All the roads that lead to Amazon Aurora

John Russell (he/him)

Aurora Developer Advocate
AWS

Dustin Brown (he/him)

Sr SMB Database Specialist SA
AWS

Who are we?

- John Russell:
  - AWS Developer Advocate for Aurora. Gathering community feedback on developer experience, doing outreach.
  - Former documentation writer for Aurora, MySQL, Oracle.

- Dustin Brown:
  - AWS Database Solutions Architect. Providing customers with prescriptive guidance for database architecture and migration patterns to Aurora.
  - Former DBA and technical leader for various companies including the airline, real estate, and genealogy industries.

What is Amazon Aurora?

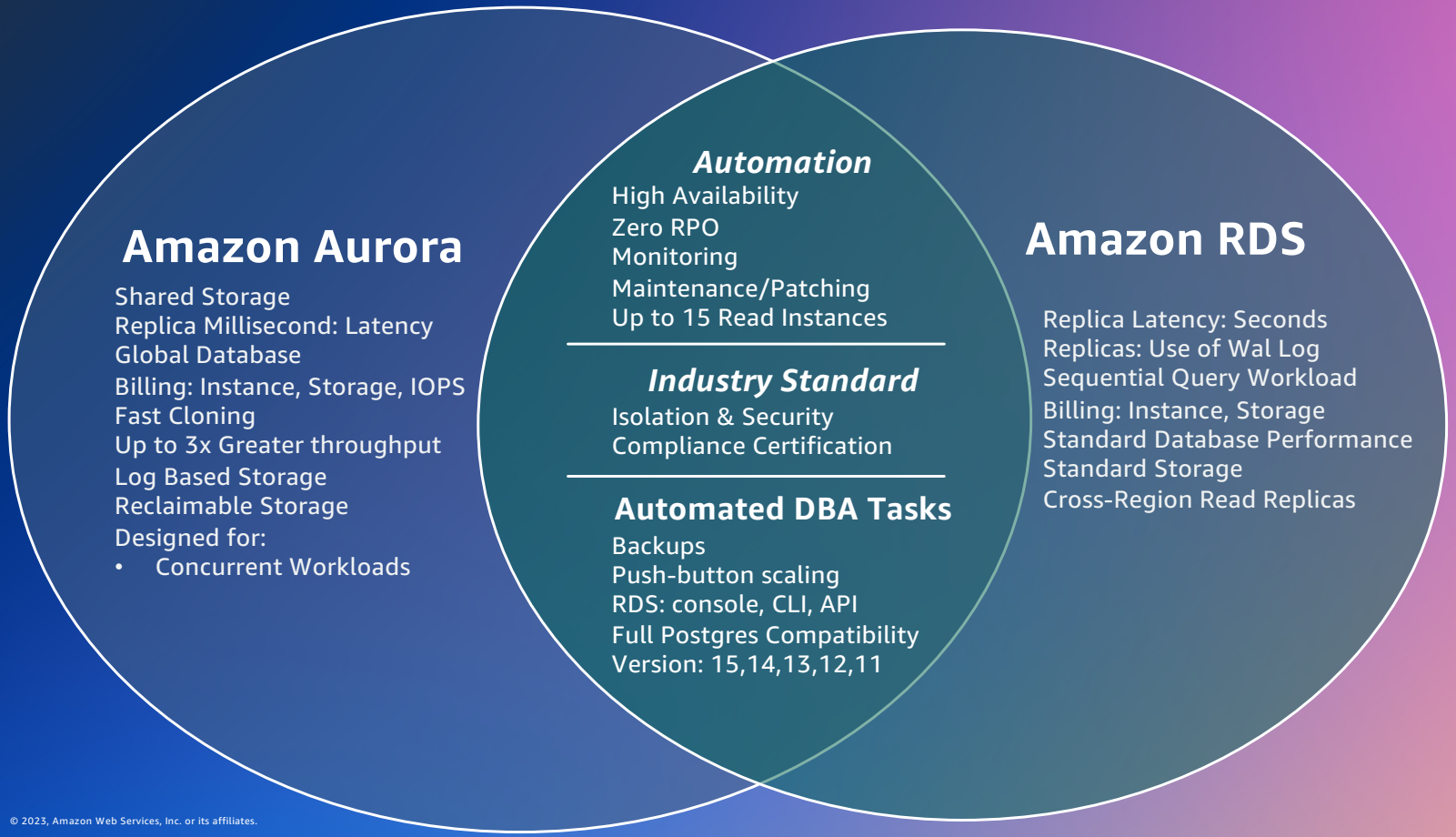
- Amazon Aurora is a relational database. The PostgreSQL-compatible edition is built on the community Postgres engine.
- It runs on AWS as a managed service.
- It features built-in physical replication, continuous backup, multi-AZ high availability, geo-replication, and all sorts of scalability features.
- The architectural innovations are based on the principle, “separation of compute and storage”.

What is this talk about?

- If you end up running Aurora PostgreSQL-compatible edition, that means you've embarked on one, two, or all three of these journeys:
  - Porting your schema & application from another database engine to the open source PostgreSQL engine.
  - Moving from on-premises deployment to a managed service on AWS.
  - Optimizing your topology, HA strategy, and workload for the Aurora architecture.
- This talk will give you guidance to point you in the right direction on each journey. To save you time, effort, $$$, and surprises.

Journey #1: Porting & Migration

AWS Relational Database Service



Decision Process to Pick Aurora for your Workload

SQL

```
SELECT c, c.name, o.orders_product
FROM customers c
JOIN orders o ON c.id = o.customer_id
WHERE c.country = 'USA'
LIMIT 100;
```

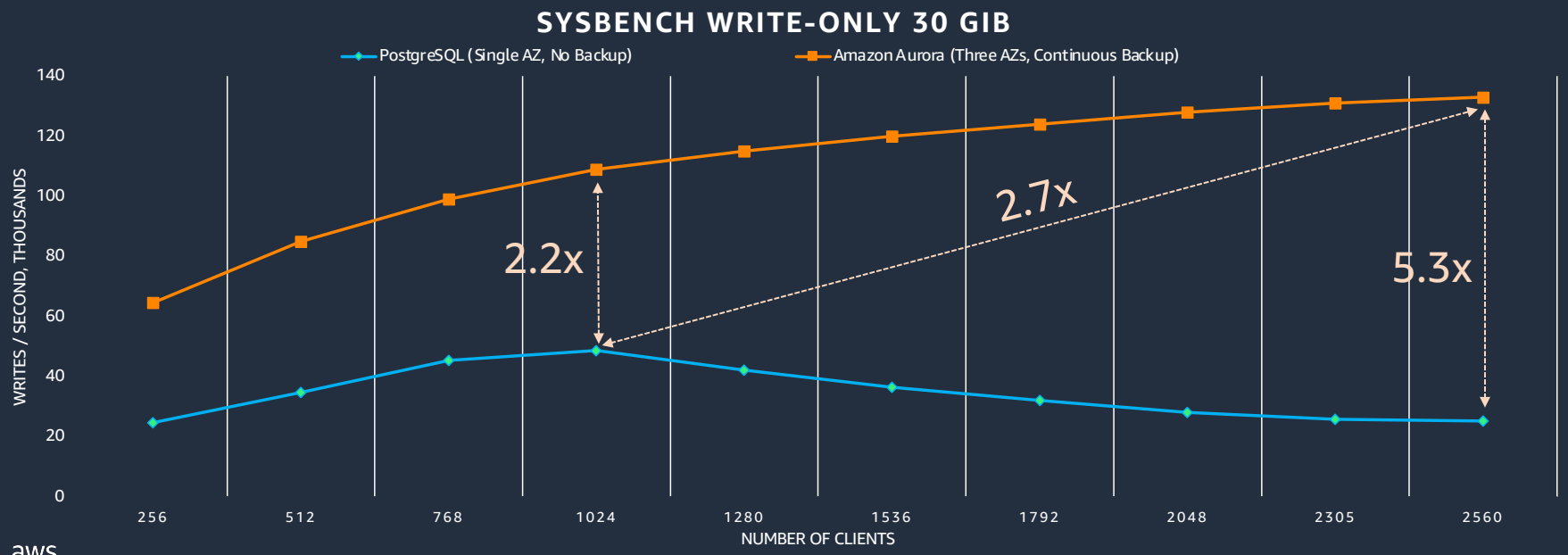
Optimization

- JOIN: customer_id foreign key index
- WHERE: filters on country
- LIMIT: less than 10 orders

Aurora Pattern	Aurora Anti-Pattern
OLTP	OLAP/ETL
Heavy Read Traffic	Light Read Traffic
Concurrent Workload	Sequential
(Read Query) Cached Workloads	Read From Disk
Short Transactions	Long Running Transaction Idle-In-Transactions Heavy Joining
Combine Queries into Single Transactions	
> 100 GB data Set	< 10 GB Data Set
> 16 Parallel Threads	Low Concurrency

Aurora throughput for PostgreSQL Sysbench

Amazon Aurora delivers >2x the absolute peak of PostgreSQL and 5x at high client counts



Porting PostgreSQL Applications to Aurora

- Aurora supports the full SQL dialect of the corresponding community PostgreSQL version, for versions 11-15.
- Each Aurora version has a set of supported extensions and foreign data wrappers.
  - Only trusted languages allowed.
  - Previously: AWS was the gatekeeper.
  - Now: the Trusted Language Extensions open source project lets you write your own extensions or use ones from the community.
- Read-write splitting is good for read-intensive applications – via application code, ORM, or proxy layer.

Trusted Language Extensions (TLE)

- Currently, there are ~85 PostgreSQL extensions approved for use with Aurora.
- Want to write your own and run it with Aurora? Use TLE.
- TLE: open-source SDK for writing extensions in trusted languages.
- Write in PLPgSQL, SQL, Javascript, Perl, Tcl. The community is working on Rust support.
- Buggy extensions can’t harm the server or other connections.
- You can define the upgrade path between extension versions.
- You can create a TLE wrapper for an extension from someone else.

Migrating schema & data into Aurora

- From on-premises PostgreSQL or RDS PostgreSQL: native tools
  - Native backup & restore: pg_dump/pg_restore.
  - Logical replication with Aurora as the destination.

- AWS purpose-built migration tools:
  - For most engines, you can use the Schema Conversion Tool (SCT) to convert tables to a PostgreSQL-compatible schema.
  - Data Migration Service (DMS) can transfer the data from one engine to another. DMS can also perform CDC to enable up-to-date syncing until switchover.

Migrate and Modernize Oracle and SQL Server Workloads and their Applications to Aurora

What's the effort?

Steps in migration	Effort breakdown
1. "Future state" architectural design	2%
2. Database schema conversion	14%
3. Application conversion / remediation	25%
4. Scripts/ETL/reports conversion	7%
5. Integration with 3rd party applications	3%
6. Data migration mechanism	4%
7. Testing and bug fixing	29%
8. Performance tuning	2%
9. Integration and deployment	7%
10. Documentation and knowledge transfer	2%
11. Project management / version control	2%
12. Post production support	3%

Porting T-SQL Applications to Aurora + Babelfish

- To avoid rewriting a SQL Server application written in T-SQL, Babelfish can help:
  - You keep the majority of the T-SQL schema and SQL statements.
  - Aurora does the DB processing behind the scenes.
- The Babelfish Compass tool analyzes the T-SQL source, identifies supported/unsupported things, estimates the effort to rewrite.
- You use Aurora and Postgres features for administration tasks such as backups and high availability.

Journey #2: On-Premises to Managed Service

Getting Started with Aurora as a Managed Service

- You might have to revisit long-held assumptions. There's a learning curve.

- No login to the host that runs the DB server.
  - You don't edit the configuration file directly.
  - You don't run any co-located tools.

- Management is through the AWS console GUI. For automation or devops, the AWS CLI has commands that do all the same things. Both are built on a management API that has many language SDKs (e.g. boto3 for Python).

Considerations for DBAs: “do less”

- Storage capacity for table data isn’t a day-to-day concern.
- Consider Aurora Serverless v2 to auto-adjust instance capacity.
- Backups happen continuously, in the background.
  - Now responsibilities are how/when to restore, high-level DR strategy.

Considerations for DBAs: “do things differently”

- Configuration Management via parameter groups.
- Database capacity and health involves multiple DB instances.
  - Aurora reader instances instead of, or alongside, logical replication.
  - Aurora global database: identical data in multiple AWS Regions.
- Learning other AWS services that integrate with Aurora.

Considerations for Developers

- Super-easy to spin up a new system preloaded with data.
- Super-easy to change capacity to match your current needs.
  - Manually by changing instance class, or automatically via Serverless v2.
- Connection considerations are different:
  - Network setup to be able to connect securely.
  - Read-write splitting for read-intensive applications. (2 endpoints.)
  - Less hardcoding for connection details.

Terminology used in Aurora

Be aware of nuances when you see terms in an Aurora context:

- “cluster” – storage volume + variable number of DB instances.
- “replica/replication” – Aurora refers to “writer instance” and “reader instance” to distinguish Aurora physical replication from Postgres logical replication.
- “backup” – it’s something you have, more than something you do.
- “Serverless” – servers still exist, they just grow or shrink capacity depending on load. Aurora Serverless v2 is newer & better for production systems.

Journey #3a: Aurora scalability and HA features

Leveraging and Optimizing for Aurora

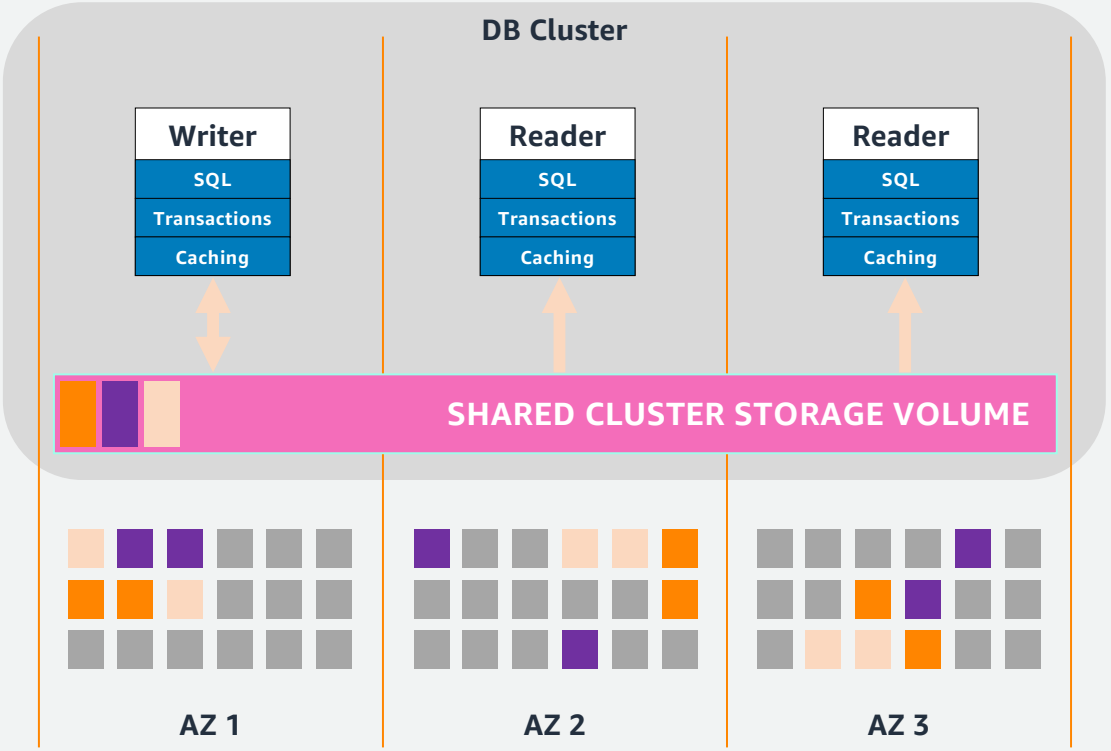
- What are the big levers you can pull to balance efficiency and HA?
- Take advantage of Aurora physical replication.
  - And Aurora global database for cross-Region replication.
- Find the right capacity to meet your needs with efficient price/performance:
  - Choose instance classes wisely.
  - Aurora Serverless v2: useful in many scalability scenarios.

Amazon Aurora cluster topology

Up to 16 DB instances/nodes in a **regional** cluster, spanning multiple AZs

One is always the writer/primary. Failover changes which instance is the writer.

Storage volume shared with readers. Readers open volume in read-only mode (PostgreSQL: `transaction_read_only = on`).



Separation of compute and storage

- An Aurora *cluster* is a whole lot of storage (up to 128 TB), accessed by a variable number of DB instances.
- Your data is *safe* regardless of how many DB instances are in the cluster.
- Multi-AZ configurations are the foundation of Aurora HA.
  - Have at least one standby server for production deployments.
  - Aurora makes multi-AZ simple. One checkbox or menu choice.

Replication in Aurora: Let's Get Physical

- Aurora maintains 6 copies of all the data in a cluster, across 3 AZs.
- This physical replication is offloaded to the servers that run Aurora storage. That makes the writes low-latency and low-overhead.
- All the DB instances see exactly the same data and schema.
  - The readers don’t copy data or replay statements. “Replica lag” means how long before a reader instance evicts stale data from its buffer cache.

Journey #3b:
PostgreSQL tuning for Aurora

Performance for Aurora Applications

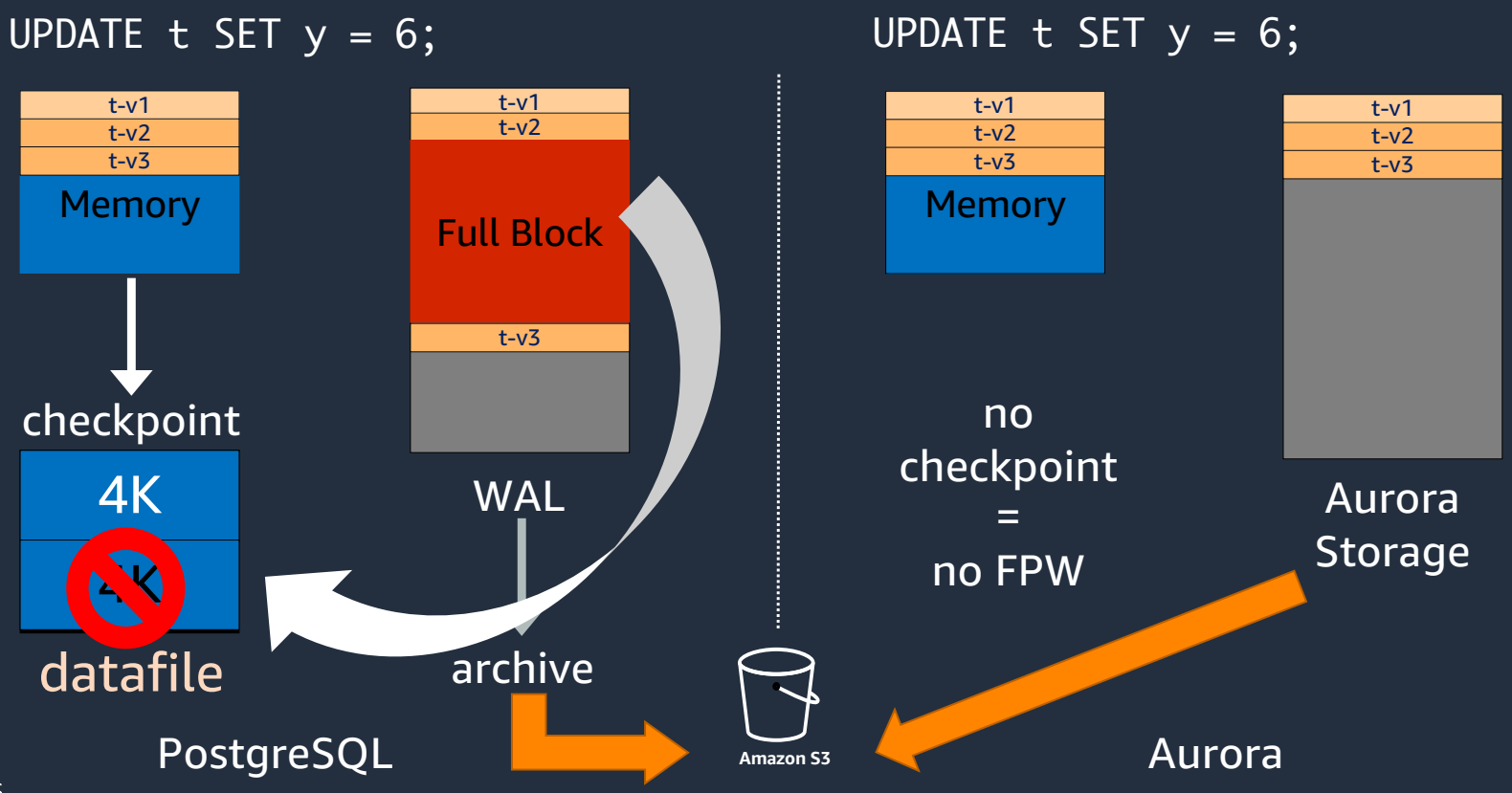
Aurora uses the same SQL dialect as the community PostgreSQL engine.

- CREATE INDEX / EXPLAIN the same as always.

Aurora has its own storage layer. The changes to physical reads, network, and I/O mean the causes of bottlenecks are different.

- For low-level tuning, you delve into *wait events.*
- For detailed yet still user-friendly performance work, you use CloudWatch, Performance Insights, and DevOps Guru for alarms, visualizations, and recommendations.
- Enhanced Monitoring shows OS-level performance info.

How Aurora optimizes I/O



Performance Tuning – Guidelines for Parameters

- Start with default values for parameters to define a baseline.
- Parameters are usually tuned by default for the instance class you choose
- Understand the impact of Aurora parameter changes.
- Parameters can be granular:
  - Some parameters can be set in a session or for a user e.g. work_mem, maintenance_work_mem.
  - Some parameters can be tweaked for tables e.g. vacuum and autovacuum related parameters.

Parameter Group Best Practices

- Create a custom parameter group for any production system
- Avoid attaching multiple cluster/instances to 1 parameter group
- Some parameters can only be changed at cluster level and some only at instance level
    - Use same parameter group for writer and the readers you prefer for failover
- Never customize a parameter in both cluster parameter group & instance parameter group

Tuning at the Instance Level – Crucial Metrics

- Right-size your instance so that working set fits in-memory.
  - Small ReadIOPS & high BuffercacheHitRatio
- Monitor key performance and utilization indicators.
  - Query latency, CPUUtilization, FreeableMemory, DBConnections, ReadThroughput, WriteThroughput, DBLoad

Don’t lock yourself up!

PostgreSQL Vacuum

- Don’t kill or disable auto-vacuum.
- Avoid VACUUM FULL

Aurora uses the same SQL dialect as the community PostgreSQL engine.

- CREATE INDEX / EXPLAIN the same as always.
- Avoid REINDEX
    - CREATE INDEX CONCURRENTLY
        - Version 12 supports REINDEX CONCURRENTLY.
        - DROP old index.
        - RENAME if needed.

Journey #3c: Monitoring

Aurora Purpose Built Monitoring Tools

- CloudWatch
- Enhanced Monitoring
- Performance Insights
    - Database Load
    - Counters
    - CPU Bottleneck
    - Wait Bottleneck

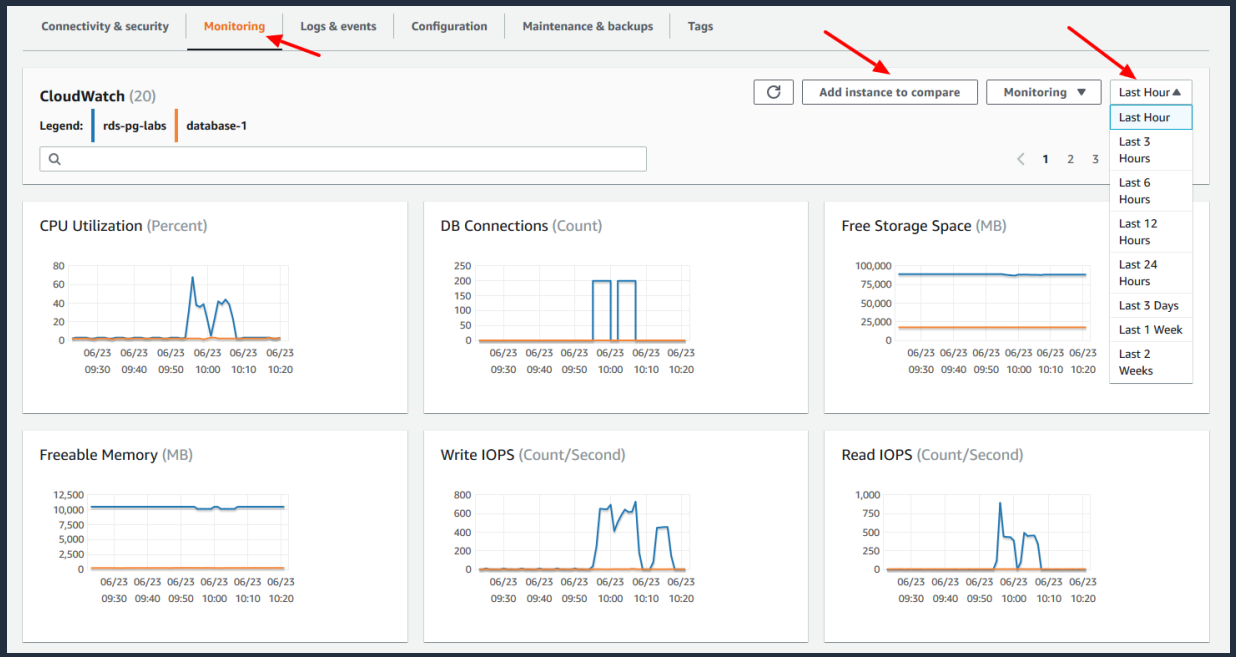
CloudWatch Metrics

CloudWatch also gathers metrics on the host underlying the RDS database. You can view these metrics in the RDS console under the monitoring tab.

CloudWatch Metrics:

- CPU Utilization
- DB Connections
- Free Storage
- Free Memory
- Billable Write IPOS
- Billable Read IOPS

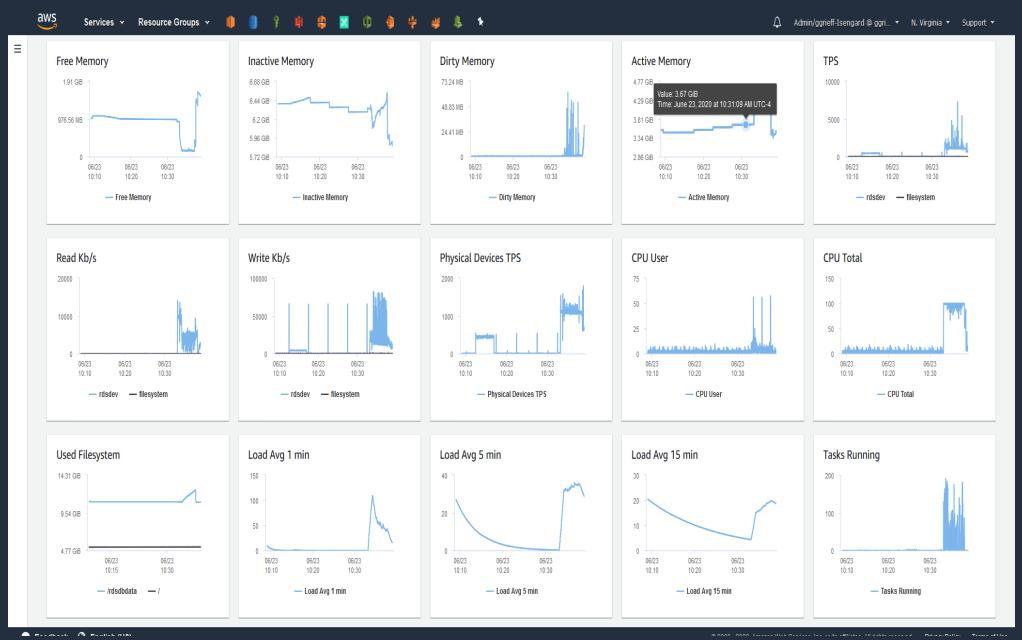
- Filter last hour to 2 weeks
- Compare RDS instances



Enhanced Monitoring

Gathers finer grained OS metrics from an agent installed on the RDS host.

- By default metrics are stored for 30 days. Governed by RDSOSMetrics log group in CloudWatch
- Incurs additional CloudWatch costs based on granularity (from 1 to 60 seconds).



Enhanced Monitoring – OS Process List

Enhanced Monitoring also includes the Process list, reachable from the monitoring dropdown. Sort the list by metric (e.g. CPU), filter for a particular user or database.

Process Groups

- RDS Child processes
- RDS Processes
- OS Processes

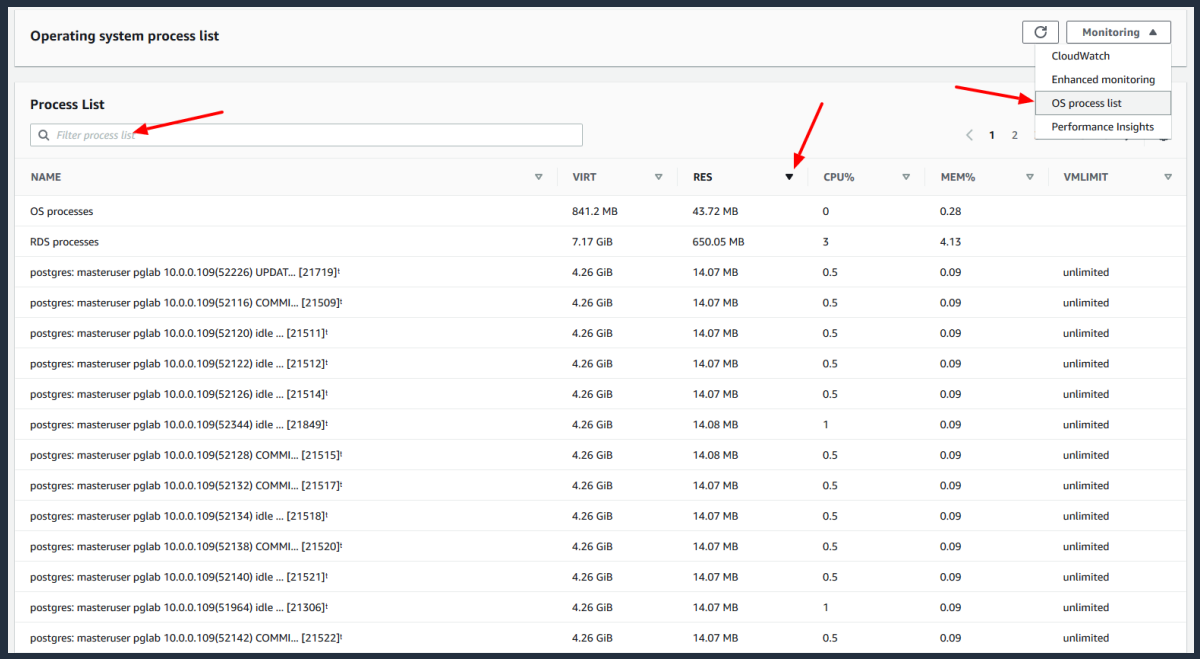
Items Listed

VIRT – Virtual size of process

RES – Physical memory used

CPU% - Total CPU bandwidth

MEM – Total memory used



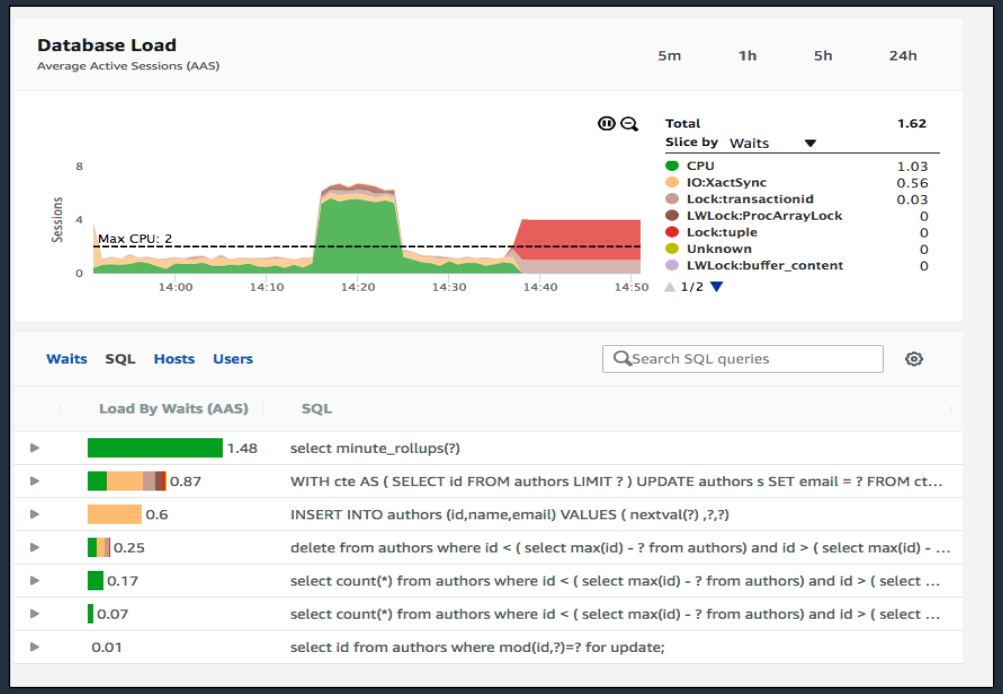
NAME	VIRT	RES	CPU%	MEM%	VMLIMIT
OS processes	841.2 MB	43.72 MB	0	0.28	
RDS processes	7.17 GiB	650.05 MB	3	4.13	
postgres: masteruser pglab 10.0.0.109(52226) UPDAT... [21719]	4.26 GiB	14.07 MB	0.5	0.09	unlimited
postgres: masteruser pglab 10.0.0.109(52116) COMMI... [21509]	4.26 GiB	14.07 MB	0.5	0.09	unlimited
postgres: masteruser pglab 10.0.0.109(52120) idle ... [21511]	4.26 GiB	14.07 MB	0.5	0.09	unlimited
postgres: masteruser pglab 10.0.0.109(52122) idle ... [21512]	4.26 GiB	14.07 MB	0.5	0.09	unlimited
postgres: masteruser pglab 10.0.0.109(52126) idle ... [21514]	4.26 GiB	14.07 MB	0.5	0.09	unlimited
postgres: masteruser pglab 10.0.0.109(52344) idle ... [21849]	4.26 GiB	14.08 MB	1	0.09	unlimited
postgres: masteruser pglab 10.0.0.109(52128) COMMI... [21515]	4.26 GiB	14.08 MB	0.5	0.09	unlimited
postgres: masteruser pglab 10.0.0.109(52132) COMMI... [21517]	4.26 GiB	14.07 MB	0.5	0.09	unlimited
postgres: masteruser pglab 10.0.0.109(52134) idle ... [21518]	4.26 GiB	14.07 MB	0.5	0.09	unlimited
postgres: masteruser pglab 10.0.0.109(52138) COMMI... [21520]	4.26 GiB	14.07 MB	0.5	0.09	unlimited
postgres: masteruser pglab 10.0.0.109(52140) idle ... [21521]	4.26 GiB	14.07 MB	0.5	0.09	unlimited
postgres: masteruser pglab 10.0.0.109(51964) idle ... [21306]	4.26 GiB	14.07 MB	1	0.09	unlimited
postgres: masteruser pglab 10.0.0.109(52142) COMMI... [21522]	4.26 GiB	14.07 MB	0.5	0.09	unlimited

Performance Insights

- Dashboard
  - DB load
  - Adjustable timeframe
  - Filterable by attribute (SQL, User, Host, Wait)
  - SQL causing load
- Phased Amazon RDS delivery
  - Amazon Aurora, Amazon RDS for MySQL, PostgreSQL, Oracle, SQL Server, MariaDB
- Guided discovery of performance problems
  - For both beginners & experts
  - Core metric “**database load**”

What is “database load”?

- All engines have a connections list showing
  - Active, Idle
- We sample every second
  - For each active session, collect
    - SQL
    - State: CPU, I/O, lock, commit log wait, etc
    - Host
    - User
- Expose as “average active sessions” (AAS)



Monitoring and Tuning Recap

- 7 Takeaways for PostgreSQL tuning with Aurora:
  - Tuning #1: Ensure that there is sufficient RAM
  - Tuning #3: Check index availability and usage
  - Tuning #4: Maintenance (Vacuuming)
  - Tuning #5: Limit temp table usage
  - Tuning #6: Scalability through join decomposition
  - Tuning #7: Recommended PostgreSQL settings for Aurora
  - Tuning #8: Utilize Performance Insights and Enhanced Monitoring

Managing costs for Aurora

Managing Costs with Aurora

- Aurora charges fall into 3 buckets:
  - Instances: Under your control. You can influence via # of instances, instance class, stop/start.
  - Storage: Predictable. You can influence by cleaning up (e.g. dropping unused indexes, vacuuming) and archiving (e.g. unneeded or too-old data).
  - I/O: Varies based on usage. You can influence by tuning queries, sizing instances for big enough buffer cache, avoiding wasteful copying.

Aurora Best Practices – Performance & Cost Optimization

Best Practice	Result
Allocate enough RAM	Goal is for working set to reside completely in memory.
Monitor workload • VolumeReadIOPS • BufferCacheHitRatio	Aurora bills at the storage I/O layer. Monitor query activity at the disk layer.
Cached workloads	Sporadic workloads on large databases.
Warm the buffer cache before use	Cold cache could give false readings when testing load.
Keep transactions short	Reduce replication lag and allow data to stay in cache longer.
Set (TTL) value of less than 30 seconds	Reduce connection failures.
Test DB failovers	Verify that writer and readers are sized correctly, and your application is resilient to instances switching roles.

Scenario: “signing off for the night”

Don’t need full DB capacity running 24x7? Here are some ways to save on instance charges during idle times:

- Stop the cluster. No instance charges while it’s in “stopped” state.
  - For long-term stoppage: save snapshot, delete cluster, restore snapshot later.
- Downsize the instance class for any instances in the cluster.
- Reduce the number of instances in the cluster, to 1 or even 0.
- Adjust capacity range in Serverless v2. (Set a low number for minimum ACUs.)
- Turn on auto-pause in Serverless v1.

Scenario: “choosing capacity for database instance”

- The Aurora pricing page, and the AWS pricing API, tell you hourly charges for instance classes: by engine, version, and AWS Region.
- Pick whatever capacity you need during active periods, use one of the cost-saving techniques (from previous slide) during idle times.
- Size instances up temporarily during intensive workloads.
- Or, let Serverless v2 adjust capacity range based on load.
  - You pick a sensible floor & ceiling for the capacity range.
- Interesting classes: T (for dev/test only), X2 (extra memory), R6g (price/performance), R6i.32xlarge (biggest available today).

Scenario: “a dev sets up a new DB server”

- Clone an existing cluster.
  - Super-fast, saves on I/O during setup, saves on storage that’s identical.
  - Best for environments that aren’t long-lived and data doesn’t diverge greatly.
- Restore a snapshot.
  - Faster and cheaper than reloading the data.
  - You can save a manual snapshot forever, or restore based on any time within the retention interval.
- The cloned or restored clusters can use different instance classes, cluster topologies, be upgraded to a higher version.
- Do you need a new environment if there’s ~128 TB free in your old one?

Demo – reducing costs for idle dev/test instances

Where in the world am I using Aurora?

What are all the places I *could* be using Aurora?

```
aws ec2 describe-regions --region us-east-1 --query 'Regions[*].[RegionName]' --output
text | sort
```

```
af-south-1
ap-east-1
ap-northeast-1
ap-northeast-2
...
us-east-1
us-east-2
us-west-1
us-west-2
```

Where in the world am I using Aurora?

What are all the places I *actually* am using Aurora? Run in a loop...

```
aws rds describe-db-instances --region $region --query '*[].[DBInstanceIdentifier]' --
output text
```

```
AWS Region ap-southeast-4: 1 DB instance(s)
AWS Region ca-central-1: 1 DB instance(s)
AWS Region us-east-1: 23 DB instance(s)
AWS Region us-west-1: 1 DB instance(s)
```

Where do I have Aurora instances that are running?

Because I care about instances with status “available”, not “stopped”.

```
aws rds describe-db-instances --region "$region" --query
'DBInstances[*].{DBInstanceIdentifier:DBInstanceIdentifier,DBInstanceClass:DBInstanceClas
s,Engine:Engine,EngineVersion:EngineVersion,DBClusterIdentifier:DBClusterIdentifier,DBIns
tanceStatus:DBInstanceStatus}|[?DBInstanceStatus == `available`]|[?Engine == `aurora-
postgresql`]|[].[DBInstanceIdentifier,DBInstanceClass,Engine,EngineVersion,DBClusterIdent
ifier,DBInstanceStatus]' --output table
```

```
-------------------------------------------------------------------------------------------------------------
|                                                DescribeDBInstances                                        |
+-------------------------------+---------------+--------------------+-------+----------------------+-------------+
|  apg14-instance               |  db.t4g.medium |  aurora-postgresql |  14.6 |  apg14               |  available  |
|  babelfish-apg145-instance    |  db.t4g.large  |  aurora-postgresql |  14.5 |  babelfish-apg145    |  available  |
|  babelfish-pg-14-instance     |  db.t3.large   |  aurora-postgresql |  14.5 |  babelfish-postgres-14 |  available  |
|  instance-2023-03-13-8569     |  db.r5.4xlarge |  aurora-postgresql |  14.6 |  tpch-100g-apg       |  available  |
|  my-second-babelfish-instance |  db.t4g.medium |  aurora-postgresql |  13.6 |  my-second-babelfish |  available  |
+-------------------------------+---------------+--------------------+-------+----------------------+-------------+
```

What's the hourly charge for each instance class?

These numbers vary by AWS Region, are subject to change, and don't reflect discounts from reserved instances or other pricing arrangements. Think of this as a worst-case scenario. Always consult the latest pricing information for your AWS Region!

```
aws pricing get-products --service-code AmazonRDS <too many more parameters to fit here>

18.56 - db.r6i.32xlarge
13.92 - db.r6i.24xlarge
13.92 - db.r5.24xlarge
...
0.146 - db.t4g.large
0.082 - db.t3.medium
0.073 - db.t4g.mediu
```

Compare and contrast instance classes

For Aurora, the T classes are recommended only for dev/test:

```
0.164 – db.t3.large
0.146 – db.t4g.large
0.082 – db.t3.medium
0.073 – db.t4g.medium
```

The sizes and prices within each family tend to increase consistently:

```
8.306 – db.r6g.16xlarge
6.229 – db.r6g.12xlarge
4.153 – db.r6g.8xlarge
2.076 – db.r6g.4xlarge
1.038 – db.r6g.2xlarge
0.519 – db.r6g.xlarge
0.26 – db.r6g.large
```

Compare and contrast instance classes

Within a given size, the latest generation tends to be a better deal than older ones. (More horsepower at same price.) Also x2g offers 2x the RAM, r6g offers good price/performance:

```
6.032 – db.x2g.8xlarge
4.64 – db.r6i.8xlarge
4.64 – db.r5.8xlarge
4.64 – db.r4.8xlarge
4.153 – db.r6g.8xlarge
3.016 – db.x2g.4xlarge
```

What if all those steps were combined?

Imagine a script that went through this process and calculated a total. Would this be useful?

```
--- instance-2023-03-13-8569 ---
Hourly instance price for Aurora PostgreSQL, db.r5.4xlarge, us-east-1 in US
dollars: $2.32
If you leave an instance like that running until tomorrow 9 AM (starting now),
that will cost roughly: $___
--- my-second-babelfish-instance ---
Hourly instance price for Aurora PostgreSQL, db.t4g.medium, us-east-1 in US
dollars: $0.073
If you leave an instance like that running until tomorrow 9 AM (starting now),
that will cost roughly: $___

=== Summary of overnight charges ===
Your total overnight charges for idle DB instances in the us-east-1 AWS Region,
in US dollars, could be roughly: $___
```

Resources

- Trusted Language Extensions (TLE) project: https://github.com/aws/pg_tle
- Babelfish project: https://babelfishpg.org/
- Supported PostgreSQL extensions: https://docs.aws.amazon.com/AmazonRDS/latest/PostgreSQLReleaseNotes/postgresql-extensions.html
- AWS instance classes for Aurora: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Concepts.DBInstanceClass.html
- Aurora pricing page: https://aws.amazon.com/rds/aurora/pricing/
- Aurora product page: https://aws.amazon.com/rds/aurora/
- Dustin's blog on SQL Server to Aurora migration: https://aws.amazon.com/blogs/database/migrate-sql-server-to-amazon-aurora-postgresql-using-best-practices-and-lessons-learned-from-the-field/
- John's video on SQL Server to Aurora migration using Babelfish: https://www.youtube.com/watch?v=f9YC5NyNzAE

Thank you!

John Russell
johrss@amazon.com
@max_webster

Dustin Brown
dusbr@amazon.com