Databases
Part 3: NoSQL DBs

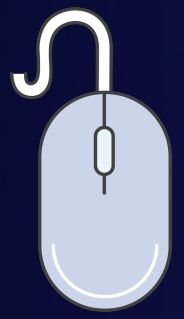
Aleksandr Bernadskii

Solutions Architect
Amazon Web Services

AWS Purpose built databases

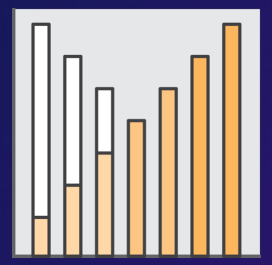
Relational	Key-value	Document	In-memory	Graph	Time-series	Ledger	Wide column
Referential integrity, ACID transactions, schema-on-write	High throughput, low-latency reads and writes, endless scale	Store documents; quickly access querying on any attribute	Query by key with microsecond latency	Quickly and easily create and navigate relationships between data	Collect, store, and process data sequenced by time	Complete, immutable and verifiable history of all changes to application data	Scalable, highly available, and managed Apache Cassandra-compatible service
Aurora, RDS	DynamoDB	DocumentDB	ElastiCache	Neptune	Timestream	QLDB	Amazon Keyspaces
Lift and shift, ERP, CRM, finance	Real-time bidding, shopping cart, social, product catalog, customer preferences	Content management, personalization, mobile	Leaderboards, real-time analytics, caching	Fraud detection, social networking, recommendation engine	IoT applications, event tracking	Systems of record, supply chain, health care, registrations, financial	Build low-latency applications, leverage open source, migrate Cassandra to the cloud

Amazon DynamoDB

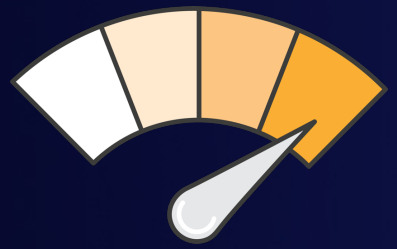


Fully managed NoSQL

Document or key-value



Scales to any workload



Fast and consistent

Access control

Event-driven programming

Horizontal scaling with DynamoDB

Workload:
data volume, reads, writes

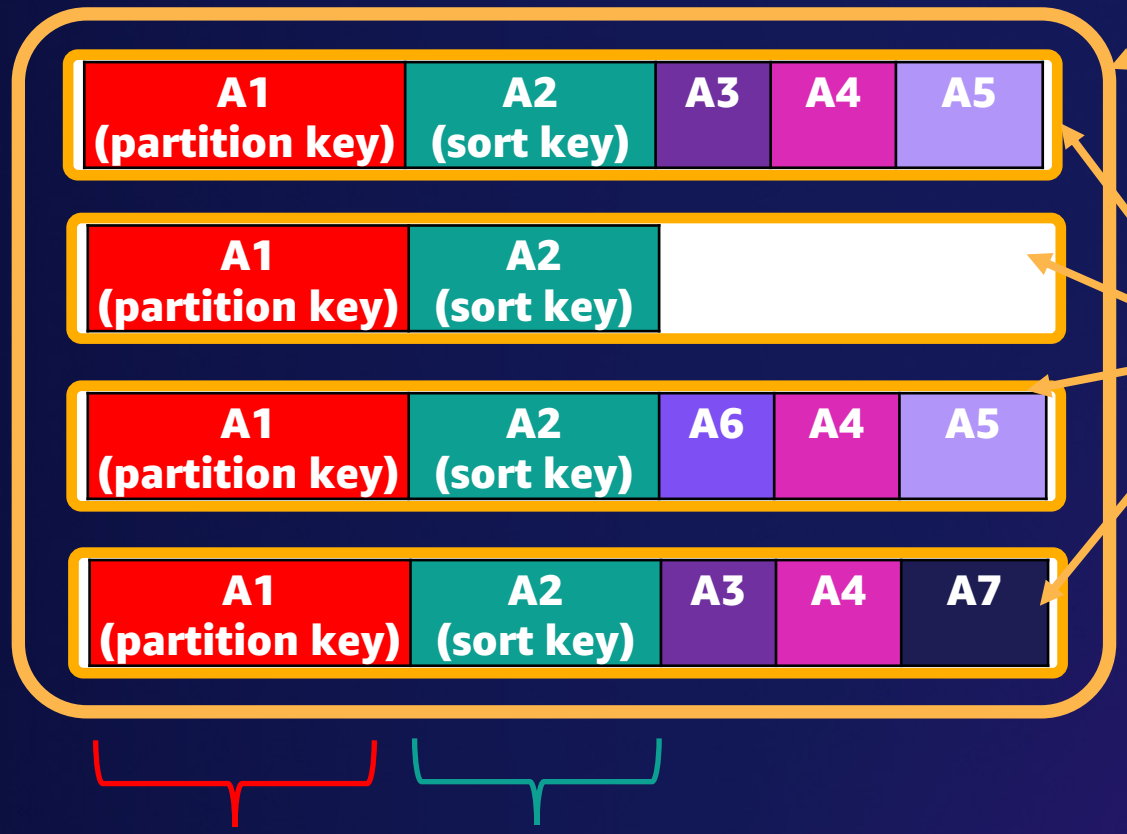
DynamoDB resources:
storage, read, and write capacity

Read/Write Capacity Modes

- Provisioned Mode
  - Specify the maximum amount of read and write capacity for a table or index
  - Use auto scaling to adjust your table's provisioned capacity automatically in response to traffic changes

- On-demand Mode
  - No capacity planning is required
    - Pay per request pricing

DynamoDB Table



Table

Items

Partition Key

Sort Key

Mandatory
Key-value access pattern
Determines data distribution

Optional
Model 1:N relationships
Enables rich query capabilities

All items for key
==, <, >, >=, <=
"begins with"
"between"
"contains"
"in"
sorted results
counts
top/bottom N values

Adaptive Capacity - Core Functions

- Dynamic partitioning
- High-traffic item isolation
- Throughput boosting

Local secondary index (LSI)

Alternate sort key attribute

Index is local to a partition key

10 GB maximum per partition key; LSIs limit the number of range keys!

Table

A1 (partition)	A2 (sort)	A3	A4	A5

LSIs

A1 (partition)	A3 (sort)	A2 (item key)

KEYS_ONLY

A1 (partition)	A4 (sort)	A2 (item key)	A3 (projected)

INCLUDE A3

A1 (partition)	A5 (sort)	A2 (item key)	A3 (projected)	A4 (projected)

ALL

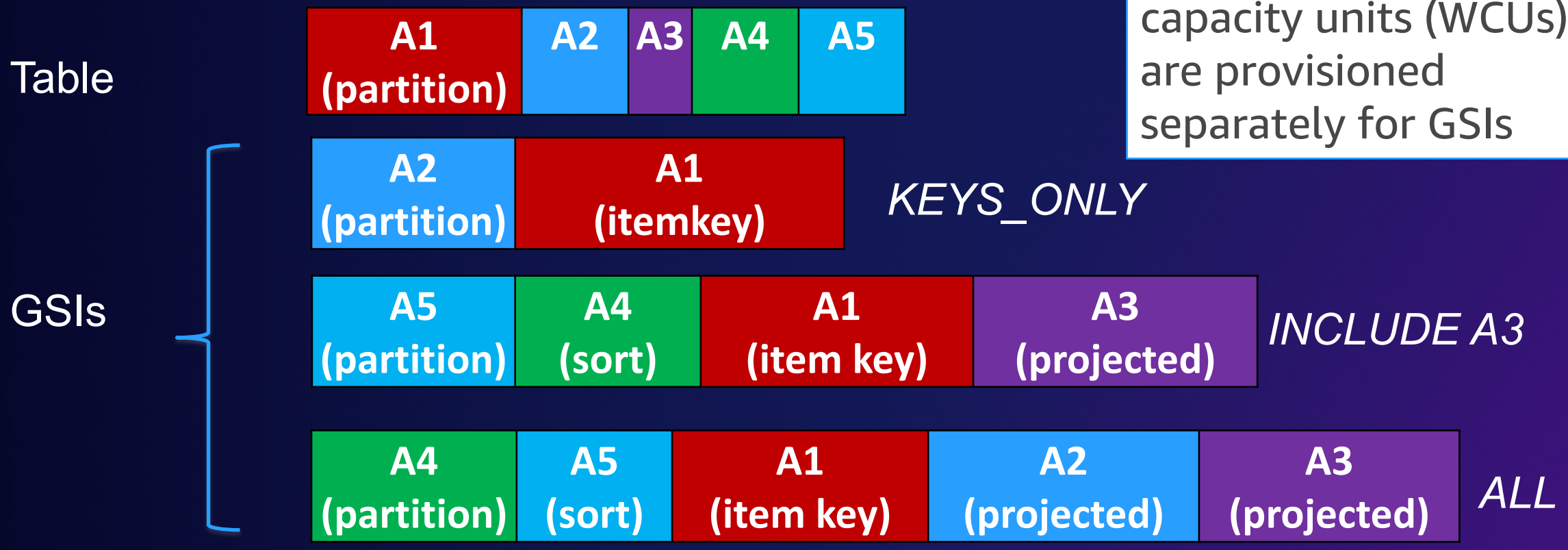
Global Secondary Index (GSI)

Online indexing

Alternate partition and/or sort key

Index is across all partition keys

Read capacity units (RCUs) and write capacity units (WCUs) are provisioned separately for GSIs



LSI or GSI?

LSI	GSI
Create at table creation	Create any time
Shares WCU/RCU with table	WCU/RCU independent of table
Size <= 10GB*	No size limits
Limit = 5	Limit = 20
Strong Consistency	Eventual Consistency

*10GB size limit is for a item collection size with an LSI

Tenets of DynamoDB data modeling

- Understand the use case
- Identify the access patterns
  - Read/write workloads
  - Query dimensions and aggregations
- Data modeling
  - Using NoSQL design patterns
- Review -> Repeat -> Review

- Nature of the data
- Relationships between the entities
- What does concurrent access look like?
- Time series data
- Archiving needs, etc.

Tenets of DynamoDB data modeling

- Understand the use case
- Identify the access patterns
  - Read/write workloads
  - Query dimensions and aggregations
- Data modeling
  - Using NoSQL design patterns
- Review -> Repeat -> Review

- Source data analysis (write workload)
- Reading one item versus multiple items (read workload)
- Query aggregations and KPIs

Tenets of DynamoDB data modeling

- Understand the use case
- Identify the access patterns
  - Read/write workloads
  - Query dimensions and aggregations
- Data modeling
  - Using NoSQL design patterns
- Review -> Repeat -> Review

- 1:1, 1:n, m:n relationships
- **1 application = 1 table**
  - Avoid unnecessary fetches
  - Simplify access patterns
- Identify primary key
  - Partition key and Sort key
- Query dimensions using LSIs and GSIs

Tenets of DynamoDB data modeling

- Understand the use case
- Identify the access patterns
  - Read/write workloads
  - Query dimensions and aggregations
- Data modeling
  - Using NoSQL design patterns
- Review -> Repeat -> Review

Example – Device Log

High cardinality

Sorted by date time

Filter by Status

Primary key		
Partition key: DeviceID	Sort key: Date	
d#12345	2020-04-24T14:40:00	State
		WARNING1
	2020-04-24T14:45:00	State
		WARNING1
	2020-04-24T14:50:00	State
		WARNING1
	2020-04-24T14:55:00	State
		NORMAL
d#54321	2020-04-11T05:50:00	State
		WARNING3
	2020-04-11T05:55:00	State
		WARNING3
	2020-04-11T06:00:00	State
		NORMAL

Access Pattern: Fetch all warning logs for a device that are sorted in descending order

- SELECT * FROM DeviceLog
- WHERE DeviceID = ‘d#12345'
- ORDER BY Date DESC
- FILTER ON State=‘WARNING1'

Returned

Filtered

Primary key		
Partition key: DeviceID	Sort key: Date	
d#12345	2020-04-24T14:40:00	State
		WARNING1
	2020-04-24T14:45:00	State
		WARNING1
	2020-04-24T14:50:00	State
		WARNING1
	2020-04-24T14:55:00	State
		NORMAL
d#54321	2020-04-11T05:50:00	State
		WARNING3
	2020-04-11T05:55:00	State
		WARNING3
	2020-04-11T06:00:00	State
		NORMAL

```
aws dynamodb query
--table-name DeviceLog
--key-condition-expression "#dID = :dID"
--no-scan-index-forward
--filter-expression "#s = :s"
--expression-attribute-names ‘{“#dID": "DeviceID", “#s": “State}’
--expression-attribute-values ‘{“:dID": {“S":"d#12345"}, “:s": {“S":"WARNING1"}}’
```

Use Composite Sort Key instead

Primary key	
Partition key: DeviceID	Sort key: State#Date
d#12345	NORMAL#2020-04-24T14:55:00
	WARNING1#2020-04-24T14:40:00
	WARNING1#2020-04-24T14:45:00
	WARNING1#2020-04-24T14:50:00
d#54321	NORMAL#2020-04-11T06:00:00
	NORMAL#2020-04-11T09:30:00
	WARNING2#2020-04-11T09:25:00
	WARNING3#2020-04-11T05:50:00
	WARNING3#2020-04-11T05:55:00

```
aws dynamodb query
--table-name DeviceLog
--no-scan-index-forward
--key-condition-expression "#dID = :dID  AND begins_with(#s, :sd)"
--expression-attribute-names '{"#cId": "DeviceID", "#s": "State#Date"}'
--expression-attribute-values '{":cId": {"S":"d#12345"}, ":sd": {"S":"WARNING1#"}}'
```

Access Pattern: Fetch all device logs for a given operator between two dates

Base Table

Primary key		Attributes		
Partition key: DeviceID	Sort key: State#Date			
d#12345	NORMAL#2020-04-24T14:55:00	Operator	Date	
		Liz	2020-04-24	
	WARNING1#2020-04-24T14:45:00	Operator	Date	
		Liz	2020-04-24	
	WARNING1#2020-04-24T14:50:00	Operator	Date	
		Liz	2020-04-24	
d#54321	NORMAL#2020-04-11T06:00:00	Operator	Date	
		Liz	2020-04-11	
	NORMAL#2020-04-11T09:30:00	Operator	Date	
		Sue	2020-04-11	
	WARNING2#2020-04-11T09:25:00	Operator	Date	
		Sue	2020-04-11	
	WARNING3#2020-04-11T05:55:00	Operator	Date	
		Liz	2020-04-11	
d#11223	WARNING4#2020-04-27T16:10:00	Operator	Date	
		Sue	2020-04-27	
	WARNING4#2020-04-27T16:15:00	Operator	Date	EscalatedTo
		Sue	2020-04-27	Sara

GSI-Operator

Primary key		Attributes	
Partition key: Operator	Sort key: Date		
Liz	2020-04-11	State#Date	DeviceID
		WARNING3#2020-04-11T05:55:00	d#54321
	2020-04-11	State#Date	DeviceID
		NORMAL#2020-04-11T06:00:00	d#54321
	2020-04-24	State#Date	DeviceID
		WARNING1#2020-04-24T14:45:00	d#12345
	2020-04-24	State#Date	DeviceID
		WARNING1#2020-04-24T14:50:00	d#12345
	2020-04-24	State#Date	DeviceID
		NORMAL#2020-04-24T14:55:00	d#12345
Sue	2020-04-11	State#Date	DeviceID
		WARNING2#2020-04-11T09:25:00	d#54321
	2020-04-11	State#Date	DeviceID
		NORMAL#2020-04-11T09:30:00	d#54321
	2020-04-27	State#Date	DeviceID
		WARNING4#2020-04-27T16:10:00	d#11223
	2020-04-27	State#Date	DeviceID
		WARNING4#2020-04-27T16:15:00	d#11223

Access Pattern: Fetch all device logs for a given operator between two dates

Primary key		Attributes	
Partition key: Operator	Sort key: Date		
	2020-04-11	State#Date	DeviceID
		WARNING3#2020-04-11T05:55:00	d#54321
	2020-04-11	State#Date	DeviceID
		NORMAL#2020-04-11T06:00:00	d#54321
Liz	2020-04-24	State#Date	DeviceID
		WARNING1#2020-04-24T14:45:00	d#12345
	2020-04-24	State#Date	DeviceID
		WARNING1#2020-04-24T14:50:00	d#12345
	2020-04-24	State#Date	DeviceID
		NORMAL#2020-04-24T14:55:00	d#12345
Sue	2020-04-11	State#Date	DeviceID
		WARNING2#2020-04-11T09:25:00	d#54321
	2020-04-11	State#Date	DeviceID
		NORMAL#2020-04-11T09:30:00	d#54321
	2020-04-27	State#Date	DeviceID
		WARNING4#2020-04-27T16:10:00	d#11223
	2020-04-27	State#Date	DeviceID
		WARNING4#2020-04-27T16:15:00	d#11223

GSI – Operator

```
aws dynamodb query
--table-name DeviceLog
--index-name GSI–Operator
--key-condition-expression "#op = :op AND #d  between :d1 AND :d2"
--expression-attribute-names '{"#op": "Operator" , "#d": "Date"}'
--expression-attribute-values '{":op": {"S":"Liz"} , ":d1": {"S":"2020-04-20"}, ":d2":{"S":"2020-04-25"}}'
```

Access Pattern: Fetch all escalated device logs for a given supervisor

GSI-Supervisor

Primary key		Attributes	
Partition key: EscalatedTo	Sort key: State#Date	DeviceID	Operator
Sara	WARNING4#2020-04-27T16:15:00	d#11223	Sue

Base Table

Primary key		Attributes		
Partition key: DeviceID	Sort key: State#Date			
d#12345	NORMAL#2020-04-24T14:55:00	Operator	Date	
		Liz	2020-04-24	
	WARNING1#2020-04-24T14:45:00	Operator	Date	
		Liz	2020-04-24	
	WARNING1#2020-04-24T14:50:00	Operator	Date	
		Liz	2020-04-24	
d#54321	NORMAL#2020-04-11T06:00:00	Operator	Date	
		Liz	2020-04-11	
	NORMAL#2020-04-11T09:30:00	Operator	Date	
		Sue	2020-04-11	
	WARNING2#2020-04-11T09:25:00	Operator	Date	
		Sue	2020-04-11	
	WARNING3#2020-04-11T05:55:00	Operator	Date	
		Liz	2020-04-11	
d#11223	WARNING4#2020-04-27T16:10:00	Operator	Date	
		Sue	2020-04-27	
	WARNING4#2020-04-27T16:15:00	Operator	Date	EscalatedTo
		Sue	2020-04-27	Sara

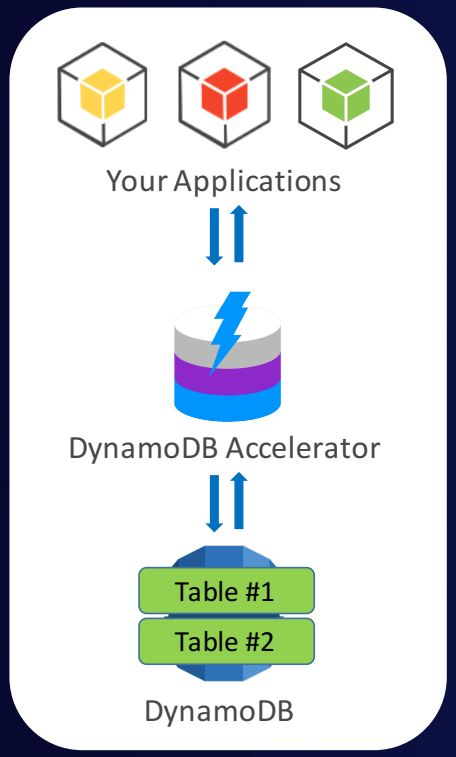
Sparse GSI: Only items that match the GSI index are projected.

Good for:

- ‘Needle in the haystack’
- Cost effective ‘scans’
- Item management

DynamoDB Accelerator (DAX)

In-memory cache



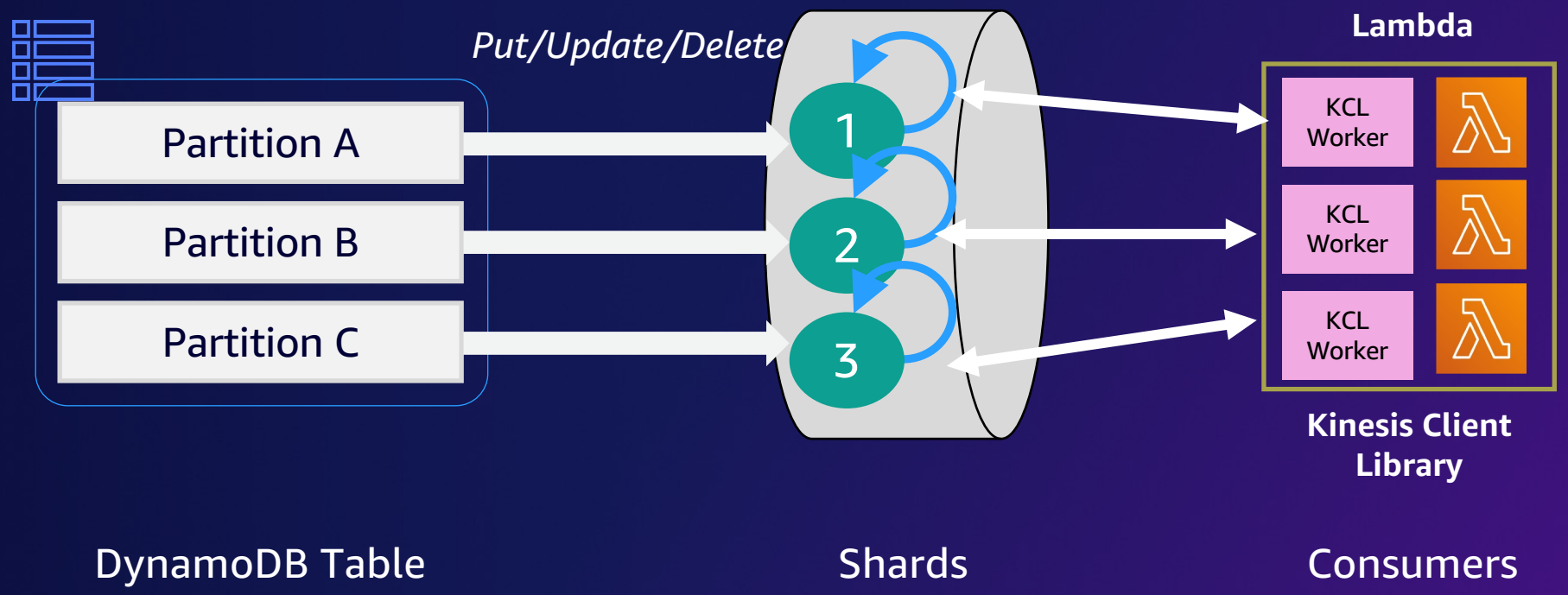
Features

- **Fully managed, highly available:** handles all software management, fault tolerant, replication across multi-AZs within a region
- **DynamoDB API compatible:** seamlessly caches DynamoDB API calls, no application re-writes required
- **Write-through:** DAX handles caching for writes
- **Flexible**: Configure DAX for one table or many
- **Scalable**: scales-out to any workload with up to 9 read replicas
- **Manageability:** fully integrated AWS service: Amazon CloudWatch, Tagging for DynamoDB, AWS Console
- **Security:** Amazon VPC, AWS IAM, AWS CloudTrail, AWS Organizations, Encryption in Transit

DynamoDB Streams – Event-Driven Architecture

DynamoDB Streams

- ✓ Ordered stream of item changes
- ✓ Exactly once, strictly ordered by key
- ✓ Highly durable, scalable
- ✓ 24 hour retention
- ✓ Sub-second latency
- ✓ Compatible with Kinesis Client Library



Time-to-live (TTL)

TTL Attribute

ID	Name	Size	Expiry
1234	A	100	1456702305
2222	B	240	1456702400
3423	C	150	1459207905

Features

- Automatically delete items from a table based on expiration timestamp
- User defined TTL attribute in epoch time format
- TTL activity recorded in DynamoDB Streams

Key Benefits

- Reduce costs by deleting items no longer needed
- Optimize application performance by controlling table size growth
- Trigger custom workflows with Streams and Lambda

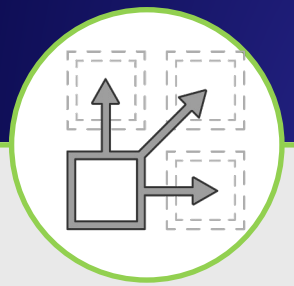
Amazon DocumentDB

Fast, scalable, and fully managed MongoDB-compatible database service

Fast

Millions of requests per second with millisecond latency

Scalable



Separation of compute and storage scales both independently; scale out to 15 read replicas in minutes

Fully managed

Managed by AWS: no hardware provisioning; auto patching, quick setup, secure, and automatic backups

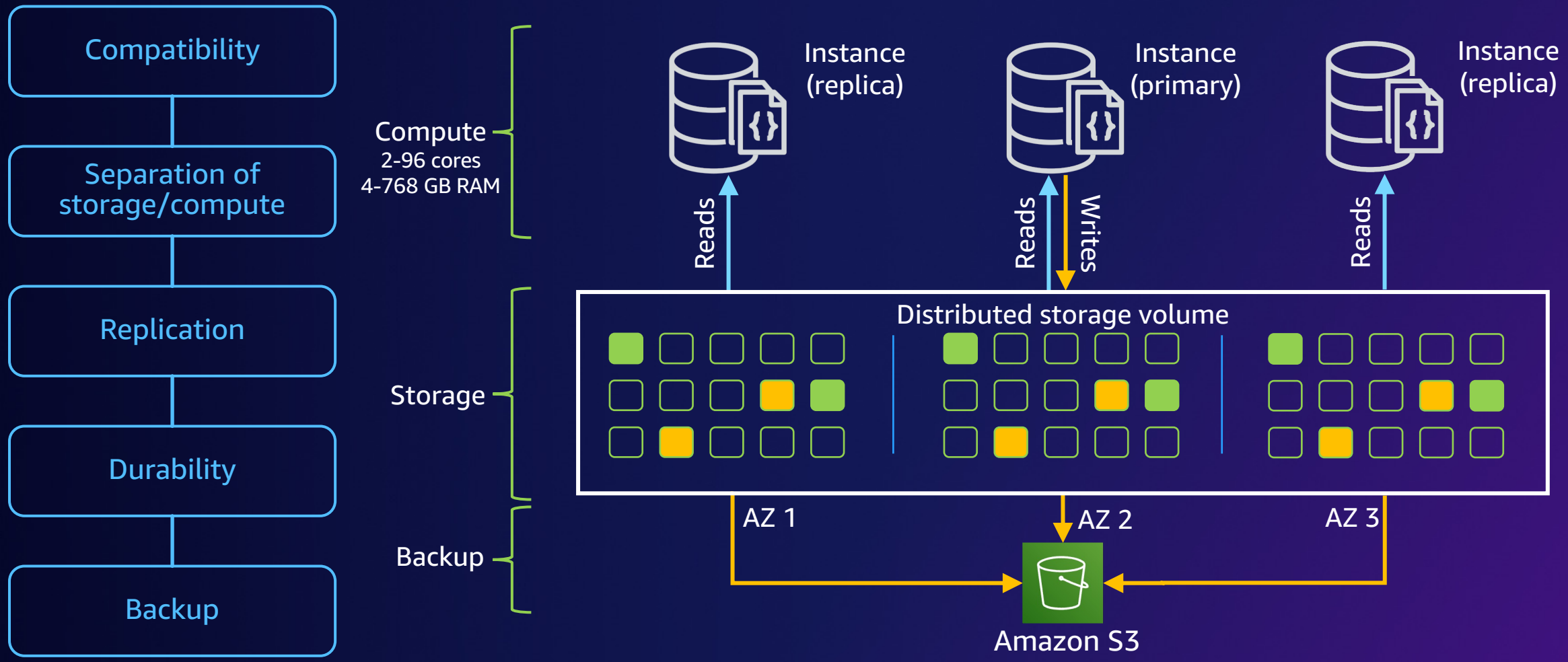
MongoDB compatible

Compatible with MongoDB 3.6 and 4.0; use the same SDKs, tools, and applications with Amazon DocumentDB. Migrate workloads with AWS DMS.

Purpose-built document database engineered for the cloud

aws

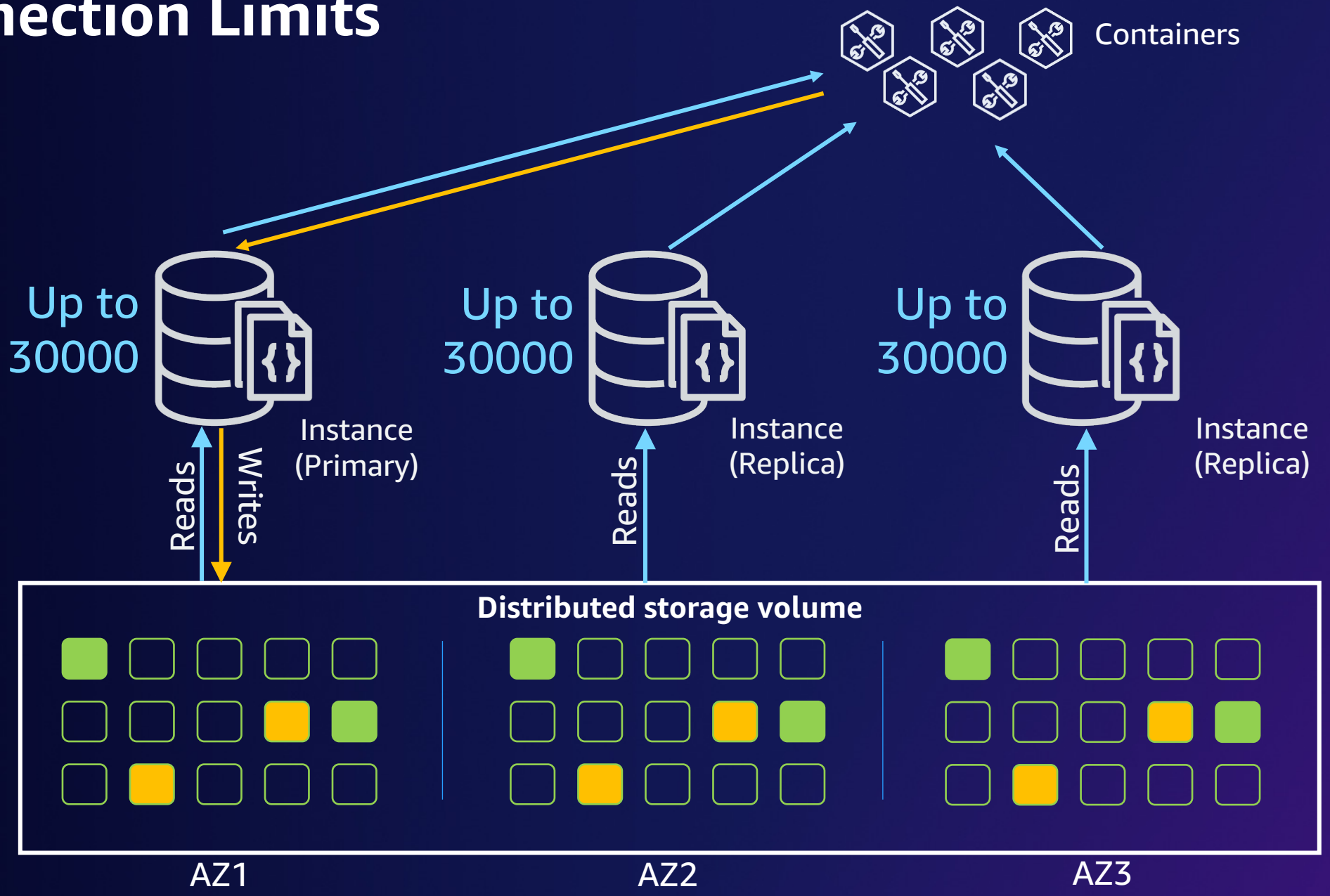
Amazon DocumentDB: Cloud Native Architecture



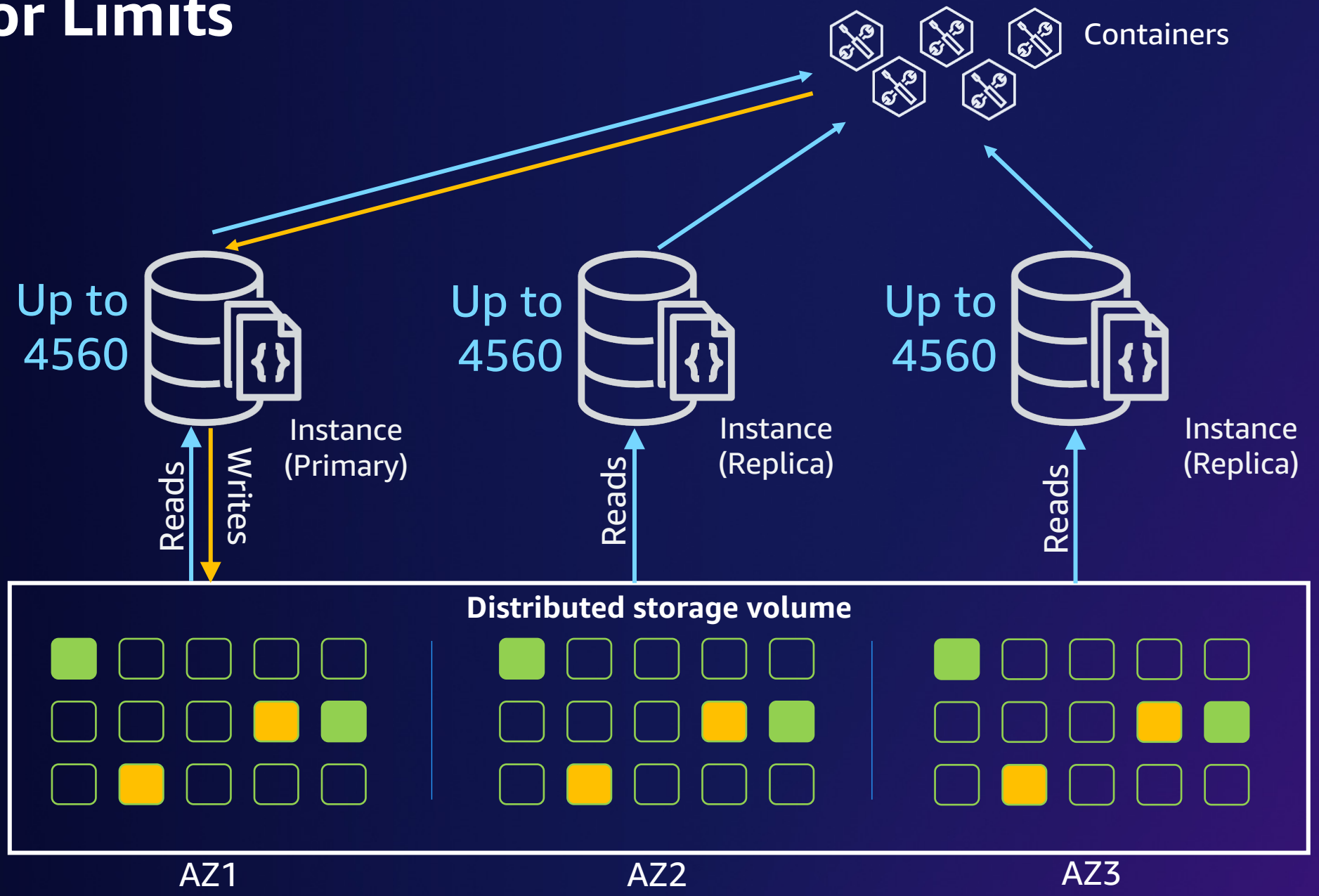
Demo code

```python
import pymongo, boto3
from secretsManager import get_secret

secret = get_secret() ## Method to get secrets from Secrets Manager

## Create mongo client with username and password from AWS Secrets Manager
client = pymongo.MongoClient(
    secret['host'],
    secret['port'],
    username=secret['username'],
    password=secret['password'],
    ssl='true', ## TLS Enabled by default
    ssl_ca_certs='rds-combined-ca-bundle.pem',
    retryWrites='false',
    replicaSet='rs0', ## Connect as a replica set
    readPreference='secondaryPreferred' ## Reads are sent to replicas
   ## DocumentDB implments the best practice of highly durable writes (write quorum of 4)
   ## w='majority',
   ## j = true
    )

db = client.test ##Get the test database
db.col.insert_one({'x':'AmazonDocumentDB'}) ## Insert a doc(request routed to Primary)
x = db.col.find_one() ## Find a document (request routed to replica)
print(x) ## Print to screen
client.close() ## Close Client
```

Connection Limits



Cursor Limits



Links

Docs:

- https://docs.aws.amazon.com/dynamodb/index.html
- https://docs.aws.amazon.com/documentdb/?id=docs_gateway
- https://aws.amazon.com/blogs/database/

Workshops:

- https://amazon-dynamodb-labs.workshop.aws/
- https://catalog.us-east-1.prod.workshops.aws/workshops/464d6c17-9faa-4fef-ac9f-dd49610174d3/en-US

Thank you!