Microsoft

Azure Cosmos DB for PostgreSQL

Technical Deep Dive

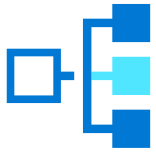
Frédéric Gisbert
Cloud Solution Architect - frgisber@microsoft.com

AGENDA

- Overview - Cosmos DB For PostgreSQL (Powered by Citus)
- Cluster Manageability
- Data Distribution
- Data Modeling
- Querying (with App Patterns)
- Performance Tuning
- Monitoring & Alerting
- Troubleshooting
- Programming (App Stacks/ORM)
- Migration Consideration
- New UX post Relaunch

Azure Cosmos DB for Postgres

A fully managed **relational distributed database** preserving ACID capabilities of SQL realm alongside enterprise proven, global distribution & elasticity of Azure Cosmos DB.



Distributed Postgres

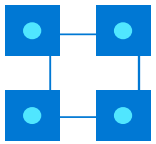
Break free from the limits of single-node Postgres & scale out across 100s of nodes

Ideal for SaaS

Easier implementation both for Building for Scale or Building for Isolation with finer control on individual shards.

Global Scalability

Global availability of data through Cross-region replicas



Common Source for OLTP & OLAP

Save time & cost. Run both transactions and analytics in one database. Also, avoid the painful costs of manual sharding

Parallelized performance

Ingest & query your database in real time, with sub-second responses across billions of rows

Stay current with PostgreSQL innovations

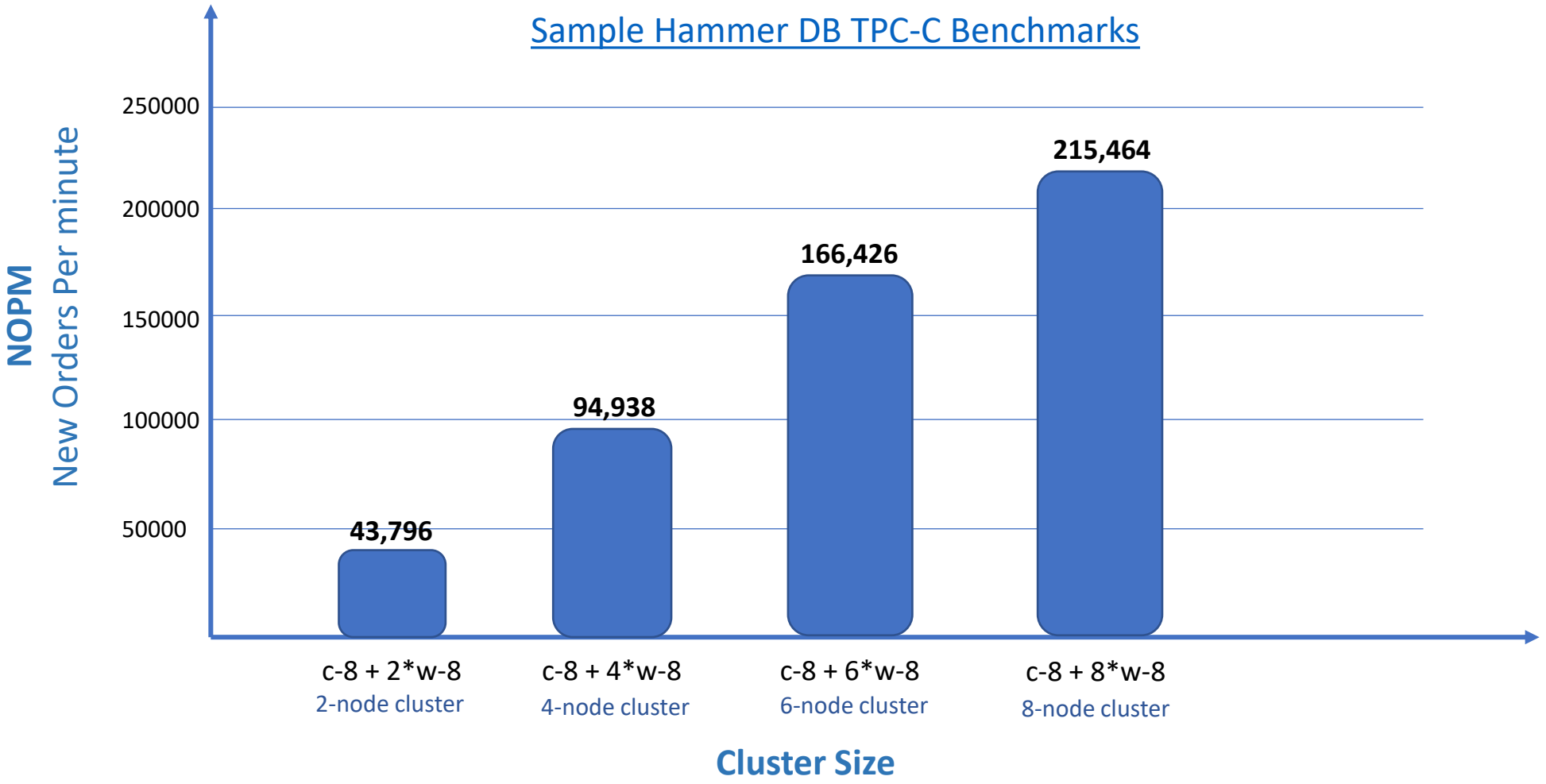
Developed as an open source extension (not a fork), leverage all your PostgreSQL expertise and its latest innovations

Fully Managed Distributed SQL build upon Open Source PostgreSQL

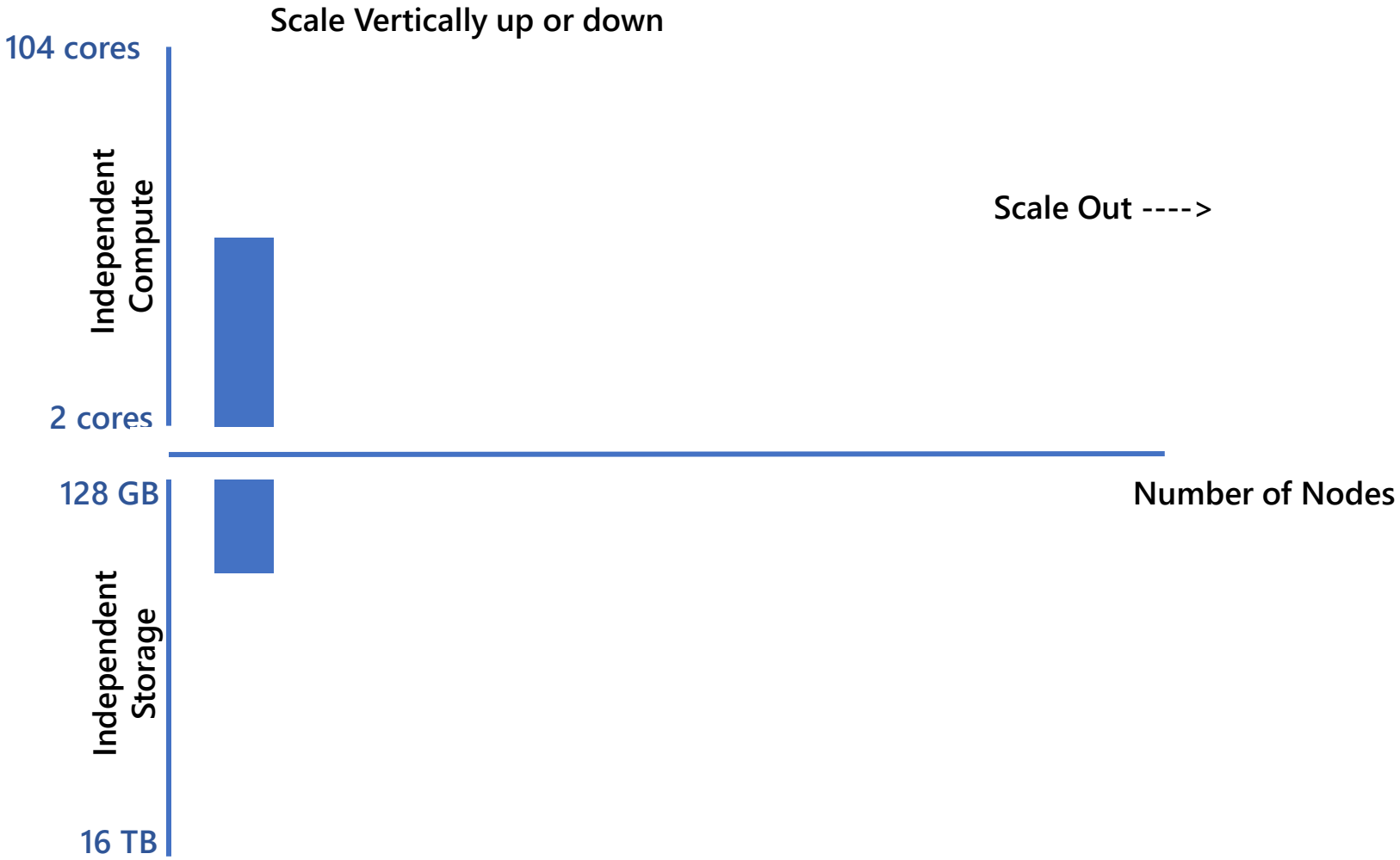
Overview

Linearly Scaled Throughput

Delivers **2 Million NOPM** with 20 nodes cluster having 64 vcores each



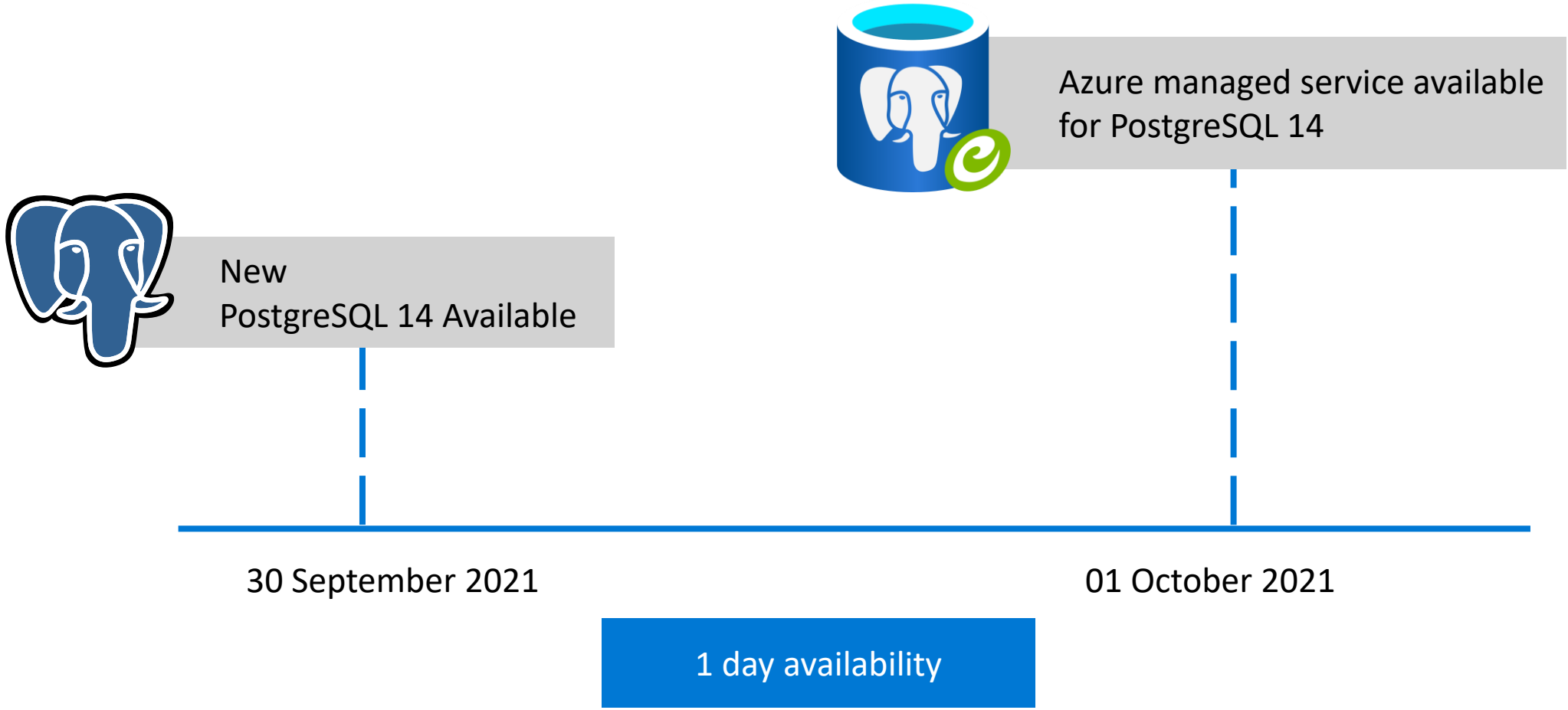
Elastic compute & Storage options



This model allows you to independently choose compute and storage resources.

Best for customers who value flexibility - control and transparency

Azure Cosmos DB for Postgres Availability

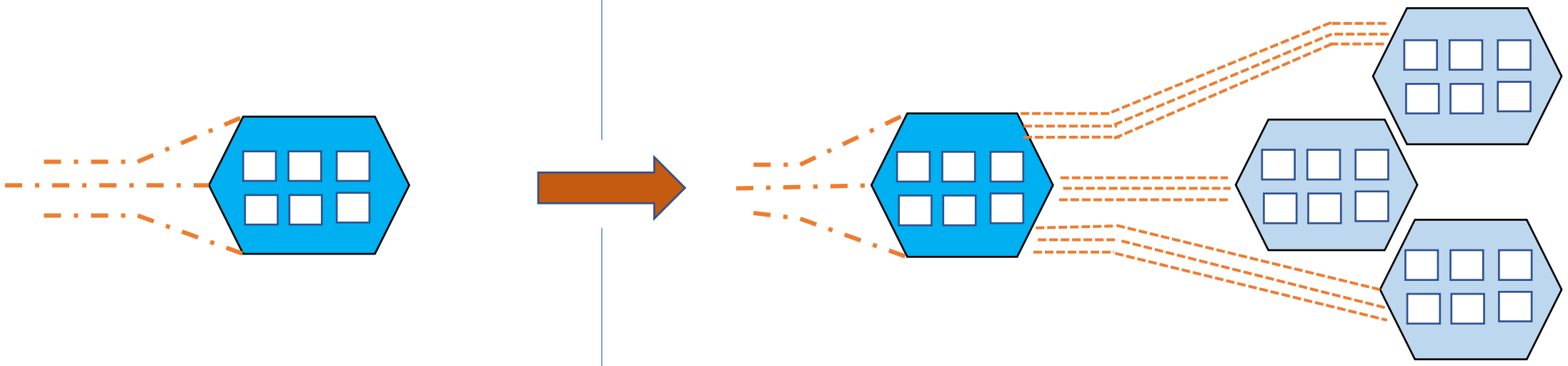


Online Scaling

(Powered by Citus)

Start Building apps on Single node

Seamlessly scale to Multiple nodes

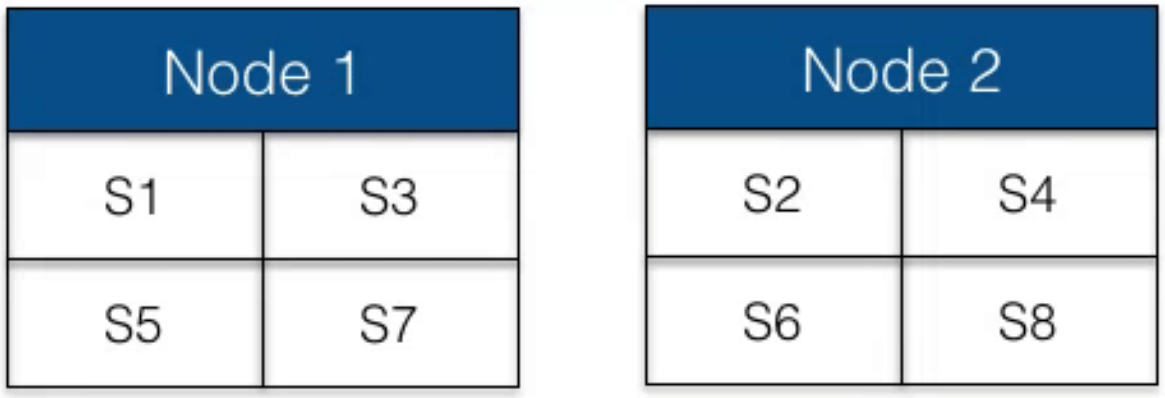


Online Tenant Isolation

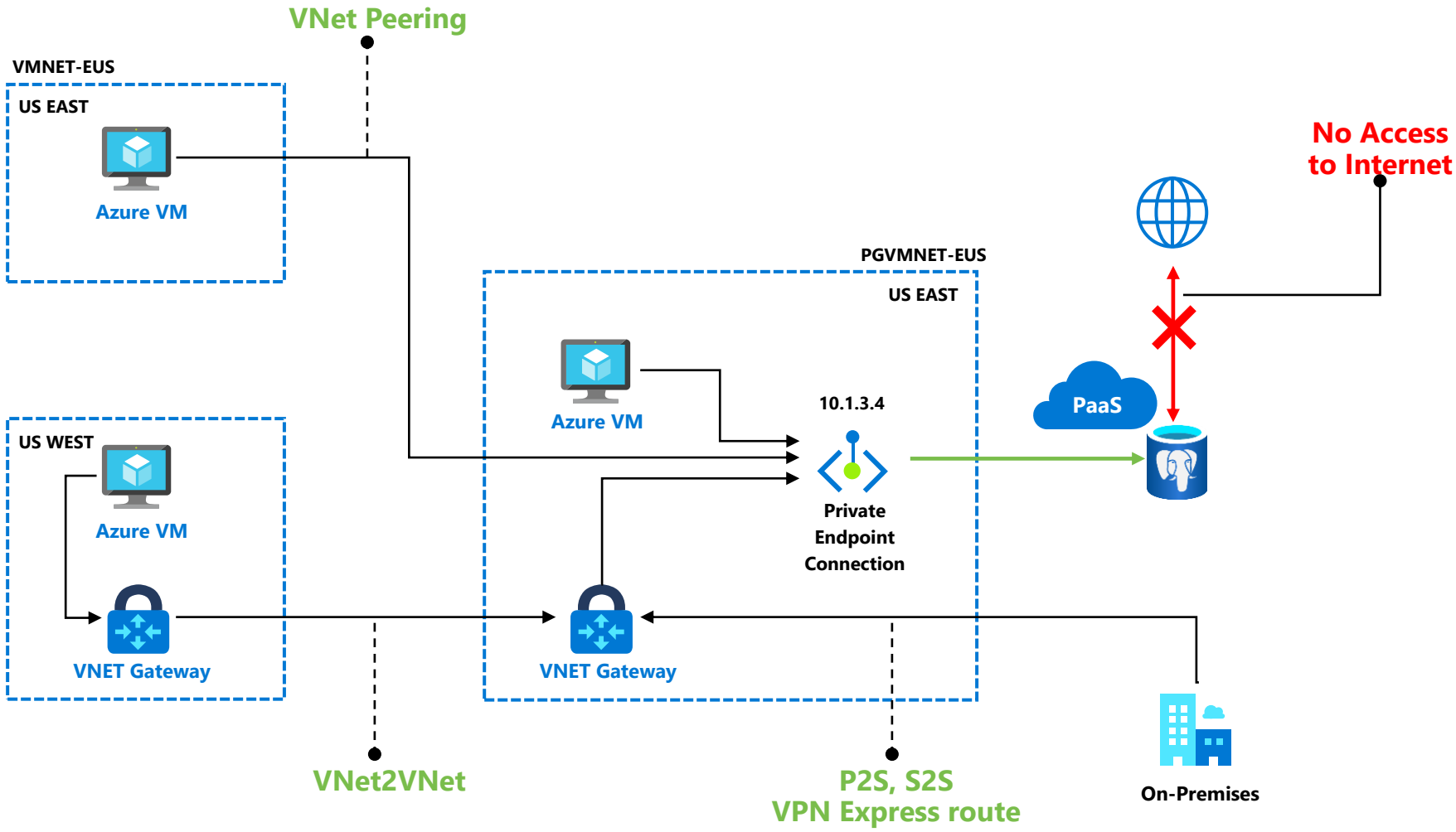
> Isolating a tenant allows allocating dedicated resources to handle growing needs for an individual tenant.

```
SELECT isolate_tenant_to_new_shard('table_name', tenant_id);

SELECT master_move_shard_placement(shard_id, source_node_name, source_node_port,
target_node_name, target_node_port);
```



Private access through Private Link



Simplified Management

Automatic shard distribution with Multi-node set up

1

Free backup storage with PITR up to 35 days

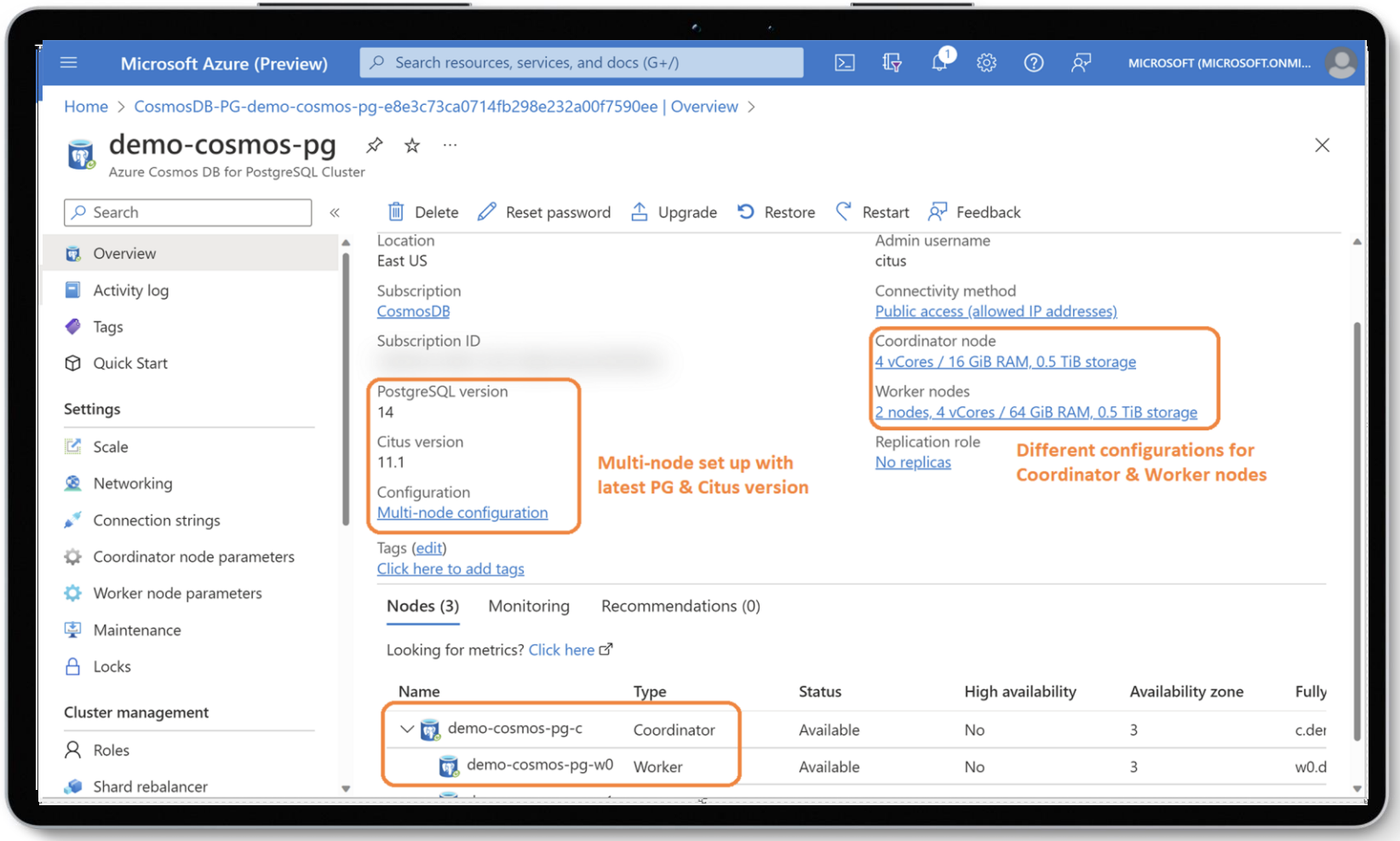
2

Columnar storage for higher level of compression

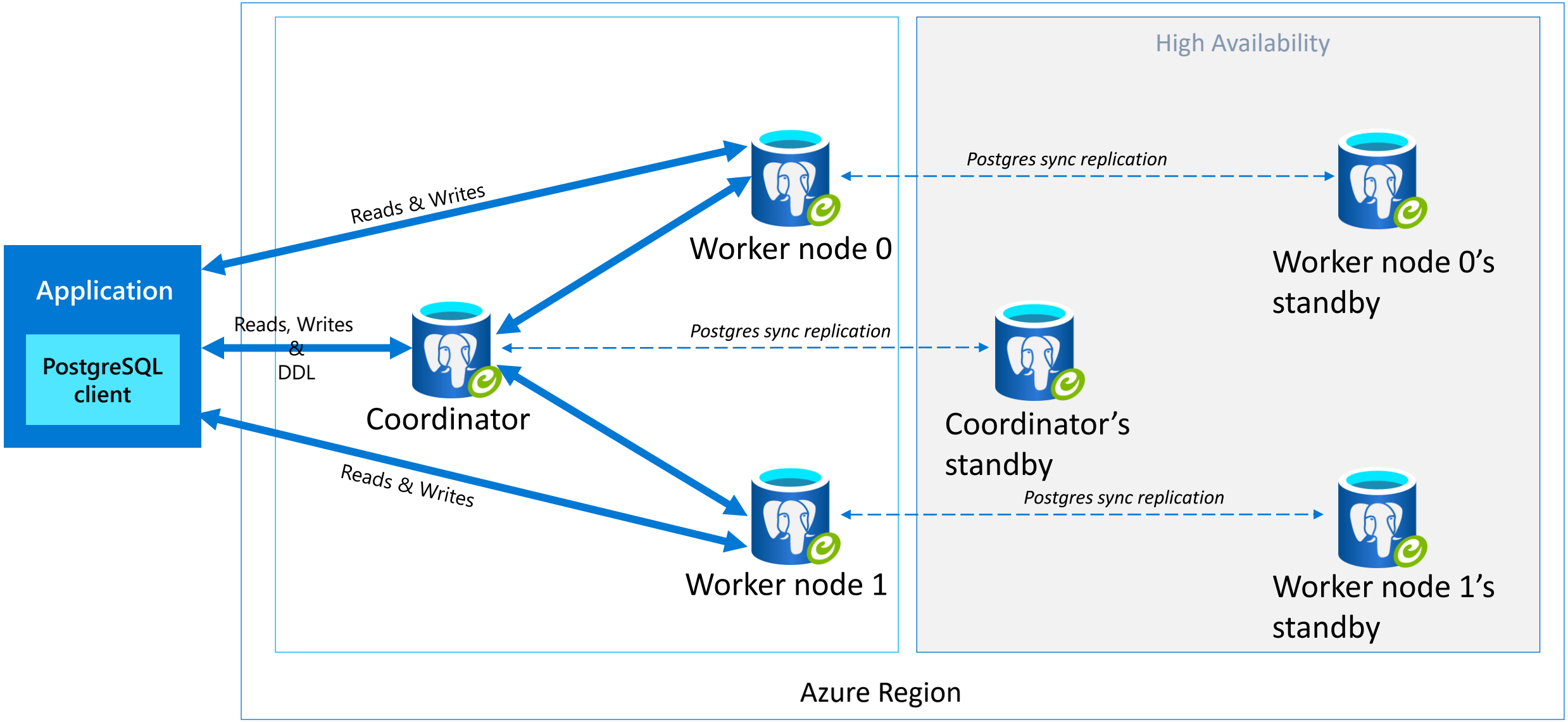
Portal Access

Goals in Power BI

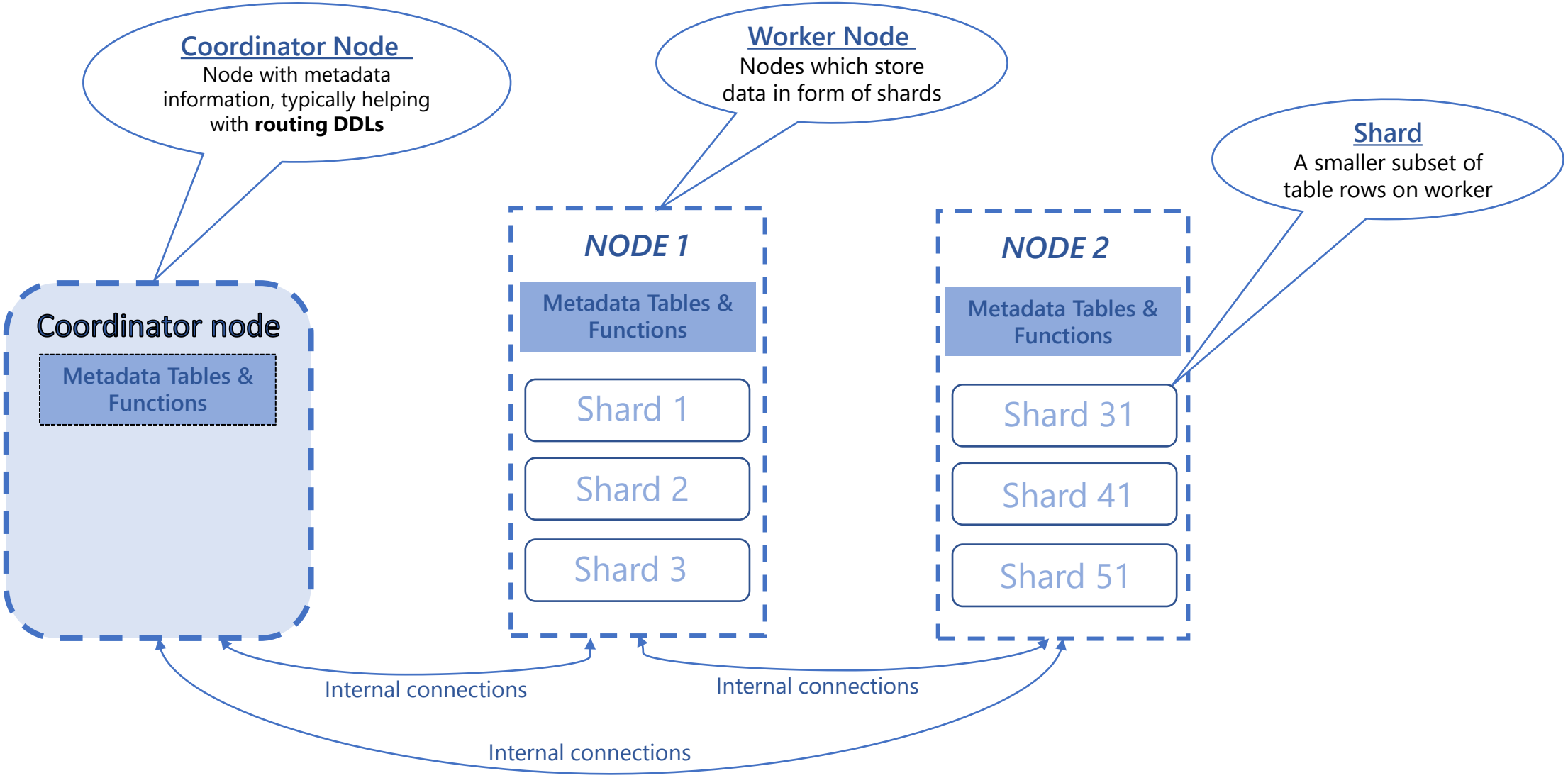
- Driven by data
- Built for teams
- AI powered
- Automated action



Architecture Overview



Data Distribution & Terminology



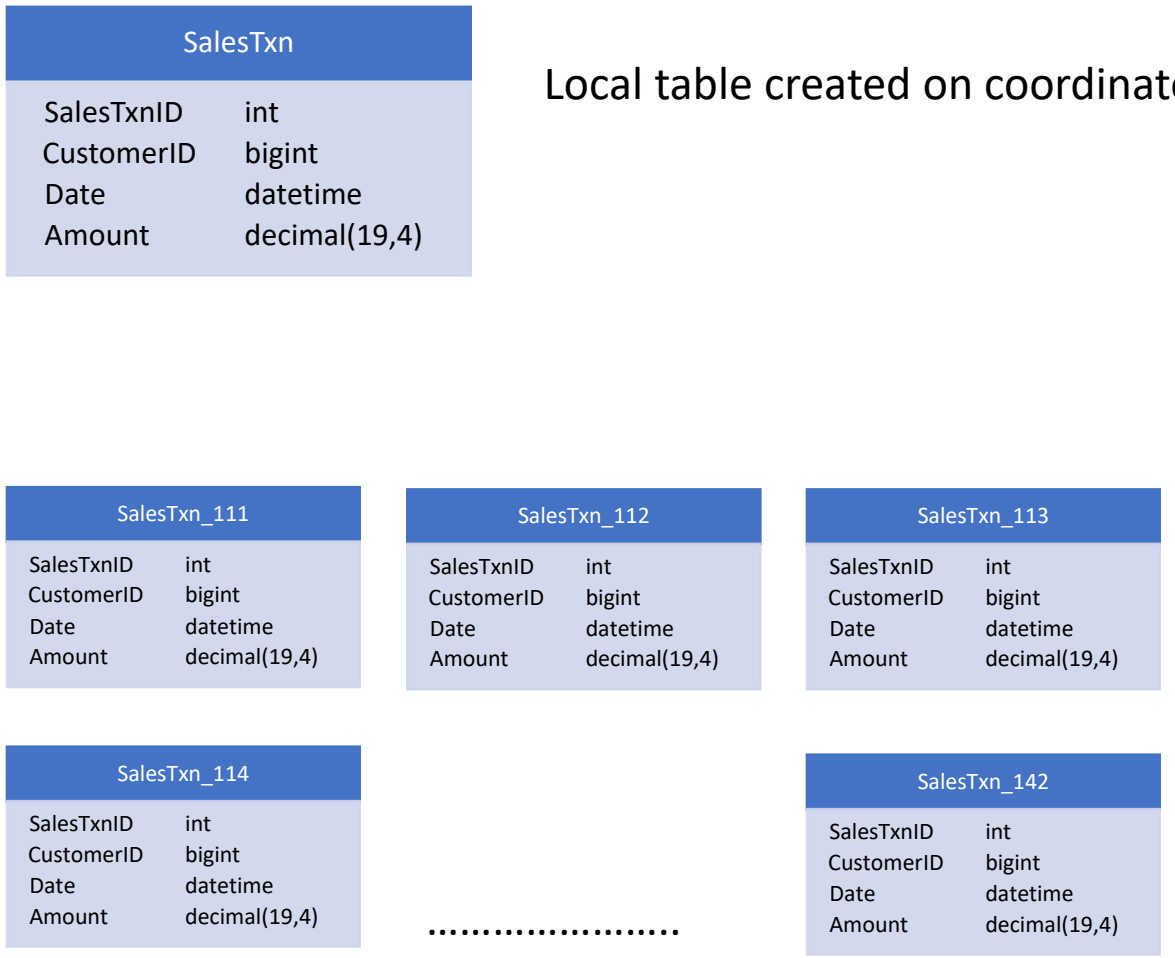
Data Distribution

```
/* Create Postgres table */

CREATE TABLE SalesTxn
          (
             SalesTxnID      int
             CustomerID      bigint
             Date            datetime
             Amount          decimal(19,4)
           )
```

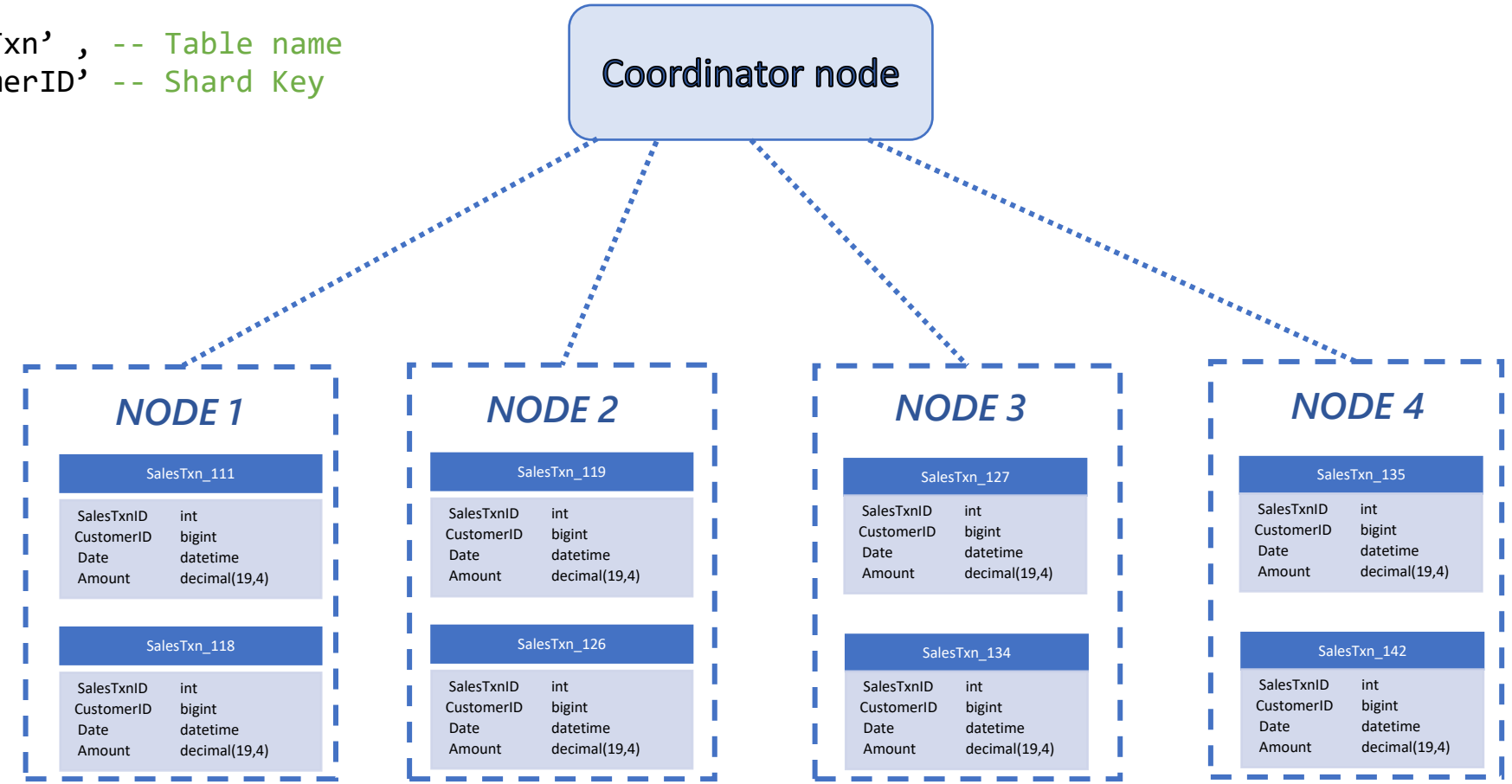
```
/* Inbuilt Automatic Online Sharding */

SELECT create_distributed_table_concurrently
          (
             'SalesTxn' , -- Table_name
             'CustomerID' -- Shard Key
          )
```



Data Distribution (Node level)

```
SELECT create_distributed_table_concurrently
        (
           'SalesTxn' , -- Table name
           'CustomerID' -- Shard Key
        )
```

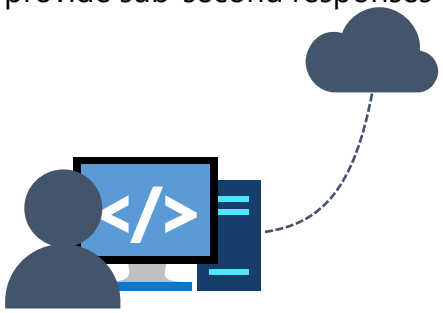


Distribution\Shard key – Column (**CustomerID** in above) used to distribute data among nodes

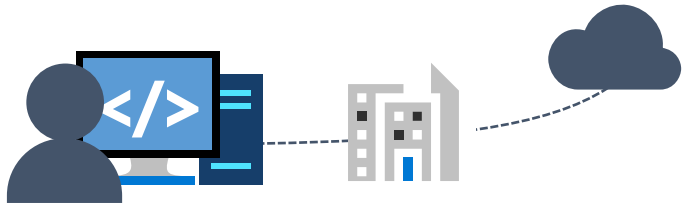
Common development scenarios

Real-time data analytics

Customer facing dashboards need to provide sub-second responses



Enterprise HTAP application



Common app types

e-Commerce app
Multi-tenant app
Analytics sites
Finance
Gaming/voting
Sales\Marketing app
...and many others

IOT based apps

Multi-tenant SaaS

SaaS app grows fast & need to scale while still providing snappy experience

Real-time / timeseries operational analytics and reporting

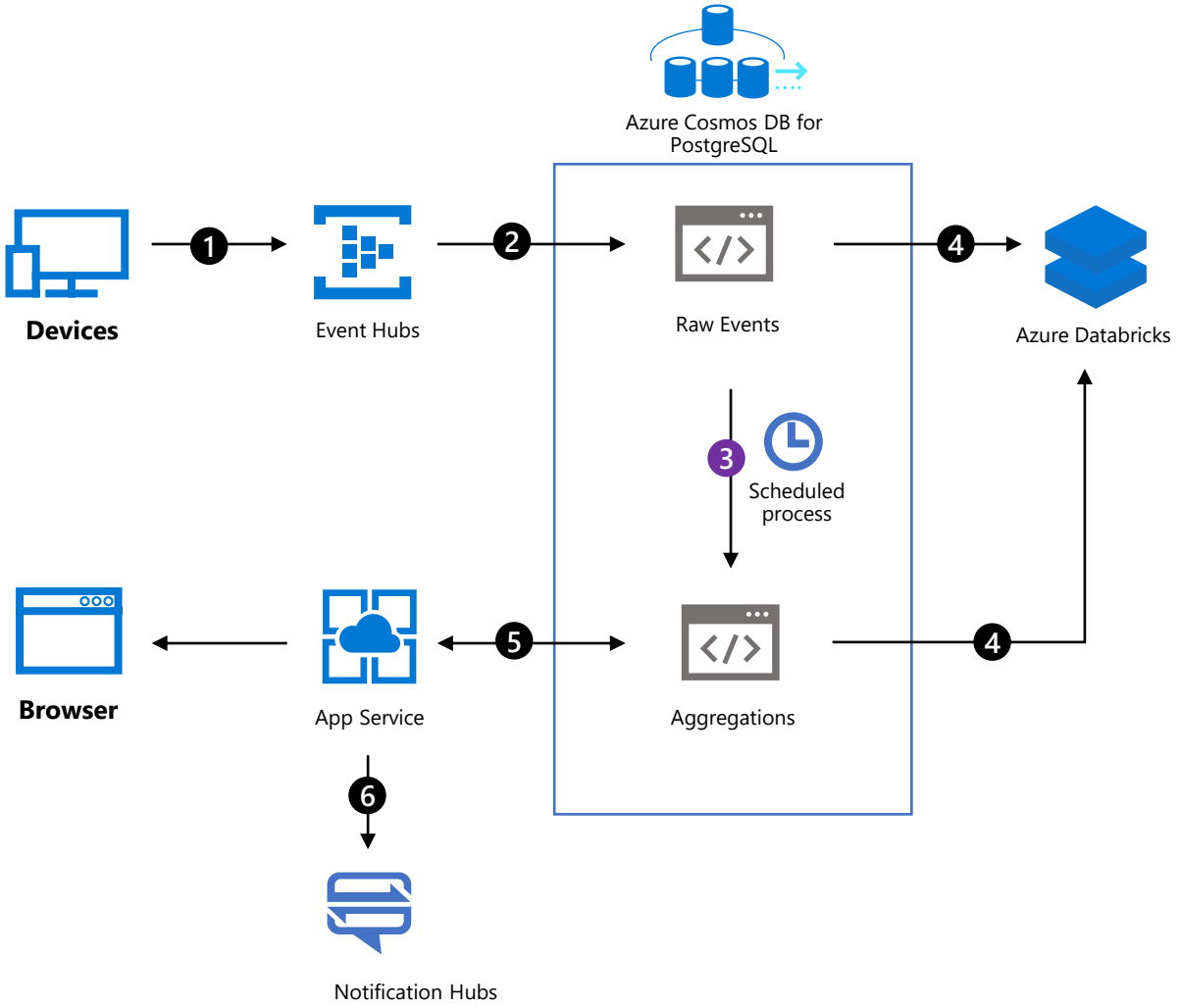
Interactive analytics dashboard: Need to visualize and query data, with sub-second latency.

Highly concurrent needs : Provide access to lots of concurrent users querying the dashboard at the same time.

Demanding performance expectations: Your users need sub-second response times for queries (sometimes millisecond response times) even when handling hundreds of analytical queries per second.

Data needs to be "fresh": Your users need to query the data while you're simultaneously ingesting new data, **this is the "real-time" bit**.

Event or time series data: Your data captures the many things that have happened (events) along with their associated timestamps, and you want to analyze the data.



Choosing a shard key for real-time analytics & timeseries workloads

Shard Key Criteria

- Pick a column that enable queries to be parallelized across the cluster, columns having:
  - high cardinality as the distribution column
  - even distribution i.e. lesser skew
  - present in the **GROUP BY** clause—or are queried **as list of values**
- For timeseries data, Use sharding in combination with native Postgres partitioning
- Use reference tables to enable JOINs

Pitfalls to Avoid

- Do not choose a timestamp as the shard key, in timeseries use case.
- Why? Data is preserved is based on hash value of distribution key (and not on the distribution key value itself. Results in skew on data towards single shard.

Scaling multi-tenant & SaaS applications

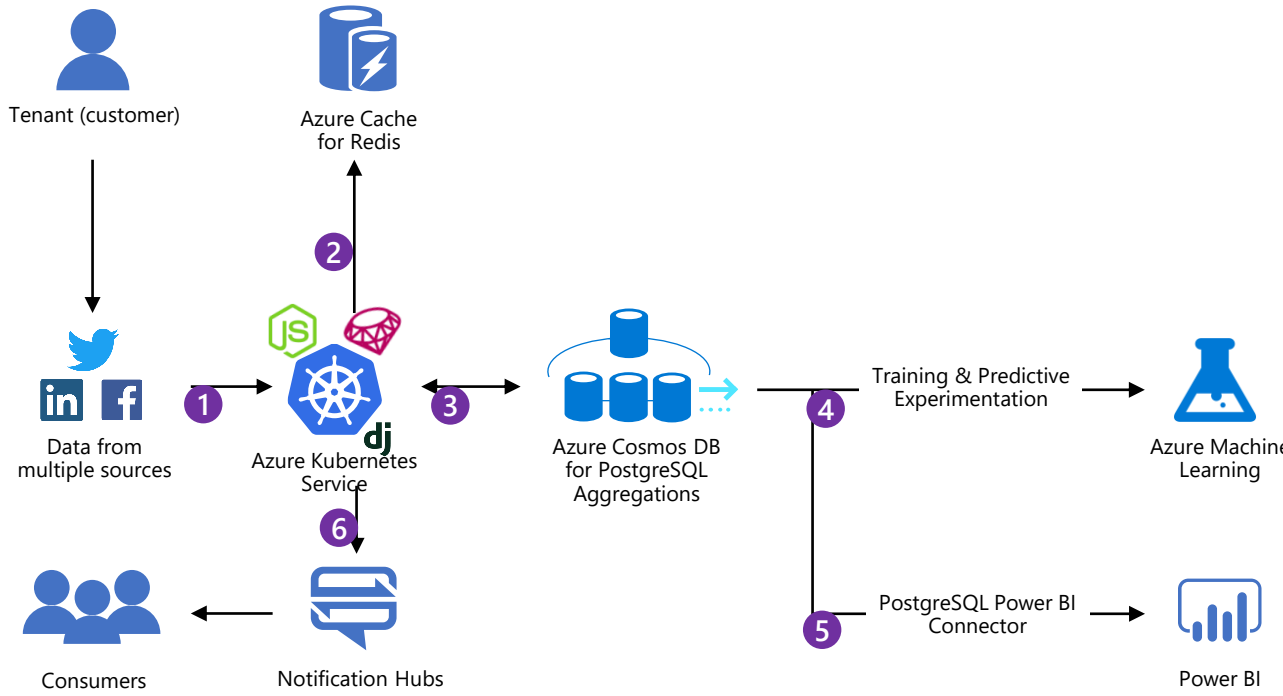
Tenants need isolation : Your SaaS customers only need to read/write their own data and should not have access to data from your other SaaS customers.

Application is growing fast: Your application is growing fast, in terms of number of users, size of database, or amount of activity—hence the number of monthly active users (MAU) or daily active users (DAU) is increasing.) More specifically, your database is 100s of GBs and growing, your SaaS app has 1000s of customers, you have 100,000+ users (or more.)

Performance issues, especially with concurrency : You're starting to run into performance issues during times with lots of concurrency.

Single digit millisecond latency : Multi-tenant SaaS apps are primarily operational/transactional, with single digit millisecond latency requirements for their database queries

Want to keep relational database semantics: You don't want to give up foreign keys for referential integrity, nor give up things like database constraints or secondary indexes. So the cost of migrating to a NoSQL data store to get scale is just not worth it to you.



Choosing a shard key for Multi-Tenant SaaS workloads

Shard Key Criteria

- Pick a column that enable queries to be parallelized across worker nodes
  - Partition distributed tables by a common tenant_id column
  - Convert small cross-tenant tables to reference tables, example country table.
  - All application queries to include the distribution column.

Pitfalls to Avoid

- Tables larger than 10 Gb are mostly observed unsuitable as reference table as a rule of thumb.
- Highly transactional table being kept as reference table.

IOT workloads – needing a transactional database

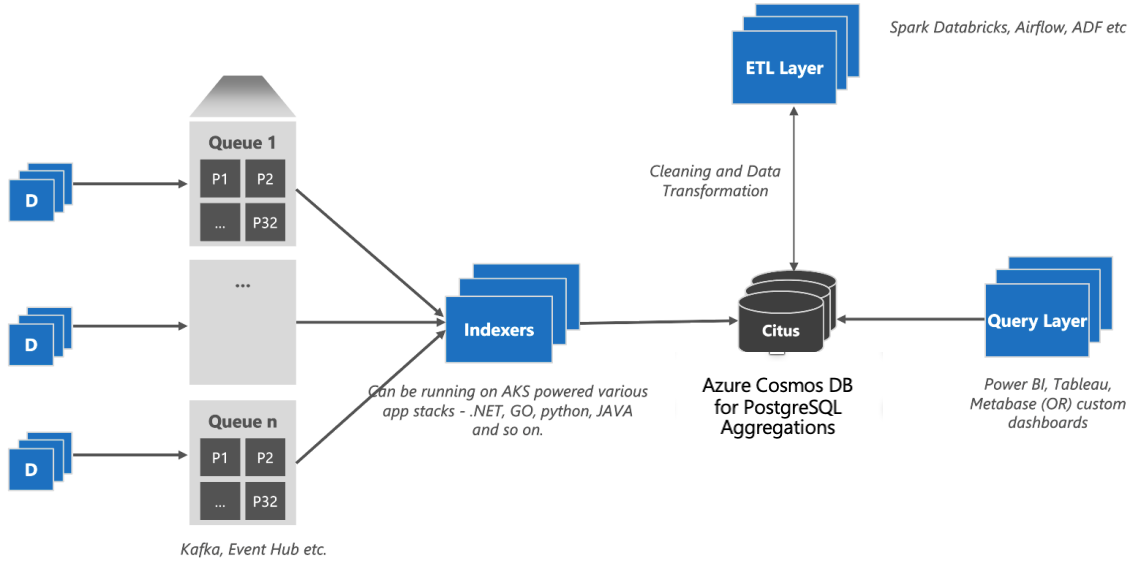
Need for an RDBMS and leverage **PostgreSQL features** – To leverage SQL expressiveness, support UPDATE/DELETE workloads, ACID etc. Also leverage unique PostgreSQL features such as JSONB, PostGIS (geospatial) and so on.

Large numbers of devices: we've seen customers with tens of thousands of devices, as well as millions of devices.

Need real-time, high-throughput ingest: In order to manage what's going on with your fleet of devices, you need the data now. Which means you need a database that can ingest and write with high throughput and low-latency. Example: 2 billion measurements per hour, which equates to roughly ~500,000 measurements ingested per second.

Query response times in the single digit second or milliseconds: Whether you are managing wind farms or manufacturing devices or fleets of smart meters, one sign that your IOT application could benefit from Azure Cosmos DB for PostgreSQL is when your users need their query responses (on fresh data) in the single digit seconds or even in milliseconds.

Run NoSQL/document style workloads, but require PostgreSQL features such as transactions, foreign/primary keys, triggers, extension like PostGIS, etc.



Choosing a shard key for IOT Workloads

Shard Key Criteria

- Distribute large tables on a common column that is central piece of the app, and the column that your app mostly queries.
- As IOT apps have a time dimension, partition your distributed tables based on time. You can use native Azure Cosmos DB for PostgreSQL time series capabilities to create and maintain partitions.
  - Partitioning helps efficiently filter data for queries with time filters.
  - Expiring old data is also fast, using the DROP vs DELETE command.
- Use the JSONB datatype to store semi-structured data. Device telemetry data is typically not structured.
- If your IoT app requires geospatial features, you can use the PostGIS extension, which Azure Cosmos DB for PostgreSQL supports natively.

Pitfalls to Avoid

- Tables larger than 10 Gb are mostly observed unsuitable as reference table as a rule of thumb.
- Highly transactional table kept as reference table.

High-throughput transactional apps

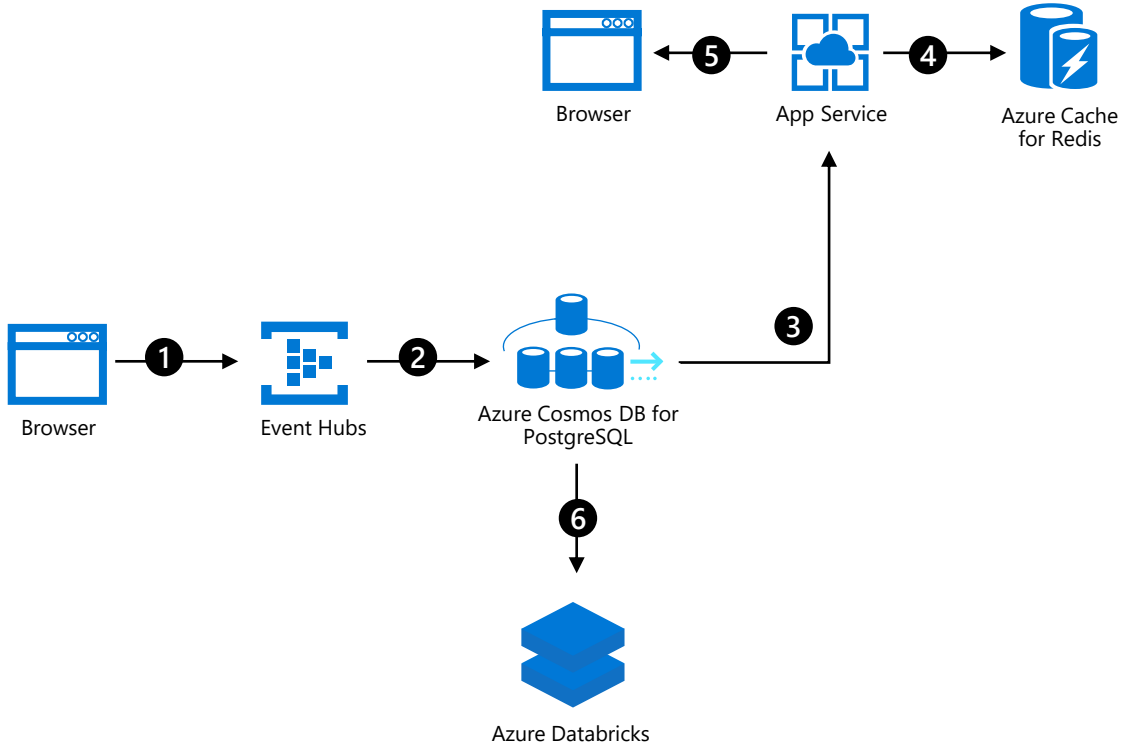
Primarily transactional application: Your app is primarily transactional in nature, with creates, reads, updates, and deletes—without the need for many complex queries.

Semi-structured data like JSON: The objects you're managing are semi-structured formats like JSON (which Postgres has robust support for.)

Single key: Your workload is mostly based on a single key, which you just have to create, read, update, and delete. (Therefore the majority of your transactions will only need to go to a single shard in the distributed Citus cluster.)

High throughput: Your throughput requirements are demanding and cannot be met by a single database server, on the order of 1000s or 10s of thousands of transactions per second.

Need relational database features: Some teams use NoSQL key-value stores for these types of semi-structured data-intensive workloads—but if you find yourself unwilling to go to NoSQL because there are relational database features you need, then Hyperscale (Citus) might be a good fit. Examples of key relational database features you might want to retain are strong consistency (not that eventual consistency compromise), foreign keys for referential integrity, triggers, and secondary indexes.



Choosing a shard key for Transactional / Operational Workloads

Shard Key Criteria

- Pick a column that application filters on most of the times (over 80% of the times) – **WHERE** clause filter on the column
  - Ex: SELECT * FROM events where device_id=1;
  - Avoids network/connection overhead, as query hits only one worker where distribution column exists – important for transactional workloads.
- If there is no such key, slight modification to below parameter should help
  - Adaptive executor is optimized to reduce network/connection overhead.
  - Can explicitly citus.**max_adaptive_executor_pool_size** to 1.

Shard Key Selection Summary

- **Multi-tenant –** "Column that is at the top of hierarchy OR the one that identifies the tenant"
  - Use foreign key graph and joins

- **Real-time Operational Analytics –** "Common column that co-locates large tables"
  - Use foreign keys and joins

- **High Throughput Transactional–** "Column that the application mostly filters on"
  - Use WHERE clause filters

Data Modelling

Table Types in Azure Cosmos DB for PostgreSQL

Distributed Table

A table split into multiple shards spread across multiple nodes.

Typical size > 10 GB

Reference Table

A table that is placed on all nodes in a Citus cluster.

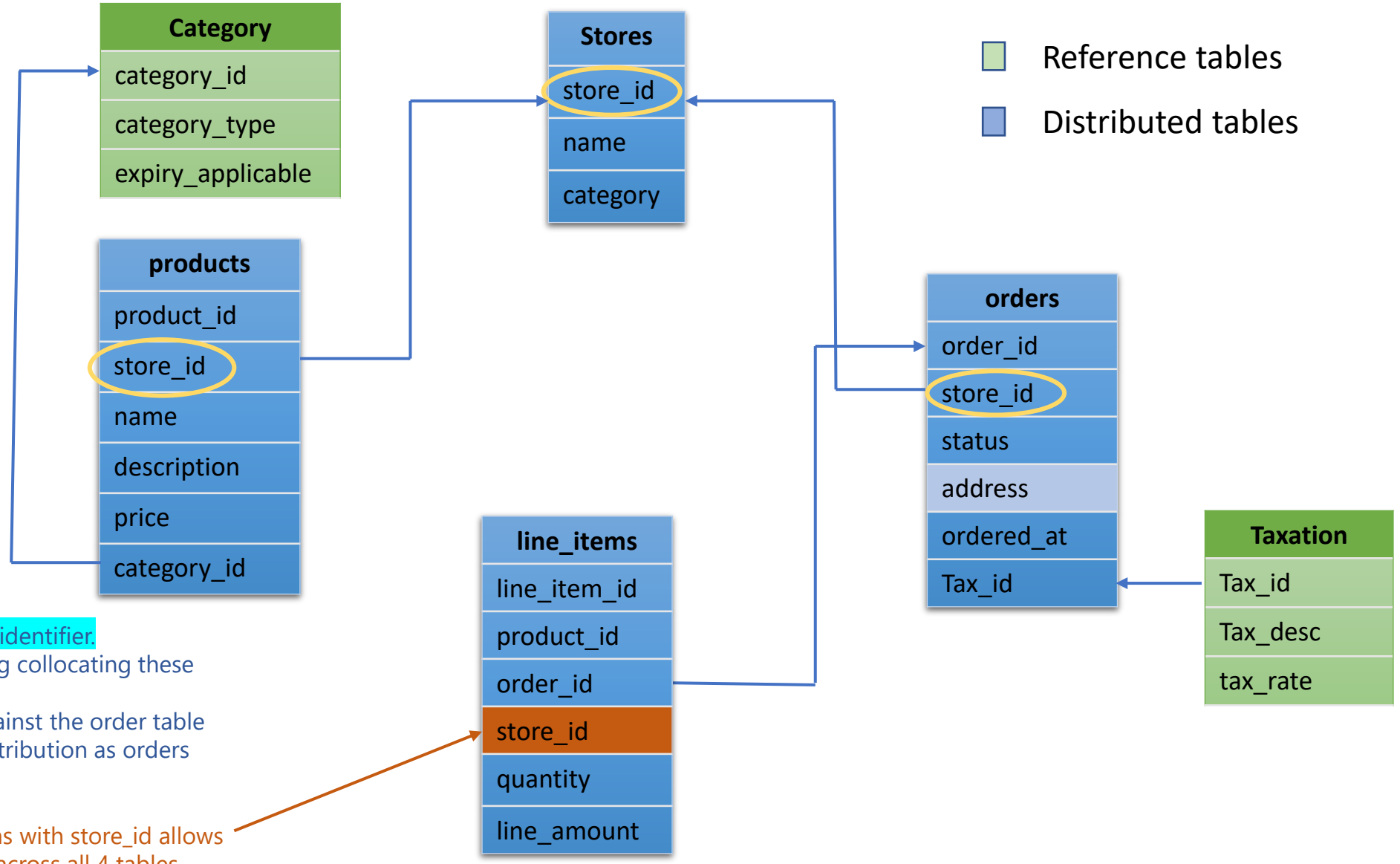
Typical size < 10 GB

Local Table

Tables that are created for administrative sections of your application, typically stored on coordinator

Data Modeling

Example:
Multi-Tenant e-commerce store



- Store_id appears as the tenant identifier.
- It's available in 3 tables allowing collocating these tables together
- Line_items would be joined against the order table often & thus requires same distribution as orders table.
  - Add field **store_id**
  - **Backfilling** the line items with store_id allows even distribution of set across all 4 tables

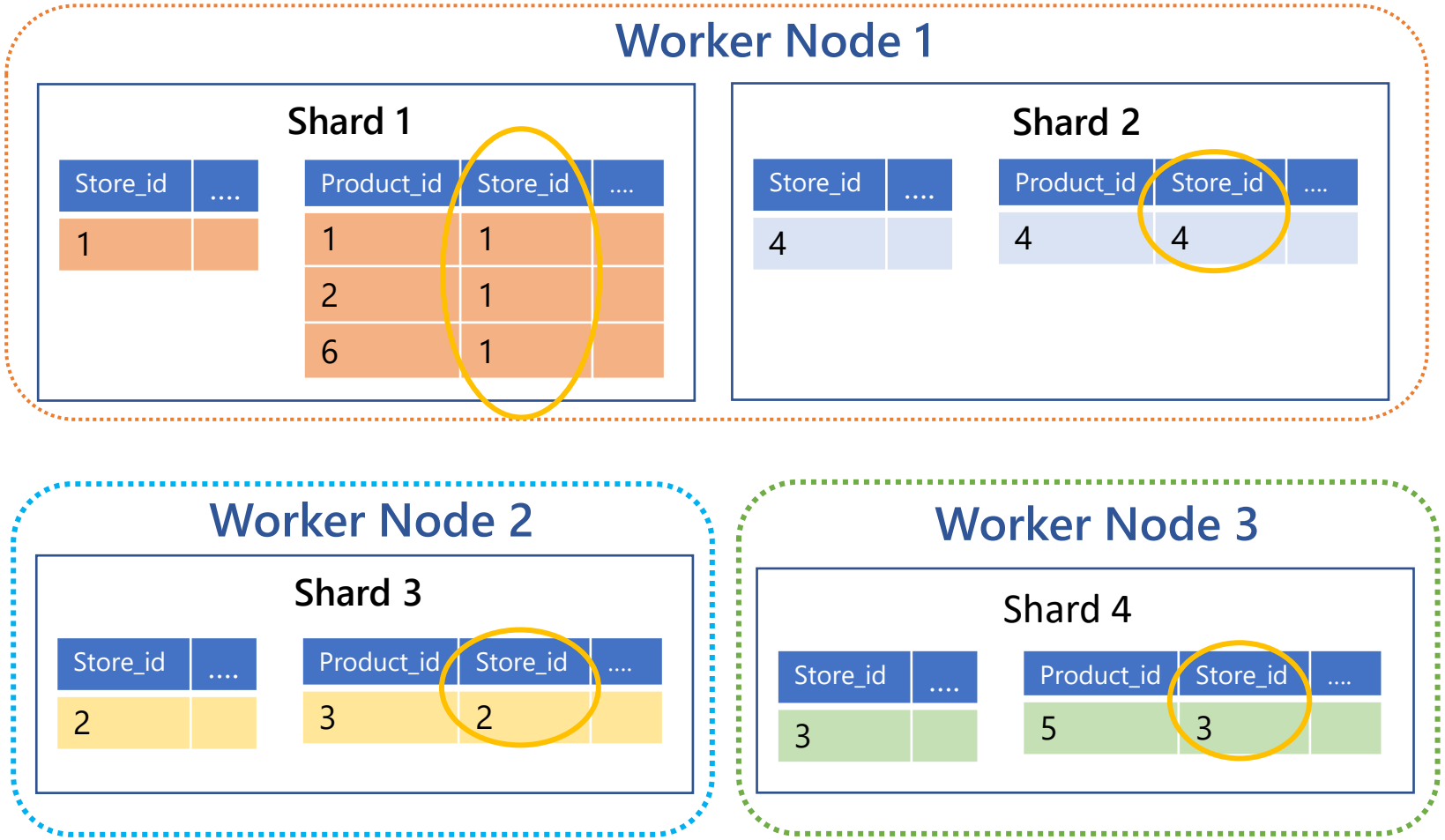
Multi-Tenant data placement

Application view

Product_id	Store_id	....
1	1	
2	1	
3	2	
4	4	
5	3	
6	1	

Store_id	....
1	
2	
3	
4	

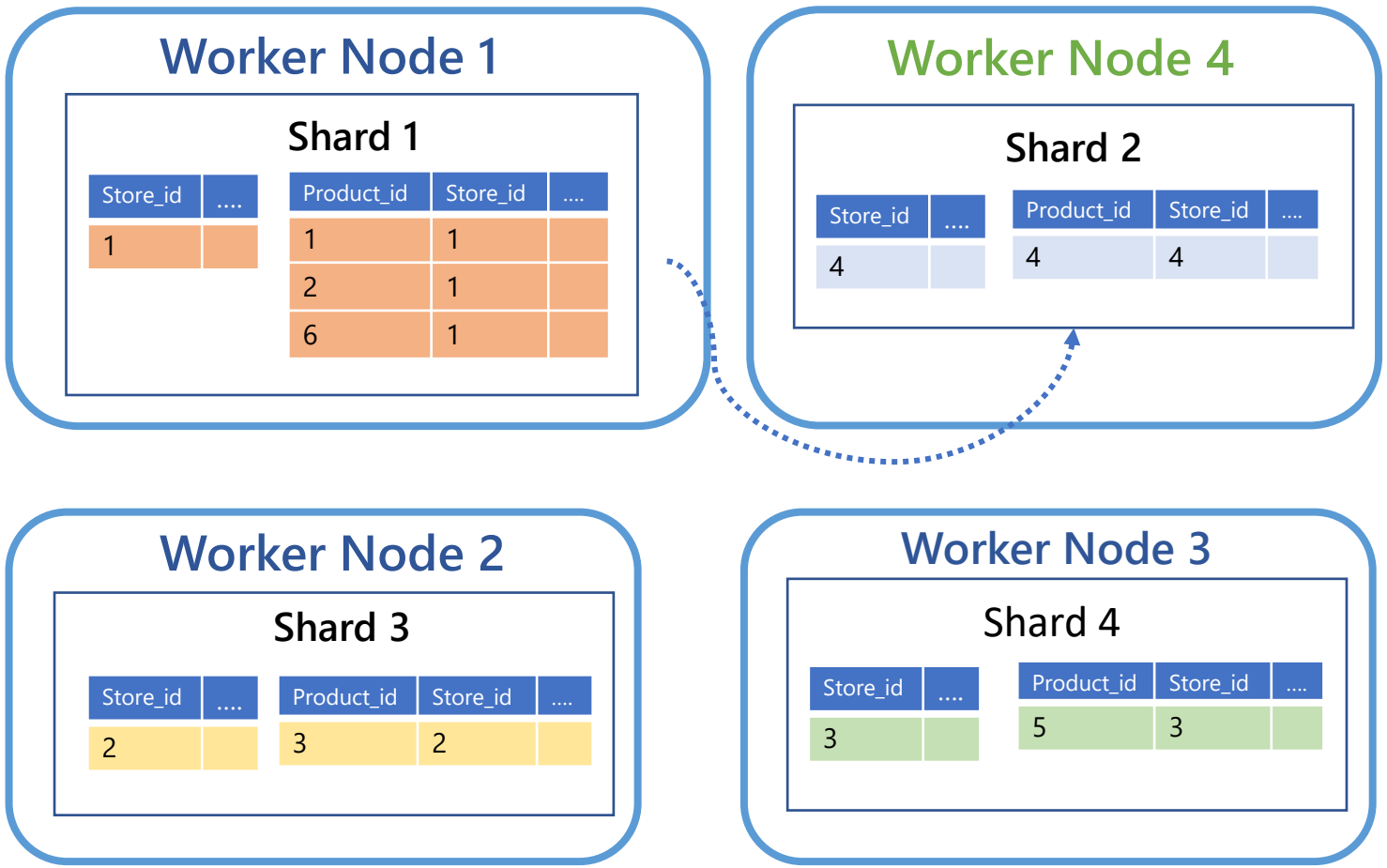
Internal Management in Database



Distribution Key : **Store_id**, thus we can see data distributed for different tenants being stored into separate shard.

NOTE : All 4 tables on previous slide will have same data placement based on store_id

Multi-Tenant Rebalancer



-- As store sales increases for stores we can scale out the cluster in following steps :

Step 1 : Provisioning an additional worker node through portal.

Step 2 : Trigger shard rebalancer

-- Data from Node 1 will be moved to Node 4

```
SELECT master_move_shard_placement(
shard_id, source_node_name,
source_node_port, target_node_name,
target_node_port);
```

Note: By default Shard rebalancer balances equal number of shards over the cluster without any parameters

Data Modeling (Multi-Tenant)

Querying Scenarios :

- Pull details for an order placed to a store. (Single shard query hits one node)

```
SELECT * FROM orders WHERE order_id = 123 and store_id = 3;
```

- Product category offered across stores (Parallelized search & Co-located data aggregation)

```
SELECT product_id, category_id, s.name, count(*) FROM products p
JOIN stores s on p.store_id = s.store_id
GROUP BY product_id, category_id, s.name;
```

- Look for product within a store \ brand (Single node Query with aggregation)

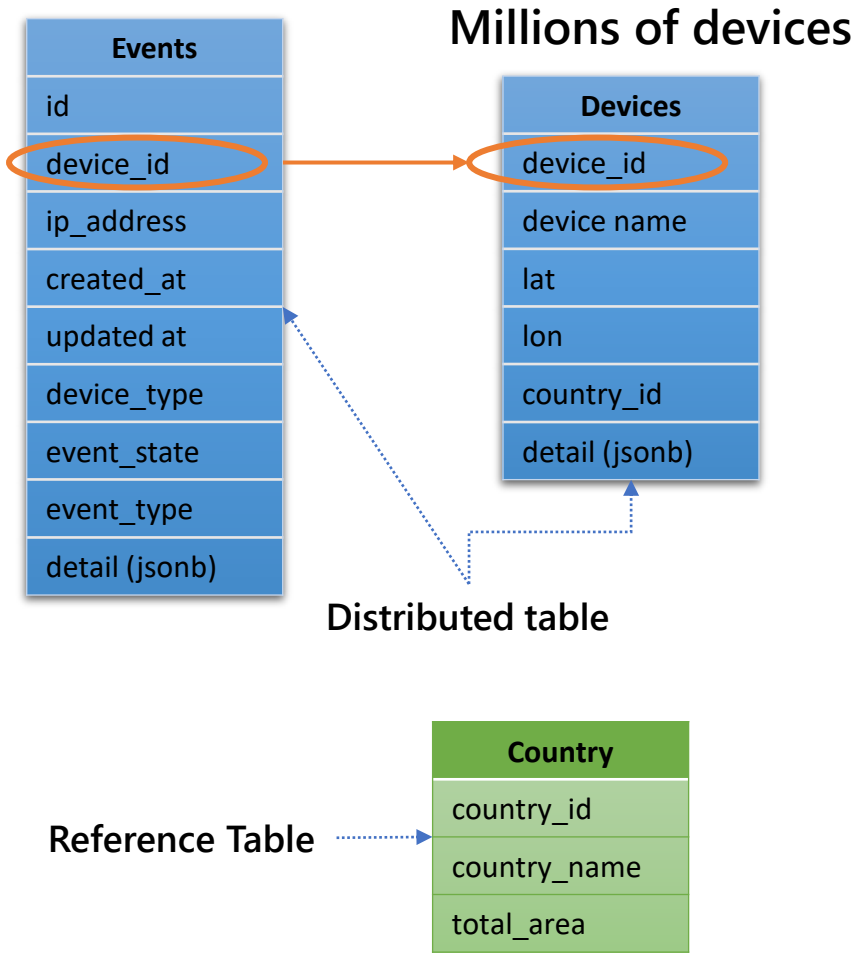
```
SELECT count(*) FROM products p
JOIN stores s on p.store_id = s.store_id
WHERE store_id = 123;
```

- Identify sales for given Store (Single node multi-shard query)

```
SELECT s.store_id, count(distinct p.product_id), count(distinct o.order_id)
FROM products p
JOIN stores s on p.store_id = s.store_id
JOIN orders o on s.store_id = o.order_id
WHERE store_id = 123
GROUP BY s.store_id;
```

Data Modeling (HTAP / Timeseries application)

>1 billion events



Query scenarios :

- Verify number of devices across a region.

```
SELECT COUNT(1) FROM Devices
WHERE ST_WITHIN(ST_MakePoint(lat, lon), YOUR_BOUNDARY_GEOM);
```

- Review device health by monitoring device specific events.

```
SELECT COUNT(1) FROM Devices d
 JOIN Events e ON d.device_id = e.device_id
 WHERE d.device_id = ‘123’;
```

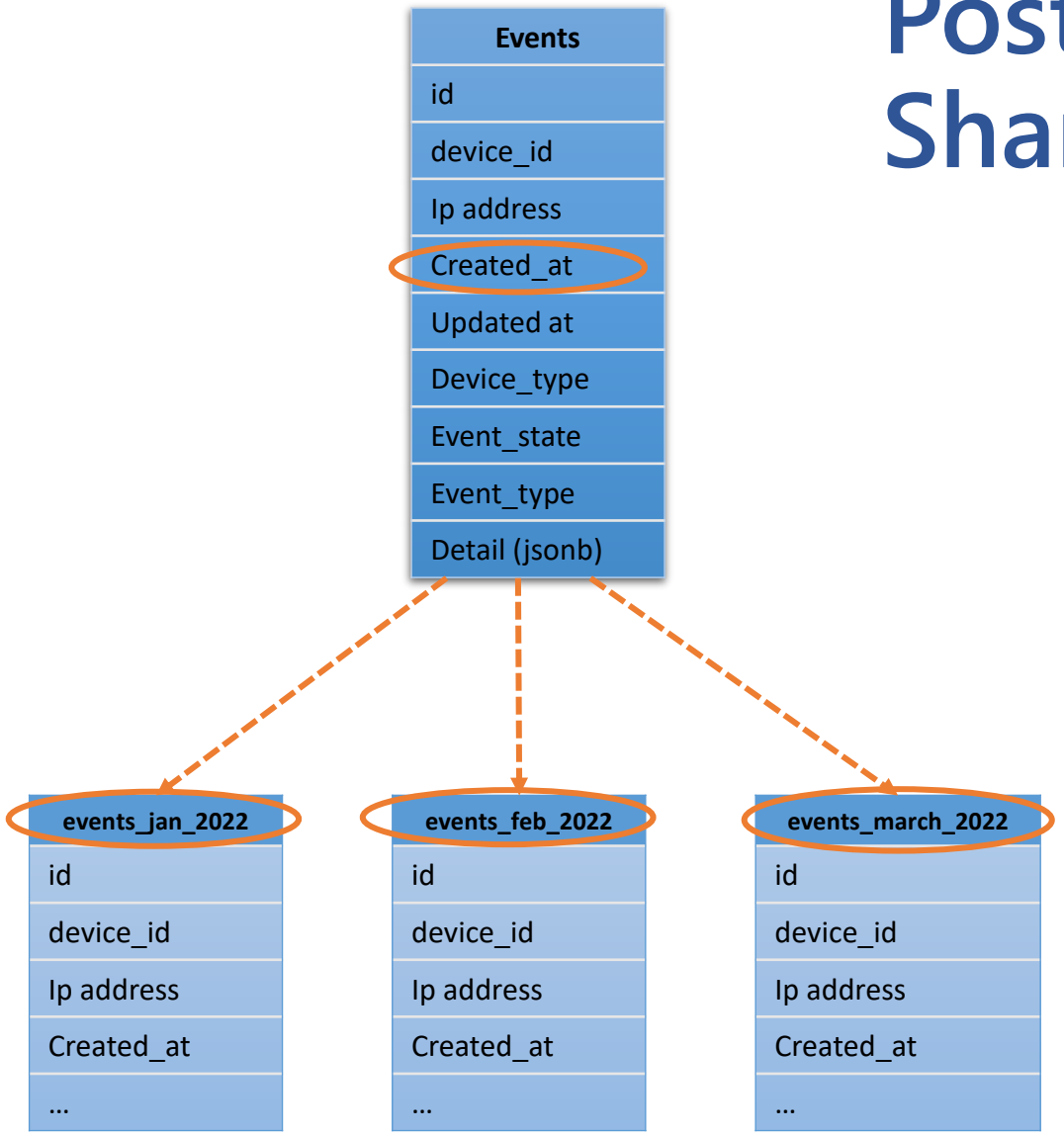
- Analyze device events within geography

```
SELECT event_type,eventstate,COUNT(1) FROM Devices d
JOIN Events e ON d.device_id = e.device_id
WHERE d.device_id = ‘123’ and
ST_WITHIN(ST_MakePoint(lat, lon), YOUR_BOUNDARY_GEOM)
GROUP BY event_type,eventstate;
```

- Review specific event types

```
SELECT event_type, Count(device_id) FROM Events
WHERE created_at > current_date - interval '7 days’
GROUP BY event_type;
```

Postgres Partitioning post Sharding



In addition to **sharding the tables on device_id** and Co-locating devices and users table, another level of distribution could be achieved through partitioning.

- *Core benefits of the approach is to limit the query to latest data*
- *Easy archival of the dataset*
- *Easier management through Scheduling with Pg_cron for activities such as indexing, vacuuming of datasets*

.....

```
SELECT create_time_partitions(
  table_name          := 'Events',
  partition_interval  := '1 month',
  end_at              := now() + '12 months'
);
```

Parameter tuning for specific workload

- citus.max_adaptive_executor_pool_size : Total number of connections that a single query (cross shard) can generate per worker node. *If there are performance issues with higher concurrency, try to reduce this to smaller number.* Low value like 1 or 2 for transactional workloads with short queries (<20 ms) should help.

- citus.task_assignment_policy : Affects queries with reference table access by choosing to read data in round robin manner. Reducing load on single worker node.

- citus.explain_all_tasks : Details, all tasks generated in execution plan.

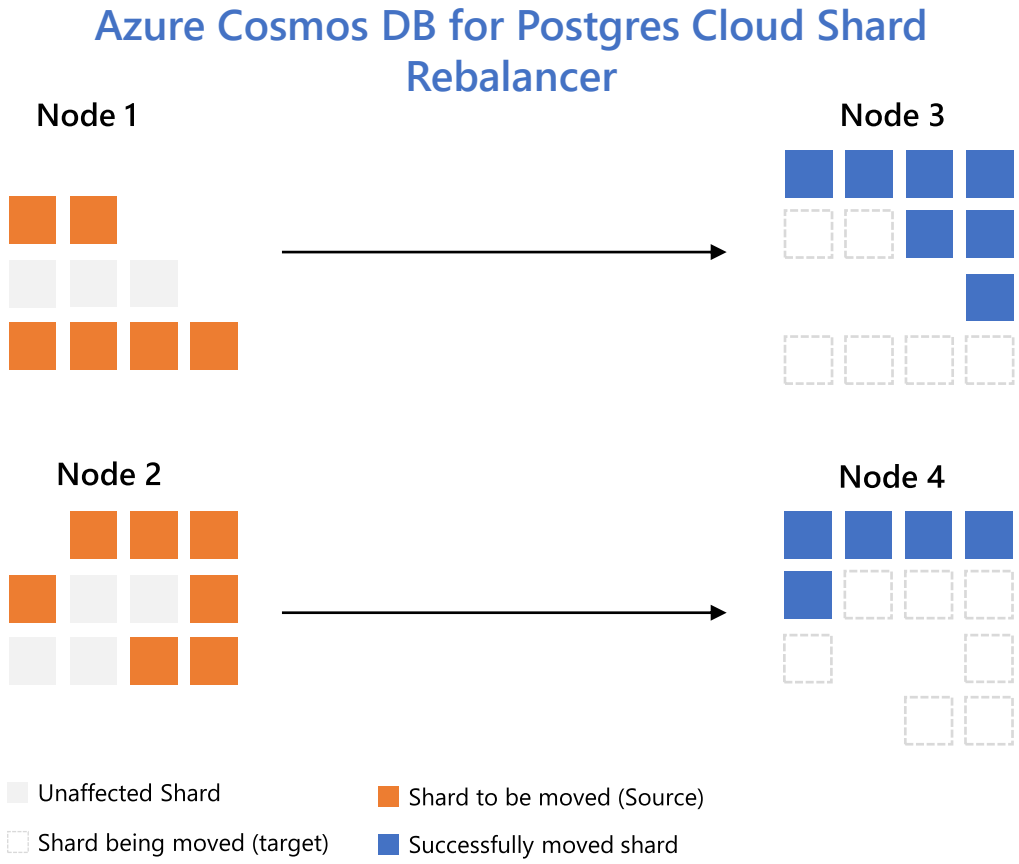
- citus.max_intermediate_result_size : The maximum size in KB of intermediate results for CTEs or sub-queries that are unable to be pushed down to worker nodes for execution. Network becomes bottleneck if intermediate result is large. The default is 1GB.

- citus.node_connection_timeout : Sets the maximum duration (in milliseconds) to wait for connection establishment. This GUC affects connections from the coordinator to workers, and workers to each other. Default : 30 secs.

- citus.max_cached_conns_per_worker : Each backend opens connections to the workers to query the shards. At the end of the transaction, the configured number of connections is kept open to speed up subsequent commands. Increasing this value will reduce the latency of multi-shard queries.

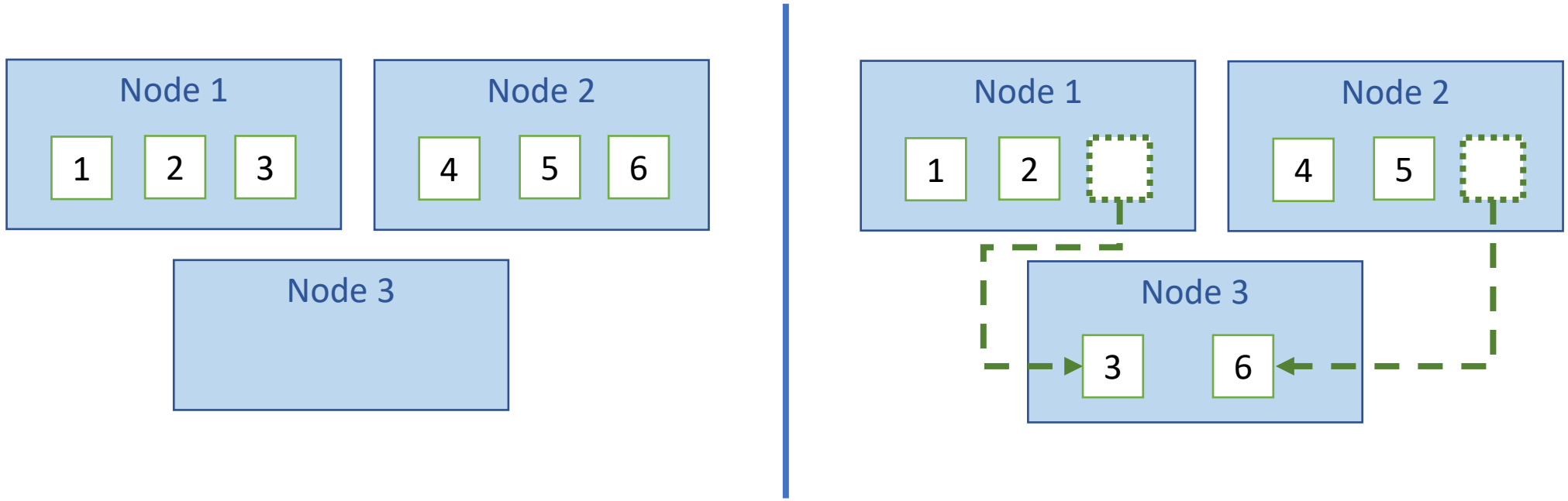
Online Rebalancer
Powered By Citus

- Adding a new node to cluster **doesn't trigger** rebalancing by default.
- Shard rebalancer redistributes shards across old and new worker nodes for **balanced data scale-out**
- Shard rebalancer will **recommend** rebalance when shards can be placed more evenly
- For more control, use tenant isolation to easily allocate dedicated resource to specific tenants with greater needs



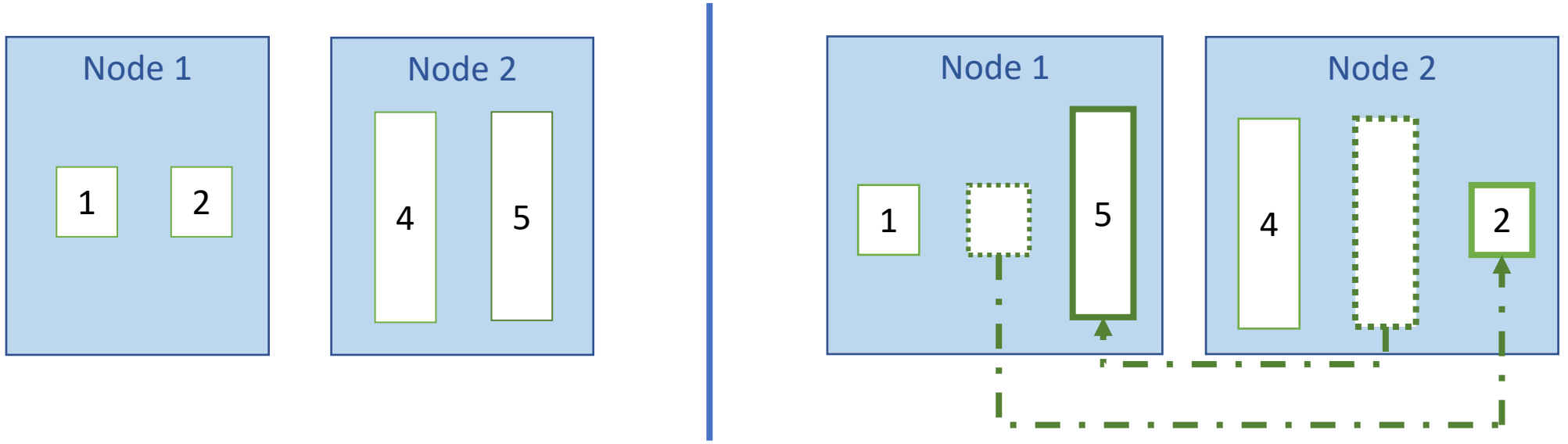
Data Distribution Strategy

```
SELECT rebalance_table_shards(rebalance_strategy := 'by_shard_count');     (Default Strategy)
```



Data Distribution Strategy

```
SELECT rebalance_table_shards(rebalance_strategy := 'by_disk_size');
```



There is flexibility for users to create self defined custom strategies through UDFs

Query Patterns

Shard Creation

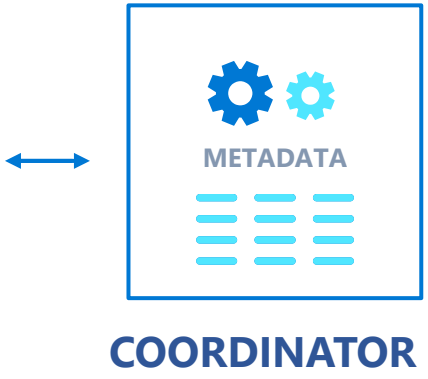
Distributes tables across the cluster

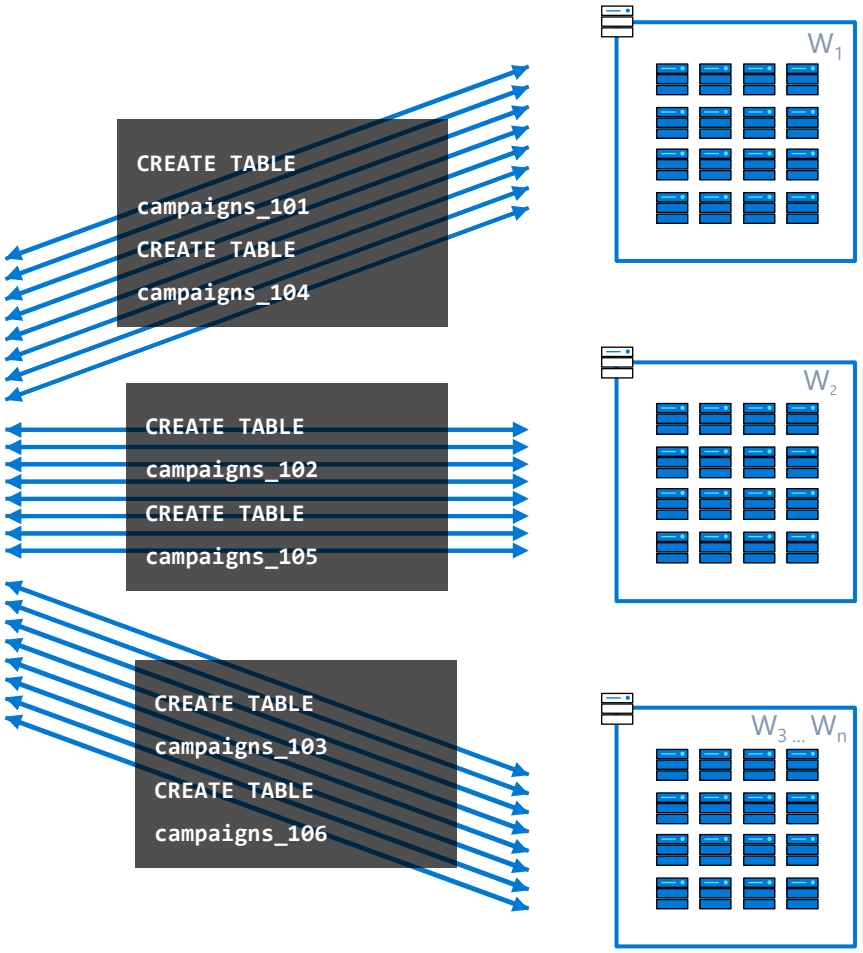
APPLICATION

Decorative

```
CREATE TABLE campaigns (…);

   SELECT create_distributed_table(
          'campaigns','company_id',
           shard_count:= 48);
```





Sample table shards

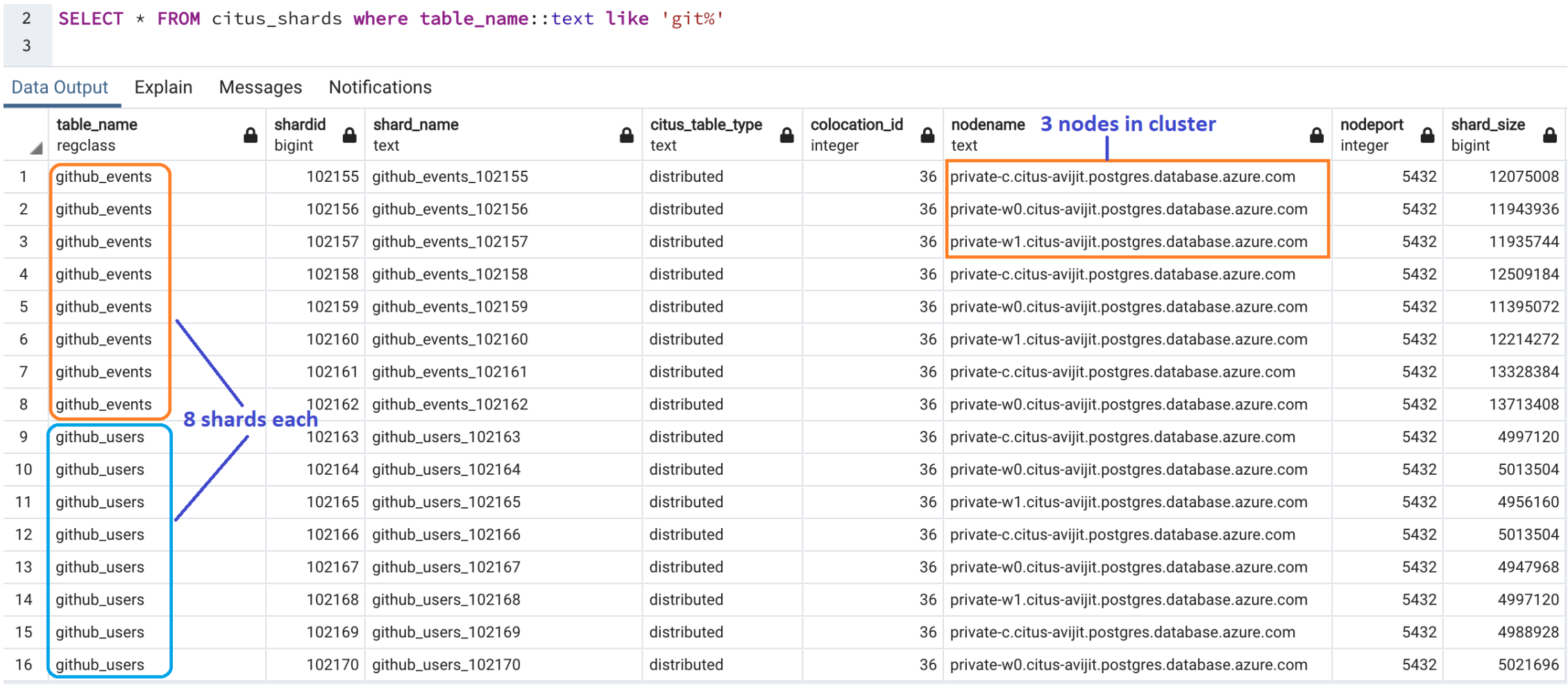
SELECT create_distributed_table_concurrently('github_users','event_id', shard_count:= 8);

SELECT create_distributed_table_concurrently('github_events','event_id', shard_count:= 8);

```
2  SELECT * FROM citus_shards where table_name::text like 'git%'
3
```

Data Output | Explain | Messages | Notifications

	table_name regclass	shardid bigint	shard_name text	citus_table_type text	colocation_id integer	nodename text	nodeport integer	shard_size bigint
1	github_events	102155	github_events_102155	distributed	36	private-c.citus-avijit.postgres.database.azure.com	5432	12075008
2	github_events	102156	github_events_102156	distributed	36	private-w0.citus-avijit.postgres.database.azure.com	5432	11943936
3	github_events	102157	github_events_102157	distributed	36	private-w1.citus-avijit.postgres.database.azure.com	5432	11935744
4	github_events	102158	github_events_102158	distributed	36	private-c.citus-avijit.postgres.database.azure.com	5432	12509184
5	github_events	102159	github_events_102159	distributed	36	private-w0.citus-avijit.postgres.database.azure.com	5432	11395072
6	github_events	102160	github_events_102160	distributed	36	private-w1.citus-avijit.postgres.database.azure.com	5432	12214272
7	github_events	102161	github_events_102161	distributed	36	private-c.citus-avijit.postgres.database.azure.com	5432	13328384
8	github_events	102162	github_events_102162	distributed	36	private-w0.citus-avijit.postgres.database.azure.com	5432	13713408
9	github_users	102163	github_users_102163	distributed	36	private-c.citus-avijit.postgres.database.azure.com	5432	4997120
10	github_users	102164	github_users_102164	distributed	36	private-w0.citus-avijit.postgres.database.azure.com	5432	5013504
11	github_users	102165	github_users_102165	distributed	36	private-w1.citus-avijit.postgres.database.azure.com	5432	4956160
12	github_users	102166	github_users_102166	distributed	36	private-c.citus-avijit.postgres.database.azure.com	5432	5013504
13	github_users	102167	github_users_102167	distributed	36	private-w0.citus-avijit.postgres.database.azure.com	5432	4947968
14	github_users	102168	github_users_102168	distributed	36	private-w1.citus-avijit.postgres.database.azure.com	5432	4997120
15	github_users	102169	github_users_102169	distributed	36	private-c.citus-avijit.postgres.database.azure.com	5432	4988928
16	github_users	102170	github_users_102170	distributed	36	private-w0.citus-avijit.postgres.database.azure.com	5432	5021696

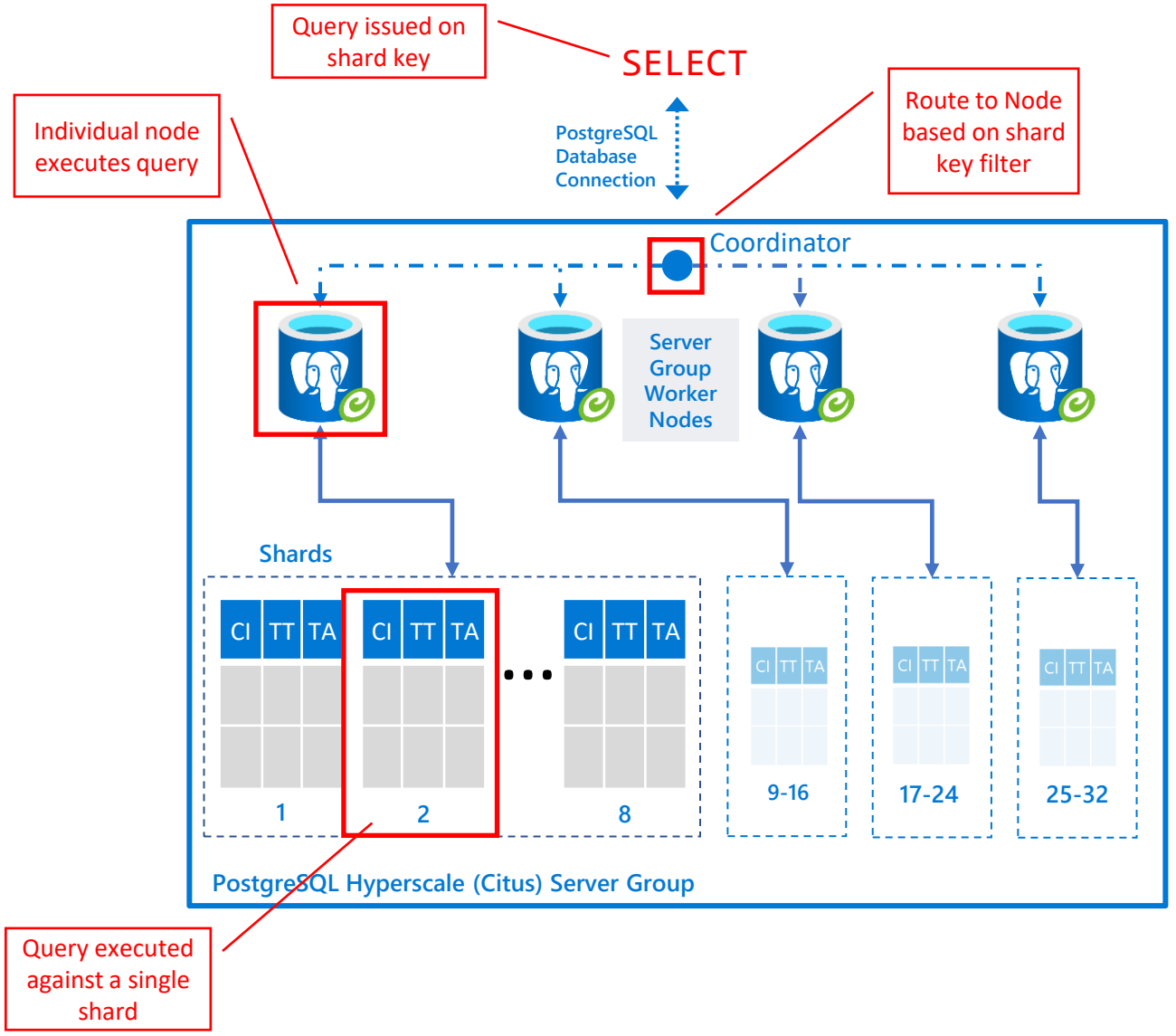


Query with Filter

APPLICATION

Coordinator node determines which node owns the shard and routes the query

Query is executed on the worker node that owns the specific shard



Query plan

Single node native Postgres

```
EXPLAIN ANALYSE
SELECT * FROM github_events;
```

ta Output | Explain | Messages | Notifications

QUERY PLAN
text

Seq Scan on github_events (cost=0.00..13350.45 rows=126245 width=161) (actual time=0.006..26.018 rows=126245 loops=1)

Planning Time: 1.274 ms

Execution Time: 30.345 ms

All data resides on same node, resources available on single node gets allocated

Azure Cosmos DB for PostgreSQL (Multi-Node)

```
EXPLAIN ANALYSE
SELECT * FROM github_events;
```

Output | Explain | Messages | Notifications

QUERY PLAN
text

Custom Scan (Citus Adaptive) (cost=0.00..0.00 rows=100000 width=161) (actual time=4225.904..4331.864 rows=126245 loops=1)

[...] Task Count: 8

[...] Tuple data received from nodes: 280 MB

[...] Tasks Shown: All

[...] -> Task

[...] Tuple data received from node: 36 MB

[...] Node: host=private-c.citus-avijit.postgres.database.azure.com port=5432 dbname=citus

[...] -> Seq Scan on github_events_102161 github_events (cost=0.00..1797.34 rows=17034 width=832) (actual time=0.006..3.059 rows=17034 loop...

[...] Planning Time: 0.829 ms

[...] Execution Time: 16.862 ms

[...] -> Task

[...] Tuple data received from node: 41 MB

[...] Node host=private-w0.citus-avijit.postgres.database.azure.com port=5432 dbname=citus

[...] -> Seq Scan on github_events_102162 github_events (cost=0.00..1947.55 rows=17355 width=858) (actual time=0.009..2.510 rows=17355 loop...

[...] Planning Time: 0.595 ms

[...] Execution Time: 13.926 ms

[...] -> Task

[...] Tuple data received from node: 34 MB

[...] Node host=private-c.citus-avijit.postgres.database.azure.com port=5432 dbname=citus

[...] -> Seq Scan on github_events_102158 github_events (cost=0.00..1685.77 rows=15877 width=860) (actual time=0.050..2.456 rows=15877 loop...

[...] Planning Time: 0.934 ms

[...] Execution Time: 13.844 ms

[...] -> Task

[...] Tuple data received from node: 35 MB

[...] Node: host=private-w1.citus-avijit.postgres.database.azure.com port=5432 dbname=citus

[...] -> Seq Scan on github_events_102160 github_events (cost=0.00..1642.55 rows=15155 width=877) (actual time=0.004..1.798 rows=15155 loop...

[...] Planning Time: 0.542 ms

[...] Execution Time: 11.258 ms

Different shards on same node

Task created per shard hosted on different nodes. Multiple tasks created for case where multiple shards hosted on same node

Query plan (Count(*))

Single node native Postgres

```
EXPLAIN
SELECT COUNT(*) FROM github_events
```

a Output | Explain | Messages | Notifications

QUERY PLAN
text

Aggregate (cost=13653.06..13653.07 rows=1 width=8)

[...] -> Seq Scan on github_events (cost=0.00..13337.45 rows=126245 width=0)

Azure Cosmos DB for Postgres (Multi-Node)

```
EXPLAIN
SELECT COUNT(*) FROM github_events
```

Output | Explain | Messages | Notifications

QUERY PLAN
text

Aggregate (cost=250.00..250.02 rows=1 width=8)

[...] -> Custom Scan (Citus Adaptive) (cost=0.00..0.00 rows=100000 width=8)

[...] Task Count: 8

8 threads created to execute in parallel across nodes

[...] Tasks Shown: All

[...] -> Task

[...] Node: host=private-c.citus-avijit.postgres.database.azure.com port=5432 dbname=citus

[...] -> Aggregate (cost=318.08..318.09 rows=1 width=8)

[...] -> Index Only Scan using event_type_index_102441 on github_events_102441 github_events (cost=0.29..278.40 rows=15874 width...

[...] -> Task

[...] Node host=private-w0.citus-avijit.postgres.database.azure.com port=5432 dbname=citus

[...] -> Aggregate (cost=2435.40..2435.41 rows=1 width=8)

[...] -> Seq Scan on github_events_102442 github_events (cost=0.00..2265.12 rows=68112 width=0)

[...] -> Task

[...] Node host=private-w1.citus-avijit.postgres.database.azure.com port=5432 dbname=citus

[...] -> Aggregate (cost=2455.39..2455.40 rows=1 width=8)

[...] -> Seq Scan on github_events_102443 github_events (cost=0.00..2283.71 rows=68671 width=0)

[...] -> Task

[...] Node host=private-w2.citus-avijit.postgres.database.azure.com port=5432 dbname=citus

[...] -> Aggregate (cost=2430.79..2430.80 rows=1 width=8)

[...] -> Seq Scan on github_events_102444 github_events (cost=0.00..2260.83 rows=67983 width=0)

Each node receiving request in parallel.

Query plan (Where)

Query with filter on shard key

```
EXPLAIN ANALYSE
SELECT * FROM github_events
WHERE user_id = 139;
```

a Output | Explain | Messages | Notifications

QUERY PLAN
text

Custom Scan (Citus Adaptive) (cost=0.00..0.00 rows=0 width=0) (actual time=53.185..53.186 rows=0 loops=1)
[...] Task Count: 1
[...] Tuple data received from nodes: 0 bytes
[...] Tasks Shown: All
[...] -> Task
[...] Tuple data received from node: 0 bytes
[...] Node host=private-c.citus-avijit.postgres.database.azure.com port=5432 dbname=citus
[...] -> Seq Scan on github_events_102155 github_events (cost=0.00..1670.92 rows=2 width=834) (actual time=5.412..5.413 rows=0 loops=1)
[...] Filter: (user_id = 139)
[...] Rows Removed by Filter: 15754
[...] Planning Time: 26.741 ms
[...] Execution Time: 5.450 ms
Planning Time: 1.135 ms
Execution Time: 53.268 ms

Single node being hit scanning only 1 shard.

Query without filter

```
EXPLAIN ANALYSE
SELECT * FROM github_events;
```

Output | Explain | Messages | Notifications

QUERY PLAN
text

Custom Scan (Citus Adaptive) (cost=0.00..0.00 rows=100000 width=161) (actual time=4225.904..4331.864 rows=126245 loops=1)
[...] Task Count: 8
[...] Tuple data received from nodes: 280 MB
[...] Tasks Shown: All
[...] -> Task
[...] Tuple data received from node: 36 MB
[...] Node: host=private-c.citus-avijit.postgres.database.azure.com port=5432 dbname=citus
[...] -> Seq Scan on github_events_102161 github_events (cost=0.00..1797.34 rows=17034 width=832) (actual time=0.006..3.059 rows=17034 loop...
[...] Planning Time: 0.829 ms
[...] Execution Time: 16.862 ms
[...] -> Task
[...] Tuple data received from node: 41 MB
[...] Node host=private-w0.citus-avijit.postgres.database.azure.com port=5432 dbname=citus
[...] -> Seq Scan on github_events_102162 github_events (cost=0.00..1647.55 rows=17355 width=858) (actual time=0.009..2.510 rows=17355 loop...
[...] Planning Time: 0.595 ms
[...] Execution Time: 13.926 ms
[...] -> Task
[...] Tuple data received from node: 34 MB
[...] Node host=private-c.citus-avijit.postgres.database.azure.com port=5432 dbname=citus
[...] -> Seq Scan on github_events_102158 github_events (cost=0.00..1685.77 rows=15877 width=860) (actual time=0.050..2.456 rows=15877 loop...
[...] Planning Time: 0.934 ms
[...] Execution Time: 13.844 ms
[...] -> Task
[...] Tuple data received from node: 35 MB
[...] Node: host=private-w1.citus-avijit.postgres.database.azure.com port=5432 dbname=citus
[...] -> Seq Scan on github_events_102160 github_events (cost=0.00..1642.55 rows=15155 width=877) (actual time=0.004..1.798 rows=15155 loop...
[...] Planning Time: 0.542 ms
[...] Execution Time: 11.258 ms

Different shards on same node

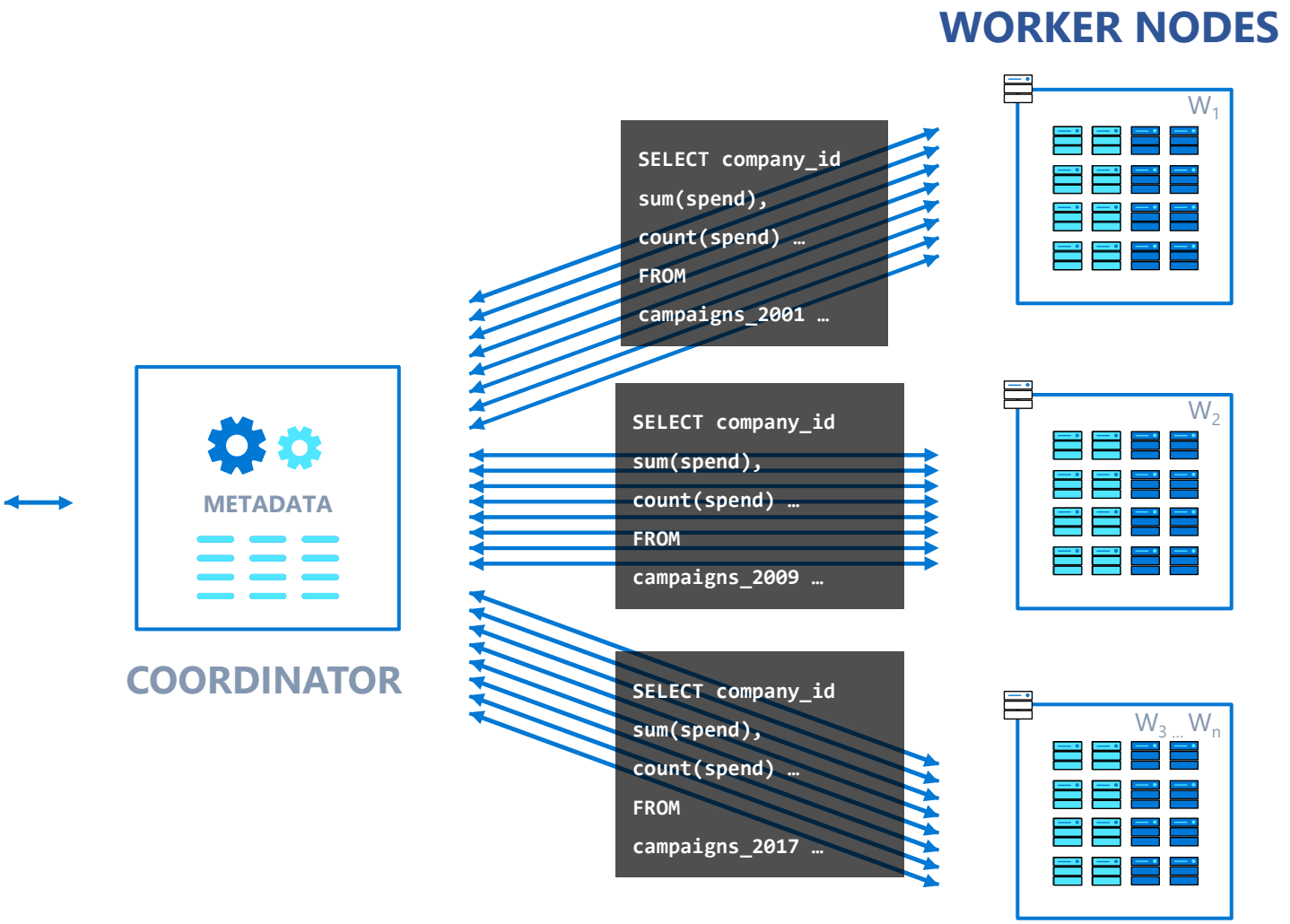
Task created per shard hosted on different nodes. Multiple tasks created for case where multiple shards hosted on same node

Query with Group By

Distributes queries across the cluster

APPLICATION

```
SELECT  company_id,
        avg(spend) AS avg_campaign_spend
  FROM  campaigns
GROUP BY company_id;
```



Query plan (Group By)

“Group By” on Single node Postgres

```
EXPLAIN ANALYSE
SELECT user_id,count(*) FROM github_events
where user_id in (2354108,10810283)
GROUP BY user_id;
```

a Output | Explain | Messages | Notifications

1 row returned for each user. All records accessed in succession

QUERY PLAN
text

GroupAggregate (cost=13710.06..13721.53 rows=200 width=16) (actual time=43.671..43.75 rows=2 loops=1)
[...] Group Key: user_id
[...] -> Sort (cost=13710.06..13713.22 rows=1262 width=8) (actual time=43.642..43.67 rows=765 loops=1)
[...] Sort Key: user_id
[...] Sort Method: quicksort Memory: 60kB
[...] -> Seq Scan on github_events (cost=0.00..13645.06 rows=1262 width=8) (actual time=0.575..43.530 rows=765 loops=1)
[...] Filter: (user_id = ANY ('{2354108,10810283}'::bigint[]))
[...] Rows Removed by Filter: 125480
Planning Time: 0.325 ms
Execution Time: 43.789 ms

“Group By” on Azure Cosmos DB for Postgres

```
EXPLAIN ANALYSE
SELECT user_id,count(*) FROM github_events
where user_id in (2354108,10810283)
GROUP BY user_id;
```

ta Output | Explain | Messages | Notifications

QUERY PLAN
text

Custom Scan (Citus Adaptive) (cost=0.00..0.00 rows=100000 width=16) (actual time=3.482..3.483 rows=2 loops=1)
Aggregated data returned for 2 users
[...] Task Count: 2
[...] Tuple data received from nodes: 32 bytes
[...] Tasks Shown: All
[...] -> Task
[...] Tuple data received from node: 16 bytes
[...] Node: host=private-w0.citus-avijit.postgres.database.azure.com port=5432 dbname=citus
[...] -> HashAggregate (cost=490.65..492.03 rows=138 width=16) (actual time=1.798..1.799 rows=1 loops=1)
[...] Group Key: user_id
[...] Batches: 1 Memory Usage: 40kB
[...] -> Seq Scan on github_events_102374 github_events (cost=0.00..489.94 rows=142 width=8) (actual time=0.011..1.763 rows=140 loops=1)
[...] Filter: (user_id = ANY ('{2354108,10810283}'::bigint[]))
[...] Rows Removed by Filter: 3935
User data split across 2 nodes, parallel execution for respective dataset
[...] Planning Time: 0.086 ms
[...] Execution Time: 1.831 ms
[...] -> Task
[...] Tuple data received from node: 16 bytes
[...] Node: host=private-c.citus-avijit.postgres.database.azure.com port=5432 dbname=citus
[...] -> HashAggregate (cost=469.72..475.07 rows=535 width=16) (actual time=1.566..1.570 rows=1 loops=1)
[...] Group Key: user_id
[...] Batches: 1 Memory Usage: 49kB
[...] -> Seq Scan on github_events_102353 github_events (cost=0.00..466.59 rows=627 width=8) (actual time=0.007..1.475 rows=625 loops=1)
[...] Filter: (user_id = ANY ('{2354108,10810283}'::bigint[]))
[...] Rows Removed by Filter: 3742
[...] Planning Time: 0.066 ms
[...] Execution Time: 1.599 ms
Planning Time: 0.377 ms
Execution Time: 3.502 ms

Parallelize execution with Group by

```
EXPLAIN ANALYSE
SELECT user_id,count(*) FROM github_events
where user_id in (2354108,10810283)
GROUP BY user_id;
```

ta Output | Explain | Messages | Notifications

QUERY PLAN
text

```
Custom Scan (Citus Adaptive)  (cost=0.00..0.00 rows=100000 width=16) (actual time=3.482..3.483 rows=2 loops=1)
[...] Task Count: 2
[...] Tuple data received from nodes: 32 bytes
[...] Tasks Shown: All
[...] ->  Task
[...] Tuple data received from node: 16 bytes
[...] Node: host=private-w0.citus-avijit.postgres.database.azure.com port=5432 dbname=citus
[...] ->  HashAggregate  (cost=490.65..492.03 rows=138 width=16) (actual time=1.798..1.799 rows=1 loops=1)
[...] Group Key: user_id
[...] Batches: 1  Memory Usage: 40kB
[...] ->  Seq Scan on github_events_102374 github_events  (cost=0.00..489.94 rows=142 width=8) (actual time=0.011..1.763 rows=140 loops=1)
[...] Filter: (user_id = ANY ('{2354108,10810283}'::bigint[]))
[...] Rows Removed by Filter: 3935
[...] Planning Time: 0.086 ms
[...] Execution Time: 1.831 ms
[...] ->  Task
[...] Tuple data received from node: 16 bytes
[...] Node: host=private-c.citus-avijit.postgres.database.azure.com port=5432 dbname=citus
[...] ->  HashAggregate  (cost=469.72..475.07 rows=535 width=16) (actual time=1.566..1.570 rows=1 loops=1)
[...] Group Key: user_id
[...] Batches: 1  Memory Usage: 49kB
[...] ->  Seq Scan on github_events_102353 github_events  (cost=0.00..466.59 rows=627 width=8) (actual time=0.007..1.475 rows=625 loops=1)
[...] Filter: (user_id = ANY ('{2354108,10810283}'::bigint[]))
[...] Rows Removed by Filter: 3742
[...] Planning Time: 0.066 ms
[...] Execution Time: 1.599 ms
Planning Time: 0.377 ms
Execution Time: 3.502 ms
```

Aggregated data returned for 2 users

User data split across 2 nodes, parallel execution for respective dataset

CO-LOCATION

> Data distributed against common shard key are collocated on same shards

	Github_User		Github_Event	
	Shard	Hash Range	Shard	Hash Range
NODE 1	1	-2147483648 <= x <= -1610612737	11	-2147483648 <= x <= -1610612737
	3	-1073741825 <= x <= -536870913	12	-1073741825 <= x <= -536870913
NODE 2	2	-1610612737 <= x <= -1073741825	23	-1610612737 <= x <= -1073741825
	4	-536870913 <= x <= -1	30	-536870913 <= x <= -1

Query plan (JOIN)

Single node Postgres

```
EXPLAIN ANALYSE
SELECT * FROM github_events ge
JOIN github_users gu ON ge.user_id = gu.user_id
where gu.user_id in (2354108,10810283);
```

Output | Explain | Messages | Notifications

QUERY PLAN
text

Much larger number of rows reviewed resulting in higher execution time.

```
Gather  (cost=14250.55..21997.10 rows=8890 width=1016) (actual time=118.621..134.878 rows=765 loops=1)
[...] Workers Planned: 2
[...] Workers Launched: 2
[...] ->  Parallel Hash Join  (cost=13250.55..20108.10 rows=3704 width=1016) (actual time=92.044..93.186 rows=255 loops=3)
[...] Hash Cond: (gu.user_id = ge.user_id)
[...] ->  Parallel Seq Scan on github_users gu  (cost=0.00..5912.38 rows=832 width=168) (actual time=12.899..12.910 rows=1 loops=3)
[...] Filter: (user_id = ANY ('{2354108,10810283}'::bigint[]))
[...] Rows Removed by Filter: 88102
[...] ->  Parallel Hash  (cost=12593.02..12593.02 rows=52602 width=848) (actual time=78.443..78.444 rows=42082 loops=3)
[...] Buckets: 131072  Batches: 1  Memory Usage: 97472kB
[...] ->  Parallel Seq Scan on github_events ge  (cost=0.00..12593.02 rows=52602 width=848) (actual time=0.004..6.763 rows=42082 loops=3)
Planning Time: 0.236 ms
Execution Time: 134.972 ms
```

Azure Cosmos DB for Postgres

```
EXPLAIN ANALYSE
SELECT * FROM github_events ge
JOIN github_users gu ON ge.user_id = gu.user_id
where gu.user_id in (2354108,10810283);
```

ta Output | Explain | Messages | Notifications

QUERY PLAN
text

```
Custom Scan (Citus Adaptive)  (cost=0.00..0.00 rows=100000 width=329) (actual time=37.807..37.881 rows=765 loops=1)
[...] Task Count: 2
[...] Tuple data received from nodes: 2624 kB
[...] Tasks Shown: All
[...] ->  Task
[...] Tuple data received from node: 631 kB
[...] Node: host=private-w0.citus-avijit.postgres.database.azure.com port=5432 dbname=citus
[...] ->  Hash Join  (cost=236.75..1111.70 rows=6027 width=329) (actual time=1.443..3.519 rows=140 loops=1)
[...] Hash Cond: (ge.user_id = gu.user_id)
[...] ->  Seq Scan on github_events_102438 ge  (cost=0.00..626.34 rows=18834 width=161) (actual time=0.010..1.184 rows=4075 loops=1)
[...] ->  Hash  (cost=235.95..235.95 rows=64 width=168) (actual time=1.416..1.417 rows=1 loops=1)
[...] Buckets: 1024  Batches: 1  Memory Usage: 9kB
[...] ->  Seq Scan on github_users_102406 gu  (cost=0.00..235.95 rows=64 width=168) (actual time=0.330..1.410 rows=1 loops=1)
[...] Filter: (user_id = ANY ('{2354108,10810283}'::bigint[]))
[...] Rows Removed by Filter: 8415
[...] Planning Time: 0.800 ms
[...] Execution Time: 3.747 ms
[...] ->  Task
[...] Tuple data received from node: 1993 kB
[...] Node: host=private-c.citus-avijit.postgres.database.azure.com port=5432 dbname=citus
[...] ->  Hash Join  (cost=229.16..1024.49 rows=5319 width=329) (actual time=1.285..3.088 rows=625 loops=1)
[...] Hash Cond: (ge.user_id = gu.user_id)
[...] ->  Seq Scan on github_events_102417 ge  (cost=0.00..570.57 rows=17157 width=161) (actual time=0.007..0.920 rows=4367 loops=1)
[...] ->  Hash  (cost=228.39..228.39 rows=62 width=168) (actual time=1.265..1.266 rows=1 loops=1)
[...] Buckets: 1024  Batches: 1  Memory Usage: 9kB
[...] ->  Seq Scan on github_users_102385 gu  (cost=0.00..228.39 rows=62 width=168) (actual time=0.721..1.261 rows=1 loops=1)
[...] Filter: (user_id = ANY ('{2354108,10810283}'::bigint[]))
[...] Rows Removed by Filter: 8155
[...] Planning Time: 0.529 ms
[...] Execution Time: 3.512 ms
Planning Time: 0.811 ms
Execution Time: 37.957 ms
```

Parallel execution along with avoiding cross node queries, running in parallel for different users

Scope JOIN within node

```
EXPLAIN ANALYSE
SELECT * FROM github_events ge
JOIN github_users gu ON ge.user_id = gu.user_id
where gu.user_id in (2354108,10810283);
```

ta Output | Explain | Messages | Notifications

```
QUERY PLAN
text
Custom Scan (Citus Adaptive)  (cost=0.00..0.00 rows=100000 width=329) (actual time=37.807..37.881 rows=765 loops=1)
[...] Task Count: 2
[...] Tuple data received from nodes: 2624 kB
[...] Tasks Shown: All
[...] ->  Task
[...] Tuple data received from node: 631 kB
[...] Node: host=private-w0.citus-avijit.postgres.database.azure.com port=5432 dbname=citus
[...] ->  Hash Join  (cost=236.75..1111.70 rows=6027 width=329) (actual time=1.443..3.519 rows=140 loops=1)
[...] Hash Cond: (ge.user_id = gu.user_id)
[...] ->  Seq Scan on github_events_102438 ge  (cost=0.00..626.34 rows=18834 width=161) (actual time=0.010..1.184 rows=4075 loops=1)
[...] ->  Hash  (cost=235.95..235.95 rows=64 width=168) (actual time=1.416..1.417 rows=1 loops=1)
[...] Buckets: 1024  Batches: 1  Memory Usage: 9kB
[...] ->  Seq Scan on github_users_102406 gu  (cost=0.00..235.95 rows=64 width=168) (actual time=0.330..1.410 rows=1 loops=1)
[...] Filter: (user_id = ANY ('{2354108,10810283}'::bigint[]))
[...] Rows Removed by Filter: 8415
[...] Planning Time: 0.800 ms
[...] Execution Time: 3.747 ms
[...] ->  Task
[...] Tuple data received from node: 1993 kB
[...] Node: host=private-c.citus-avijit.postgres.database.azure.com port=5432 dbname=citus
[...] ->  Hash Join  (cost=229.16..1024.49 rows=5319 width=329) (actual time=1.285..3.088 rows=625 loops=1)
[...] Hash Cond: (ge.user_id = gu.user_id)
[...] ->  Seq Scan on github_events_102417 ge  (cost=0.00..570.57 rows=17157 width=161) (actual time=0.007..0.920 rows=4367 loops=1)
[...] ->  Hash  (cost=228.39..228.39 rows=62 width=168) (actual time=1.265..1.266 rows=1 loops=1)
[...] Buckets: 1024  Batches: 1  Memory Usage: 9kB
[...] ->  Seq Scan on github_users_102385 gu  (cost=0.00..228.39 rows=62 width=168) (actual time=0.721..1.261 rows=1 loops=1)
[...] Filter: (user_id = ANY ('{2354108,10810283}'::bigint[]))
[...] Rows Removed by Filter: 8155
[...] Planning Time: 0.529 ms
[...] Execution Time: 3.512 ms
Planning Time: 0.811 ms
Execution Time: 37.957 ms
```

Parallel execution along with avoiding cross node queries, running in parallel for different users

Performance Tuning

Distributed Query Scoping

```
-- Missing filter on distribution column

UPDATE ads SET impressions_count = impressions_count+1
WHERE id = 42;


-- Missing distribution column in join

SELECT page_id, count(event_id) FROM page
LEFT JOIN event USING (page_id)
WHERE tenant_id = 6 AND path LIKE '/blog%’
GROUP BY page_id;


-- logically correct, but slow

;WITH single_ad AS
( SELECT * FROM ads WHERE id=1 )
SELECT * FROM single_ad s
JOIN campaigns c ON (s.campaign_id=c.id);
```

```
-- Added filter on distribution column

UPDATE ads SET impressions_count = impressions_count+1
WHERE id = 42 AND company_id = 1;


-- Added distribution column in join

SELECT page_id, count(event_id) FROM page
LEFT JOIN event USING (tenant_id, page_id)
WHERE tenant_id = 6 AND path LIKE '/blog%’
GROUP BY page_id;


-- faster, joining on distribution column

;WITH single_ad AS
( SELECT * FROM ads WHERE id=1 and company_id=1 )
SELECT * FROM single_ad s
JOIN campaigns c ON (s.campaign_id=c.id)
WHERE s.company_id=1 AND c.company_id = 1;
```

System health and locks

```
-- Equivalent of pg_stat_statement with additional detail on source node containing data
-- Applicable for Citus 11 and above

SELECT global_pid, nodeid, is_worker_query, query FROM citus_stat_activity
WHERE global_pid = 110000000123;
[ RECORD 1 ]---+--------------------------------------------------
global_pid     | 110000000123
nodeid         | 11
is_worker_query| f
query          | UPDATE tbl SET b = 100 WHERE a = 0;


-- Update command here is blocking Delete

SELECT * FROM citus_lock_waits;
[ RECORD 1 ]-------------------------+----------------------------
waiting_gpid                         | 20000000345
blocking_gpid                        | 110000000123
blocked_statement                    | DELETE FROM tbl WHERE a = 0;
current_statement_in_blocking_process| UPDATE tbl SET b = 100 WHERE a = 0;
waiting_nodeid                       | 2
blocking_nodeid                      | 11
```

Vacuuming

Vacuum not triggered enough	
autovacuum_vacuum_scale_factor	Lower the value to trigger vacuuming more frequently, useful for larger tables with more updates / deletes.
autovacuum_vacuum_insert_scale_factor	Lower the values to trigger vacuuming more frequently for large, insert-heavy tables.
Vacuum too slow	
autovacuum_vacuum_cost_delay	Decrease to reduce cost limiting sleep time and make vacuuming faster.
autovacuum_vacuum_cost_limit	Increase the cost to be accumulated before vacuum will sleep, thereby reducing sleep frequency and making vacuum go faster.
autovacuum_max_workers	Increase to allow more parallel workers to be triggered by autovacuum.
max_parallel_maintenance_workers	Increase to allow `VACUUM` to vacuum more indexes in parallel.*
Vacuum isn't cleaning up dead rows	
statement_timeout	Set to automatically terminate long-running queries after a specified time.**
log_min_duration_statement	Set to log each completed statement which takes longer than the specified timeout.**
hot_standby_feedback	Set to "on" so the standby sends feedback to the primary about running queries. Decreases query cancellation, but can increase bloat. Consider switching "off" if bloat is too high.

Vacuum not triggered?

POSSIBLE FIXES

- Decrease autovacuum scale factor
- Decrease autovacuum insert scale factor

Vacuum too slow?

POSSIBLE FIXES

- Reduce impact of cost limiting
- Increase parallel workers
- Improve scan speed by prefetching tables
- Increase memory to store dead tuples
- Vacuum indexes in parallel

Vacuum not cleaning dead rows?

POSSIBLE FIXES

- Terminate or proactively time out long-running transactions
- Consider setting hot_standby_feedback = off
- Set vacuum_defer_cleanup_age to a higher value
- Drop inactive or unneeded replication slots
- Rollback hanging 2PC transactions
- Manage DDLs to give autovacuum time to do its job

Judicial Logging

Database logging is an important part of your highly available database solution, though it is important to understand that it **often become an overhead without judicial log levels**.

setting	value	reason
log_statement_stats	OFF	Avoid profiling overhead
log_duration	OFF	Don't need to know the duration of normal queries
log_statement	NONE	Don't log queries without specific reason
log_min_duration_statement	A value longer than what you think normal queries should take	Shows the abnormally long queries

Additional Tips

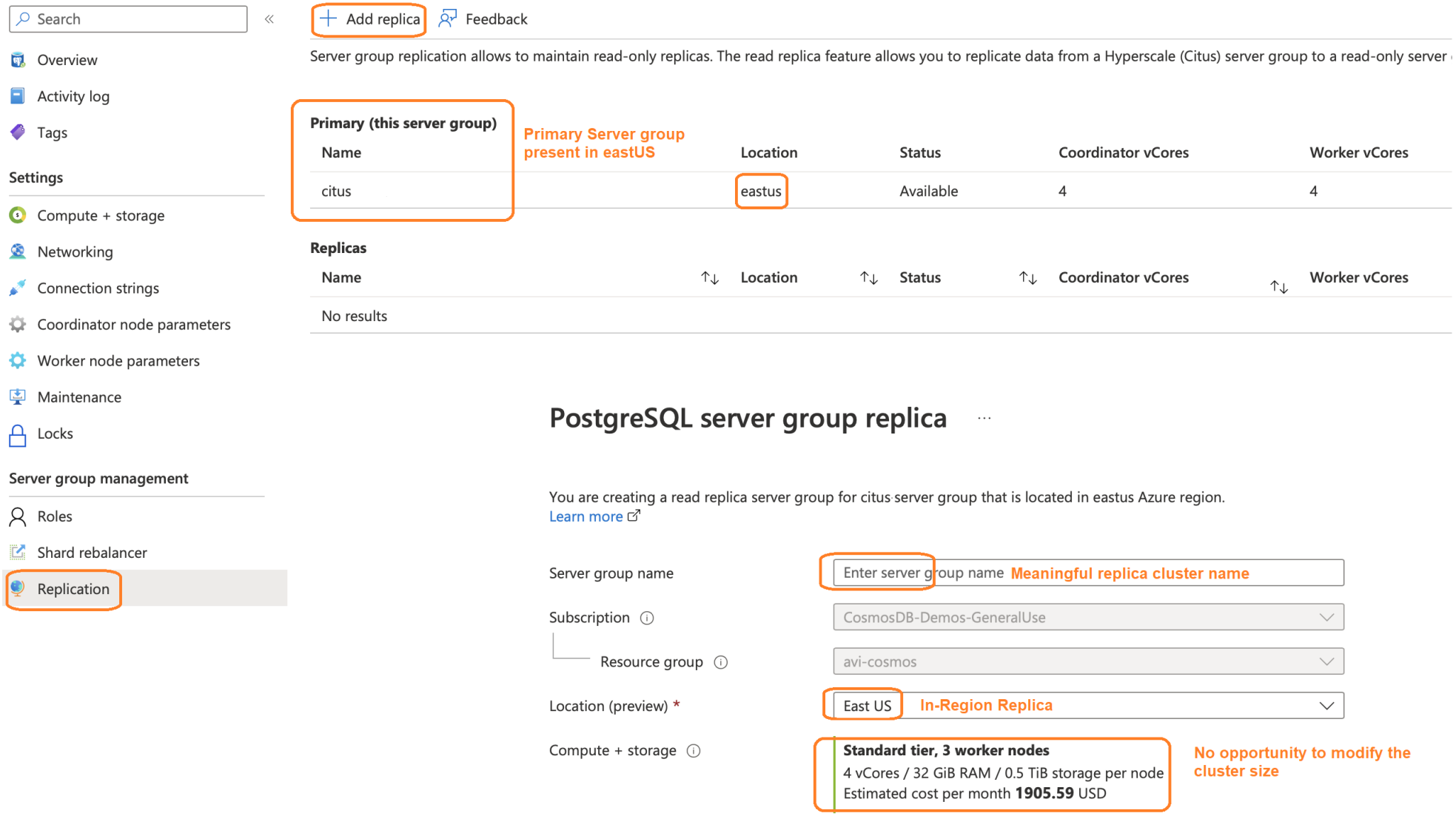
- PostgreSQL supports building indices concurrently, to avoid taking a write lock on the table.

- Consider setting lock_timeout in a SQL session prior to running a heavy DDL command. With lock_timeout, PostgreSQL will abort the DDL command if the command waits too long for a write lock. A DDL command waiting for a lock can cause later queries to queue behind itself.

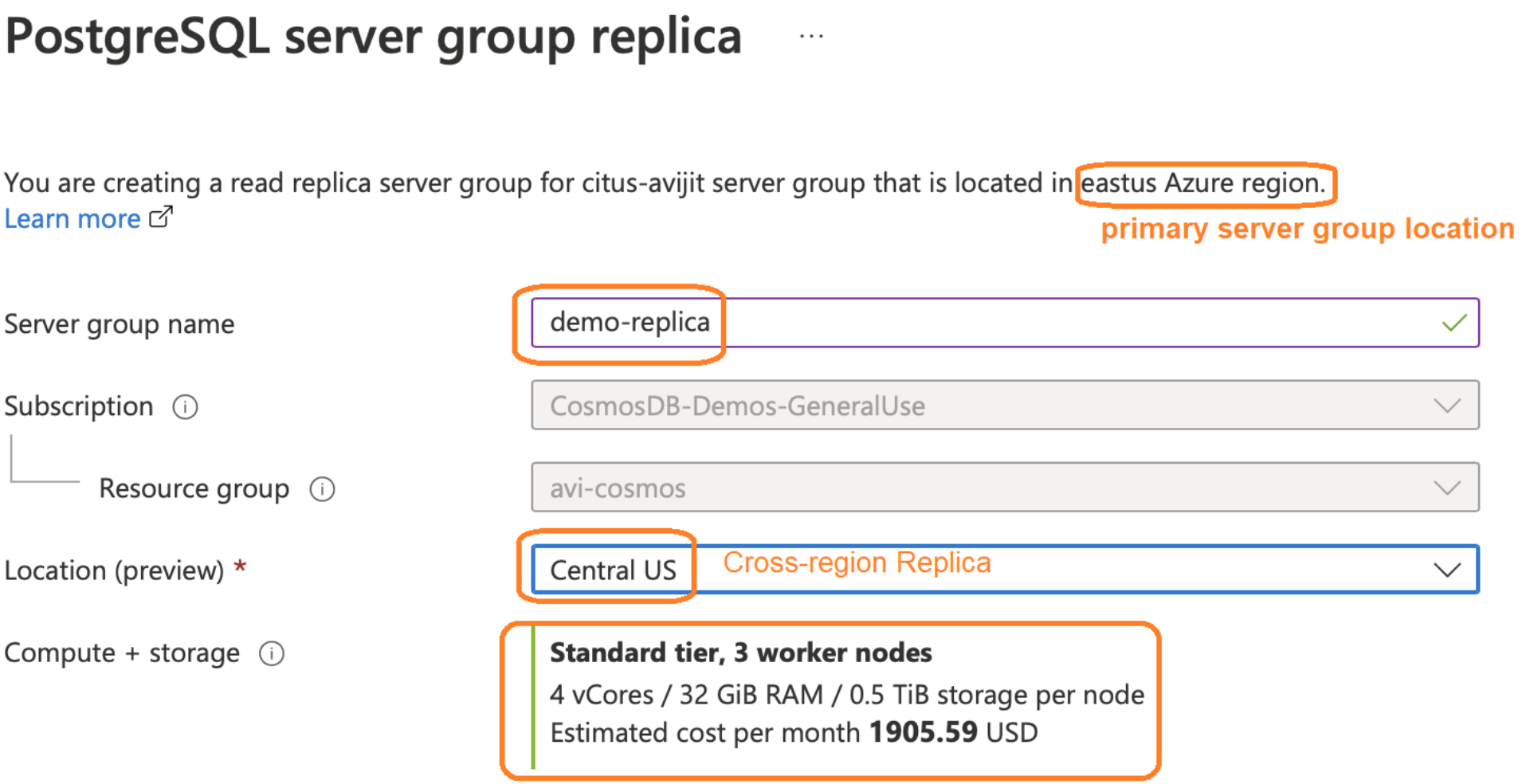
- **Client Side connection pooling :** A connection pool holds open database connections for reuse. An application requests a connection from the pool when needed, and the pool returns one that is already established if possible, or establishes a new one. When done, the application releases the connection back to the pool rather than closing it.
  Adding a client-side connection pool is an easy way to boost application performance with minimal code changes. In our measurements, running single-row insert statements goes about **24x faster** on a Hyperscale (Citus) server group with pooling enabled.

Management

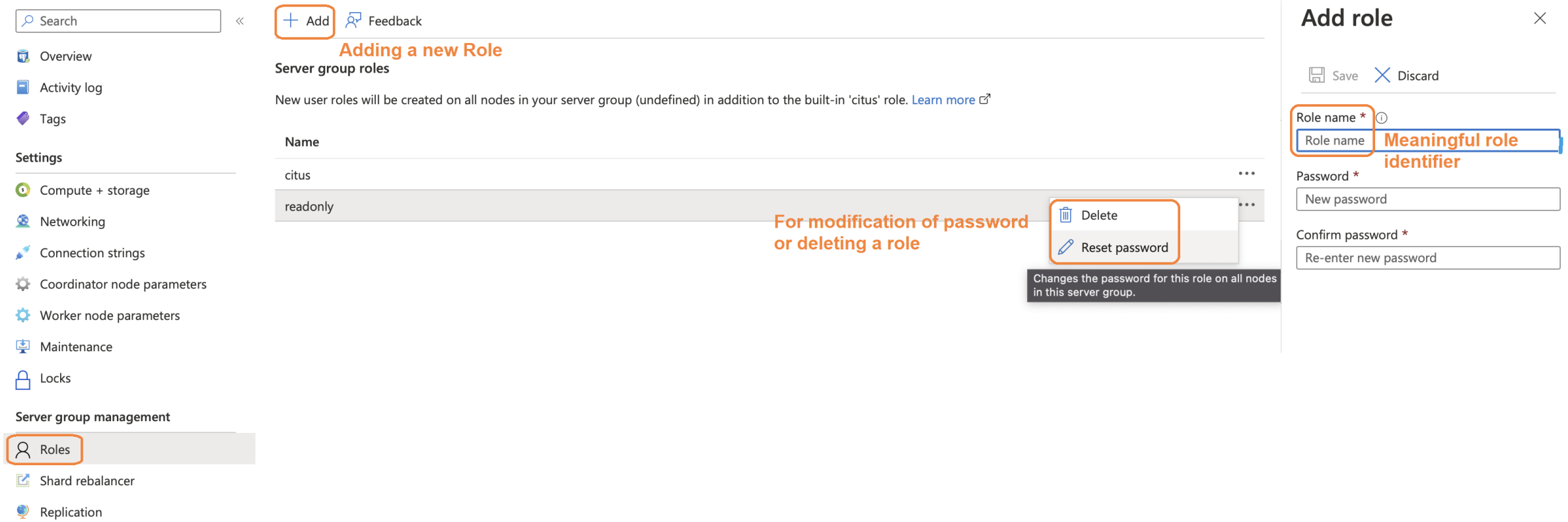
Replica Creation



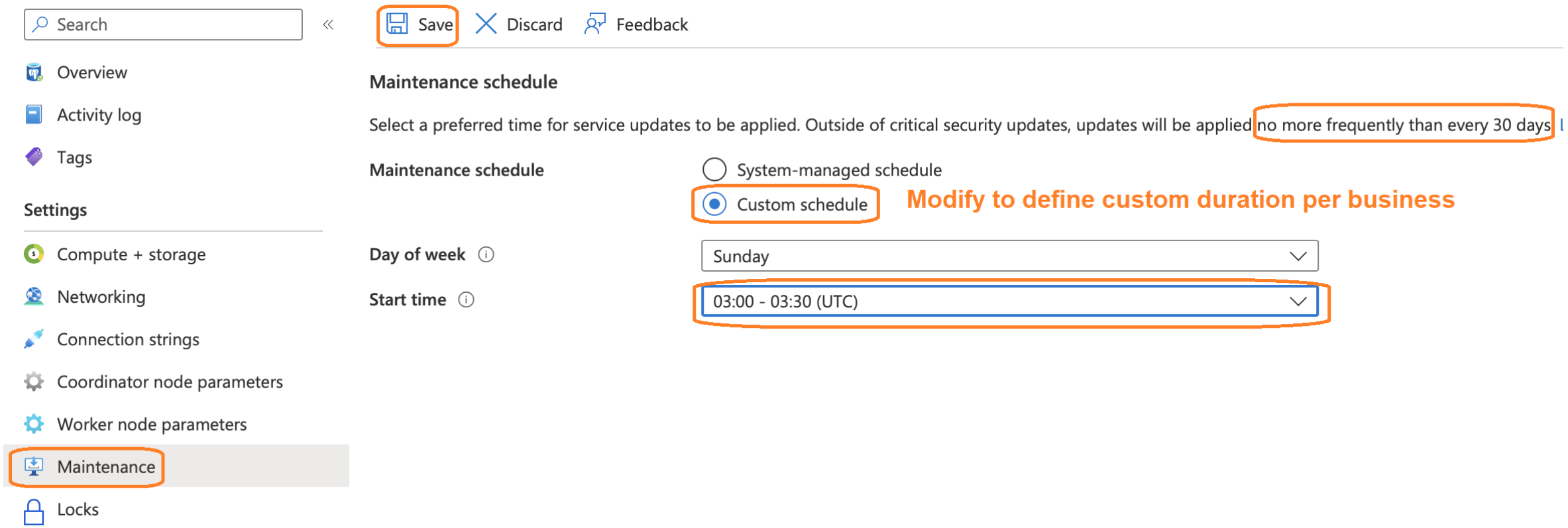
Replica Creation (Cross region)



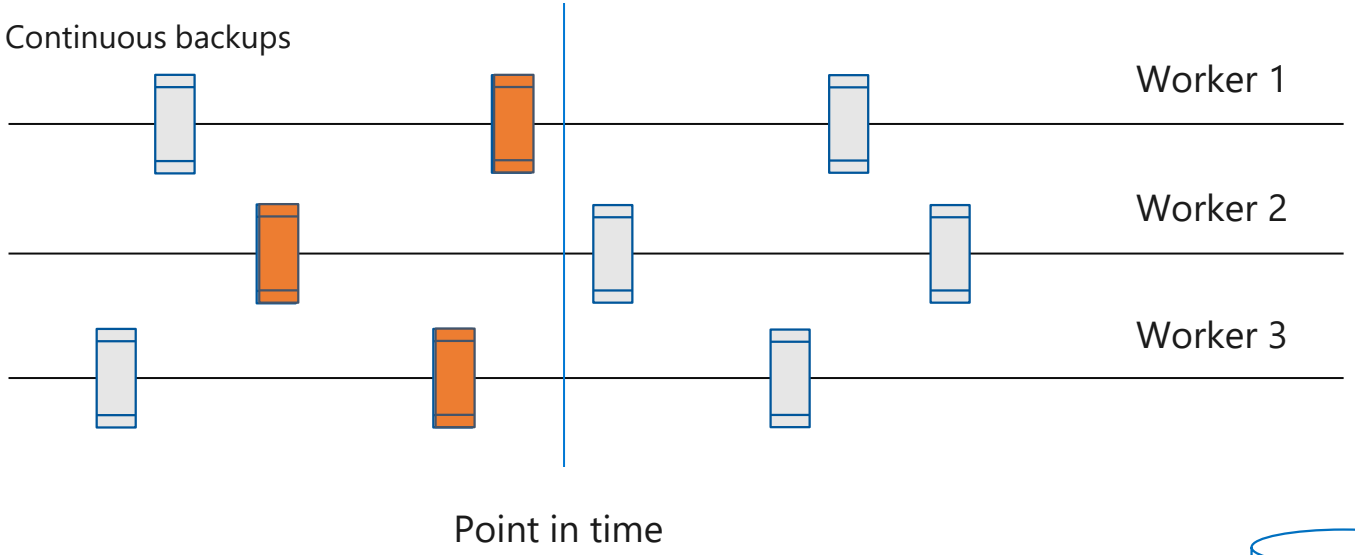
Role Creation

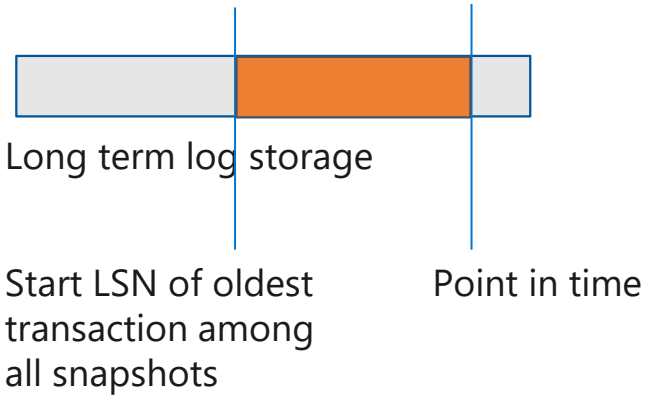


Maintenance Schedule



Backup & Restore

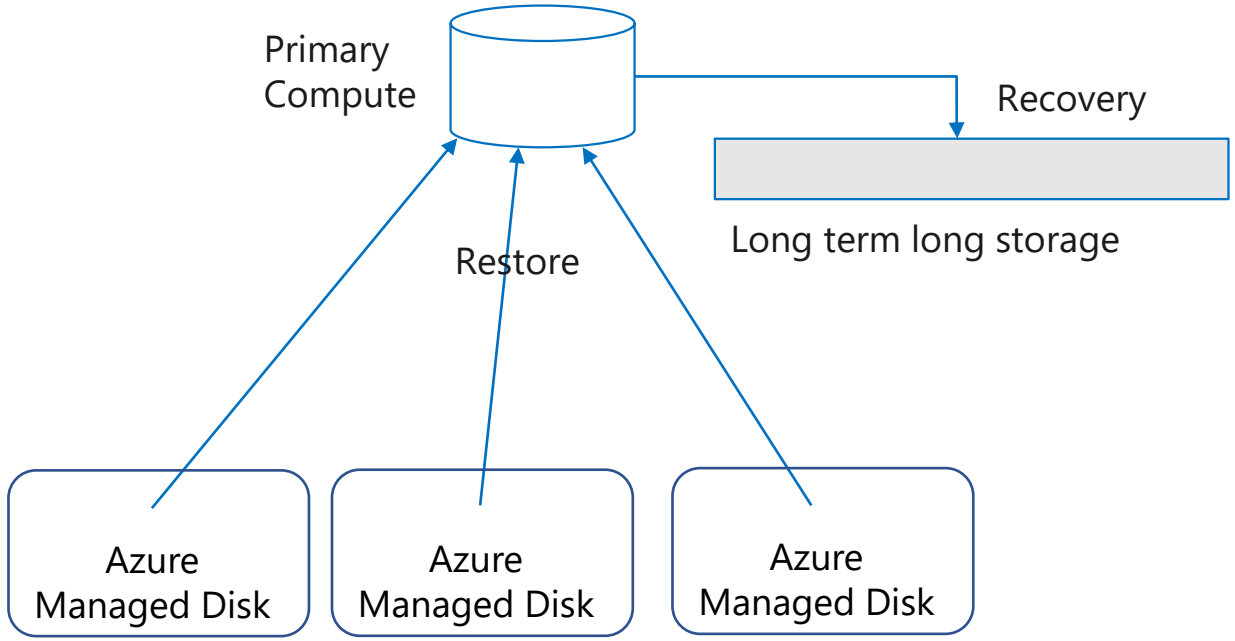




Backup has no impact to apps

Parallelized copy (metadata)

Constant time PITR



Cluster Management

Finding which shard contains data for a specific tenant

```
SELECT shardid, shardstate, shardlength, nodename, nodeport, placementid
FROM pg_dist_placement AS placement, pg_dist_node AS node
WHERE placement.groupid = node.groupid AND node.noderole = 'primary’
AND shardid = ( SELECT get_shard_id_for_distribution_column('stores', 4) );
```

Identify the size of shards

```
SELECT shardid, table_name, shard_size FROM citus_shards WHERE table_name = 'my_table';
```

Review execution plan for Rebalance strategy

```
SELECT get_rebalance_table_shards_plan( rebalance_strategy := 'by_disk_size’ );
```

Monitor the progress of every shard being planned for move

```
SELECT * FROM get_rebalance_progress();
```

Database Management Part 1

Identify, **distributed & reference** tables with corresponding sharding key & number of shards

```
SELECT table_name::text, citus_table_type, distribution_column, shard_count
FROM citus_tables;
```

Identify tables collocated & their size in Kbs

```
SELECT c.colocation_id,table_name, table_size FROM pg_dist_colocation p
JOIN citus_tables c ON p.colocationid = c.colocation_id
WHERE c.colocation_id = 42;
```

Lists queries with the number of executions & partition_key

```
SELECT * FROM citus_stat_statements;
```

Review the nodes and properties like has_meta_data, should_have_shards, node role etc.

```
SELECT * FROM pg_dist_node;
```

Database Management Part 2

Identify unused indexes

```
SELECT * FROM run_command_on_shards('my_distributed_table', $cmd$ SELECT array_agg(a) as infos FROM
( SELECT ( schemaname || '.' || relname || '##' || indexrelname || '##' ||
pg_size_pretty(pg_relation_size(i.indexrelid))::text || '##' || idx_scan::text ) AS a
FROM pg_stat_user_indexes ui
JOIN pg_index i ON ui.indexrelid = i.indexrelid
WHERE NOT indisunique AND idx_scan < 50 AND pg_relation_size(relid) > 5 * 8192
AND (schemaname || '.' || relname)::regclass = '%s'::regclass ORDER BY
pg_relation_size(i.indexrelid) / NULLIF(idx_scan, 0) DESC nulls first,
pg_relation_size(i.indexrelid) DESC ) sub $cmd$);
```

Index Hit Rate

```
-- on coordinator
SELECT 100 * (sum(idx_blks_hit) - sum(idx_blks_read)) / sum(idx_blks_hit) AS index_hit_rate
FROM pg_statio_user_indexes;

-- on workers
SELECT nodename, result as index_hit_rate
FROM run_command_on_workers($cmd$ SELECT 100 * (sum(idx_blks_hit) - sum(idx_blks_read)) /
sum(idx_blks_hit) AS index_hit_rate FROM pg_statio_user_indexes; $cmd$);
```

Database Management Part 3

Viewing active queries running

```
SELECT global_pid, query, state FROM citus_stat_activity
WHERE state != 'idle';
```

Why are the queries waiting

```
SELECT wait_event || ':' || wait_event_type AS type, count(*) AS number_of_occurences
FROM pg_stat_activity WHERE state != 'idle'
GROUP BY wait_event, wait_event_type ORDER BY number_of_occurences DESC;
```

Blocking chains could be reviewed using queries covered under [Lock Monitoring]

Output | Explain | Messages | Notifications

Open update transaction blocking delete

```
begin;
update github_events set
event_type='PullEvent' where user_id =
3226435;
```

OK

blocked_pid integer	blocked_user name	blocking_pid integer	blocking_user name	blocked_statement text
12052	citus	24195	citus	DELETE FROM public.github_events_102437 github_events WHERE (event_id OPERATOR(pg_catalog.=) '4950948973'::bigint)
13102	citus	8944	citus	Alter table github_events Add column lastupdatedat datetime;
13102	citus	8944	citus	Alter table github_events Add column lastupdatedat datetime;
13102	citus	24195	citus	Alter table github_events Add column lastupdatedat datetime;
13102	citus	24195	citus	Alter table github_events Add column lastupdatedat datetime;

Delete blocking an Alter table

[...] delete from github_events where event_id=4950948973;
[...] delete from github_events where event_id=4950948973;
[...] begin;
[...] begin;

Infrastructure Management

Query to review roles & associated privileges

```
SELECT grantee,grantor,table_name,ARRAY_AGG(privilege_type)
FROM information_schema.role_table_grants
WHERE grantee IN ('citus','readonly','avijit') and "table_name" IN ('github_events')
GROUP BY grantee,grantor,table_name
```

Validate for last back up & duration for it

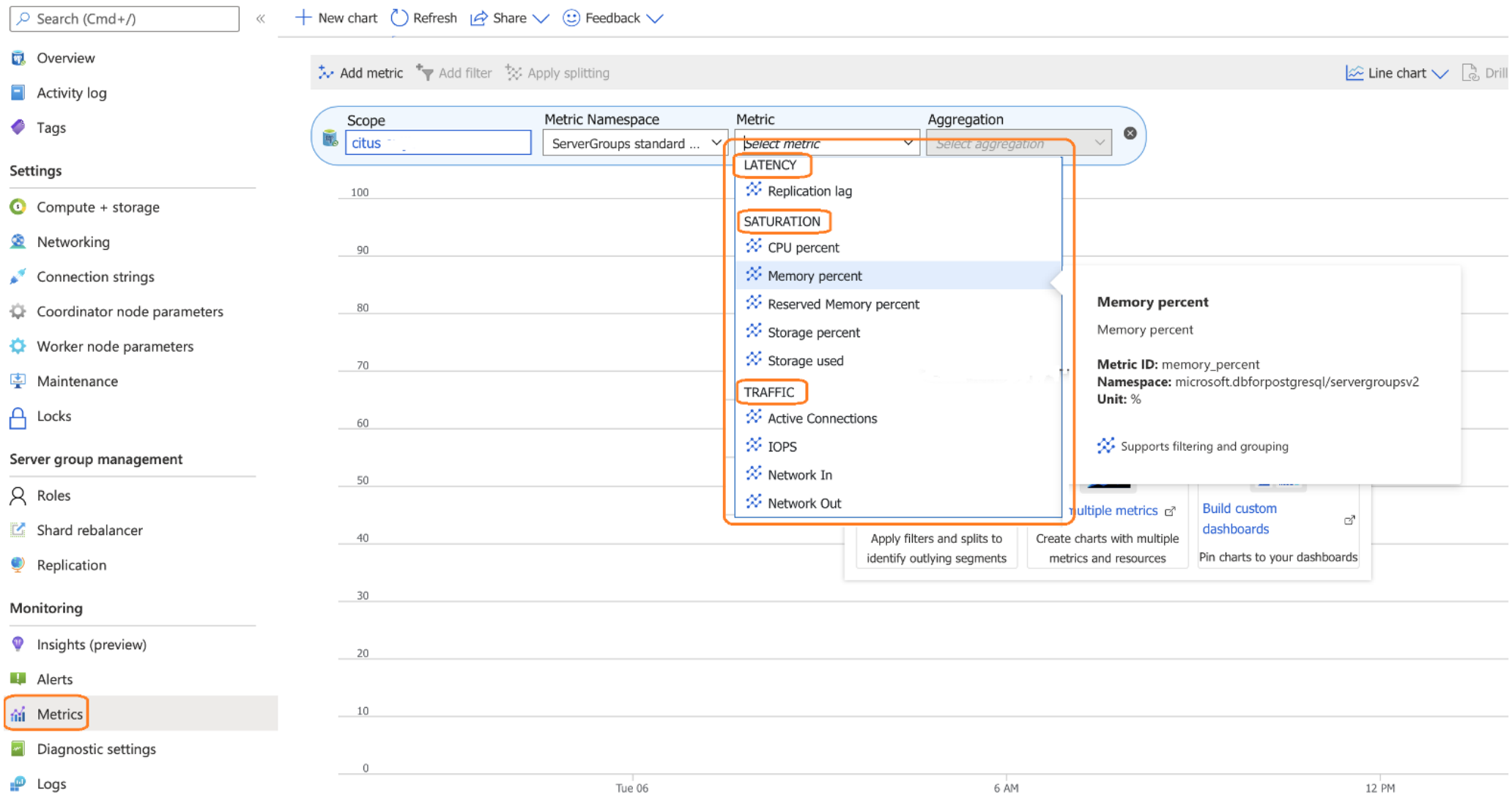
```
AzureDiagnostics
| order by TimeGenerated desc
| where LogicalServerName_s == ”demo-all-tests-c"
| where TimeGenerated > ago(7d)
| where Message contains ”back up started” OR Message contains ”back up completed”
```

Validate for last maintenance & any downtime involved

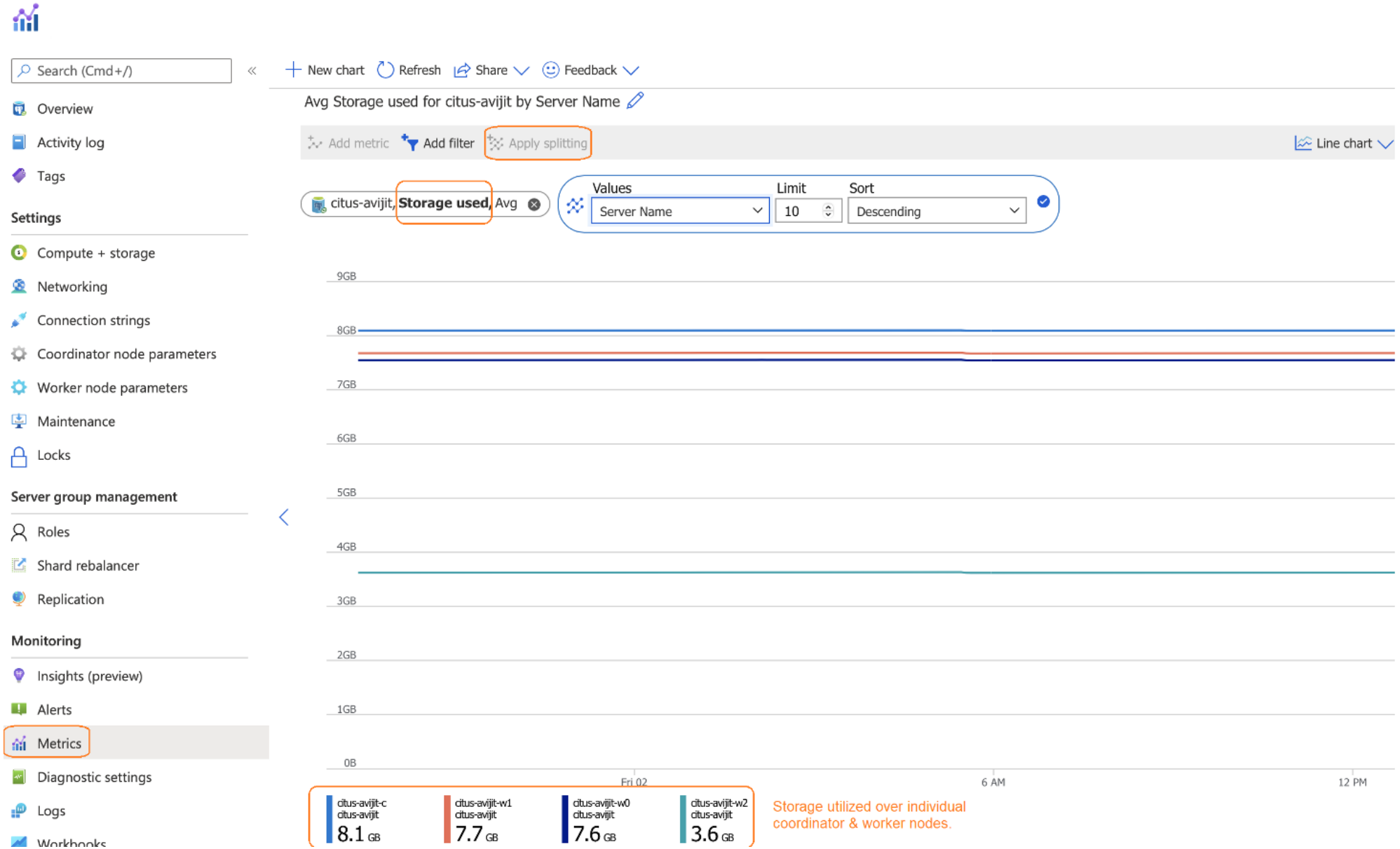
```
AzureDiagnostics
| order by TimeGenerated desc
| where LogicalServerName_s == ”demo-all-tests-c"
| where TimeGenerated > ago(30d)
| where Message contains ”Maintenance started” AND Message contains ”connection failed”
```

Monitoring

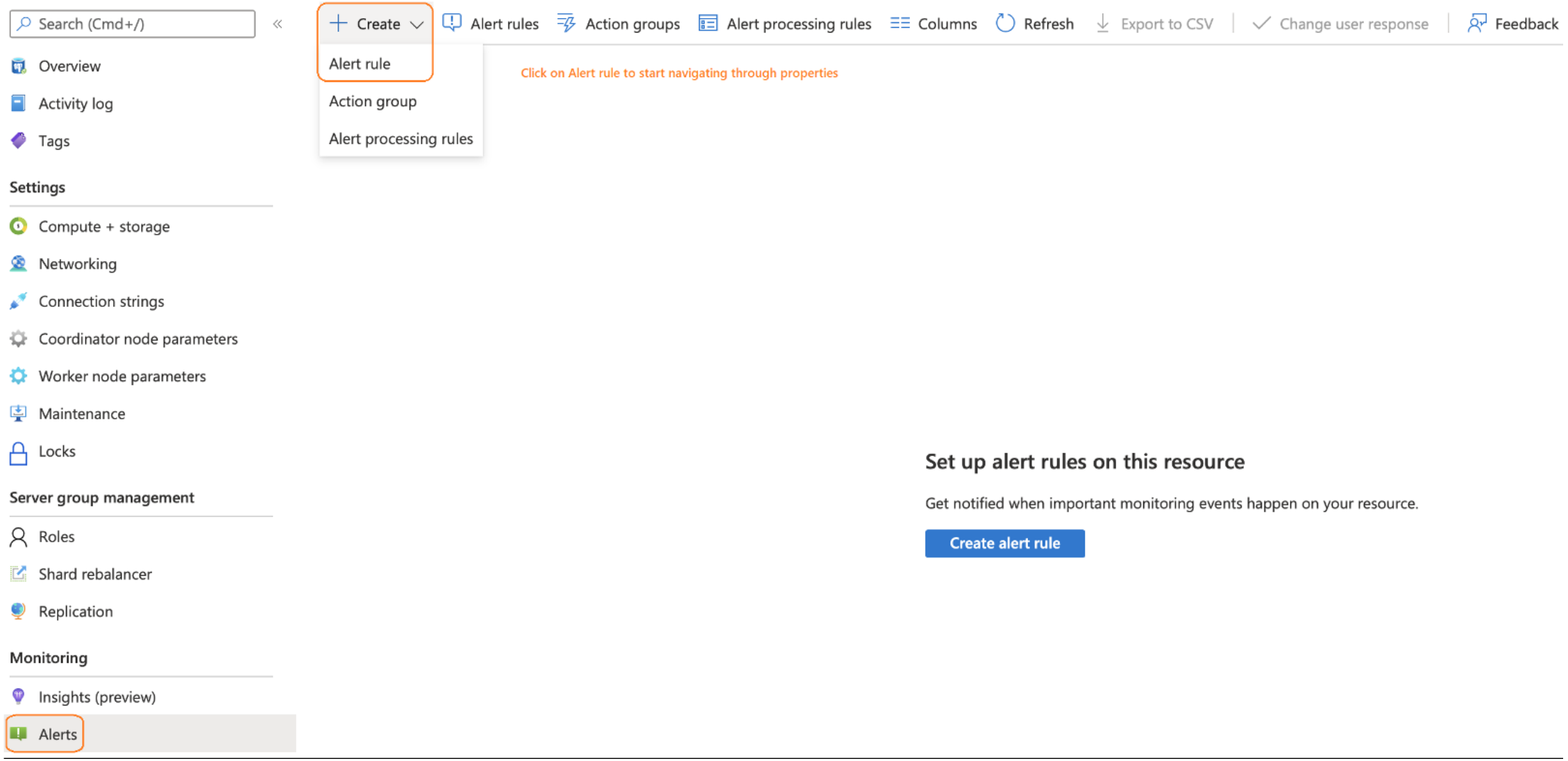
Dashboard Metrics



Dashboard Metrics



Create Alert



Define Alert Metric

Create an alert rule

Step 1

Scope | **Condition** | Actions | Details | Tags | Review + create

Configure when the alert rule should trigger by selecting a signal and defining its logic.

+ Add condition

Select a signal

Step 2

Choose a signal below and configure the logic on the next screen to define the alert condition.

Signal type ⓘ: All

Monitor service ⓘ: All

Filter by signal type

All

Metrics

Log

Activity Log

Resource Health

Metric ,Logs & traces strengthens observability of a platform. Integrated Azure monitor permits defining alerts over all three options.

Signal typ

Log

Reso

Met

Select a signal

Step 3

Choose a signal below and configure the logic on the next screen to define the alert condition.

Signal type ⓘ: Metrics

Monitor service ⓘ: All

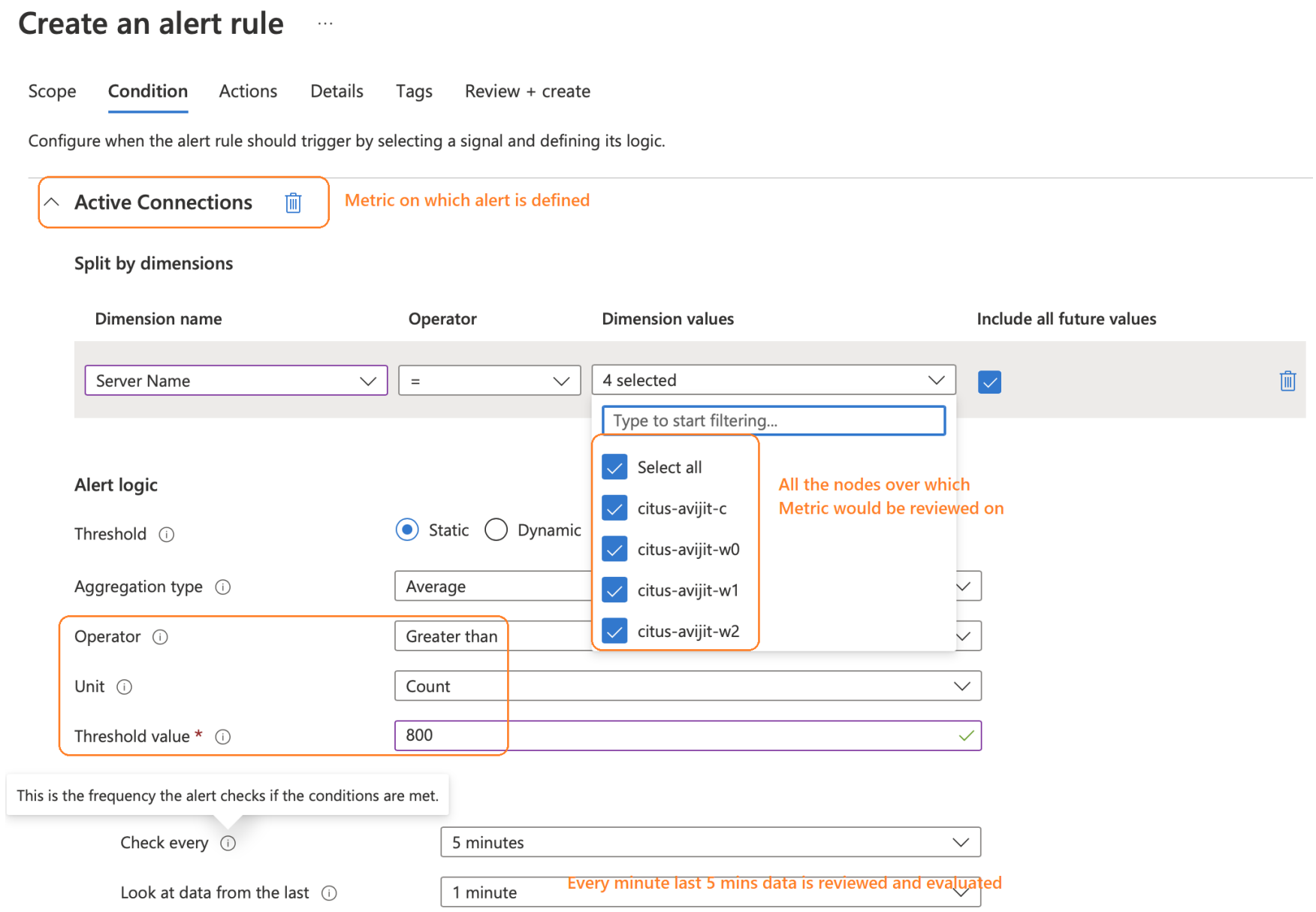
Displaying 1 - 13 signals out of total 13 signals

Search by signal name

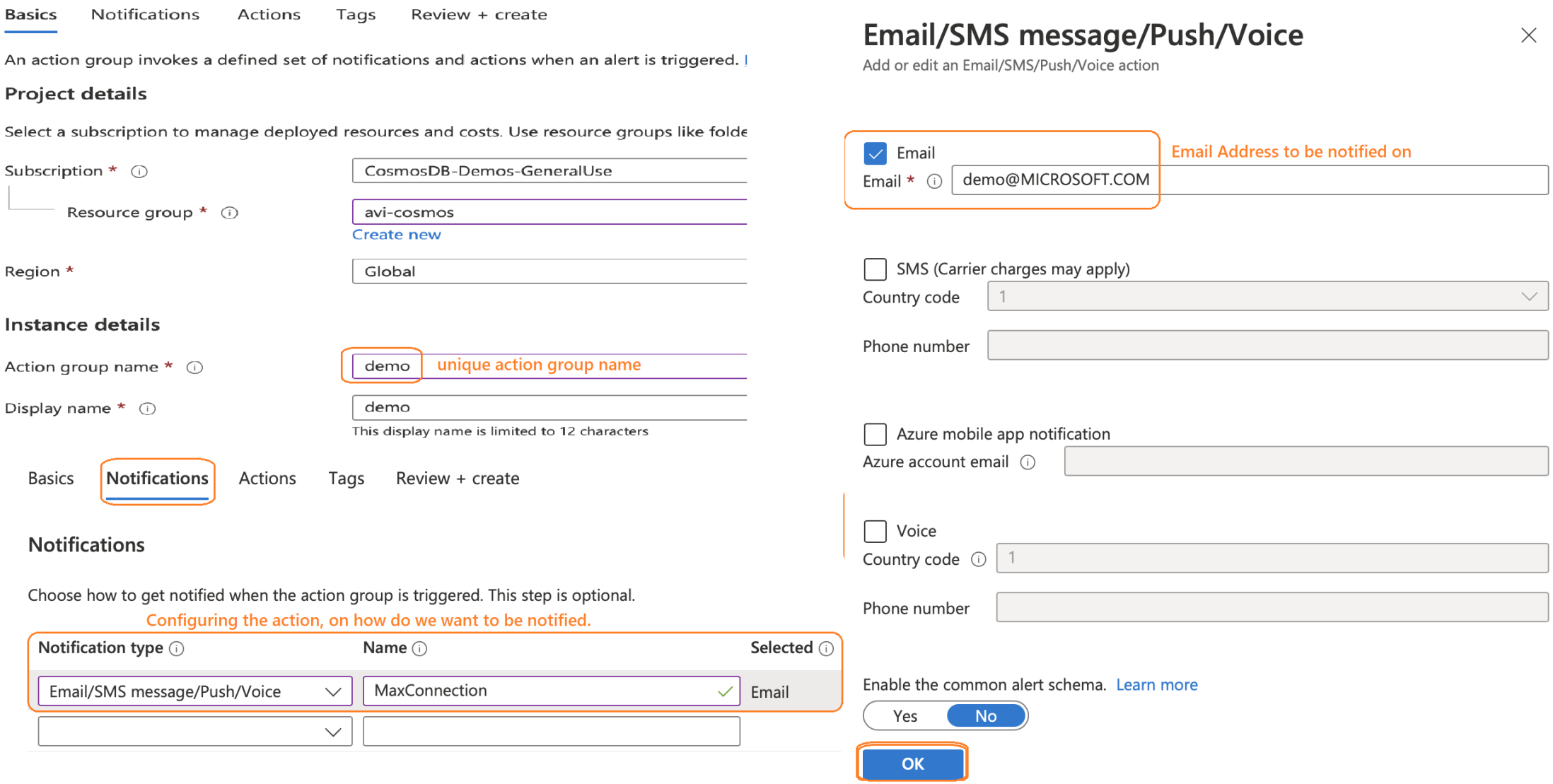
Appropriately choose a signal to generate alert on

Signal name	Signal type
Active Connections	Metrics
Reserved Memory percent	Metrics
CPU percent	Metrics
IOPS	Metrics
Memory percent	Metrics
Network Out	Metrics
Network In	Metrics
Storage percent	Metrics
Storage used	Metrics
VM Cached Bandwidth Consumed Percentage	Metrics
VM Cached IOPS Consumed Percentage	Metrics
VM Uncached Bandwidth Consumed Percentage	Metrics
VM Uncached IOPS Consumed Percentage	Metrics

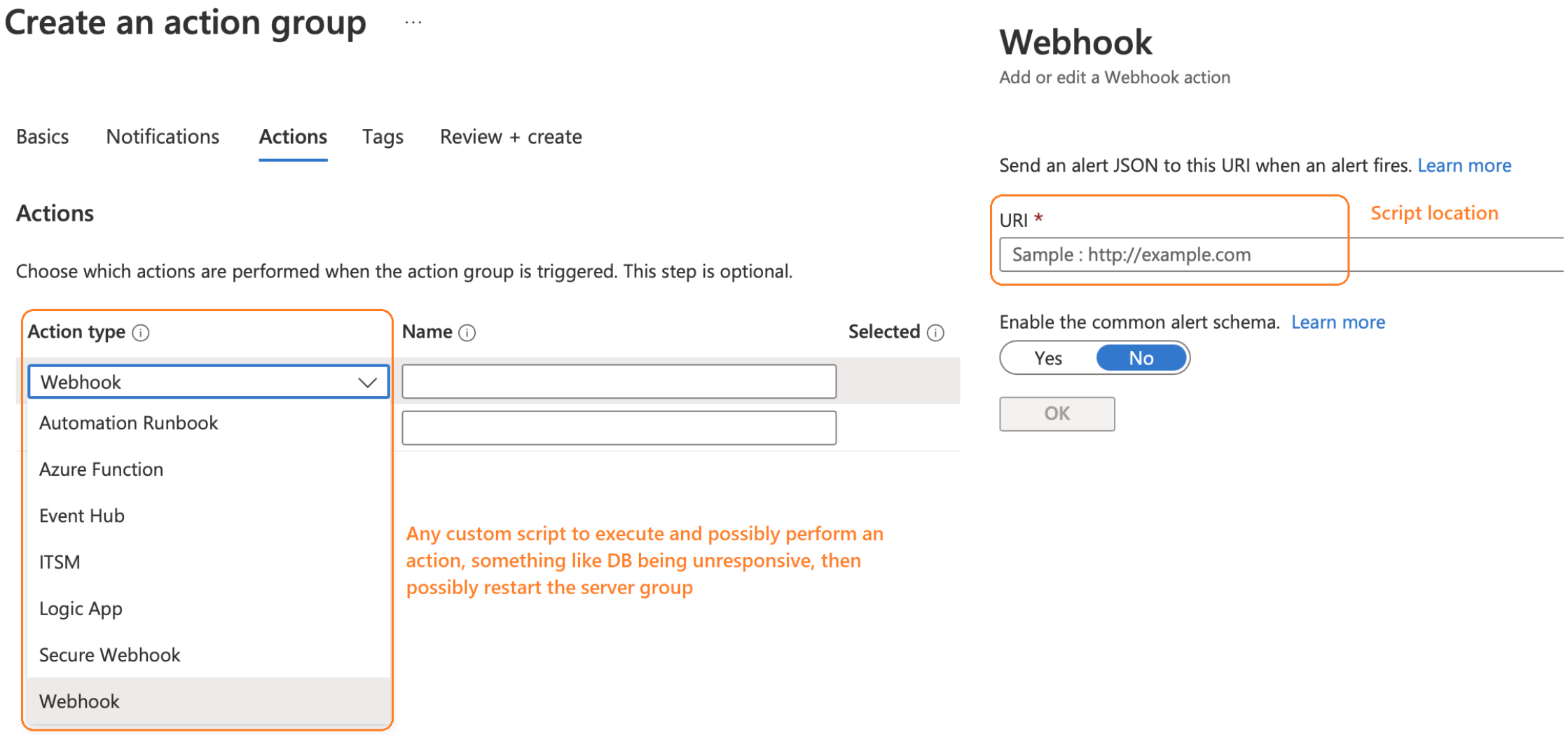
Define Alert Logic



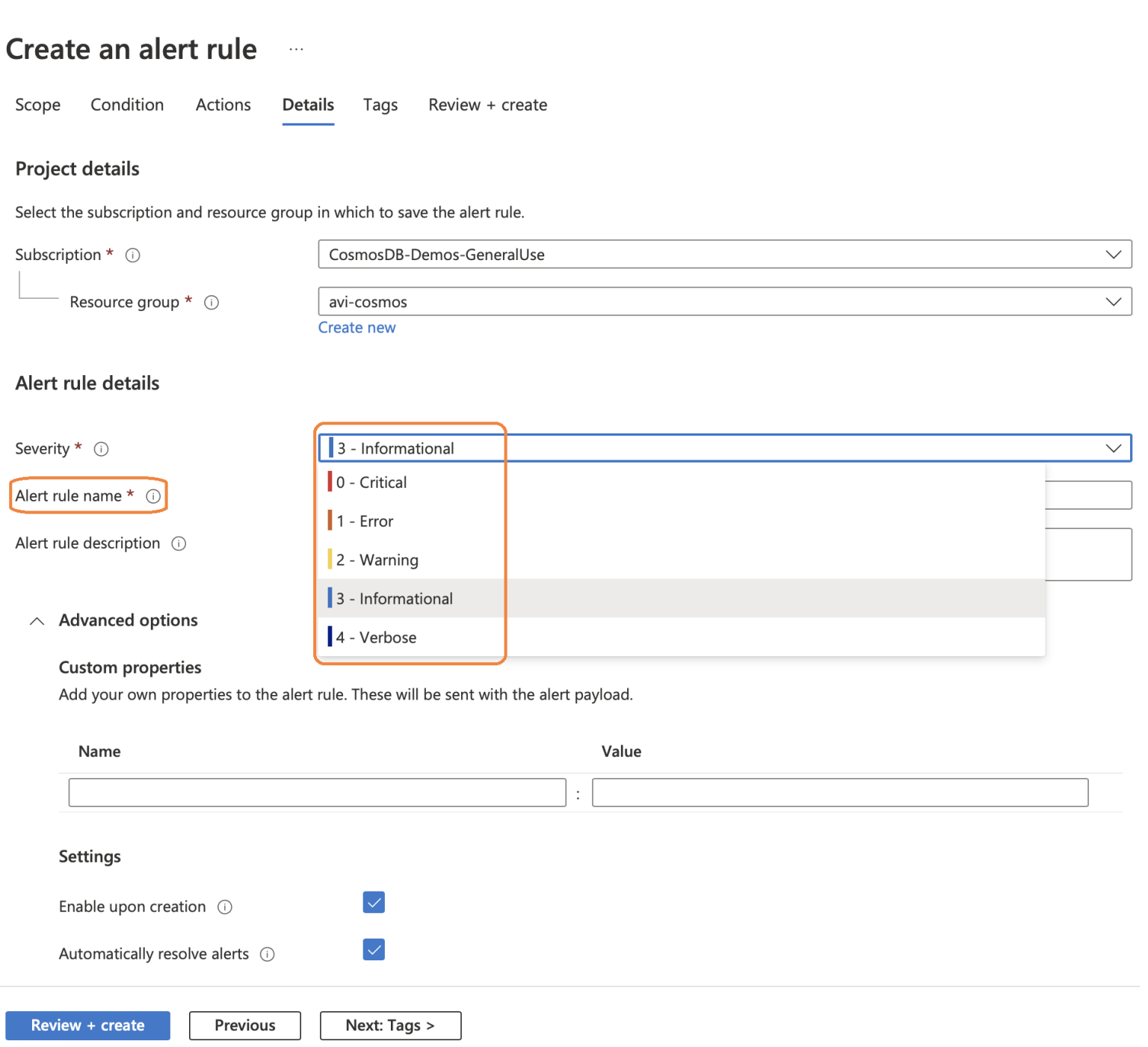
Define action group identifiers



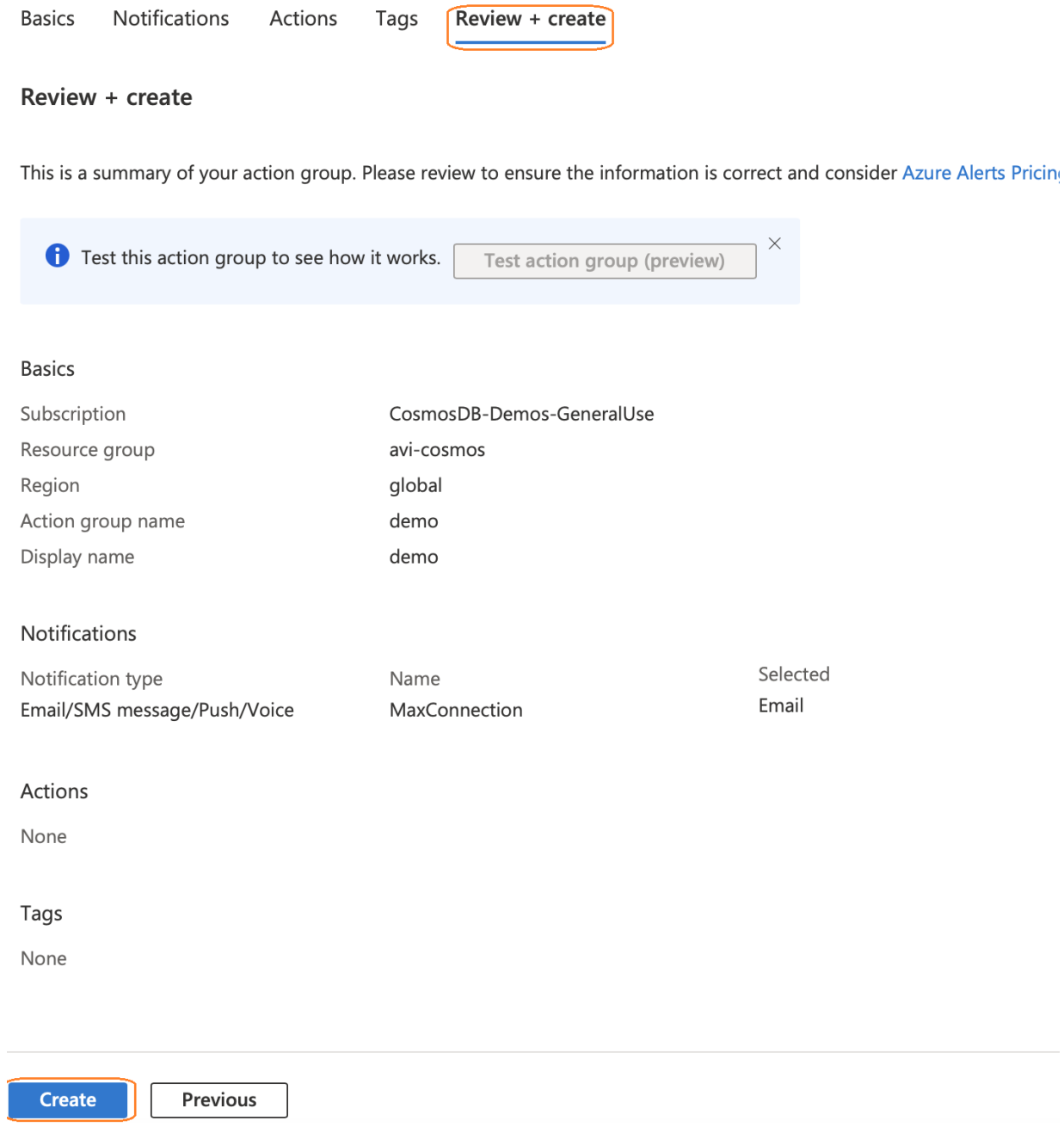
Define additional custom actions



Define severity for rule

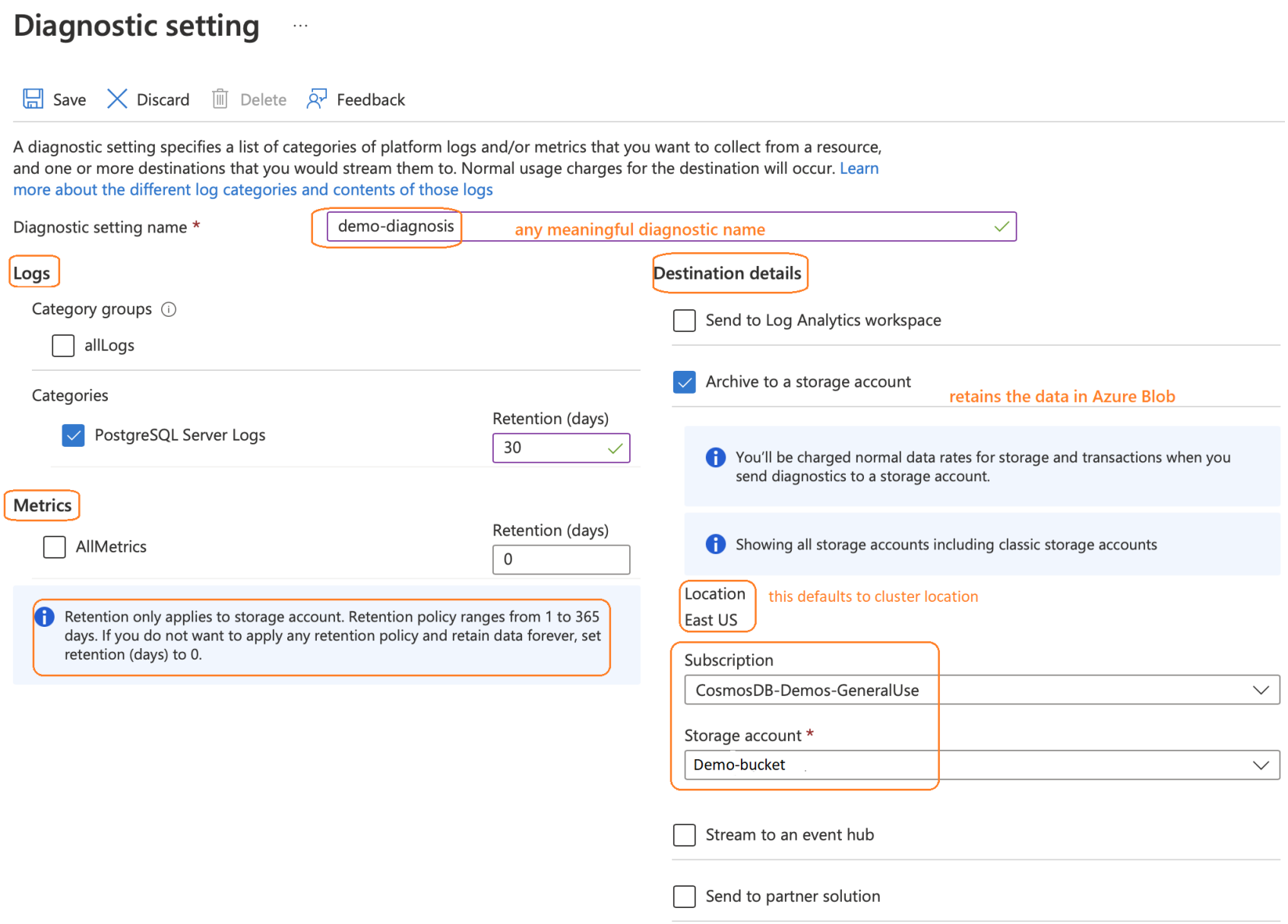


Review & Create action group

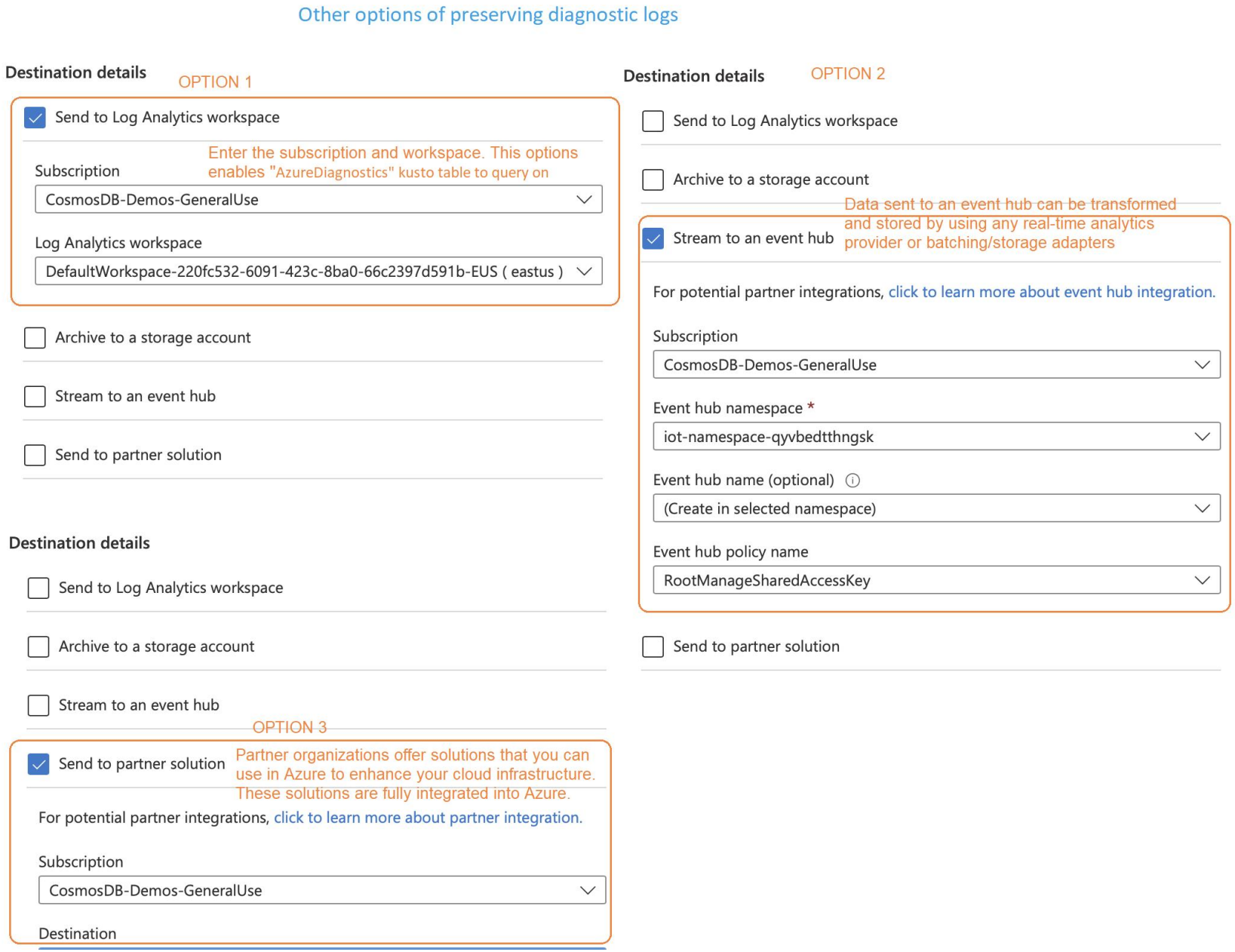


Log Analytics

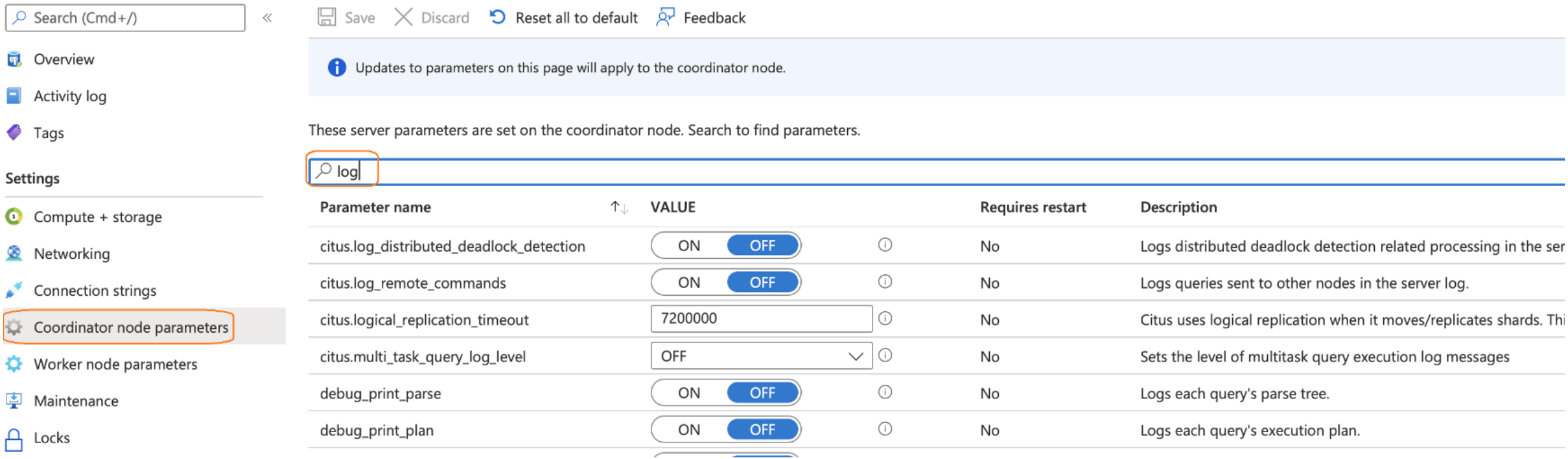
Log Analytics workspace to manage Logs



Log Analytics workspace to manage Logs contd…



Navigate to Logging Parameters

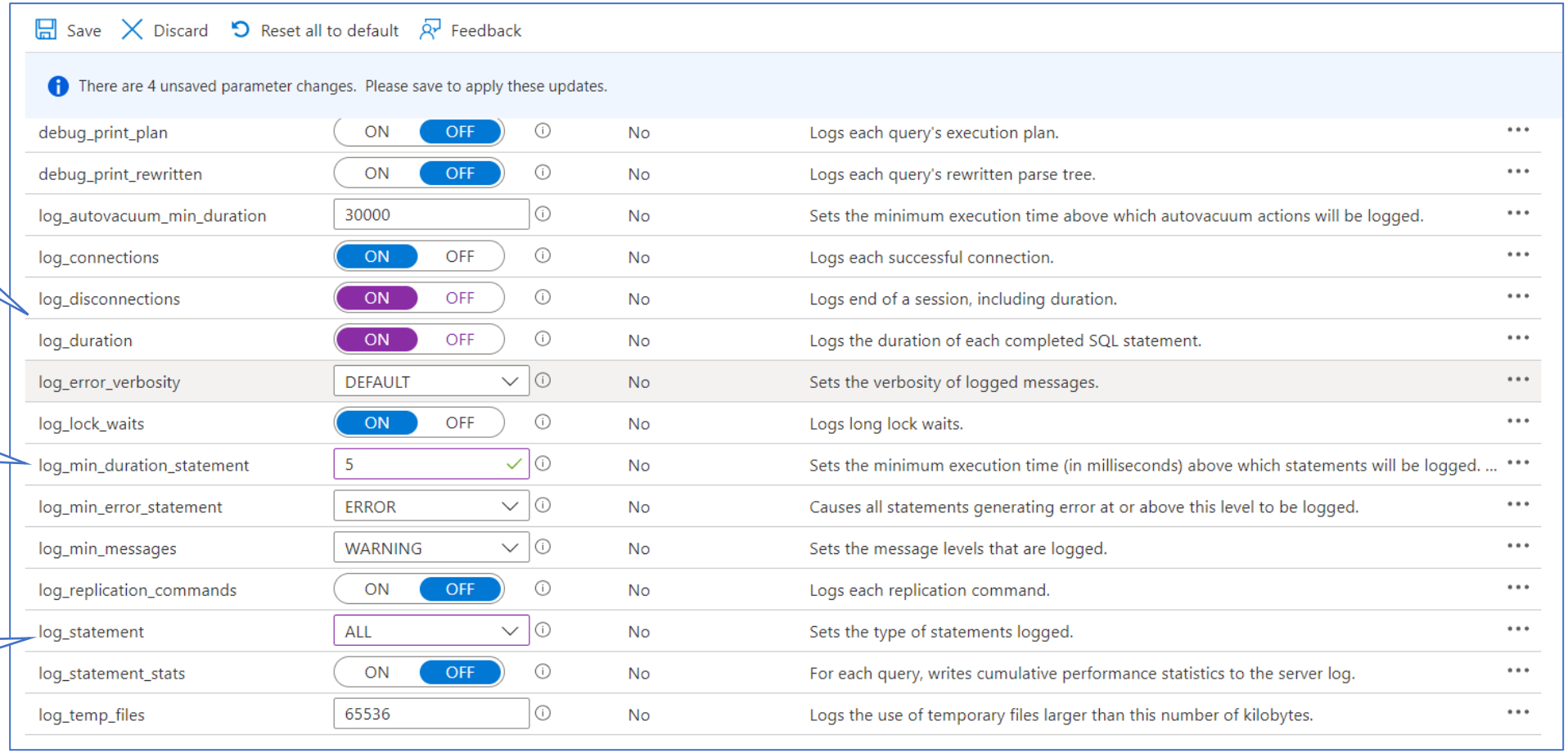


Configuring Logging Parameters

LOG_DISCONNECTIONS
Default: OFF
LOG_DURATION
Default: OFF

LOG_MIN_DURATION_STATEMENT
Default: 30000
To Disable set it to -1

LOG_STATEMENT
Default: NONE
Other Options:
DDL, MOD, ALL



Reviewing Analytics Logs for Connections

Azure data explorer query to list all incoming connections for “appuser” & “pgadminuser”

```
AzureDiagnostics
| order by TimeGenerated desc
| where LogicalServerName_s == ”demo-all-tests-c"
| where TimeGenerated > ago(1d)
| where Message contains "connection authorized: user=appuser”
        or Message contains "connection authorized: user=pgadminuser”
| project Message, originalEventTimestamp_s
| limit 100
```

Results | Chart

Message	originalEventTimestamp_s
connection authorized: user=appuser database=citus application_name=psql SSL enabled (protocol=TLSv1.2, cipher=ECDHE-RSA-AES256-GCM-SHA384, bits=256)	2022-08-26 14:32:31.827957
connection authorized: user=appuser database=citus application_name=psql SSL enabled (protocol=TLSv1.2, cipher=ECDHE-RSA-AES256-GCM-SHA384, bits=256)	2022-08-26 14:32:21.112031
connection authorized: user=pgadminuser database=citus application_name=psql SSL enabled (protocol=TLSv1.2, cipher=ECDHE-RSA-AES256-GCM-SHA384, bits=256)	2022-08-26 14:32:02.249166
connection authorized: user=pgadminuser database=citus application_name=psql SSL enabled (protocol=TLSv1.2, cipher=ECDHE-RSA-AES256-GCM-SHA384, bits=256)	2022-08-26 14:31:54.286840

Reviewing Analytics Logs for DMLs

Azure Data Explorer query to list all the DML operations (system + user) performed on Citus server

```
AzureDiagnostics
| order by TimeGenerated desc
| where LogicalServerName_s == ”demo-hsc-all-tests-c"
| where Message contains "INSERT"
        or Message has "SELECT"
        or Message has "DELETE"
        or Message has “UPDATE"
| project Message,originalEventTimestamp_s
| limit 100
```

Results | Chart

Message	originalEventTimestamp_s
duration: 4.004 ms statement: insert into tempo values (2);	2022-08-26 15:11:07.379335
duration: 3.412 ms statement: insert into tempo values (1);	2022-08-26 15:11:01.674794
duration: 5.079 ms statement: delete from books where id='bk107' ;	2022-08-26 15:10:29.010885
duration: 0.618 ms statement: SELECT pol.polname, pol.polpermissive, CASE WHEN pol.polroles = '{0}' THEN NULL ELSE pg_catalog.array_to_string(array(select rolname ...	2022-08-26 15:10:02.069160
duration: 4.426 ms statement: UPDATE books set description='Updated Value' where id='bk106' ;	2022-08-26 15:07:28.848120
duration: 18.678 ms statement: UPDATE books set description='Updated Value' where id='bk105' ;	2022-08-26 15:07:24.922375
duration: 6.113 ms statement: UPDATE books set description='Updated Value' where id='bk105' ;	2022-08-26 15:07:18.355367

PostgreSQL Extensions

PostgreSQL Extensions

Azure Cosmos DB for PostgreSQL supports several other important categories of Extentions :

- **Full Text Search** extentions like dict_int
- **Geospatial** extention like PostGIS
- **Functions** extention like autoinc, intagg, intarray, pg_partman, uuid-ossp etc
- Index Type extentions like btree_gin
- DataType extentions like hstore, ltree, topn etc
- Other Important extentions like pg_cron, pg_stat_statements, pageinspect etc

PostGIS extension real world use case

```
-- Creating extension on Citus
SELECT create_extension('postgis');
```

PostGIS – Extension adds support for geographic objects to the PostgreSQL object-relational database

Guy Carpenter – Provides data insights to major Insurance players across the world helping them analyse policies in different spheres like Reinsurance, Risk management & Financial management to its Clients. With sample queries like below they try to associate catastrophe dataset against Client's policies dataset to perform real time spatial data analysis & identify the impact of such events

```
-- Query uses geometric datatypes & spatial function
SELECT COUNT(*) ,storm_name ,migration_source_name FROM policy_polygon_data p
JOIN catastrophe_data c
WHERE st_intersect(p.shape,c.shape) and storm_name ='IRMA';

OR
WHERE st_dwithin(geography(p.shape), geography(c.shape),16090); -- helps to look in the vicinity of 16 miles.
```

Catastrophe may include Hurricane, Storms, Earthquakes, Wildfire, Flood etc

https://www.youtube.com/watch?v=vbDmwEgNFHs

TopN & HLL extension real world use case

```
-- Creating extension on Citus
SELECT create_extension('TopN');
```

TopN - Extension makes it easy to calculate the most frequently occurring values.

Algolia uses TopN to keep top items sorted by frequency. With TopN, Algolia's APIs can quickly find the most popular queries in a given time frame across the entire distributed database. This feature is useful for Algolia's customers such as Hacker News, whose search is powered by Algolia.

```
SELECT
      index,
      topn_add_agg(query) AS top_searches,
      topn_add_agg(query) FILTER (WHERE nb_hits = 0) AS top_no_result_searches,
      topn_union_agg(top_hits) AS top_hits,
      hll_add_agg(hll_hash_text(user_id)) AS user_hll
FROM searches;
```

```
SELECT
  query,
  hll_cardinality(hll_union_agg(user_hll))::bigint AS distinct_user_count
FROM
  searches;
```

https://www.youtube.com/watch?v=vbDmwEgNFHs

Customers

UK COVID-19 dashboard built using Postgres and Citus

- While ministers and scientists can see individual data sets before the public, the dashboard itself is an example of truly democratized, open-access data: the latest graphs and someone sitting at home in Newcastle sees the latest trends and graphs for the first time at 4pm, the same moment as the Prime Minister in his office in Downing Street does.
- The number of concurrent users at peak is 60,000 to 100,000 depending on day and prevalence—and that's just the people browsing the actual website.
- Support for the Python ORM mattered.
- The need for High Availability (HA) drove UKHSA to use a managed database service in the cloud.
- Postgres support for JSON and JSONB is more than just a crowd favourite.

GOV.UK Coronavirus (COVID-19)

250,000 to 300,000 hits per minute at peak.

concurrent users at peak is 60,000 to 100,000

7.5 billion records

5 million data points returned in under 10 seconds

Cache invalidation durations never exceed 2 minutes.

https://techcommunity.microsoft.com/t5/azure-database-for-postgresql/uk-covid-19-dashboard-built-using-postgres-and-citus-for/ba-p/3036276

MixRank Gets Faster Performance & Huge Cost Savings with the Citus Database

- MixRank is the Customer Discovery Platform that helps business-to-business (B2B) inside sales teams find new customers. It is the fastest way for sales reps to build prospect lists, prioritize leads, and contact decision makers.
- "The Citus open source extension to Postgres, deployed on a 20-node Citus cluster with SSDs, has enabled us to achieve phenomenal I/O parallelism and I/O throughput. We have been able to dramatically increase performance and also to cut costs. Citus has been a competitive and operational game-changer for our business."
- The most obvious benefit of the shift to Citus was that customers immediately began seeing fresher data. Because Citus did not need to pre-compute as many elements as before, customers began seeing data from minutes or even seconds before their queries. "Most of our competitors still rely on daily batch updates, which makes the Citus database a huge competitive differentiator for us.

MIXRANK

1.6 PB of time series data, including cold data

3 petabytes (3 PB) of data processed every month.

50-100 times smaller hardware budget

Data available in real-time

https://www.citusdata.com/customers/mixrank

With Citus, Algolia Speeds Up Search Analytics & Eradicates Database Management Headaches

- Algolia hosts search as a service to power search-based tools on their own websites and apps. Algolia also provides its customers with a variety of analytics about their end users' search behaviors.
- Citus handles our complex ingest and delete operations seamlessly—and that's with a workflow that creates five to ten billion new rows of data per day. The way in which Citus shards and distributes data across multiple nodes has given us the parallelization we need.
- TopN is an open source extension to Postgres, created by Citus Data. Algolia uses TopN to calculate the most frequently occurring values in a column so they can keep top items sorted by frequency.

Architecture of Algolia's Search Analytics solution

https://youtu.be/QbojoPywO8g

algolia

Scale for Algolia's Analytics platform

Challenges

- Analytics on Real-time data suffered.
- Reliability was challenge with Elastic Search Cluster.
- Estimation of pricing. Pricing for BigQuery was based on amount of data scanned.
- Clickhouse was ruled out due to missing managed cloud offering.
- Limitation to algorithm support like TopN & HyperLogLog (available through extension in PG)

Impact

- Response times for 95% of queries are less than 800 milliseconds.
- Citus Cloud enables Algolia to provide analytics at scale, for over 1B searches per day
- Support with high ingestion rate 5-10 Bn per day.
- Went from frequent downtime with previous analytics solution—to no downtime with the Citus Cloud database
- Clearer cost estimation.

1.6 PB of time series data, including cold data

3 petabytes (3 PB) of data processed every month.

50-100 times smaller hardware budget

Data available in real-time

https://youtu.be/QbojoPywO8g

Microsoft Windows relies on Hyperscale (Citus) for mission-critical decisions

"Ship/no-ship decisions for Microsoft Windows are made using Hyperscale (Citus), where our team runs on-the-fly analytics on billions of JSON events with sub-second responses.

Distributed PostgreSQL is a game changer."

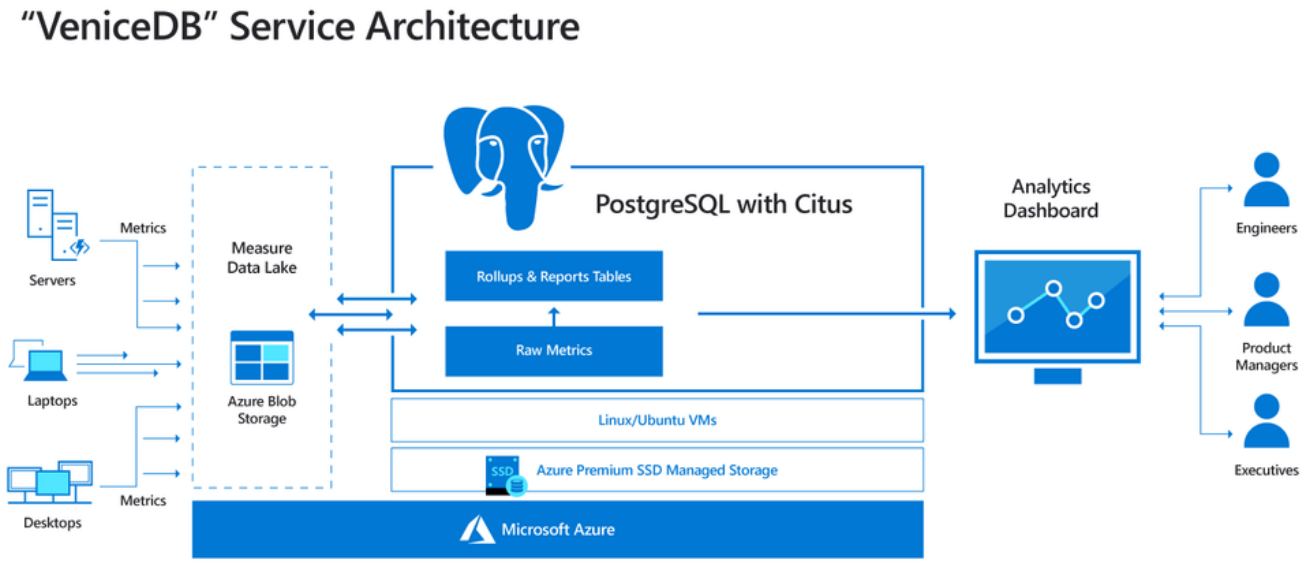
1 PB+ data

20,000 diagnostic metrics

6M+ queries per day;

75% of queries completing < 0.2 secs

2,816 Cores and 18.7 TB of RAM



https://techcommunity.microsoft.com/t5/Azure-Database-for-PostgreSQL/Architecting-petabyte-scale-analytics-by-scaling-out-Postgres-on/ba-p/969685

Marsh Mclennan (MMC) (Americas/United States)

Guy Carpenter, Advantage Point

- **GC Advantage Point –** the industry's most advanced spatial analytics platform for event response, underwriting and portfolio management. Helps their clients (large insurance firms) translate vast amount of geospatial data to insights required for hazard (like earthquakes) prediction, impact of hazard in an area and other mission critical features.
- Easier Migration of Netezza (Postgres-XL). Highly require a scalable distributed PostgreSQL database.
- Extension nature of Citus – lets them leverage latest innovations in PostgreSQL (PostGIS).
- PaaS nature of Citus – "no need to worry of maintenance"

MarshMcLennan

~30TB data size and growing drastically.

Workload has **3000 global users**, with **10-50 of them concurrently** performing complex geospatial analysis

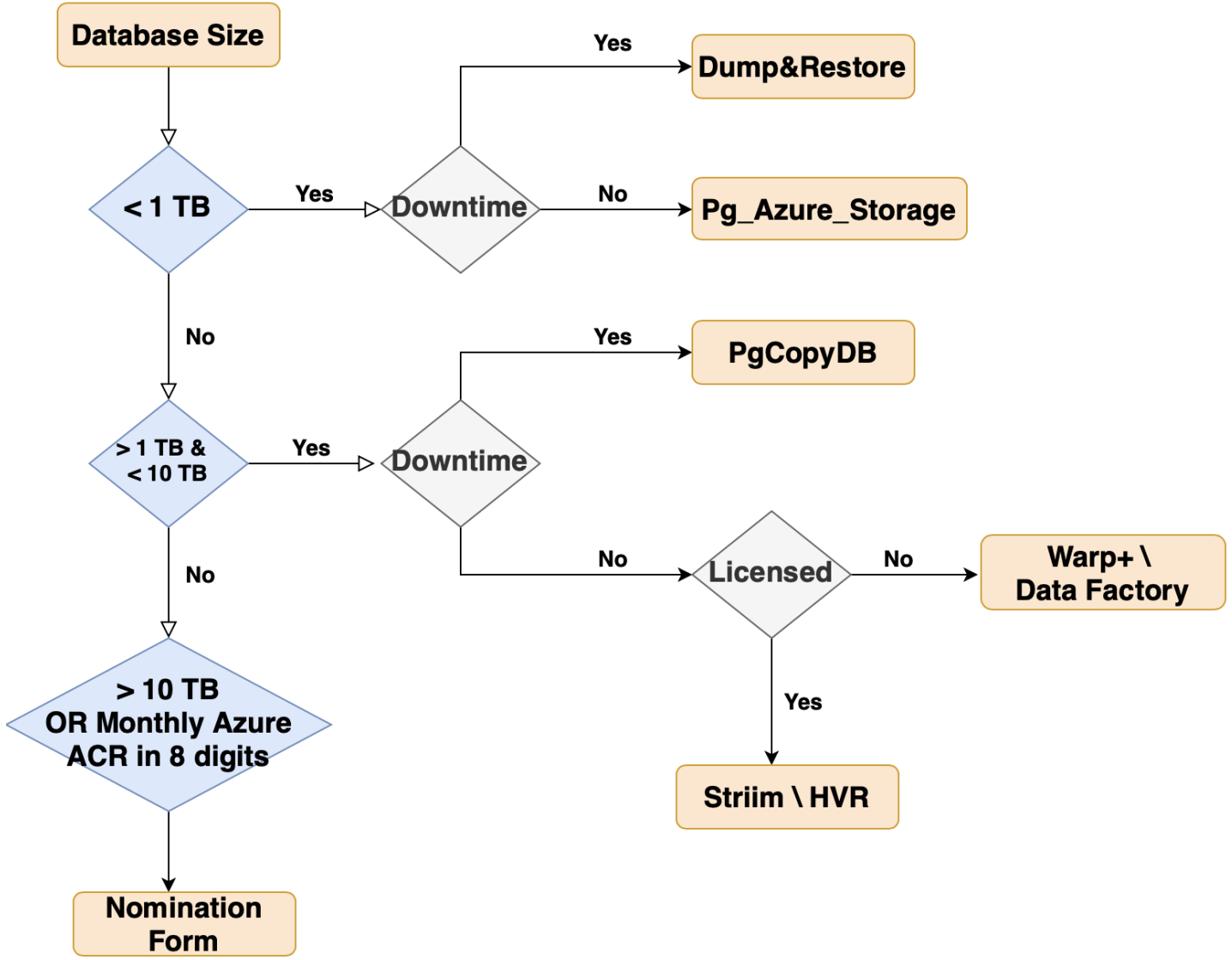
Data available in real-time

https://www.youtube.com/watch?v=vbDmwEgNFHs

MarshMcLennan

Migration

Homogeneous Migration Toolset



Hetrogeneous Migration

Pre-migration

Discover

Inventory database assets, and application stack discovery

Assess

Assess workloads and fix recommendations

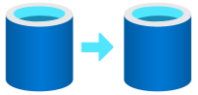
Convert

Convert the source schema to work in the target environment. This is only relevant for heterogeneous migrations.

Migration

Migrate schema, data & objects

Migrate the source schema, and then migrate the source data to the target



Data sync

Sync your target schema and data with the source. This is only relevant for minimal-downtime migrations

Cutover

Cut over from the source to the target environment. This is only relevant for minimal-downtime migrations

Post-migration

Remediate applications

Iteratively make any necessary changes to your applications

Optimize

Based on the tests you performed, address any performance issues, and then retest to confirm the performance improvements

Run functional & performance tests

Iteratively run functional and performance tests

Tooling →

- ora2pg
- DMAT
- SQLines
- Ispirer
- Pitss
- EPAM
- UnifyCloud
- Newt Global
- Striim
- HVR

- DMVA
- Newt Global

App Stacks

Frameworks & Languages Parity

Language	Citus	Postgres	Reference
Java	✅	✅	https://docs.microsoft.com/en-us/azure/postgresql/hyperscale/quickstart-app-stacks-java
Python	✅	✅	https://docs.microsoft.com/en-us/azure/postgresql/hyperscale/quickstart-app-stacks-python
C#	✅	✅	https://docs.microsoft.com/en-us/azure/postgresql/hyperscale/quickstart-app-stacks-csharp
Node.js	✅	✅	https://docs.microsoft.com/en-us/azure/postgresql/hyperscale/quickstart-app-stacks-nodejs
Ruby	✅	✅	https://docs.microsoft.com/en-us/azure/postgresql/hyperscale/quickstart-app-stacks-ruby
Go	✅	✅	Official documentation yet to be added
C	✅	✅	Official documentation yet to be added
C++	✅	✅	Official documentation yet to be added
Ruby on Rails	✅	✅	https://docs.citusdata.com/en/stable/develop/migration_mt_ror.html
Django	✅	✅	https://docs.citusdata.com/en/stable/develop/migration_mt_django.html
Asp.net	✅	✅	https://docs.citusdata.com/en/stable/develop/migration_mt_asp.html

Learn more

Learning Resources

Docs : https://learn.microsoft.com/azure/cosmos-db/postgresql/introduction

Free Trial : https://aka.ms/trycosmosdb

Citus community resources:

Blog : https://www.citusdata.com/blog/

Newsletter : https://www.citusdata.com/join-newsletter

Please fill the Nomination form and share it on cdbpg@microsoft.com , for larger opportunity size.

Criteria :

- Data size : Greater than 10 TB
- MS contracts in 8 digits
- Cluster in place and shorter time frame to go live

Nomination Form : https://aka.ms/cdbpostgreshelp