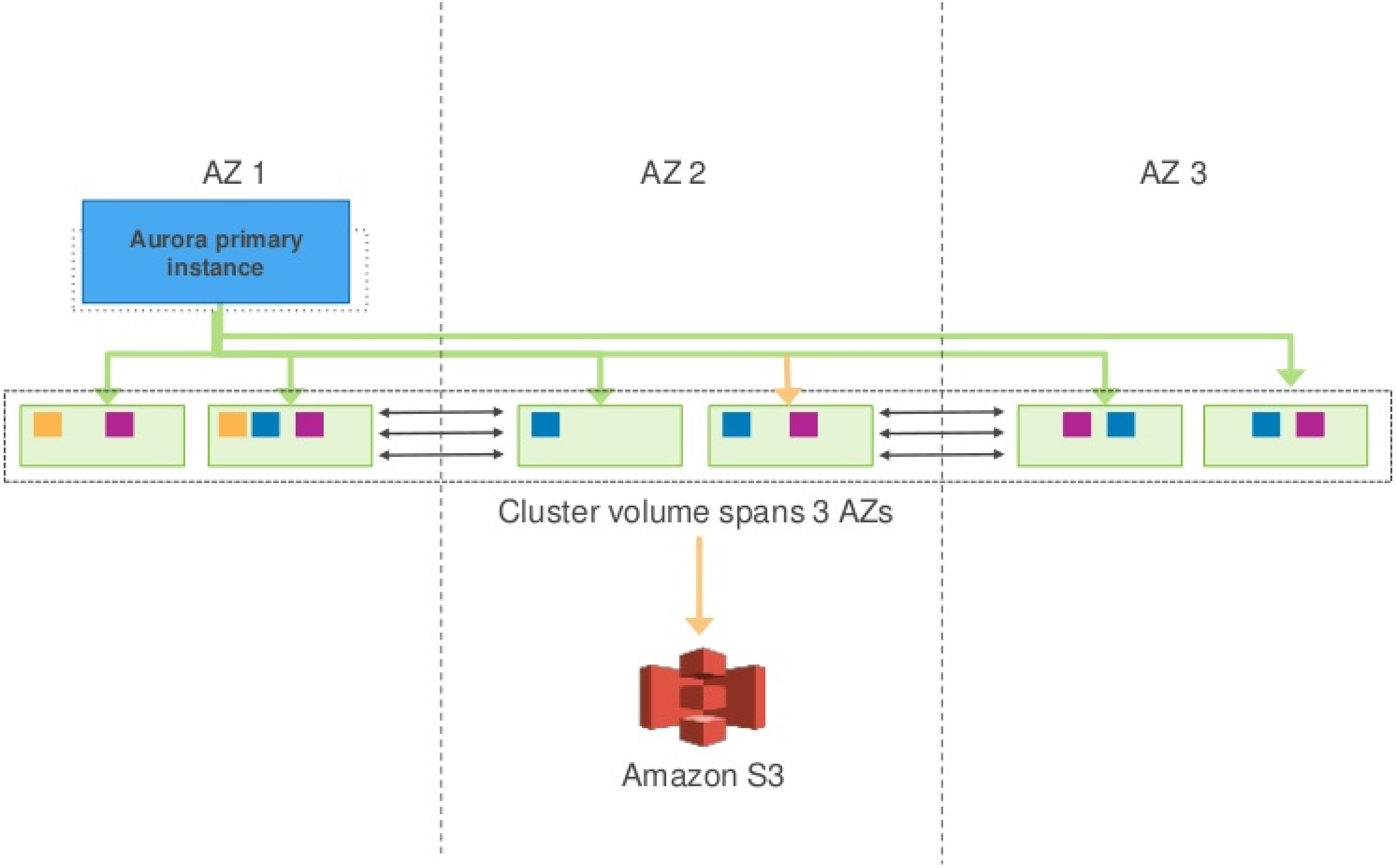
- Process asynchronously
- Reduce latency path
- Use lock-free data structures
- Batch operations together

DATABASES ARE ALL ABOUT **I/O**

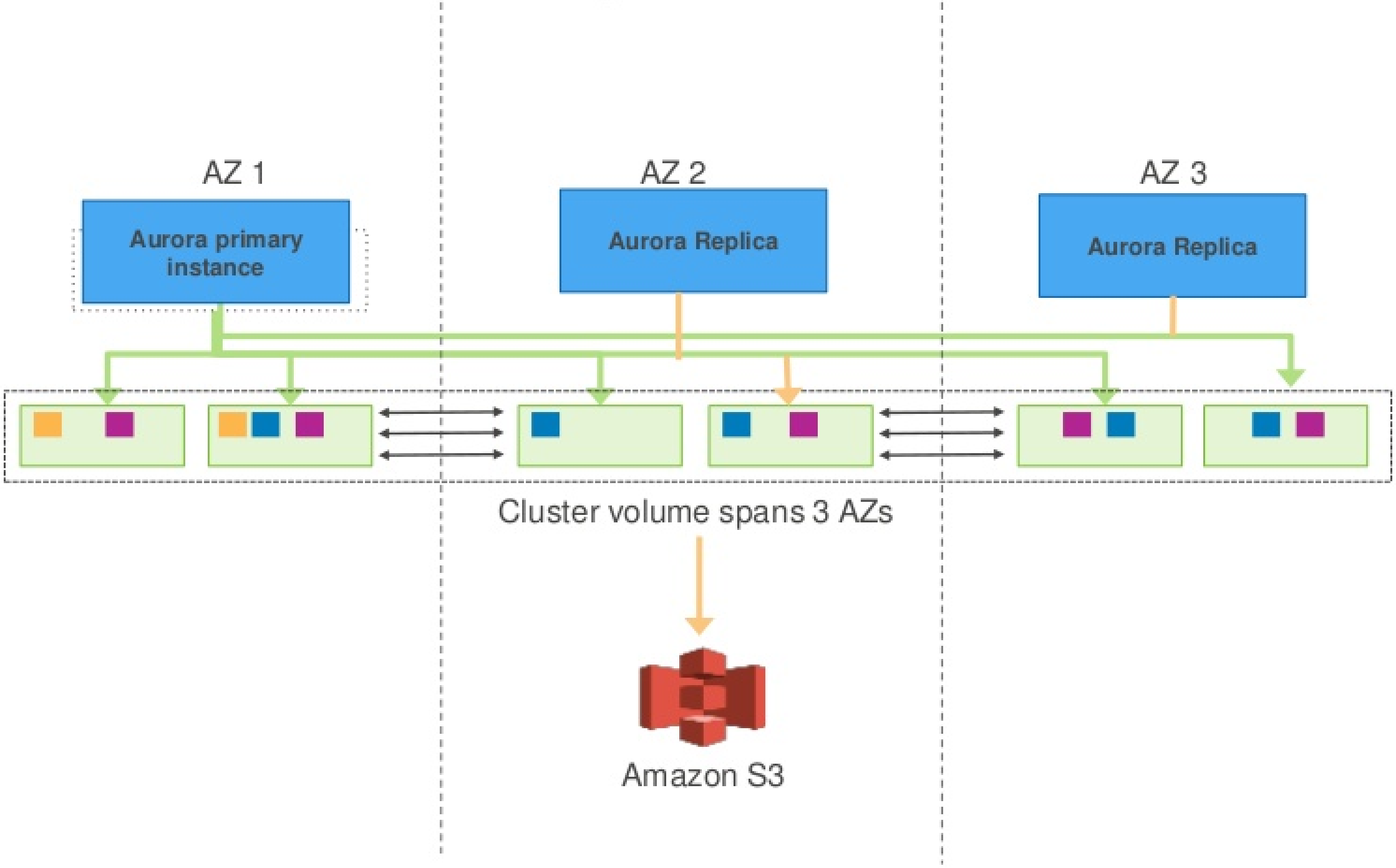
NETWORK-ATTACHED STORAGE IS ALL ABOUT **PACKETS/SECOND**

HIGH-THROUGHPUT PROCESSING DOES NOT ALLOW **CONTEXT SWITCHES**

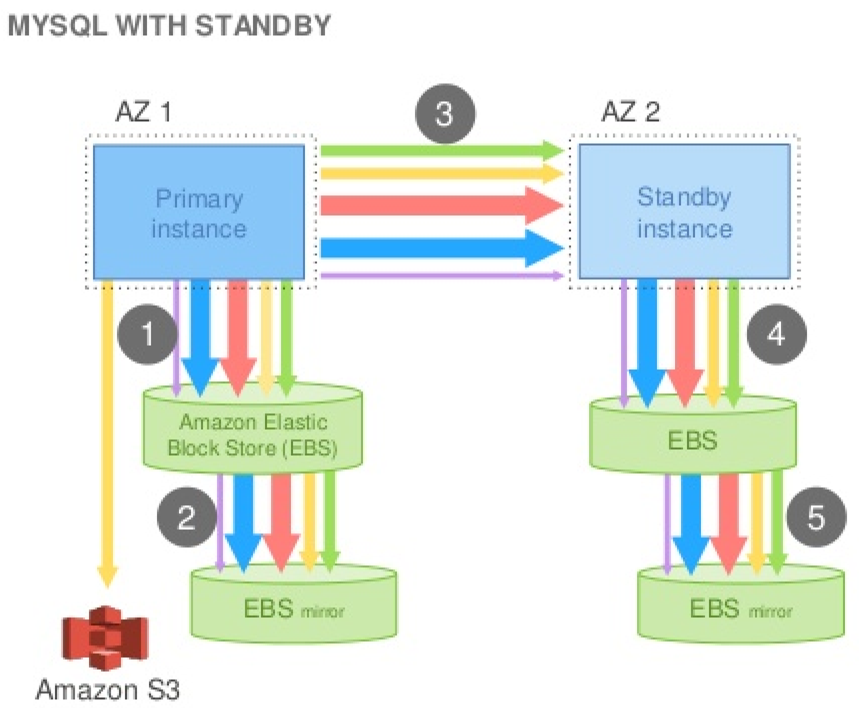
Aurora cluster



Aurora cluster with replicas



I/O traffic in RDS MySQL



I/O FLOW

Issue write to Amazon EBS—EBS issues to mirror, acknowledge when both done

Stages write to standby instance using storage level replication

Issues write to EBS on standby instance

OBSERVATIONS

Steps 1, 3, 5 are sequential and synchronous

This amplifies both latency and jitter

Many types of write operations for each user operation

Have to write data blocks twice to avoid torn write operations

PERFORMANCE

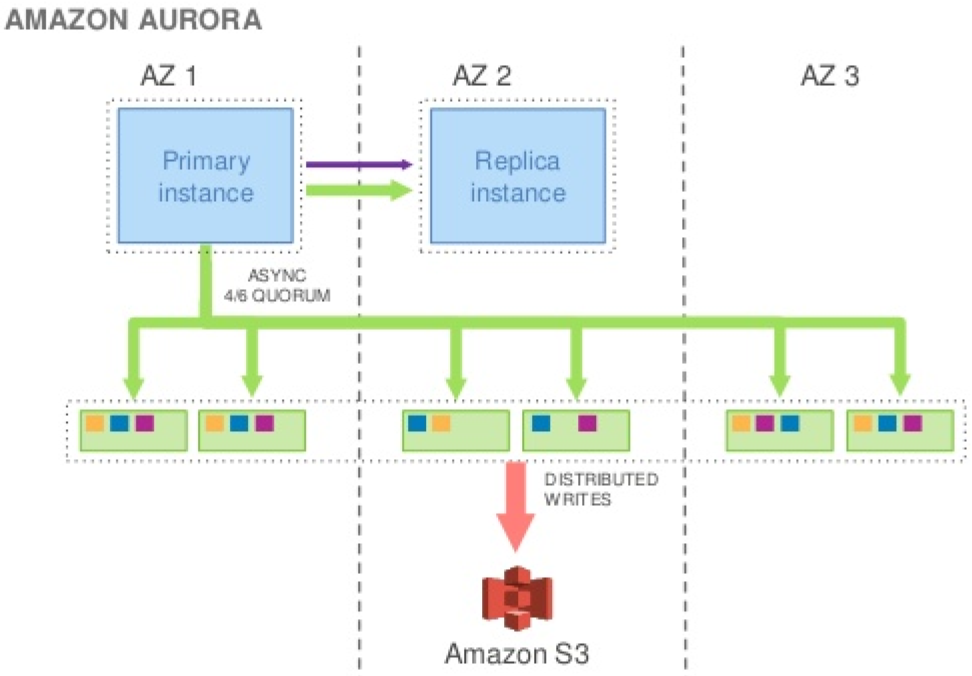
780 K transactions

7,388 K I/Os per million transactions (excludes mirroring, standby)

Average 7.4 I/Os per transaction

30 minute SysBench write-only workload, 100 GB dataset, **RDS Single AZ**, 30 K PIOPS

I/O traffic in Aurora (database)



IO FLOW

Boxcar redo log records—fully ordered by LSN
Shuffle to appropriate segments—partially ordered
Boxcar to storage nodes and issue write operations

OBSERVATIONS

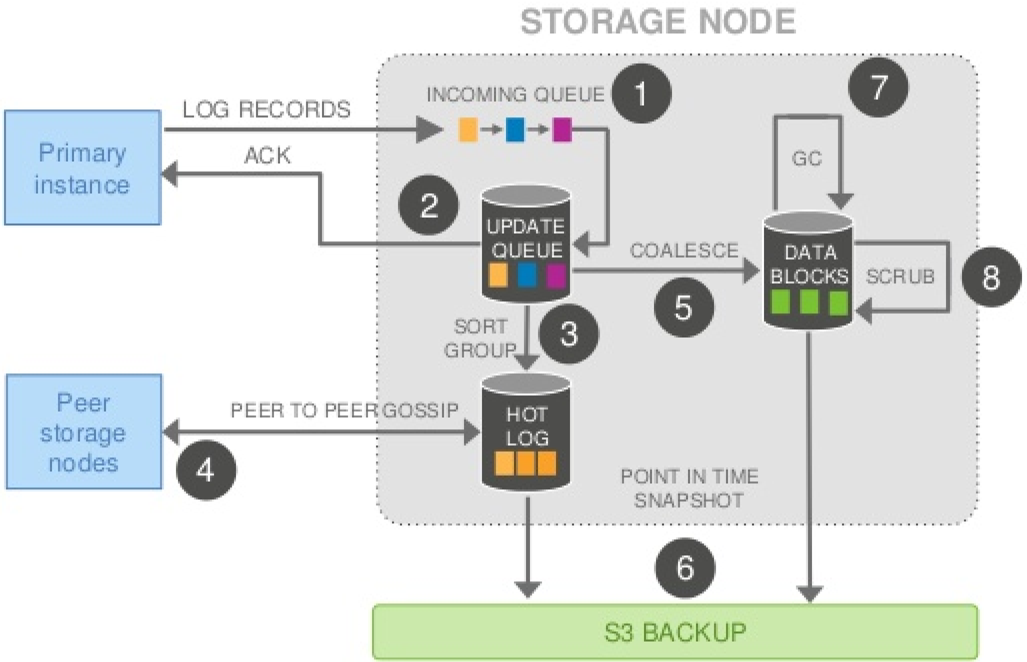
Only write redo log records; all steps asynchronous
No data block writes (checkpoint, cache replacement)
6x more log writes, but 9x less network traffic
Tolerant of network and storage outlier latency

PERFORMANCE

27,378 K transactions	35x MORE
950K I/Os per 1M transactions (6x amplification)	7.7x LESS

30 minute SysBench writeonly workload, 100GB dataset

I/O traffic in Aurora (storage node)



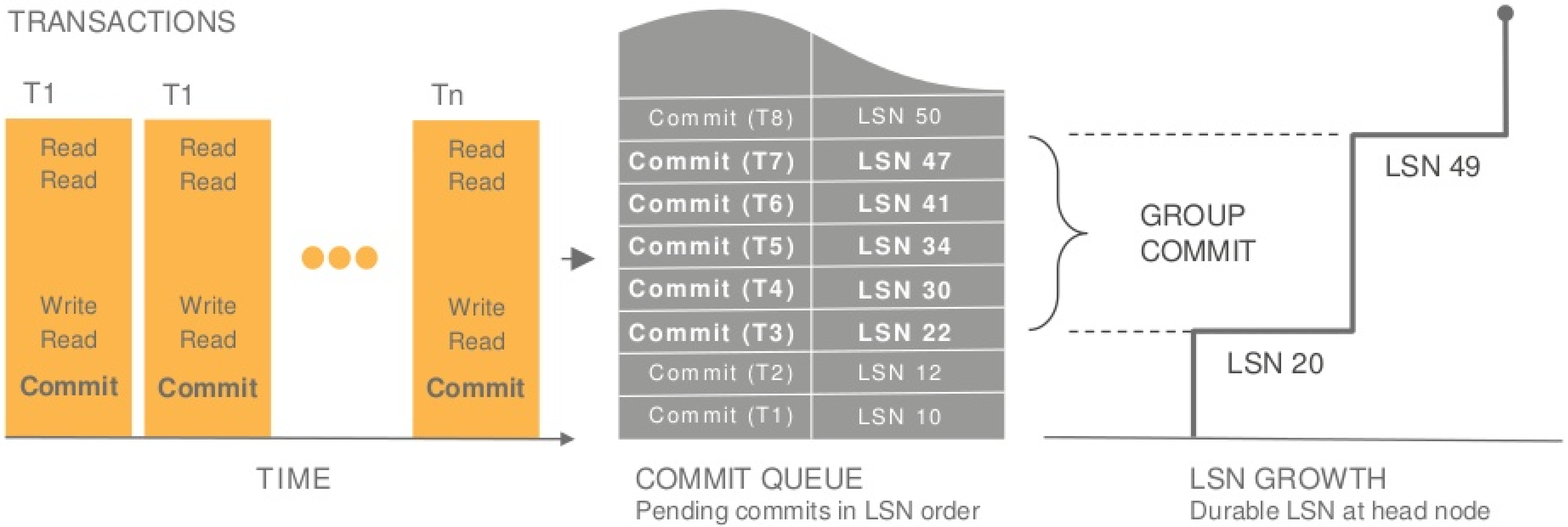
I/O FLOW

1. Receive record and add to in-memory queue
2. Persist record and acknowledge
3. Organize records and identify gaps in log
4. Gossip with peers to fill in holes
5. Coalesce log records into new data block versions
6. Periodically stage log and new block versions to S3
7. Periodically garbage-collect old versions
8. Periodically validate CRC codes on blocks

OBSERVATIONS

All steps are asynchronous
Only steps 1 and 2 are in the foreground latency path
Input queue is **46x less** than MySQL (unamplified, per node)
Favors latency-sensitive operations
Use disk space to buffer against spikes in activity

Asynchronous group commits



TRADITIONAL APPROACH

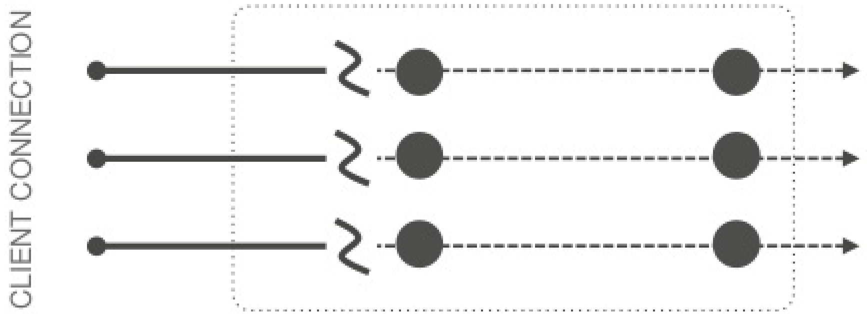
- Maintain a buffer of log records to write out to disk
- Issue write operations when buffer is full, or time out waiting for write operations
- First writer has latency penalty when write rate is low

AMAZON AURORA

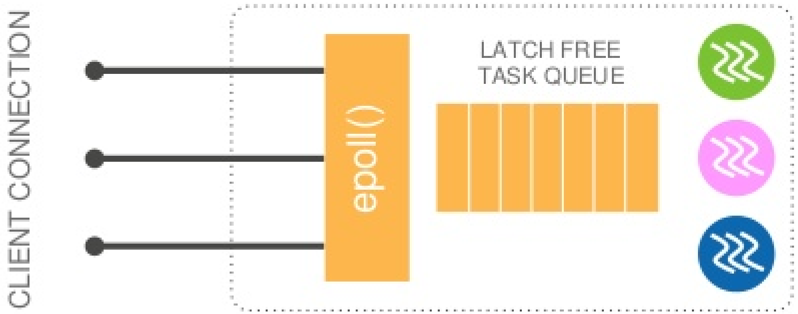
- Request I/O with first write, fill buffer till write picked up
- Individual write durable when 4 of 6 storage nodes acknowledge
- Advance DB durable point up to earliest pending acknowledgement

Adaptive thread pool

MYSQL THREAD MODEL



AURORA THREAD MODEL



Standard MySQL—one thread per connection

Doesn't scale with connection count

MySQL EE—connections assigned to thread group

Requires careful stall threshold tuning

Re-entrant connections multiplexed to active threads

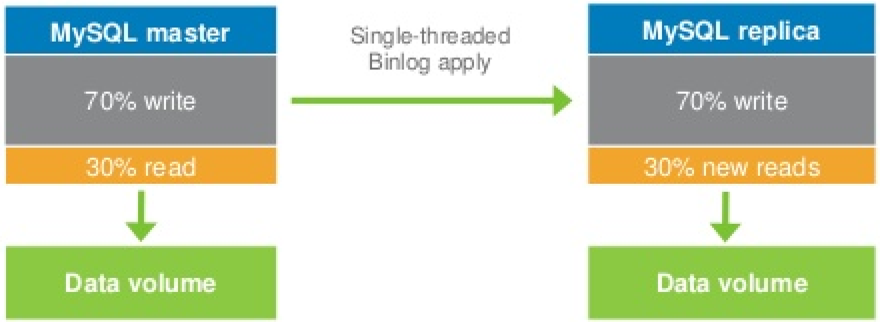
Kernel-space epoll() inserts into latch-free event queue

Dynamically size threads pool

Gracefully handles 5000+ concurrent client sessions on r3.8xl

I/O traffic in Aurora (Aurora Replica)

MYSQL READ SCALING



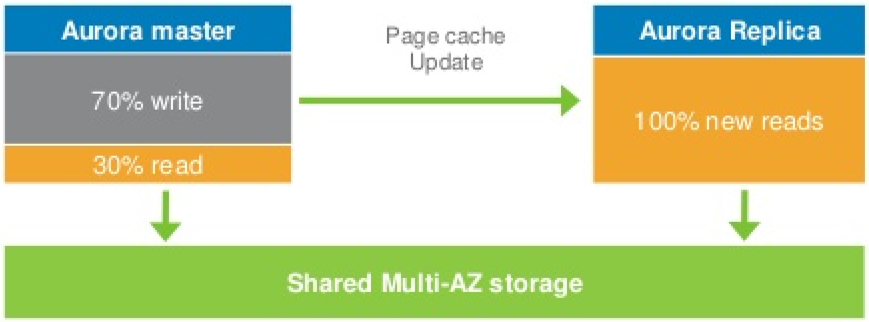
Logical: Ship SQL statements to replica

Write workload similar on both instances

Independent storage

Can result in data drift between master and replica

AMAZON AURORA READ SCALING



Physical: Ship redo from master to replica

Replica shares storage; no writes performed

Cached pages have redo applied

Advance read view when all commits seen

Availability

“Performance only matters if your database is up”

Storage node availability

Quorum system for read/write; latency tolerant

Peer-to-peer gossip replication to fill in holes

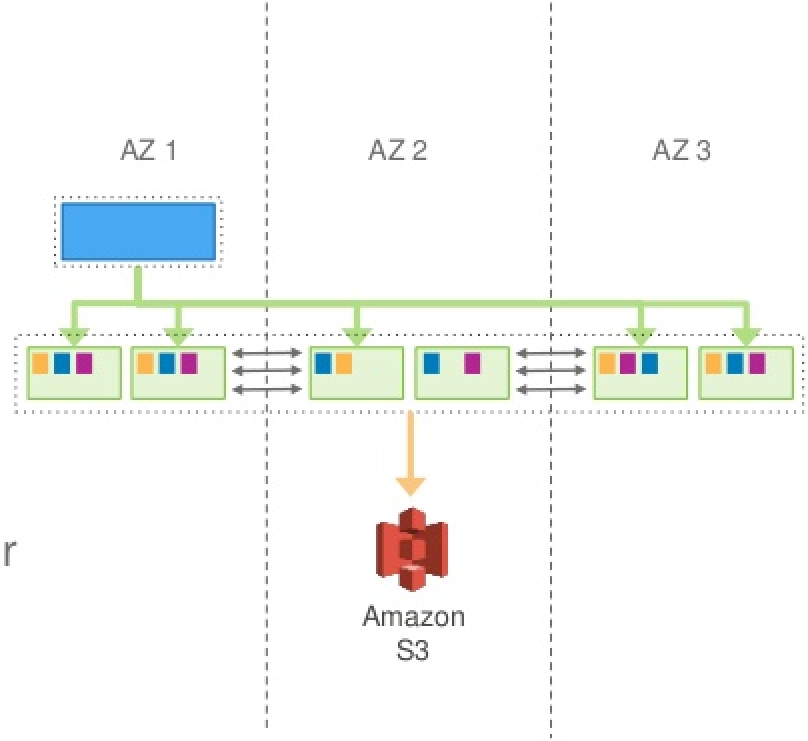
Continuous backup to S3 (designed for 11 9s durability)

Continuous scrubbing of data blocks

Continuous monitoring of nodes and disks for repair

10 GB segments as unit of repair or hotspot rebalance to quickly rebalance load

Quorum membership changes do not stall write operations



Instant crash recovery

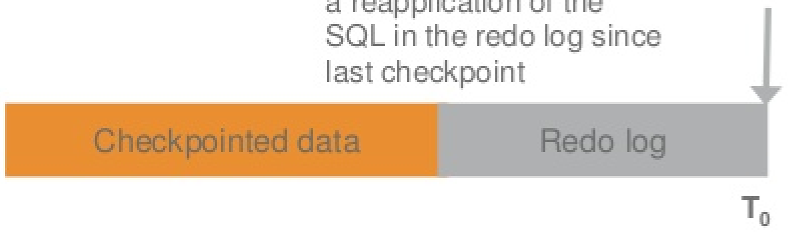
Traditional databases

Have to replay logs since the last checkpoint

Typically 5 minutes between checkpoints

Single-threaded in MySQL; requires a large number of disk accesses

Crash at T_0 requires a reapplication of the SQL in the redo log since last checkpoint



Amazon Aurora

Underlying storage replays redo records on demand as part of a disk read

Parallel, distributed, asynchronous

No replay for startup

Crash at T_0 will result in redo logs being applied to each segment on demand, in parallel, asynchronously

Survivable caches

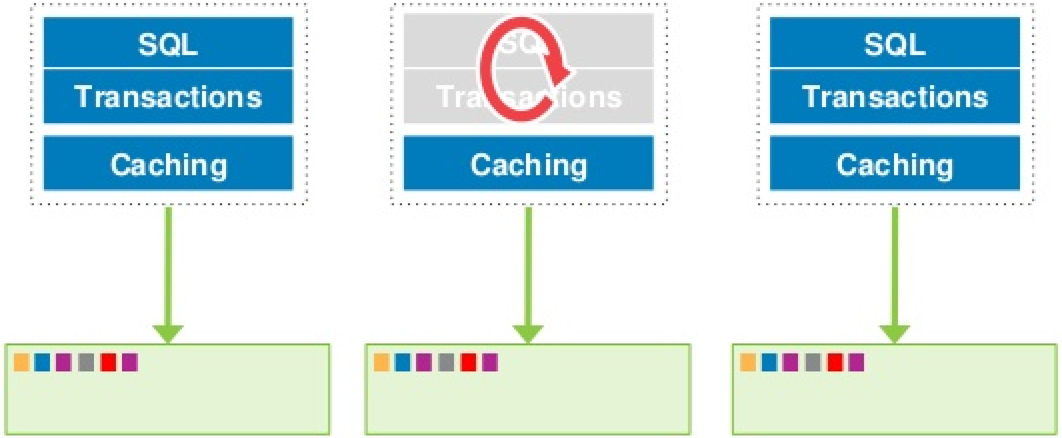
We moved the cache out of the database process

Cache remains warm in the event of a database restart

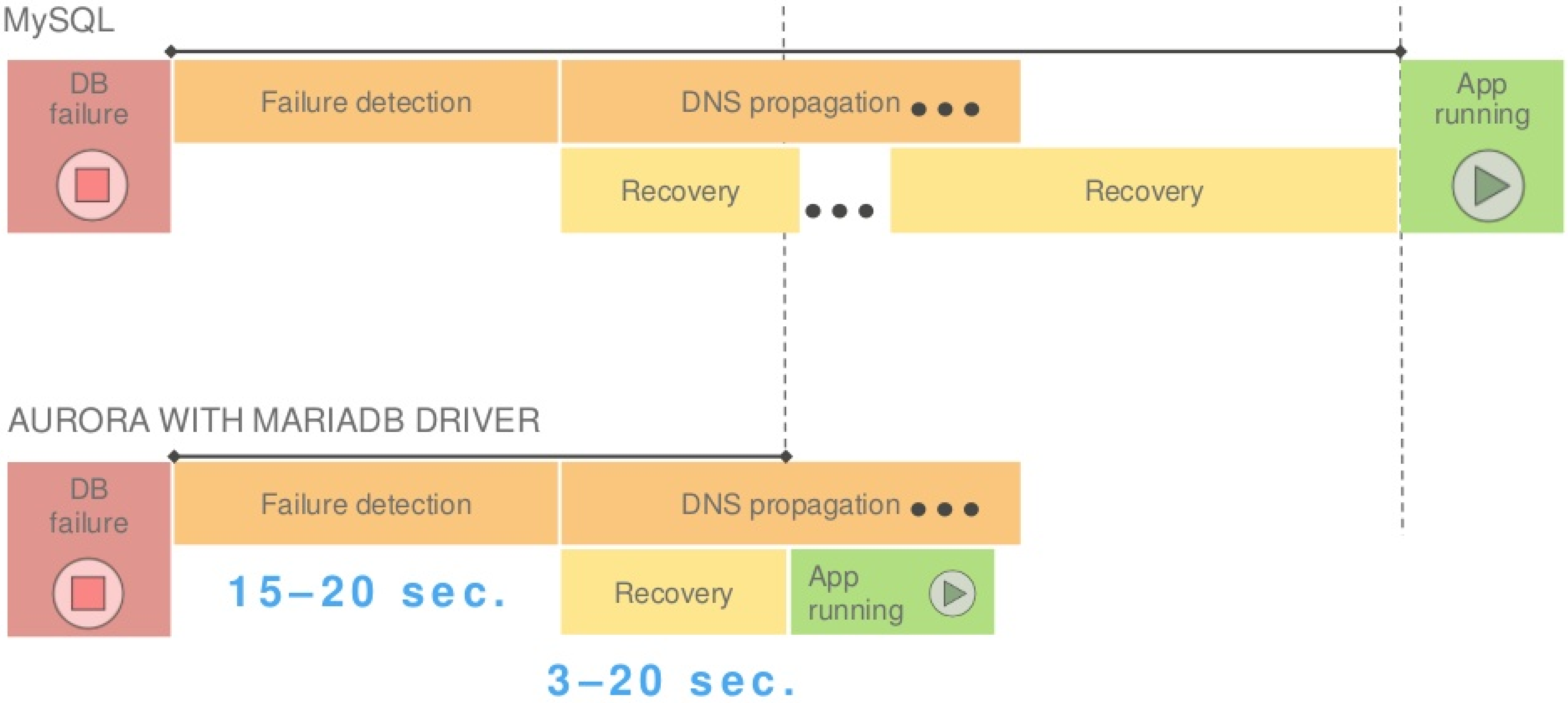
Lets you resume fully loaded operations much faster

Instant crash recovery + survivable cache = quick and easy recovery from DB failures

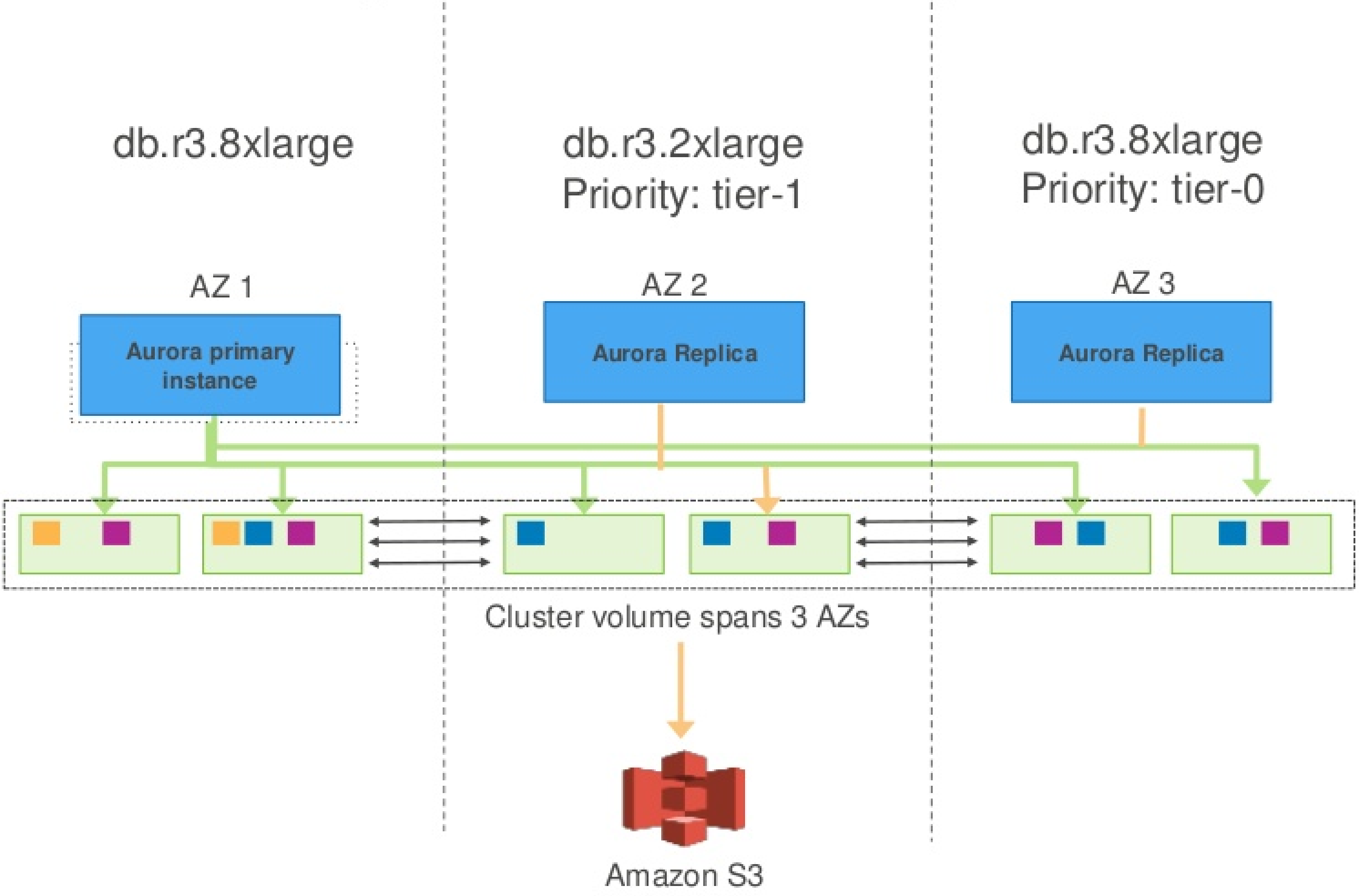
Caching process is outside the DB process and remains warm across a database restart



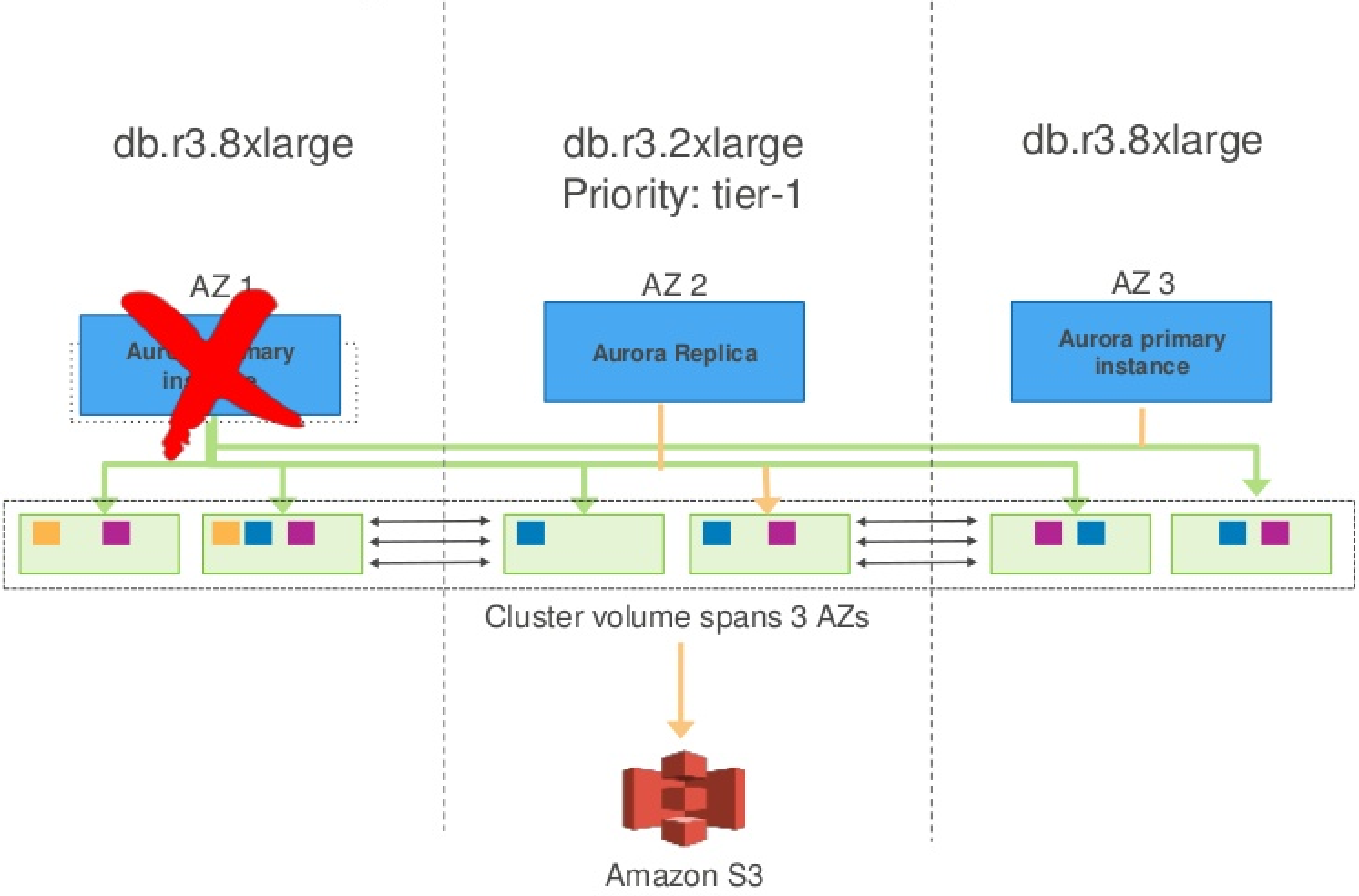
Faster, more predictable failover



High availability with Aurora Replicas



High availability with Aurora Replicas



Simulate failures using SQL

To cause the failure of a component at the database node:

```
ALTER SYSTEM CRASH [{INSTANCE | DISPATCHER | NODE}]
```

To simulate the failure of disks:

```
ALTER SYSTEM SIMULATE percent_failure DISK failure_type IN
[DISK index | NODE index] FOR INTERVAL interval
```

To simulate the failure of networking:

```
ALTER SYSTEM SIMULATE percent_failure NETWORK failure_type
[TO {ALL | read_replica | availability_zone}] FOR INTERVAL interval
```

Thank You.