Pushing past PostgreSQL's vertical scaling limits – Running Amazon RDS

Lessons and guidance on the largest PostgreSQL workloads

Andrei Dukhounik
he/him

Alisdair Owens
he/him

Agenda

- Introduction and Background
- Performance Management
- Patching
- Application Reliability
- Operational Reliability
- Wishlist

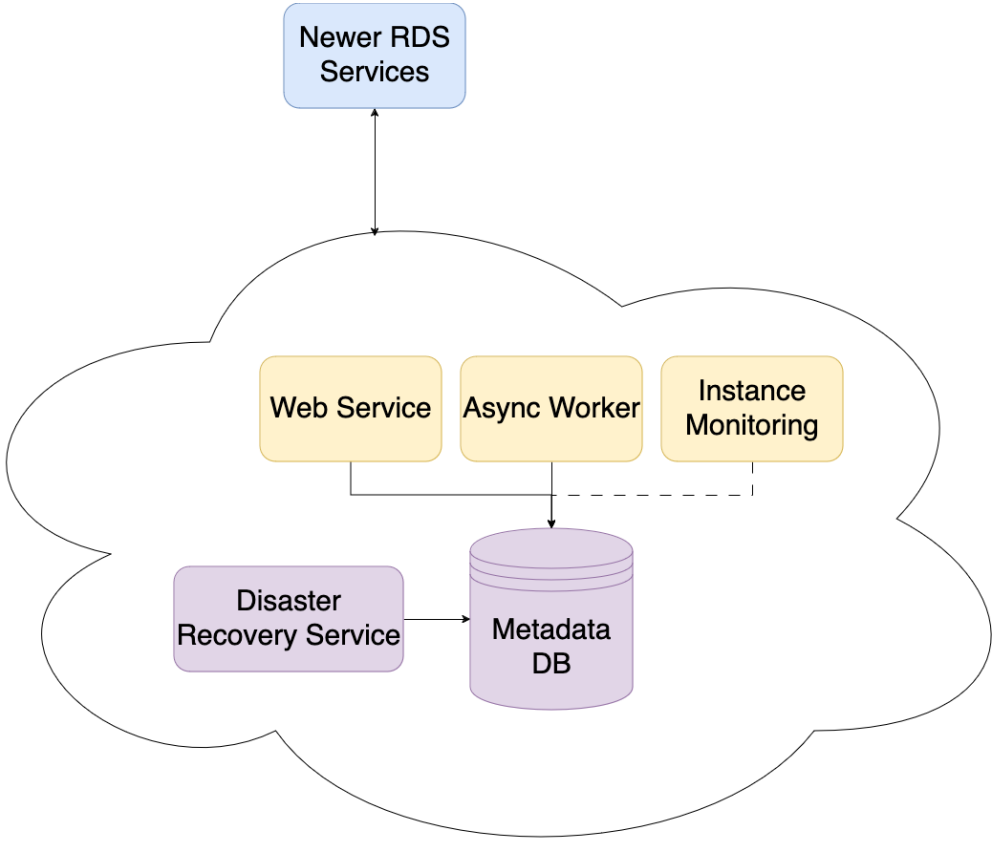
Introduction and Background

Introduction

- Amazon Relational Database Service (Amazon RDS) manages relational databases on behalf of an enormous number of customers.
- One of those customers is ourselves!
  - Core system backed by regional RDS for PostgreSQL DBs (‘RDS in RDS’)
- We’re here to talk through our performance and operability journey with Postgres
  - Along with some architectural background

Architecture

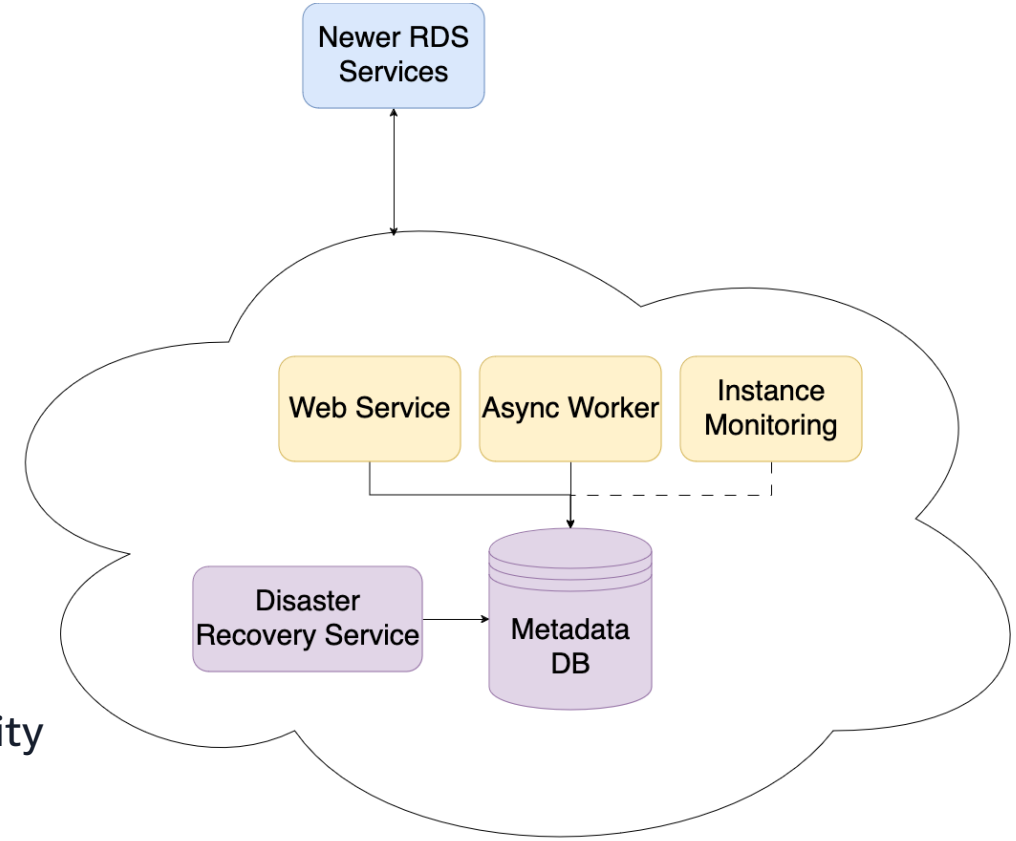
- Classic control plane architecture
  - Front end web service
  - Asynchronous worker
  - Shared metadata DB
- RDS' core control plane relies on an RDS DB
- Newer internal services have no direct access



Self-Hosting

- We get a lot from RDS!
  - Health monitoring
  - Multi-AZ/failover
  - Reliable backups
  - Replica maintenance
  - etc

- We need to supplement it to reliably self-host
  - Complex recovery scenarios rely on core metadata DB availability
  - Instance failover architected to be independent
  - We have an independent internal service to cover complex recovery scenarios



Scale

- Multi-terabyte database
- Very high TPS
- Operating with relatively minimal read I/O
- Mixed workload
  - Large majority of work is small/point lookups/writes
  - But there are some complex, many-join queries that run for > 1 minute
- Mixed table sizes
  - Up to ~TB
- Interruptions of service are unacceptable
  - 24/7 workload

Future Scaling

- RDS continues to grow
  - More and bigger customers
  - More customer automations
  - More features
- Tactical optimizations drop in effectiveness over time
  - And regressions become more impactful
- Read replicas and caching help
  - But variable read latencies cause consistency challenges

Horizontal Scaling

- Wanted to allow unlimited horizontal scaling
- RDS now operates on a sharded model
- Indefinite fleet growth
- Better predictability and testability

How We Got There

Performance Management

Performance Management

- Continuous query optimization
- Bloat management
- Caching
- Use of replicas

Monitoring

Mechanism	 Amazon RDS Performance Insights	 PostgreSQL pg_stat_statements	 Continuous Application Profiling	 Client-side SQL Logging
Characteristics	• Database-level performance stats • Periodic sampling, long-term storage	• Database-level performance stats • Aggregated until reset	• Application-level code profiler • Periodic sampling, stored indefinitely	• SQL text, (redacted) params+timing logged client side • Stored in Amazon Cloudwatch Logs
Benefits	• Zoom into historical events • Information on wait events	• Near-complete SQL list • I/O information • Execution count	• Connect DB + client side • Comprehensive history	• Includes param values • Comprehensive history

Query Optimization

- Fix queries along two axes
  - Total time consumed
  - Frequency of execution
- Partial indexes and partitioning for tables with historical data
- Watch out for ‘Performance cliffs’
  - Query plan changes
  - I/O
  - Lock contention
  - Subtransactions

Query Optimization

- Predictability is key
- Prefer many point lookups over complex queries
  - Even if it’s less efficient on average
- We prefer to avoid extensive use of custom statistics, pg_hint_plan, etc
  - RDS has many developers writing queries
  - So extensive review/support for each query scales badly

A little access doesn’t hurt…

- We have some tables that evolved organically in the past
  - Heavily indexed
  - Frequent access
- Lightweight lock contention drove up tail latencies
  - Database engineers identified a slow path involving large index counts
    - Driving contention on LWLocks
  - And a recent patch that improved latencies for large lock counts
- 50% drop in peak lock rates when running a version with the patch applied

Bloat

- Bloat maintenance is critical for controlling I/O
  - Particularly on older versions
  - Indexes could be 10x+ bloated

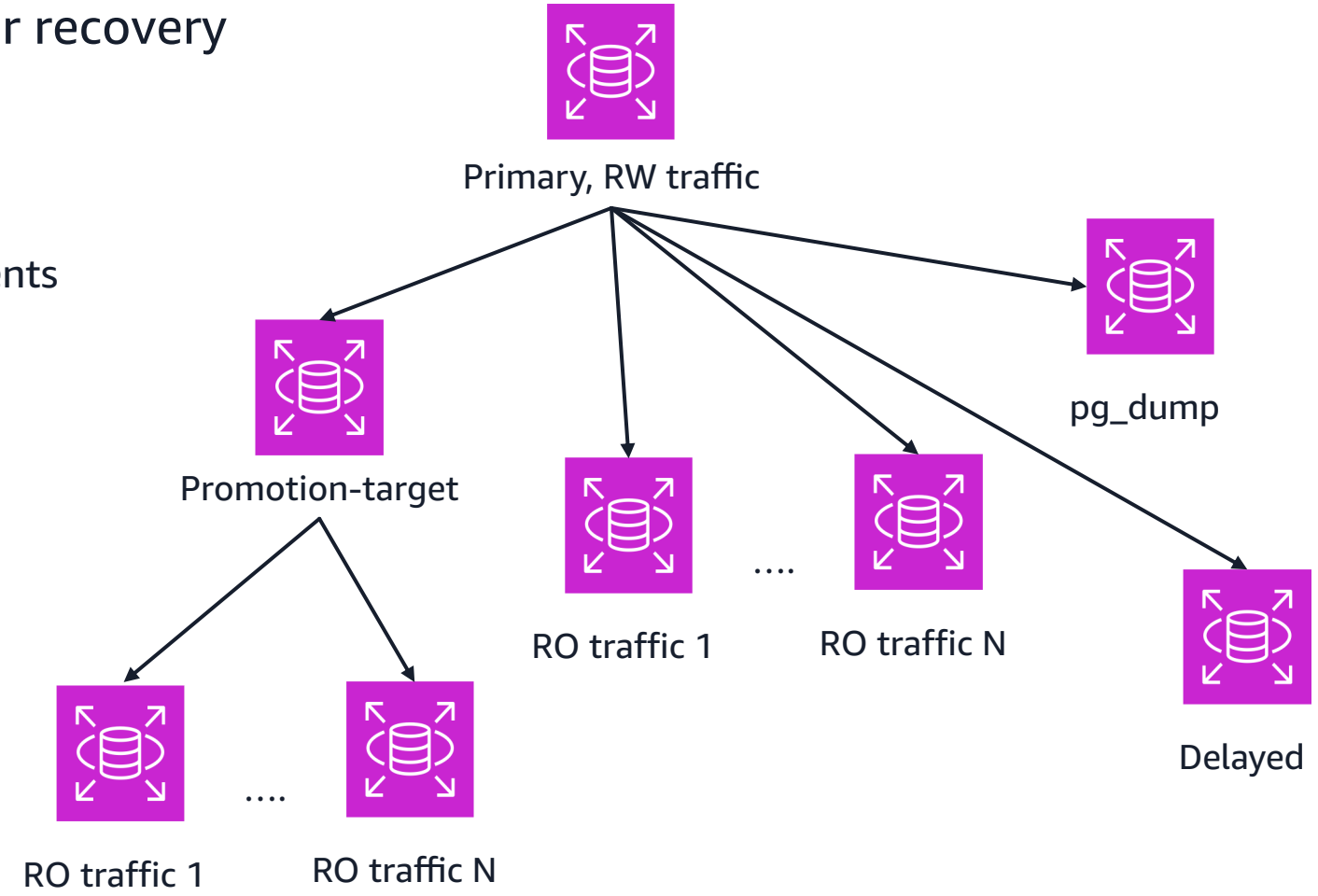
- Managing bloat
  - Control autovacuum
  - Limit table size fluctuations
  - Partitioning
  - Periodic maintenance
    - REINDEX CONCURRENTLY
    - pg_repack
    - VACUUM FULL

Caching

- An obvious win
- But watch out for the ‘thundering herd’
  - Prefer request-level caches and small shared caches
- Favor predictability over peak performance

Replicas

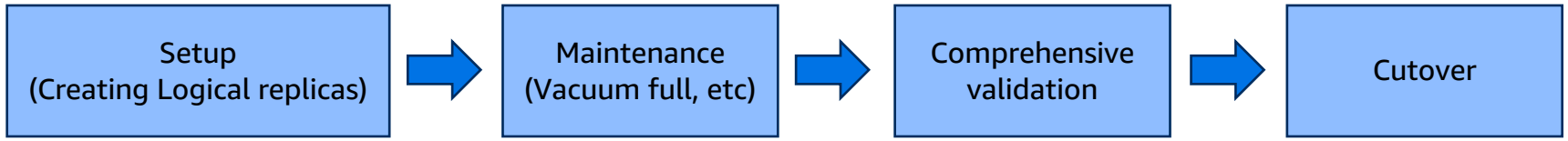
- Replicas for performance and disaster recovery
- Replicas help horizontal scaling
- But are a risk factor for consistency
  - Regressions may appear during latency events



Patching

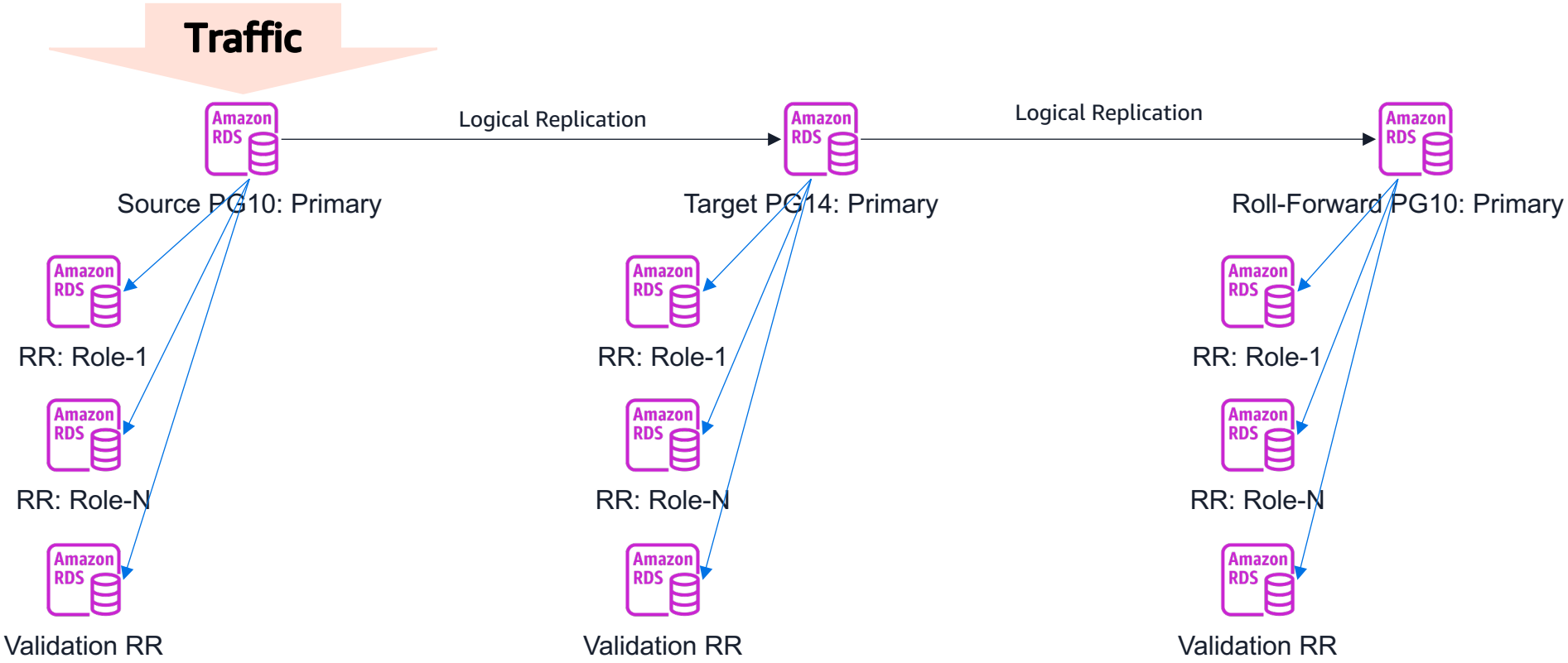
Major Version Upgrade

- Logical replication seeded from Amazon RDS Snapshot
  - Eliminates vacuum burden
- Database maintenance before cutover
- Comprehensive validation
- Performance improvements testing
- Last MVU upgrade: pg10.16 -> pg14.5 resulted in notable performance benefits



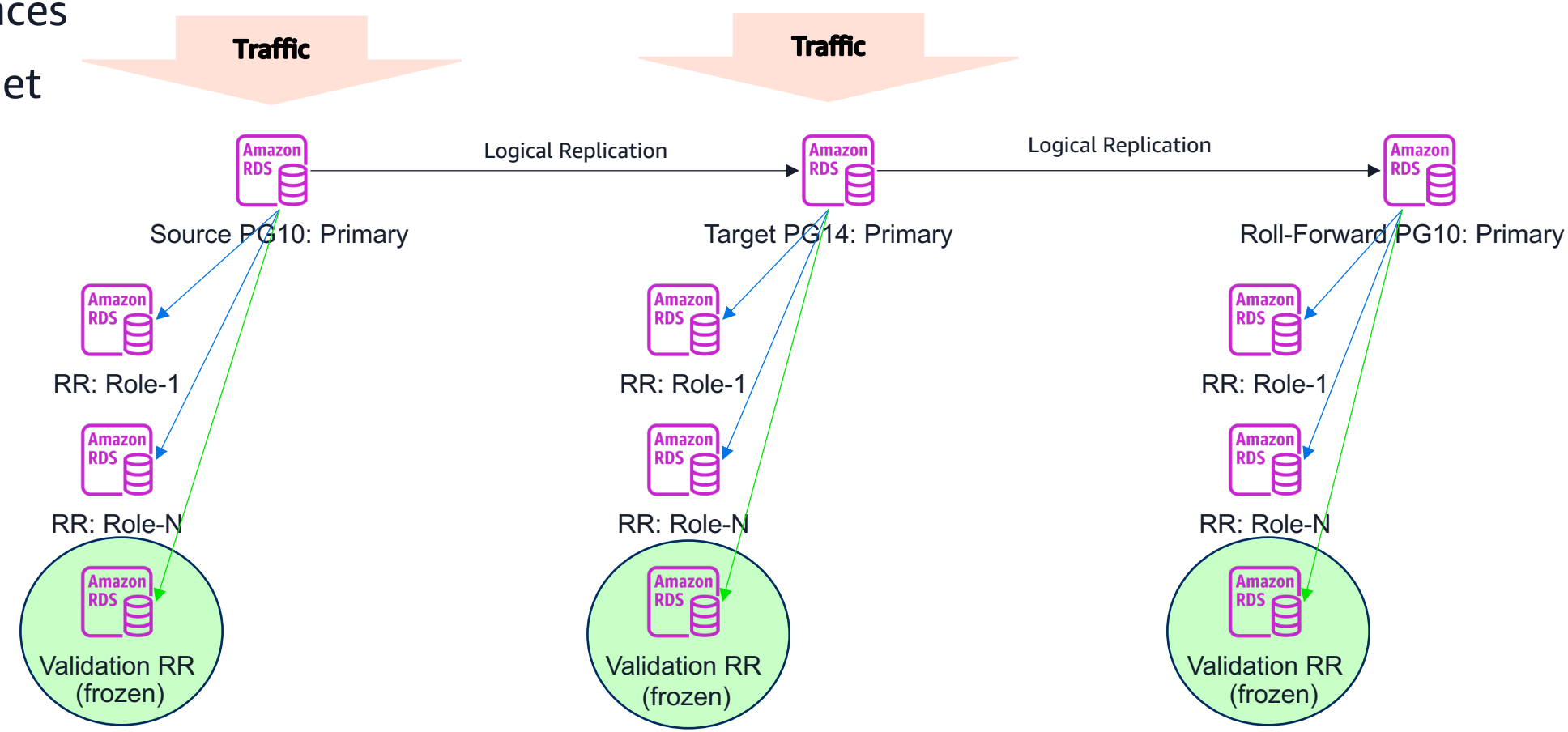
Major Version Upgrade: Setup

- Target and Roll-forward are logical replicas
- Created from RDS snapshot and logical replication resumed from the recovery point
- Roll-forward as a rollback target



Major Version Upgrade: Maintenance and Validation

- Maintenance: Vacuum full, changing column types, adding new indexes, etc
- Validation: “freeze” Validation RRs and then compare hashes of each table in three DBs
- Sync PG sequences
- Cutover to Target



Major Version Upgrade: Performance

- Performance testing in advance
  - Capture load: capture all queries to DB with their timing (!) via client level logging
  - Replay tool: replay captured load with similar simulated timing and concurrency to restored RDS snapshots on current MV and target MV
  - Result: prediction of 10-12% relative improvement (decrease)

- PG10 -> PG14 performance improvements
  - CPU: 13% relative improvement
  - IOPS (thanks to maintenance)
    - ReadIOPS decrease by 95%
    - WriteIOPS no change

Application Reliability

Application Reliability

1

Database-level timeouts

- Statement timeout
- Idle-in-transaction timeout
- Lock timeout

2

Connection-depletion avoidance

- Avoid network activity inside transactions
- Client circuit breakers
- Limit application-level pool consumption per workload

3

Query performance Management

- Vigilance on plan changes
- Rewrite unreliable queries
- Avoid subtransactions

Operational Reliability

Schema Changes

- Not all changes are allowed
  - Yes: add new column, create new index concurrently, etc
  - No: transactional DDL, change column type, naïve constraint changes, etc
  - Many changes possible if we use multiple steps
    - Add constraint with NOT VALID, then VALIDATE.
- Setup
  - Tests for spotting “bad” changes
  - Killing blocking service transactions to allow DDL changes to proceed
    - Otherwise all other operations may pile up in a queue waiting for DDL

Schema Changes: Killing long running transactions to allow DDL changes to proceed

```
select query, wait_event, wait_event_type, query_start from pg_stat_activity;
+-----------------------------------------------------------------------------+-------------------+---------------+-------------------------------+
|query                                                                        |wait_event         |wait_event_type|query_start                    |
+-----------------------------------------------------------------------------+-------------------+---------------+-------------------------------+
|SELECT a, b, c from schema_name.table_name WHERE... (slow query)             |ClientWrite        |Client         |2024-05-16 22:05:00.000000 +00:00|
|SELECT * FROM schema_name.table_name WHERE ...                               |relation           |Lock           |2024-05-16 22:05:01.000000 +00:00|
|ALTER TABLE schema_name.table_name ADD ...                                   |relation           |Lock           |2024-05-16 22:05:05.000000 +00:00|
|SELECT * FROM schema_name.table_name WHERE ...                               |relation           |Lock           |2024-05-16 22:05:05.500000 +00:00|
|UPDATE schema_name.table_name WHERE ...                                      |relation           |Lock           |2024-05-16 22:05:06.000000 +00:00|
|SELECT * FROM schema_name.table_name WHERE ...                               |relation           |Lock           |2024-05-16 22:05:06.000000 +00:00|
|INSERT INTO schema_name.table_name VALUES ...                                |relation           |Lock           |2024-05-16 22:05:06.100000 +00:00|
|SELECT * FROM schema_name.table_name WHERE ...                               |relation           |Lock           |2024-05-16 22:05:06.200000 +00:00|
|UPDATE schema_name.table_name WHERE ...                                      |relation           |Lock           |2024-05-16 22:05:06.200000 +00:00|
|SELECT * FROM schema_name.table_name WHERE ...                               |relation           |Lock           |2024-05-16 22:05:06.300000 +00:00|
|INSERT INTO schema_name.table_name VALUES ...                                |relation           |Lock           |2024-05-16 22:05:06.300000 +00:00|
|SELECT * FROM schema_name.table_name WHERE ...                               |relation           |Lock           |2024-05-16 22:05:06.400000 +00:00|
|UPDATE schema_name.table_name WHERE ...                                      |relation           |Lock           |2024-05-16 22:05:06.400000 +00:00|
... and more queries piling up ...
+-----------------------------------------------------------------------------+-------------------+---------------+-------------------------------+
```

Vacuum management

- Timeouts, alarming on long-running transactions, txids, etc.
- To supplement physical backups, we take frequent logical backups via pg_dump
  - Slow
  - Vacuum-blocking
- Also vacuum-blocking from replica
- Dump-dedicated replica
  - Pause replication during dump

Wishlist

Wishlist

1

Operational Predictability

- User-level resource limits
- Plan stability
- Transaction timeouts
- Bulletproof logical replication
- Enforced ‘read-only’ switch
- ‘safe’ schema changes

2

Performance

- Connection-tolerance
- Multi-thread-per-table pg_dump
- Incremental materialized view refresh

3

Insight

- Daily pg_stat_statements partitions
- More logging options for operations that generate excess workload
- Historical deadlock information

Thank you!

Andrei Dukhounik

dukhouni@amazon.com

Alisdair Owens

alow@amazon.com