CLOUD COMPUTING SYSTEMS

Lecture 3

Nuno Preguiça

(nuno.preguica_at_fct.unl.pt)

OUTLINE

Motivation for geo-replicated storages.

Cloud databases: first generation.

- Amazon Dynamo.

Cloud databases: current generation.

- Azure Cosmos DB.

SCALING APP SERVICES

When scaling out (running more instances) the application server of a three-tier model application, the data tier becomes a bottleneck.

Scaling up (running in more powerful machine) the database server has limits.

THE QUEST OF FASTER DATABASES

SQL databases provides strong consistency.

- Serializability requires that concurrent transactions execute as if they were executed serially.
- Common implementations use locks for achieving this.

Replicated databases with strong consistency provide the illusion that there is a single database.

- Require coordination among multiple servers.
  - Challenging in a single data center;
  - Too high latency in geo-replicated data centers.

OUTLINE

Motivation for geo-replicated storages.

Cloud databases: first generation.

- **Amazon Dynamo**.

Cloud databases: current generation.

- Azure Cosmos DB.

CLOUD DATABASES: FIRST GENERATION

Dynamo database from Amazon [2007].

Goal:

- Geo-replication with high availability and performance.

Highly influential in other cloud databases.

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,
Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall
and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face

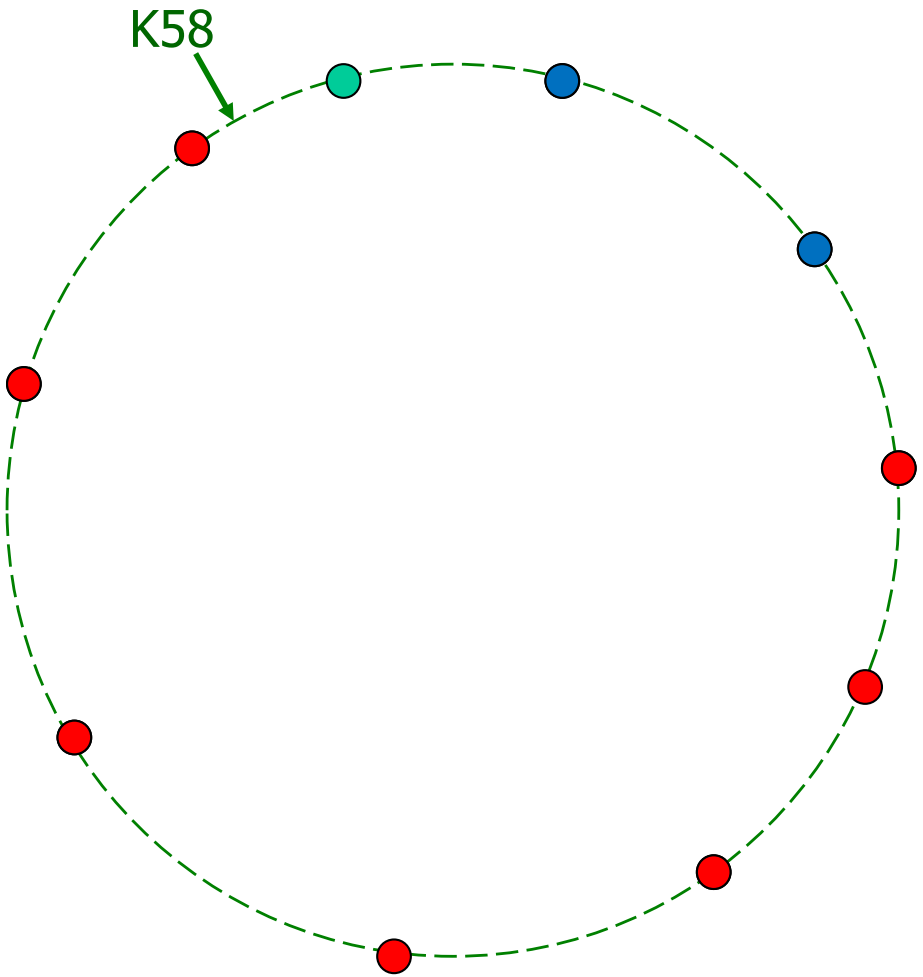
One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and

DYNAMO FEATURES: DATA MODEL

Simple data model: key-value pairs.

Simple API:

- get(key) -> (list of values, context)
- put(key, value, context)



Why?

Easy to scale.

Consistent hashing

DYNAMO FEATURES: EVENTUAL CONSISTENCY

High availability and low latency require accepting operations without coordinating with other replicas.

Consequence: concurrent updates. How to handle?

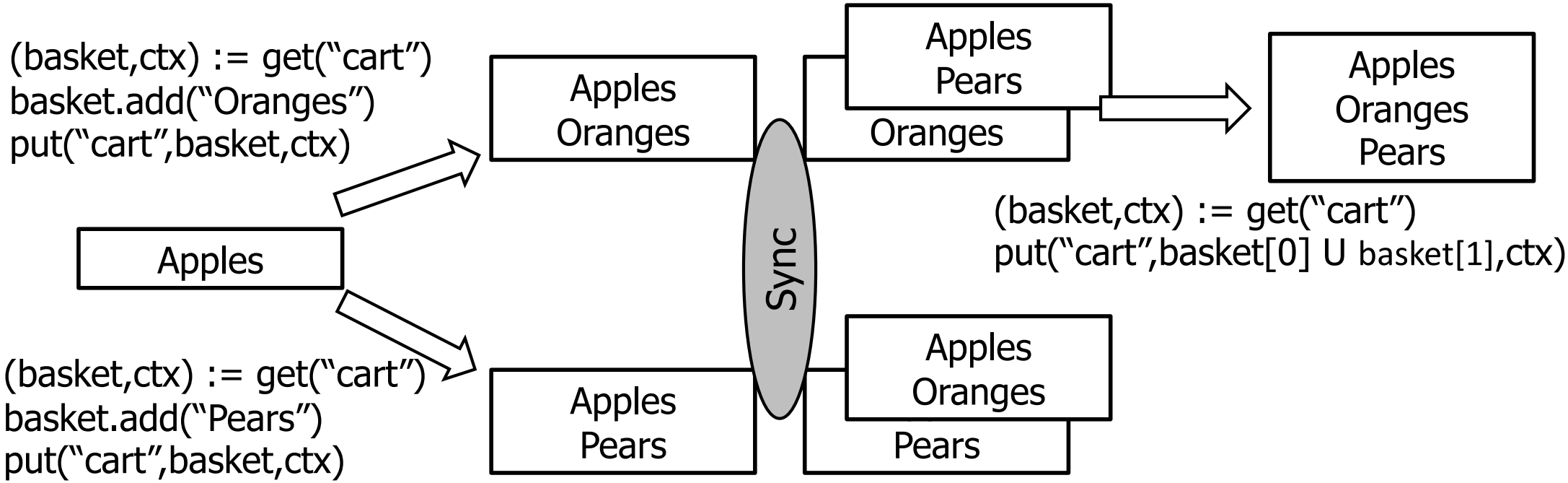
Eventual consistency model:

- Replicas can always accept updates;
- Updates are eventually propagated to all replicas;
- Replicas converge to the same state (after receiving the same set of updates).

DYNAMO FEATURES: EVENTUAL CONSISTENCY (2)

When concurrent updates occur, the system keeps the multiple versions. The application must merge versions by applying a new update.

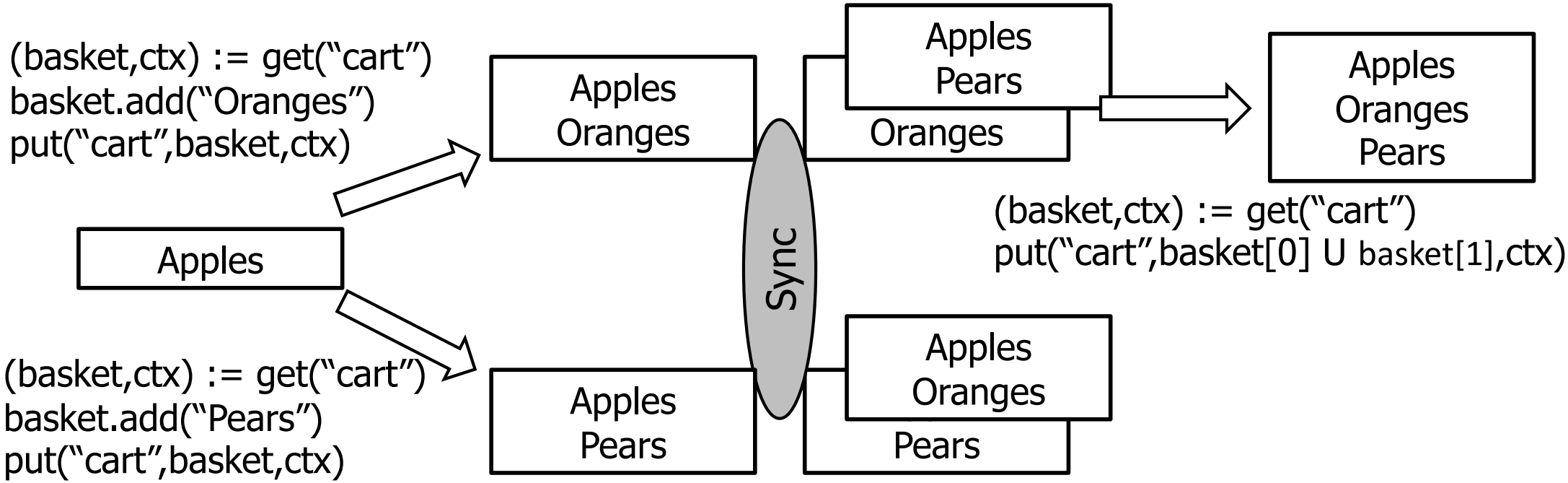
E.g. Shopping cart.



DYNAMO FEATURES: EVENTUAL CONSISTENCY (3)

get(key) -> (list of values, context)

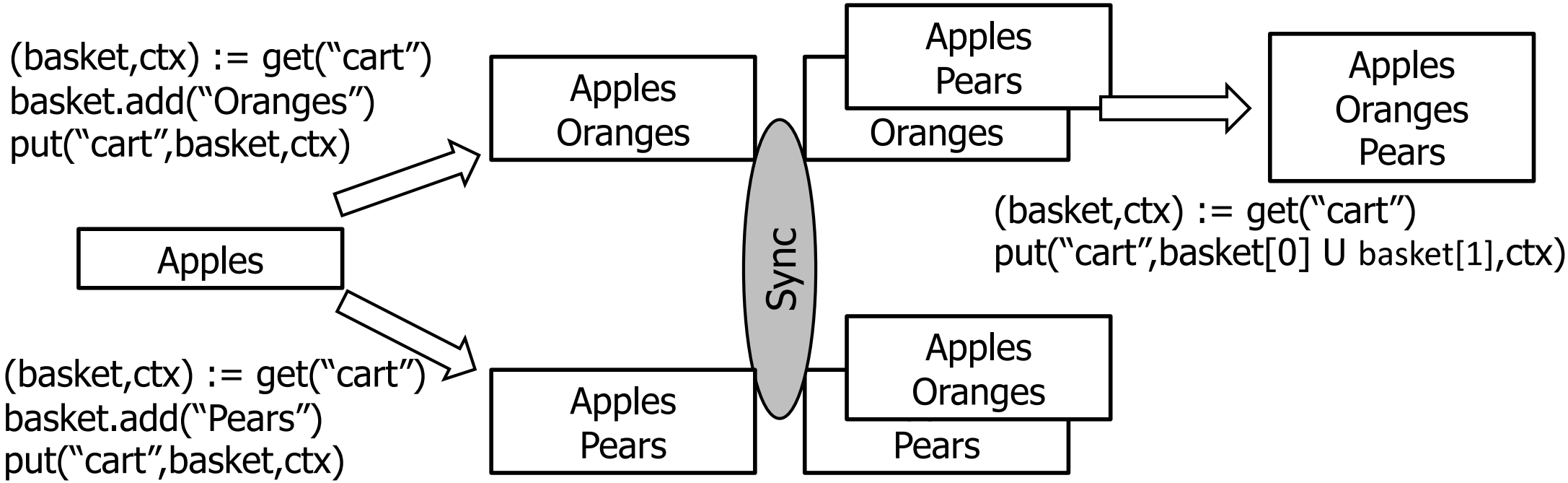
- Get will return all values associated with a key.



DYNAMO FEATURES: EVENTUAL CONSISTENCY (4)

put(key, value, context)

- Put replaces the values of the version specified by context with a new value. From what you have studied in Distributed Systems course, what is the context?
- **NOTE**: this approach guarantees that if between a get and a put from a client, another client modifies the value of the key, both version will be kep.



EVENTUAL CONSISTENCY: PROBLEMS

Not all applications/data work correctly under eventual consistency.

Examples?

Impossible to enforce some invariants – e.g. val >= 0.

- E.g.: stock maintenance.

Need "smart" reconciliation for some data – e.g. counter.

- E.g. number of likes in social networks.

More on algorithms used in eventual consistency systems in Algorithms and Distributed Systems course.

Amazon DynamoDB

Cloud database available at AWS.

Extends the initial design of Dynamo in several directions.

- Data model supports documents (like JSON documents).
- Geo-replication: possible to select regions to replicate data.
- Stronger consistency with conditional update operation.
  - Execute update if some condition on the state holds.

OUTLINE

Motivation for geo-replicated storages.

Cloud databases: first generation.

- Amazon Dynamo.

Cloud databases: current generation.

- **Azure Cosmos DB, Amazon DynamoDB.**

Azure Cosmos DB

Azure Cosmos DB is Microsoft's globally distributed, multi-model database service.

Easy to scale throughput and storage across any number of Azure regions worldwide.

Single-digit-millisecond data access using APIs that include SQL, MongoDB, Cassandra, Tables, etc.

Provides comprehensive service level agreements (SLAs) for throughput, latency, availability, and consistency guarantees.

Built heavily on research results previously published…

DATABASE API

Cosmos DB provides different APIs, with different data models:

1. Relational data model.
   - Database composed by tables and documents.
2. Document data model.
   - A document can be any JSON object.
3. Graph data model.
   - Represent data that can be modelled as a graph (e.g. social network relations, etc.)

...

How to support all these APIs in the same database?

DATA MODEL UNDER THE HOODS

Internally, the core type system of Azure Cosmos DB's database engine is atom-record-sequence (ARS) based.

Atoms consist of a small set of primitive types e.g. string, bool, number etc.

Records are structs (pairs key, value).

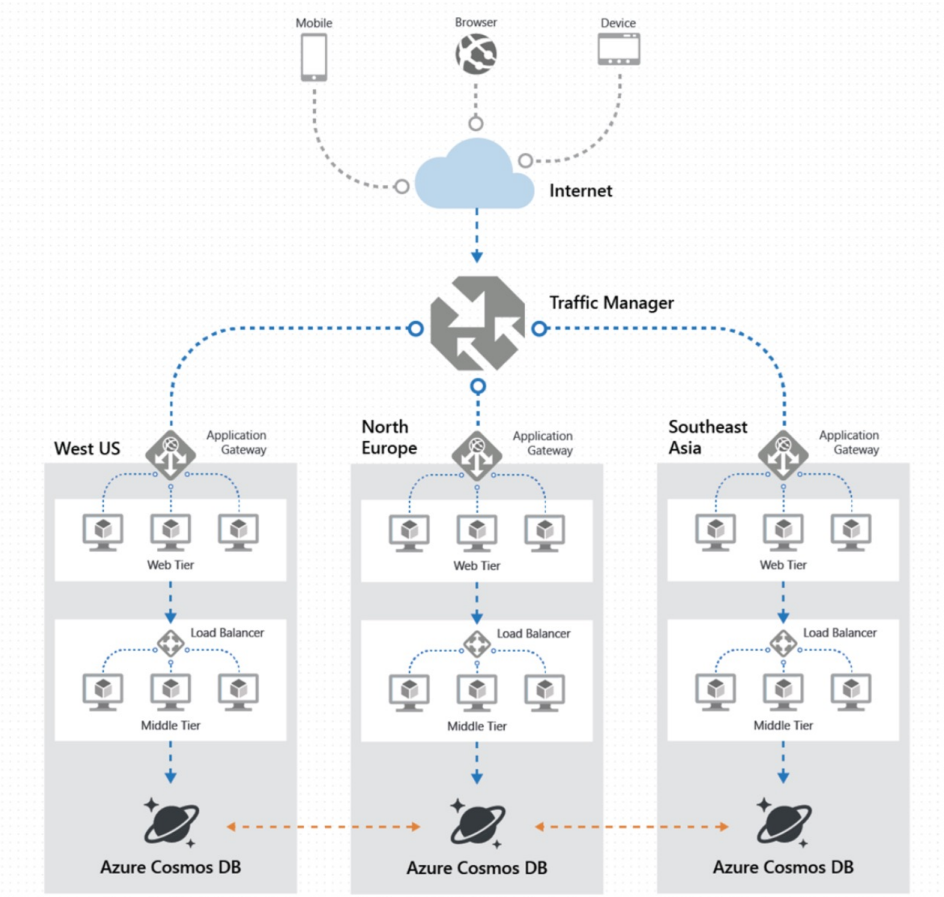
Sequences are arrays consisting of atoms, records or sequences.

GLOBAL DISTRIBUTION

Possible to configure a database to be globally distributed and available in any of the Azure regions.

How to decide where to locate replicas?

Depends on global reach and location of clients.



SUPPORT FOR GLOBAL DISTRIBUTION

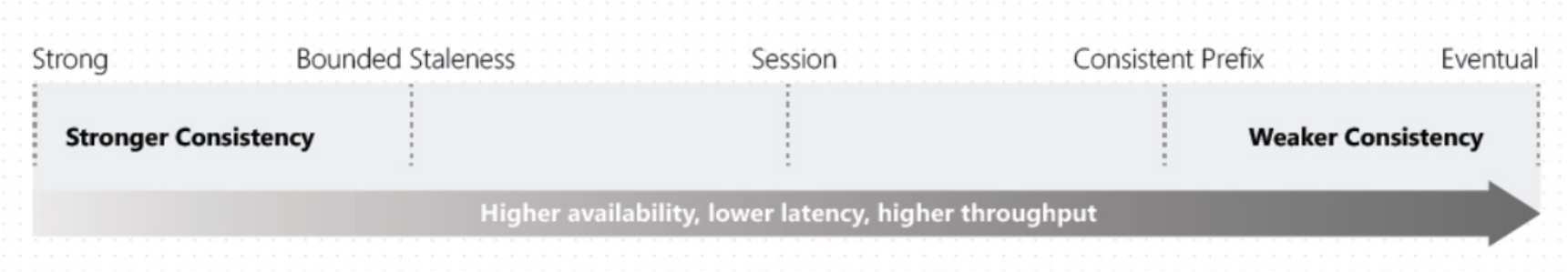
Multi-master replication.

- Multiple replicas will accept update operations (without coordinating to other replicas).

Well defined consistency-level.

Scalable read and write throughput with SLAs.

CONSISTENCY MODEL



Consistency as a spectrum, from stronger to weaker consistency.

Applications can select the consistency level appropriate for each database / operation.

Consistency level: strong consistency (intuition)

Strong consistency gives the illusion that there is a single database (despite the fact that the implementation is replicated and partitioned).

CONSISTENCY LEVEL: STRONG CONSISTENCY

Strong consistency offers linearizability.

1. A read is guaranteed to return the most recently committed version of an item.
2. A client never sees an uncommitted or partial write.

How to implement?

Replication algorithm with master replica (e.g. Paxos).

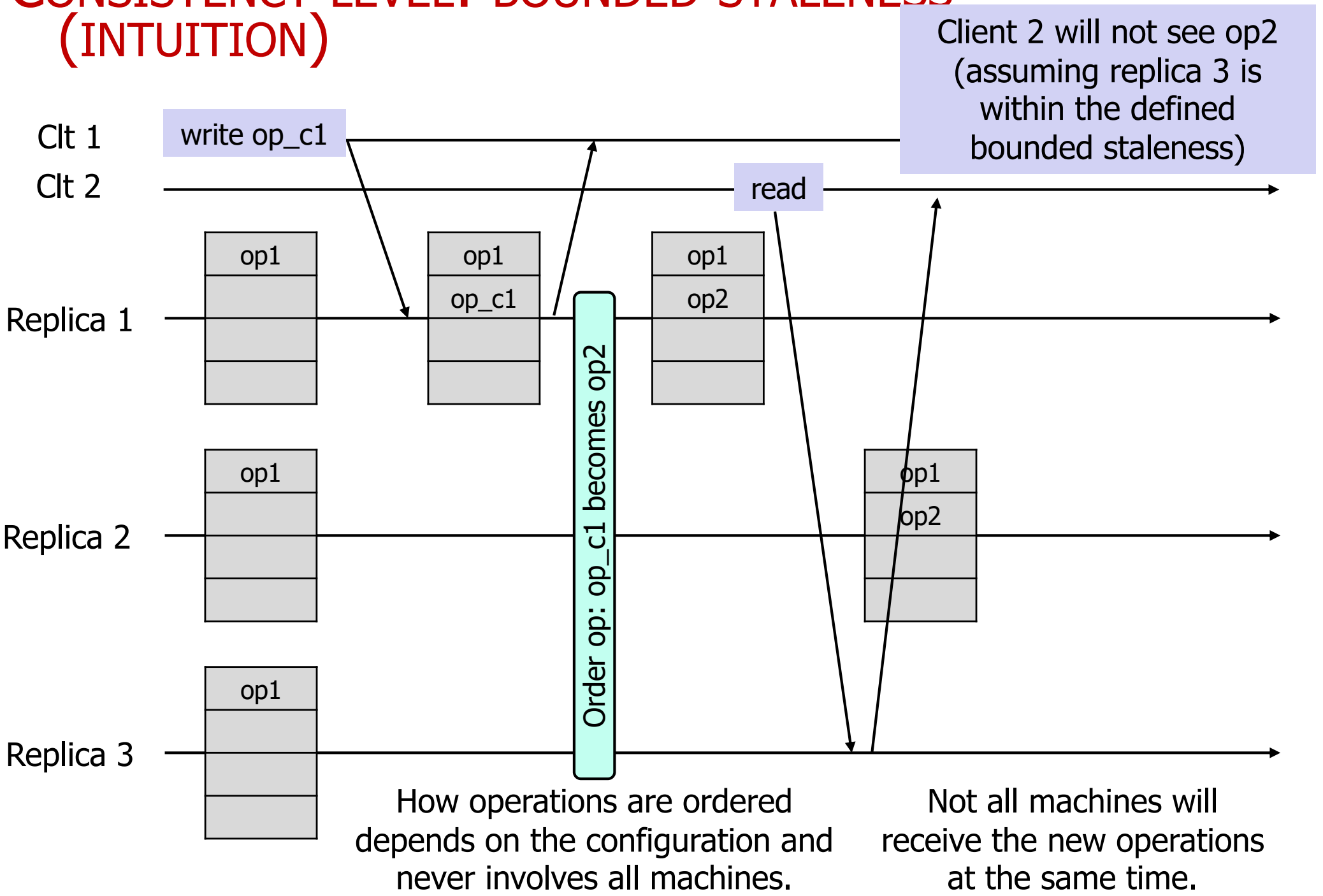
Consistency level: bounded staleness (intuition)

The database state evolves by applying all updates in a total order – all updates outside the staleness window are applied in the same total order in all replica.

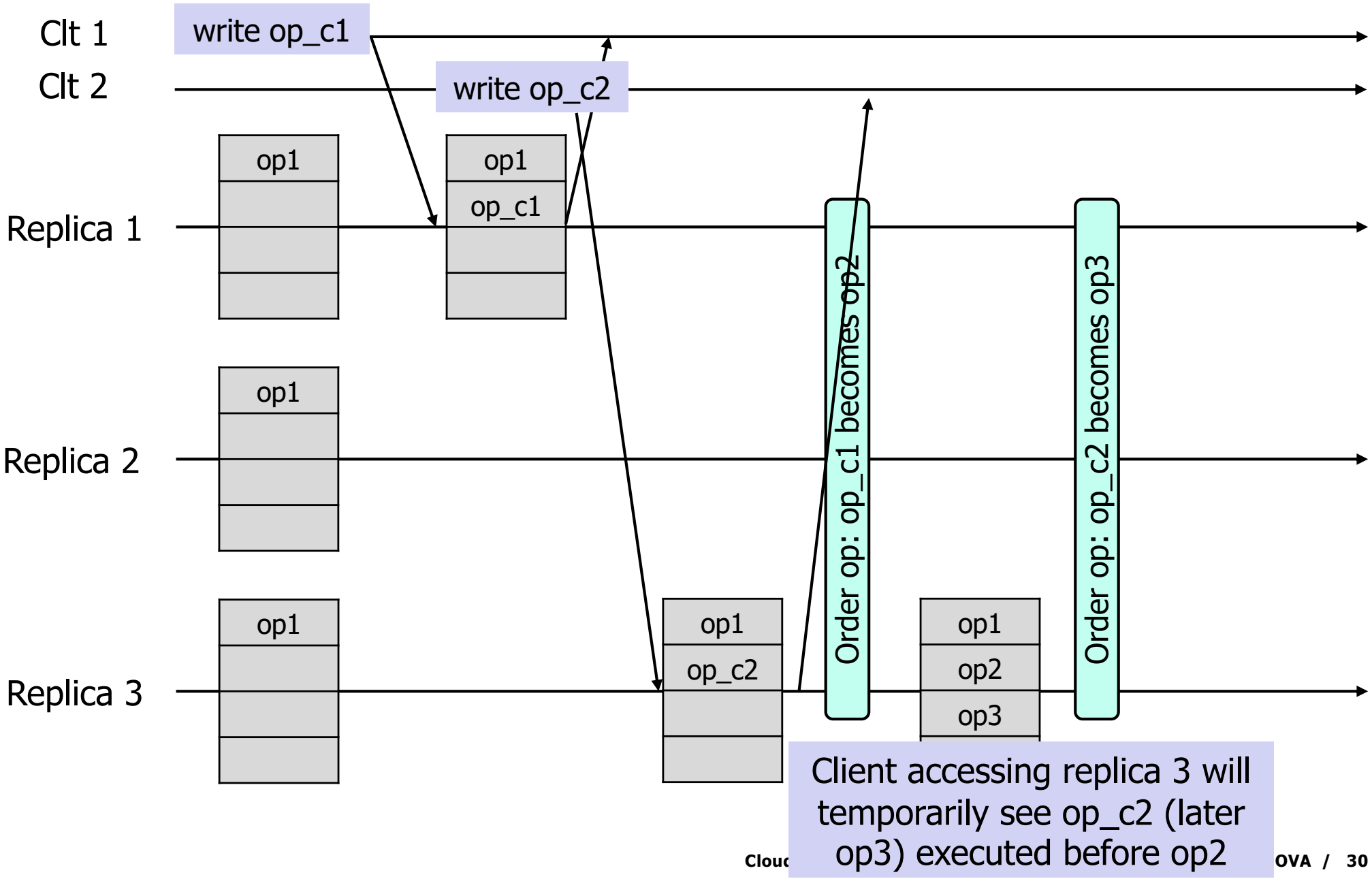
For updates inside the staleness window:

- When there is a single write region:
  - For clients in that region, updates are totally order immediately (equivalent to strong consistency)
  - For clients in other regions, clients may miss updates inside the staleness window, but they will see them in order (leading to consistent prefix)
- When there are multiple write regions:
  - For clients writing to a single region, clients will observe those updates updates in order (equivalent to consistent prefix)
  - When client write to different regions, updates are applied locally but may be reorder later (leading to eventual consistency)

CONSISTENCY LEVEL: BOUNDED STALENESS (INTUITION)



CONSISTENCY LEVEL: BOUNDED STALENESS (INTUITION)



CONSISTENCY LEVEL: BOUNDED STALENESS

Bounded staleness provides the following guarantees:

1. Reads observe a **consistent-prefix**, i.e., all updates are totally ordered except within the "staleness window".
2. **Monotonic read**, meaning that clients observe a version that is later that the previously read. Why is this interesting?
3. A read might return an old value of the data item, configured as:
    - The number of versions (K) behind the current version that the read can return;
      E.g. with K = 5, the client knows that it might miss up to 5 writes.
    - The time interval (T) for which it might miss a write.
      E.g. with T = 10ms, the client knows that it might miss updates executed in the last 10 ms.

CONSISTENCY LEVEL: BOUNDED STALENESS

How to implement?

- Master region orders and propagates updates to other regions.
- A region can receive operations that propagate to the master region to be ordered – while the order is not established by the master, these updates are visible out-of-order in the local region.
- Read can be performed in the local region, given that the bounded conditions can be established locally – e.g. from the last message received from the master, a replica know the potential staleness.

Consistency level: session (intuition)

The client has a session, where she sees the database evolving as if it was a single replica. Updates from other clients/regions are integrated in the view of the session.

Note: different clients with this consistency level may see different database states.

CONSISTENCY LEVEL: SESSION

Session consistency level provides the following guarantees:

1. Within a single client session, reads are guaranteed to honor the consistent-prefix (assuming a single "writer" session), monotonic reads, monotonic writes, read-your-writes, and write-follows-reads guarantees.

CONSISTENCY LEVEL: SESSION (2)

Monotonic writes: writes are propagated after writes that logically precede them.

- E.g. when creating an account and later changing some property, the creation is propagated always first.
- When inserting a post and adding a reference to the post in some other object, the insertion is propagated always first.

Write-follows-reads: a write is propagated always after the read updates.

- E.g.: if a user sees a post and later replies, the reply will be propagated always after the operation that created the original post.

Read your writes: a read always reflects the writes executed in the session.

- E.g.: if a user makes a post, the following reads will always return that post.

CONSISTENCY LEVEL: SESSION (CONT.)

Session consistency level provides the following guarantees:

2. Clients outside of the session performing writes will see:
   - Clients in the same region (either reading only or writing) will see updates in order (leading to consistent prefix)
   - Clients in other regions, with updates performed in a single region, will see update in order (leading to Consistent prefix)
   - For client writing in other regions, they may see their update being reorder in relation to the writes of other regions (leading to Eventual consistency)

CONSISTENCY LEVEL: SESSION (3)

How to implement?

- Client maintains version vector (token, context) with a summary of the operations observed;
- Reads request a state that is at least as recent as the vector;
- Updates in each region are ordered and this order is respected when applying them to other replicas).

Consistency level: consistent prefix (intuition)

The client sees a prefix of the updates from every region, but might miss recent updates from different regions.

CONSISTENCY LEVEL: CONSISTENT PREFIX

Consistency prefix level provides the following guarantees:

1. Results that are returned contain some prefix of all the updates, with no gaps;
2. Reads never see out-of-order writes from a region (i.e., it always observes updates executed in region in the same order).

CONSISTENCY LEVEL: CONSISTENT PREFIX

How to implement?

- Region orders updates and propagates them to the replicas;
- Reads sees the updates received by the master (in order) more local request.

Consistency level: eventual (intuition)

The client might see a state that reflects any subset of the updates.

CONSISTENCY LEVEL: EVENTUAL

Under eventual consistency, there is no ordering guarantee for reads. In the absence of any further writes, the replicas eventually converge.

CONSISTENCY LEVELS EXPLAINED THROUGH BASEBALL

	1	2	3	4	5	6	7	8	9	Runs
Visitors	0	0	1	0	1	0	0			2
Home	1	0	1	1	0	2				5

There is a single writer updating the result. Possible returns:

Consistency level	Scores (Visitors, Home)
Strong	2-5

[example from Azure documentation]

CONSISTENCY LEVELS EXPLAINED THROUGH BASEBALL

	1	2	3	4	5	6	7	8	9	Runs
Visitors	0	0	1	0	1	0	0			2
Home	1	0	1	1	0	2				5

There is a single writer updating the result. Possible returns:

Consistency level	Scores (Visitors, Home)
Bounded staleness	Scores that are at most one inning out of date: 2-3, 2-4, 2-5

[example from Azure documentation]

CONSISTENCY LEVELS EXPLAINED THROUGH BASEBALL

	1	2	3	4	5	6	7	8	9	Runs
Visitors	0	0	1	0	1	0	0			2
Home	1	0	1	1	0	2				5

There is a single writer updating the result. Possible returns:

Consistency level	Scores (Visitors, Home)
Session	• For the writer: 2-5 • For anyone other than the writer: 0-0, 0-1, 0-2, 0-3, 0-4, 0-5, 1-0, 1-1, 1-2, 1-3, 1-4, 1-5, 2-0, 2-1, 2-2, 2-3, 2-4, 2-5 • After reading 1-3: 1-3, 1-4, 1-5, 2-3, 2-4, 2-5

[example from Azure documentation]

CONSISTENCY LEVELS EXPLAINED THROUGH BASEBALL

	1	2	3	4	5	6	7	8	9	Runs
Visitors	0	0	1	0	1	0	0			2
Home	1	0	1	1	0	2				5

There is a single writer updating the result. Possible returns:

Consistency level	Scores (Visitors, Home)
Consistent prefix	0-0, 0-1, 1-1, 1-2, 1-3, 2-3, 2-4, 2-5

[example from Azure documentation]

CONSISTENCY LEVELS EXPLAINED THROUGH BASEBALL

	1	2	3	4	5	6	7	8	9	Runs
Visitors	0	0	1	0	1	0	0			2
Home	1	0	1	1	0	2				5

There is some writer recording the result. Possible returns:

Consistency level	Scores (Visitors, Home)
Eventual	0-0, 0-1, 0-2, 0-3, 0-4, 0-5, 1-0, 1-1, 1-2, 1-3, 1-4, 1-5, 2-0, 2-1, 2-2, 2-3, 2-4, 2-5

[example from Azure documentation]

What consistency model to select (according to Microsoft)

Session consistency is optimal for most applications.

If **stronger consistency** is necessary, select bounded staleness. Bounded staleness is almost as good as strong consistency if the bound is small, but reads can be processed locally.

If your application requires eventual consistency, it is recommended that you use **consistent prefix** consistency level – provide better guarantee with similar cost.

If you need the highest availability and the lowest latency, then use eventual consistency level.

What is missing?

WHAT CONSISTENCY MODEL TO SELECT

Examples.

TRANSACTIONS AND CONDITIONAL UPDATES

CosmosDB supports transactions with snapshot isolation for updates executed to a container's logical partition.

- There is no support for transactions across partitions.

CosmosDB has support for conditional updates.

CONFLICT RESOLUTION

Consistency levels allow for concurrent updates. Conflict occur when:

Insert conflicts: These conflicts can occur when an application simultaneously inserts two or more items with the same unique index in two or more regions.

Replace conflicts: These conflicts can occur when an application updates the same item simultaneously in two or more regions.

Delete conflicts: These conflicts can occur when an application simultaneously deletes an item in one region and updates it in another region.

CONFLICT RESOLUTION (2)

CosmosDB supports the following conflict resolution policies for application programs:

1. Last-Write-Wins (LWW): uses a system-defined timestamp to select which version to keep.
2. Application-defined (Custom): possible to define a merge procedure to solve conflicts. These procedures get invoked upon detection of the write-write conflicts – the procedure executes exactly-once.

Cosmos DB uses CRDTs internally to manage concurrent updates.

DURABILITY

Service-level agreement, based on:

Recovery time objective (**RTO**) is the time until an application recover from a disruptive event.

Recovery point objective (**RPO**) is the period of time for which updates might get lost in a failure.

DURABILITY (2)

Region(s)	Replication mode	Consistency level	RPO	RTO
1	Single or Multi-Master	Any Consistency Level	< 240 Minutes	<1 Week
>1	Single Master	Session, Consistent Prefix, Eventual	< 15 minutes	< 15 minutes
>1	Single Master	Bounded Staleness	*K* & *T*	< 15 minutes
>1	Single Master	Strong	0	< 15 minutes
>1	Multi-Master	Session, Consistent Prefix, Eventual	< 15 minutes	0
>1	Multi-Master	Bounded Staleness	*K* & *T*	0

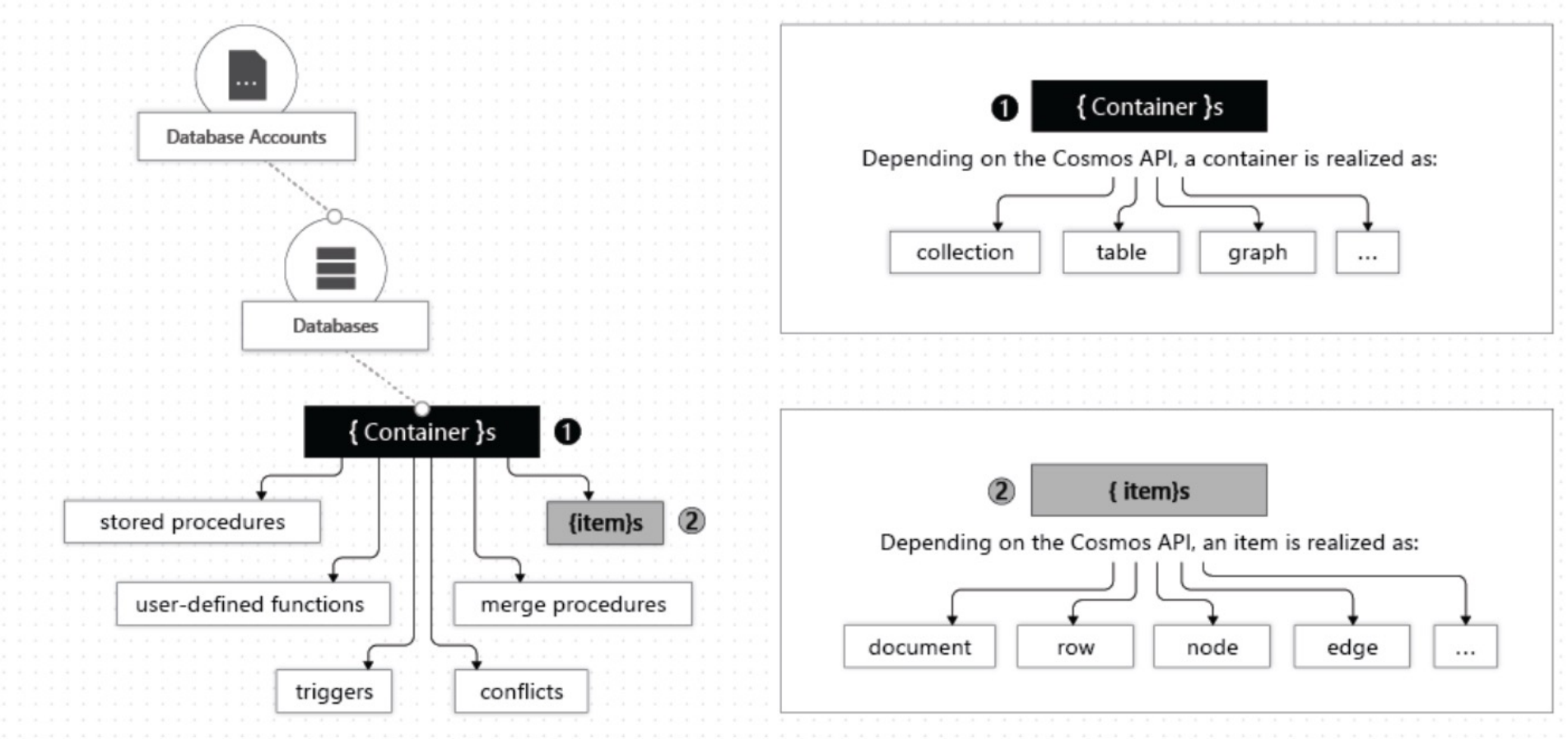
DURABILITY (3)

What justifies these values?

Region(s)	Replication mode	Consistency level	RPO	RTO
1	Single or Multi-Master	Any Consistency Level	< 240 Minutes	<1 Week
>1	Single Master	Session, Consistent Prefix, Eventual	< 15 minutes	< 15 minutes
>1	Single Master	Bounded Staleness	*K* & *T*	< 15 minutes
>1	Single Master	Strong	0	< 15 minutes
>1	Multi-Master	Session, Consistent Prefix, Eventual	< 15 minutes	0
>1	Multi-Master	Bounded Staleness	*K* & *T*	0

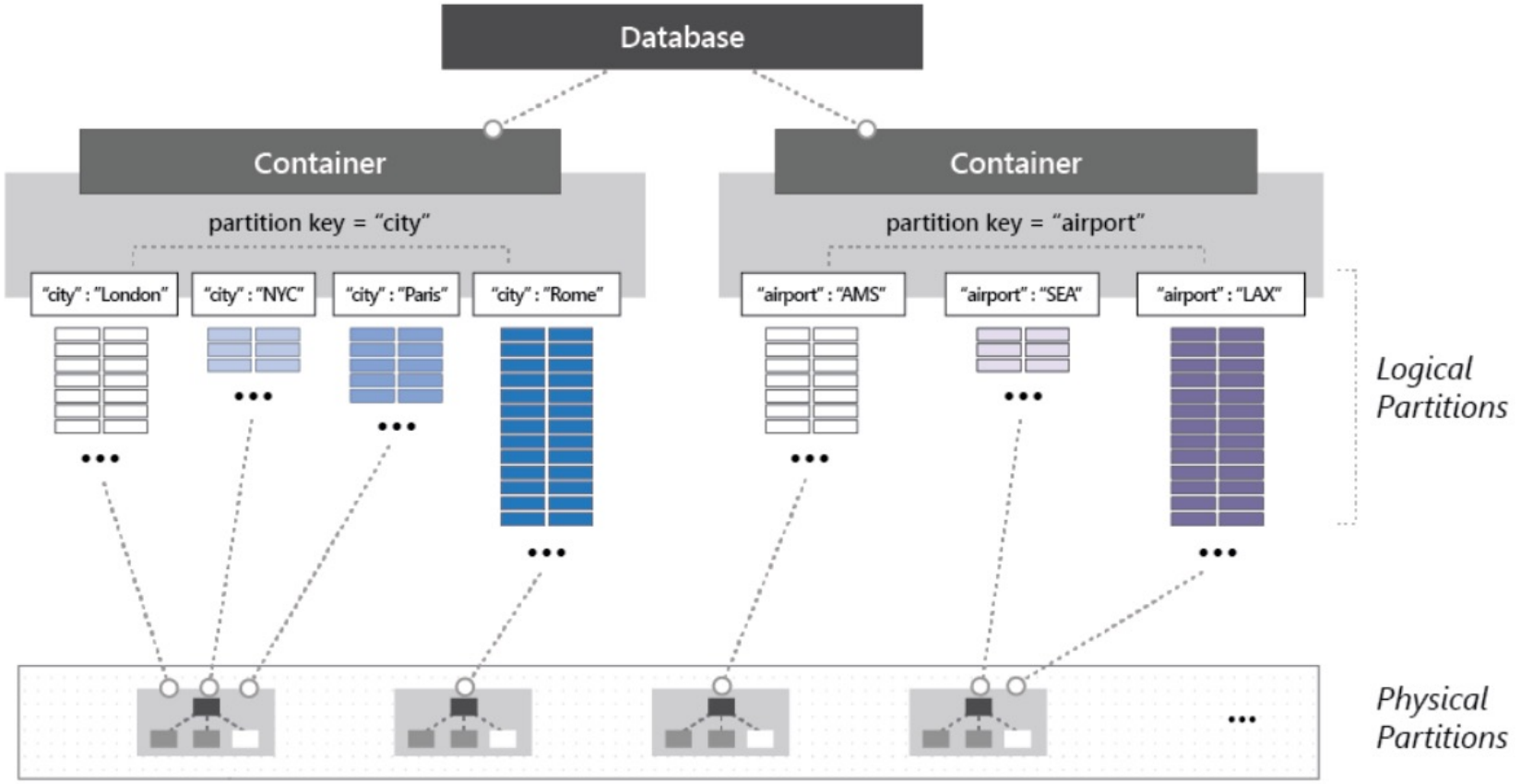
DATA ORGANIZATION UNDER THE H

A database may include a set of containers.
A container can be a collection, table, graph, etc.



SHARDING

A container is horizontally partitioned across multiple machines according to the partition key.

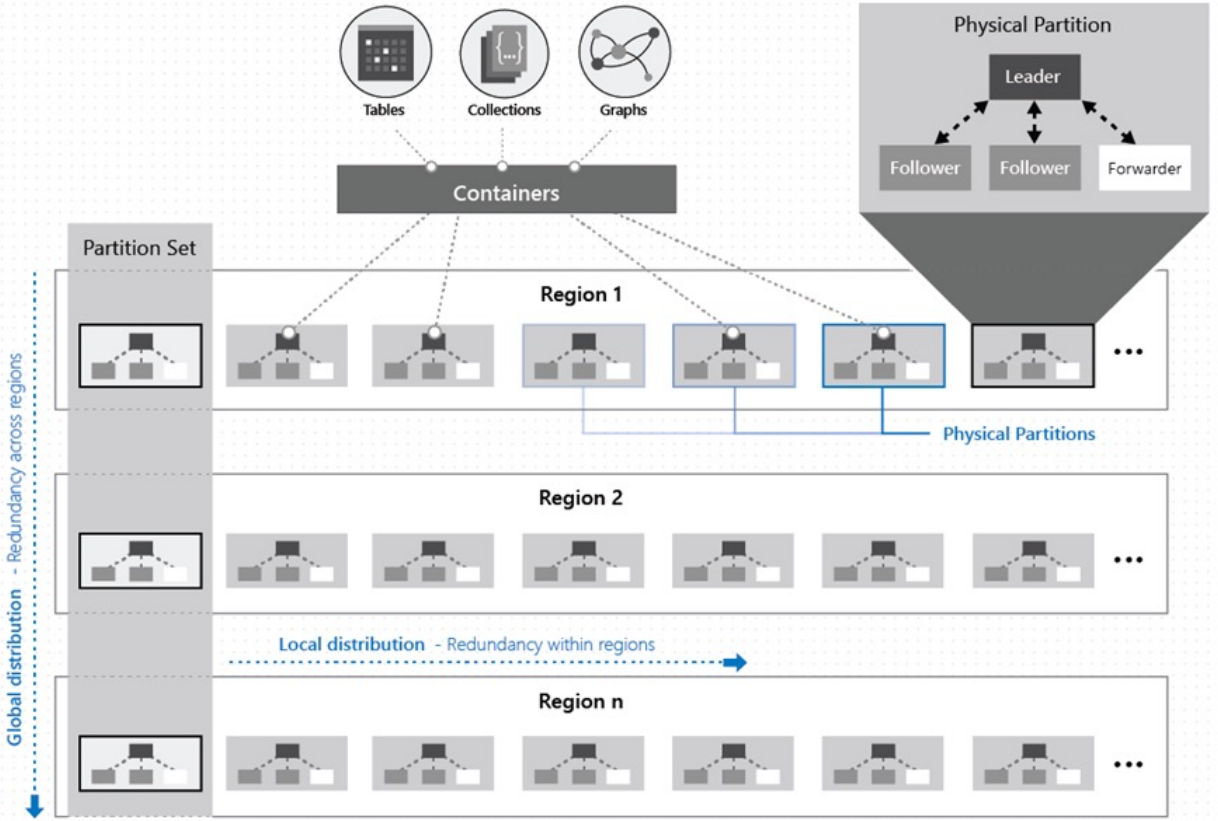


REPLICATION UNDER THE HOODS

Data is horizontally partitioned.

Within each region, every partition is replicated in a replica-set. All writes are replicated and durably committed by a majority of replicas.

Each partition is replicated across regions. Each region contains all data partitions of a Cosmos container.



For an account distributed across N regions, there will be at least N x 4 copies of the data.

CHANGE FEED

Cosmos DB allows to listen for changes in an Azure Cosmos container.

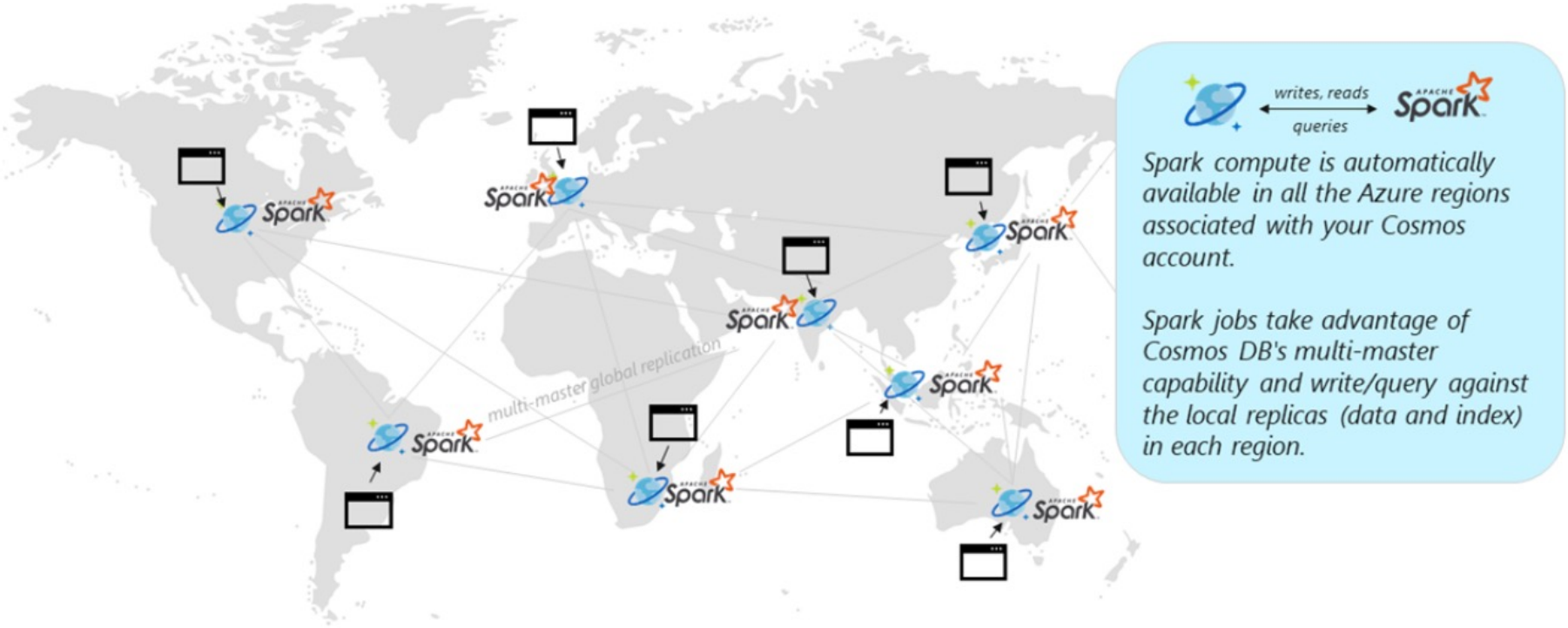
Change feed outputs the sorted list of documents that were changed in the order in which they were modified.

The changes are persisted, can be processed asynchronously and incrementally, and the output can be distributed across one or more consumers for parallel processing.

What can this be used for?

GLOBALLY DISTRIBUTED OPERATIONAL ANALYTICS

Azure supports running computations against the local replica in each region.



Cosmos account distributed across 8 Azure regions with Apache Spark jobs running against local replicas

PRICING

The cost of the throughput provisioned and the storage consumed on an hourly basis.

- Not cheap if you are not using it.

The request unit is based on the number and type of operations and the data transferred.

OTHER DATABASE SERVICES

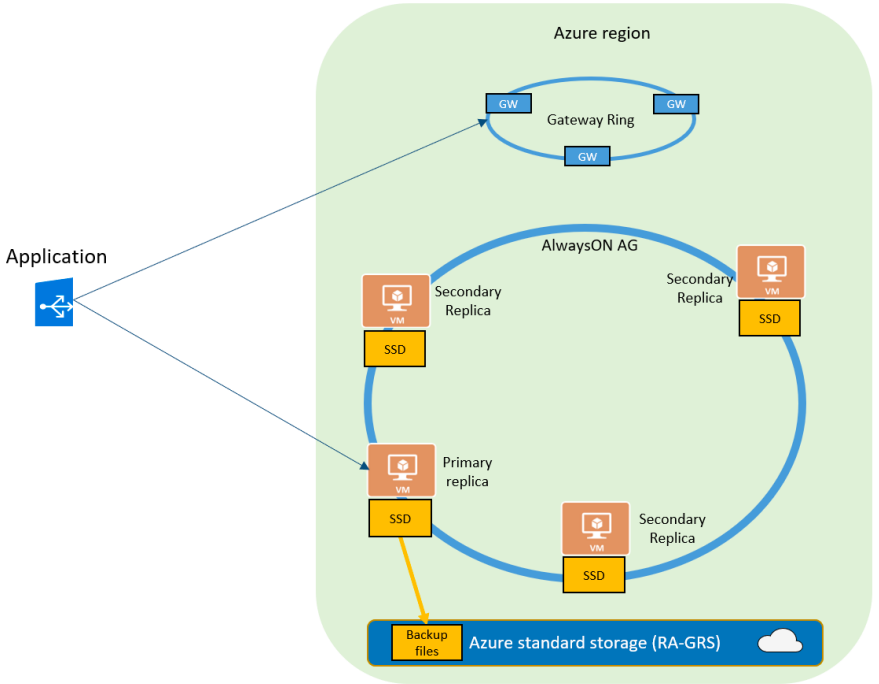
Cloud platforms also provide other database services – e.g.:

- Azure SQL database

SQL interface

Primary/backup replication

Sharding with inter-shard transactions



TO KNOW MORE

G. DeCandia, et. Al.. Dynamo: amazon's highly available key-value store. In SOSP'07.

https://azure.microsoft.com/en-us/blog/a-technical-overview-of-azure-cosmos-db/

https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels

https://docs.microsoft.com/en-us/azure/cosmos-db/conflict-resolution-policies

ACKNOWLEDGMENTS

Some text and images from Microsoft Azure online documentation.