E-book Series

Developer's Guide to Getting Started with Azure Cosmos DB

Azure Cosmos DB, the globally distributed database service from Microsoft

Who should read this?

This eBook was written for developers who are thinking of building new cloud apps or moving existing NoSQL apps to the cloud. It provides a brief primer on NoSQL, followed by an overview of Azure Cosmos DB and the value it brings to developers building apps for NoSQL workloads. We also provide an introduction to the core concepts you'll need to understand how best to put Azure Cosmos DB to use, followed by some resources to help you get started.

Why Azure Cosmos DB?

Azure Cosmos DB, the globally distributed database service from Microsoft, is unique in many ways. It is ideal for distributed apps that require extremely low latency at a global scale and enables you to avoid the all-or-nothing tradeoffs you face with most other NoSQL (nonrelational) databases by providing:

- native support for all major NoSQL data models—including key-value, document, graph, and columnar
- turnkey global distribution,
- multi-master support,
- elastic scaling of throughput and storage
- five well-defined consistency levels,
- data indexing as data is ingested, without requiring you to deal with schema or index management

it does all this with guaranteed high availability and low latency, all backed by industry-leading SLAs.

Contents

NoSQL: A quick primer 1
- NoSQL defined 1
- When to consider NoSQL.................................. 2
- How to choose a NoSQL database 2

Azure Cosmos DB: A globally distributed, multi-model database..................................... 5
- Key features and capabilities 6
- Common use cases 8

Core concepts and considerations 10
- Azure Cosmos DB accounts 10
- Resource model: Databases, containers, and items.. 11
- Partitioning and horizontal scalability 12
- Choosing a good partition key 13
- Request units and provisioned throughout 14
- Global distribution 18
- Consistency 21
- Automatic indexing.. 24

Architectural considerations 27
- Change feed.. 27
- Building serverless apps with Azure Cosmos DB and Azure Functions.................................. 28
- Server-side programming............................... 29
- Apache Spark to Azure Cosmos DB Connector 30
- Built-in operational analytics with Apache Spark (in preview) 30

Operational considerations 31
- Cost optimization with Azure Cosmos DB 31
- Security 31
- Online backup and restore 31
- Compliance 32

Building an app with Azure Cosmos DB 33
- Choosing the right API 33
- One database, multiple APIs 33
- Choosing an API.. 34

Getting Started with the SQL API..................35
- Quickstarts.. 35
- Tutorials 35
- How-to guides 36
- Additional resources.. 36

Getting Started with the Cassandra API 37
- Quickstarts.. 37
- Tutorials 37
- Cassandra and Spark.. 38
- How-to guides 38

Getting Started with the Azure Cosmos DB for MongoDB API 39
- Quickstarts.. 39
- Tutorials 40
- How-to guides 40

Getting Started with the Gremlin API...........42
- Quickstarts.. 42
- Tutorials 43
- How-to guides 43

Getting Started with the Table API 44
- Quickstarts.. 44
- Tutorials 44
- How-to guides 45

Conclusion 46

© 2019 Microsoft Corporation. All rights reserved. This document is provided "as is." Information and views expressed in this document, including URL and other internet website references, may change without notice. You bear the risk of using it. This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

NoSQL: A quick primer

If you work with databases, you've probably heard of NoSQL. Even if you haven't, odds are that you depend on NoSQL databases more than you know—if not as a developer, as an end user. It's becoming more and more popular with today's largest companies for its flexibility and scalability, in areas ranging from gaming and e-commerce to big data and real-time web apps. The use cases for NoSQL are continuing to grow, and, with the availability of NoSQL database services in the cloud, the benefits that it provides are within the reach of all.

NoSQL databases have been around since the 1960s, under various names. However, their popularity began to surge—and the NoSQL label was attached—much more recently, as leading technology companies began adopting NoSQL databases for their ability to handle petabytes of rapidly changing, unstructured data. But what exactly *is* a NoSQL database and, more importantly, what can it do for you as a developer?

NoSQL defined

NoSQL is the name for a category of databases that are nonrelational in nature, meaning that data storage and retrieval aren't handled using a predefined schema, with structured rows and columns, as with a relational database. Instead, NoSQL databases don't require a predefined schema and employ data models that make them highly effective at handling unstructured, unpredictable data—often with blazing-fast query speeds. By design, most NoSQL databases also support horizontal scalability.

Common NoSQL data models

The most common types of NoSQL data models include:

- **Key-value type,** which pairs keys and values using a hash table—in a manner similar to how a file path points to a file containing some data. The key is used to reference the value, which can include any arbitrary value—for example, an integer, string, a JSON structure (aka a document), a JPEG, an array, and so on.
- **Document** databases extend the concept of the key-value database by organizing entire documents into groups often called *collections*. A key can be any attribute within the document**,** within which data is encoded using a standardized format, such as XML or JSON. (In general, key-value stores don't support nested key-value pairs, whereas document databases do. What's more, because document databases store their data in a format that the database can understand, they allow queries on any attribute within a document.)
- **Columnar, or wide-column** databases, which generally store the values of one or more columns together in a storage block. Unlike relational databases, a columnar database can efficiently store and query across rows that contain sparsely filled columns.
- **Graph,** which uses a data model based on nodes, edges, and properties to represent interconnected data—such as relationships between people in a social network.

It's worth noting that most NoSQL databases can also handle highly structured data—they just aren't limited to it, nor do you need to define a database schema ahead of time. Similarly, if you want to add new data types to a NoSQL database, unlike with a relational database, you won't need to stop what you're

doing, add new columns, and then move your data to the new schema. This can be a big advantage when it comes to agile development and more frequent software release cycles.

Horizontal scalability

Another factor that contributes to the rapid adoption of NoSQL databases is that they're designed to scale out, or scale horizontally, which makes them capable of handling a virtually unlimited amount of data. That's not to say that you can't scale out a relational database, but it can get tricky. Many NoSQL databases, in comparison, have the inherent capabilities that allow them to scale out automatically and distribute their data over an arbitrary number of servers.

Replication

Most NoSQL databases are distributed and support some form of automatic replication, which can help maintain service availability in the event of a planned or unplanned outage. Replication also lets you distribute copies of your data across multiple geographies. For geographically distributed apps, it implies that someone using an app in one part of the world can read from a local replica and rather than waiting for data to be retrieved from the other side of the globe.

When to consider NoSQL

At this point you might be asking, "So when should I use a NoSQL database?" To answer this, it's worth starting with an acknowledgment that "NoSQL" can also mean "Not only SQL." As we've stated, NoSQL databases can also handle structured data—and can often be accessed using a structured query language like SQL. Although at first that might seem to muddy the picture when it comes to the SQL or NoSQL question, it really doesn't—it just shifts the focus to what your app needs to do, and what's required of your database to support that.

If you need to handle unstructured data at any scale, NoSQL might be a good place to start. Now consider the other characteristics of many NoSQL databases, such as low latency, horizontal scalability, and automatic replication. Clearly, these characteristics lend themselves well to a distributed app that requires fast performance across multiple geographic regions—achieved by using the enabling characteristics of NoSQL to put a copy of your data in each geography where your users reside. Similarly, the low latency of NoSQL makes it a strong candidate for delivering real-time customer experiences—like you might need for e-commerce or gaming. NoSQL is also proving popular in other scenarios, such as building serverless apps and implementing big data/analytics over operational data/transactional apps.

The takeaway here is that, at the end of the day, the decision on when to use NoSQL is about more than just whether your data is structured or not—it's about what your app needs to do, and how easily and flexibly you can achieve that.

How to choose a NoSQL database

Flexibility in handling unstructured data, inherent horizontal scalability, and built-in replication are all reasons why NoSQL databases are becoming more and more popular. And with so many of them to choose from, developers can usually find one that's well suited to their data. Such specialized, purpose-built NoSQL databases can also serve queries with blazing speed in many cases, which is critical in delivering real-time user experiences at scale—gaming and e-commerce are two good examples. However, that's not to say that there aren't some potential tradeoffs and other important

considerations associated with choosing a NoSQL database.

Programming models and APIs

If you've worked with relational databases, you're probably aware that they're not always a good match for the data structures you use when programming. Many NoSQL databases, however, are aggregate oriented, with an aggregate defined as a collection of data that you interact with as a unit—making them a much more natural fit for modern object-oriented programming languages.

As such, when it comes to choosing a NoSQL database, you'll probably want to start by choosing a data model—and then evaluate the NoSQL databases that support it, along with the programming languages and SDKs that each database supports. Does the database lock you into a given SDK and language, or will you have a choice in the matter? And does the SDK have what you need to get the most out of your distributed database—such as transparent multihoming APIs to ensure that your app can properly operate in case of a planned or unplanned failover?

Consistency vs. latency

Because a replicated NoSQL database is, in effect, a distributed system, you'll need to be aware of the CAP theorem. Also called Brewer's theorem, it states that it's impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees:

- **Consistency**—Ensuring that every request receives the most recent data
- **Availability**—Ensuring that every request receives a response
- **Partition tolerance**—Ensuring that the system continues to operate in the event of a failure between network nodes

The impossibility result of the CAP theorem proves that it's impossible for such a system to both remain highly available and deliver linearizable consistency in the event of a network failure (in which replicas are unable to talk to each other). Similarly, the CAP theorem shows that, in the absence of a network failure, you *can* achieve both availability and consistency. However, even in the absence of a network failure, you still need to consider tradeoffs between consistency and latency—formally codified in PACELC theorem—due to the fact that data packets being sent over a network wire are unable to travel faster than the speed of light.

Some NoSQL databases don't guarantee consistency. Most of them, however, let you choose from either end of the spectrum: strong consistency (you'll get the latest data, but you might need to wait) or eventual consistency (you'll get a fast response, but the data might be stale). Some NoSQL databases support other consistency levels, which typically fall in between those extremes. The key takeaway here is that, all other things considered equal, the more flexibility and control you have in terms of consistency levels—and thus the tradeoffs between consistency and latency—the better off you'll be.

On-premises vs. cloud—and which cloud?

NoSQL databases have been around for years, so you can find many that were designed to run on-premises. However, it's worth noting that NoSQL databases really started becoming popular with the advent of the cloud—and for good reason: their distributed nature and horizontal scalability make them an ideal fit. In fact, odds are that, regardless of the data model you choose, you'll find several cloud options. But as you're probably aware, all clouds are not created equal. So how do you choose?

In approaching this decision, in addition to programming languages/APIs and consistency/latency tradeoffs, you might want to consider the following:

- **Supported data models.** Does the cloud provider support all the data models that I might want to use? And if so, will I need to juggle a bunch of different database services?
- **Deployment and operations.** How easily can I deploy my database, and then replicate it to other regions if needed? How tedious are the setup and maintenance requirements? Do I get a fully managed service, or will I need to worry about patching and planned downtime?
- **Geographic presence.** Where are the cloud provider's datacenters? Can put my data where I want it? How will I handle important regulatory and data sovereignty issues, such as the European Union's new General Data Protection Regulation?
- **Ease of replication.** What's the process for replicating my database to a different geographic region? How complex is the process, and how long will it take?
- **Scalability.** How will I ensure the database resources required to ensure adequate performance—and scale for growth? Will I need to pre-provision and pay for resources that I might never use, or can I scale up and down on demand to handle unpredictable workloads?
- **High availability.** What will happen in the event of an unexpected failure? Is high availability built into the service, or will it be an added complication that I'll need to worry about?
- **Service levels.** Does the cloud service guarantee a certain level of availability? Does it have any latency guarantees? And if so, are they "empty promises" or are they financially backed?
- **Ecosystem.** How tightly integrated is the database with the rest of the cloud platform? Does it provide all the services I need, and can they be quickly stitched together to build a complete solution?

Finally, in selecting a NoSQL database service, it's worth taking a step back and examining its cloud platform as a whole. Rarely does any database exist in isolation, so you'll want to make sure the service you choose—and the platform upon which it resides—can provide everything that you'll need to put your NoSQL database to use. The specific services you'll need will depend on your app, such as the ability to integrate your NoSQL database with other app components via serverless functions. Other cloud services that you might need are more scenario specific, such as those for ingesting massive volumes of IoT data, implementing real-time streaming analytics, or building AI into your apps. And don't forget about ease of integration, such as triggering a serverless function when your NoSQL data changes. After all, even if a cloud platform provides all the services you need, you don't want to tie them together with paper clips and glue.

Azure Cosmos DB: A globally distributed, multi-model database

Azure Cosmos DB, the globally distributed database service from Microsoft, is a lot more than just another NoSQL database. it provides native support for all major NoSQL data models - key-value type, document, graph, and columnar—exposed through multiple APIs so you can use familiar tools and frameworks. Azure Cosmos DB also delivers turnkey global distribution, multi-master support, and elastic scaling of throughput and storage, making it ideal for apps that require extremely low latency, anywhere in the world.

With Azure Cosmos DB, you get things that you can't find anywhere else. It's the only database service that offers five well-defined consistency levels, enabling you to avoid the all-or-nothing tradeoffs you face with most NoSQL databases. It even indexes your data for you as it's ingested, without requiring you to deal with schema or index management. And it delivers guaranteed high availability and low latency, all backed by industry-leading service-level agreements (SLAs).

Best of all, because Azure Cosmos DB is a fully managed Microsoft Azure service, you won't need to manage virtual machines, deploy and configure software, or deal with upgrades. Every database is automatically backed up, protected against regional failures, and encrypted, so you won't have to worry about those things either—leaving you with even more time to focus on your app.

Check out our technical training series

This seven-part webinar series covers the following topics:

- Technical overview of Azure Cosmos DB
- Build real-time personalized experiences with AI and serverless technology
- Using the Gremlin and Table APIs with Azure Cosmos DB
- Build or migrate your Mongo DB app to Azure Cosmos DB
- Understanding operations of Azure Cosmos DB
- Build serverless apps with Azure Cosmos DB and Azure Functions
- Apply real-time analytics with Azure Cosmos DB and Spark

A brief history of Azure Cosmos DB

As a cloud service, Azure Cosmos DB is built from the ground up for multitenancy, elastic scalability, high availability, and global distribution—with low latencies and intuitive, predictable consistency levels. The work began in 2010, when developers at Microsoft set out to build a database that could meet those fundamental requirements for internal global apps. The result was a new fully managed nonrelational database service called Azure DocumentDB.

Seven years later, we announced Azure Cosmos DB, the first globally distributed, multi-model database service for building planet-scale apps. Since then, we've added support for new APIs, a native Apache Spark connector, the Azure Cosmos DB Change Feed Processor Library (which provides a sorted list of documents in the order in which they were modified), support for Azure Cosmos DB in

the Azure Storage Explorer, and a number of features for monitoring and troubleshooting.

In January 2018, Info World's 2018 Technology of the Year awards recognized Azure Cosmos DB, zeroing in on its "innovative approach to the complexities of building and managing distributed systems."

So just how did we achieve this? By design, Azure Cosmos DB does three things very well:

- **Partitioning,** which is what enables elastic scale out of storage and throughput.
- **Replication,** which enables turnkey global distribution—augmented with a set of well-defined consistency levels to let you tune consistency versus performance.
- **Resource governance,** through which Azure Cosmos DB can offer comprehensive SLAs encompassing the four dimensions of global distribution that customers care about the most: throughput, latency at the ninety-ninth percentile, availability, and consistency.

Key features and capabilities

To understand how you can use Azure Cosmos DB to build infinitely scalable, highly responsive global apps, it's worth looking at its key capabilities in more detail. Later in this e-book, we'll take a deeper dive into many of these same concepts.

Multiple data models. Azure Cosmos DB is the only fully managed service that natively supports document, graph, key-value, and columnar NoSQL data models—all in one place. These data models are supported through the following APIs, with SDKs available in multiple languages:

- SQL API: An API for accessing the core schema-less JSON document-oriented database engine with rich SQL querying capabilities.
- Azure Cosmos DB API for MongoDB: An API for accessing the document- oriented massively scalable MongoDB-as-a-service that you can use to easily move existing MongoDB apps to the cloud. The MongoDB API enables connectivity between Azure Cosmos DB and existing MongoDB libraries, drivers, tools, and apps.
- Cassandra API: An API for accessing the column based globally distributed Cassandra-as-a-service, which makes it easy to move existing Apache Cassandra apps to the cloud. The Cassandra API enables connectivity between Azure Cosmos DB and existing Cassandra libraries, drivers, tools, and apps.
- Gremlin (graph) API: An API to the fully managed, horizontally scalable database service that supports Open Graph APIs (based on the Apache TinkerPop specification).
- Azure Table API: An API built to provide automatic indexing, guaranteed low latency, global distribution, and other features of Azure Cosmos DB to existing Azure Table storage apps with very minimal effort.

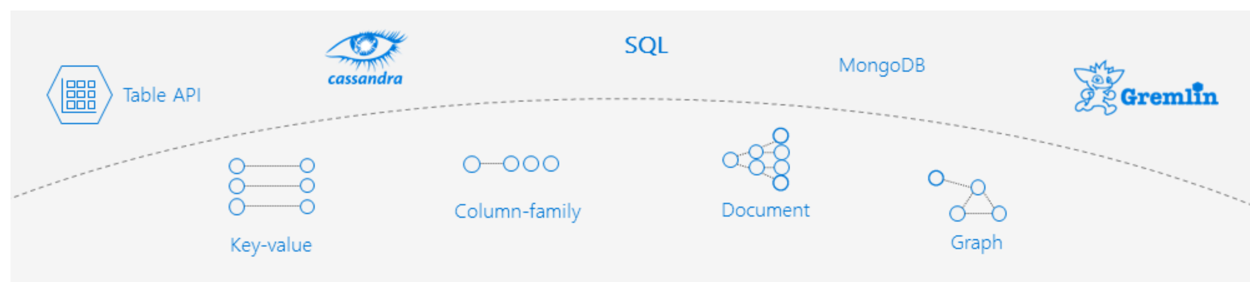


Figure 1. Azure Cosmos DB natively supports document, graph, key value, and columnar data models.

Turnkey global distribution. Azure Cosmos DB is the only database service that delivers turnkey global distribution. It lets you distribute your data to any number of Azure regions with just a few mouse clicks, keeping your data close to your users to maximize app performance. With the Azure Cosmos DB multihoming APIs, your app always knows where the nearest copy of your data resides, without any configuration changes, even as you add and remove regions.

Multi-master support. With multi-master support (multi-region writes), you can write data to any region associated with your Azure Cosmos DB account and have those updates propagate asynchronously, enabling you to seamlessly scale both write and read throughput anywhere around the world. You'll get single-digit millisecond write latencies at the ninety-ninth percentile, 99.999 percent write (and read) availability, and comprehensive and flexible built-in conflict resolution. Multi-master support is crucial for building globally distributed apps and significantly simplifies their development.

Limitless, elastic scale out of storage and throughput. With Azure Cosmos DB, you pay only for the storage and throughput that you need—and can independently and elastically scale storage and throughput at any time, across the globe.

Guaranteed low latency. With its latch-free, write-optimized database engine, Azure Cosmos DB delivers guaranteed low latency. For a typical 1-KB item, reads are guaranteed to be under 10 milliseconds at the ninety-ninth percentile; indexed writes are guaranteed to be under 10 milliseconds at the ninety-ninth percentile, within the same Azure region. Median latencies are even lower, at under 5 milliseconds.

Five well-defined consistency options. Azure Cosmos DB is the only database service that offers five well-defined, practical, and intuitive consistency levels—ranging from strong to eventual. In between those two extremes, you get three intermediate consistency levels to choose from (bounded staleness, consistent-prefix, and session), enabling you to fine-tune the tradeoffs between consistency and latency for your app.

No schema or index management. Azure Cosmos DB lets you rapidly iterate without worrying about schemas or indexes. The Azure Cosmos DB database engine is schema agnostic, and Azure Cosmos DB is the only database service that automatically indexes all the data it ingests, resulting in blazing-fast queries. It works across all supported data models, without the need for schemas or secondary indexes.

Global presence. As a foundational Azure service, Azure Cosmos DB is available in all regions where Azure is available— currently 54 regions worldwide.

Industry-leading security and compliance. When you choose Azure Cosmos DB, you run on Microsoft Azure—the world's most trusted cloud, with more compliance offerings than any other cloud provider. Data within Azure Cosmos DB is always encrypted, both at rest and in motion, as are indexes, backups, and attachments. Encryption is enabled by default, in a manner that's transparent to your app and has no impact on performance, throughput, or availability.

"Always on" availability. Azure Cosmos DB provides a 99.99 percent availability SLA for all single-region accounts and a 99.999 percent read availability SLA for all multi-region accounts. Automatic failover helps protect against the unlikely event of a regional outage, with all SLAs maintained. You can prioritize failover order for mult-iregion accounts and can manually trigger failover to test the end-to-end availability of your app—with guaranteed zero data-loss.

Unmatched, enterprise-grade SLAs. With Azure Cosmos DB, you can rest assured that your apps are running on an enterprise-grade database service. In fact, Azure Cosmos DB is *the first and only database service* to offer industry-leading, financially-backed SLAs for 99.999 percent high availability, latency at the ninety-ninth percentile, guaranteed throughput, and consistency.

Figure 2. Azure Cosmos DB offers industry-leading, financially backed SLAs

Common use cases

Now that we've covered the key features and capabilities of Azure Cosmos DB, just how can you put them to use? As a fully managed, multi-model database service, Azure Cosmos DB is a good choice for a broad range of apps. It's especially well-suited for event-driven serverless apps that require low latency and that might need to scale rapidly and globally. Add in its support for multiple data models and APIs and five consistency levels, and you have a NoSQL-compatible database service capable of supporting most any scenario where a traditional relational database isn't a good fit.

That said, here are common scenarios where Microsoft customers are using Azure Cosmos DB:

- **Globally distributed apps.** Azure Cosmos DB lets you build modern apps at a global scale, ensuring uncompromised performance no matter where your users are. You can easily put copies of your data in regions across the world, knowing you'll get guaranteed low latencies and built-in failover to ensure high availability and disaster recovery.
- **Real-time customer experiences.** The guaranteed low latency provided by Azure Cosmos DB makes it ideal for delivering real-time customer experiences and other latency-sensitive apps. And when you use Azure Cosmos DB together with Azure Databricks for its advanced analytics and machine learning capabilities, you can build apps that provide personalization and real-time recommendations.
- **Internet of things (IoT).** Azure Cosmos DB lets you accommodate diverse and unpredictable IoT workloads—enabling you to scale instantly and elastically to handle sustained, write-heavy data ingestion, all with uncompromised query performance.
- **E-commerce.** Azure Cosmos DB supports flexible schemas and hierarchical data, making it well suited for storing product catalog data where different products have different attributes. This is one of the reasons why Azure Cosmos DB is used extensively in Microsoft's own e-commerce platforms.
- **Gaming.** Modern games rely on the cloud to deliver personalized content like in-

game stats, social media integration, and leaderboards. Through its low-latency reads and writes, Azure Cosmos DB can help deliver an engaging, uncompromised in-game experience across large and changing user bases. At the same time, its instant, elastic scalability enables it to easily support the traffic spikes that are likely to occur during new game launches, online tournaments, and feature updates.

- **Serverless apps.** Azure Cosmos DB integrates natively with Azure Functions, making it easy to build event-driven, serverless apps that let you seamlessly scale data ingestion, throughput, and data volumes. Your data will be made available immediately and indexed automatically, with stable ingestion rates and query performance. And with the change feed support in Azure Cosmos DB, you can easily use changes in your data to kick off other actions and/or synchronize multiple data models in your event-driven app.
- **Big data and analytics.** Azure Cosmos DB integrates effortlessly with Azure Databricks for advanced analytics via Apache Spark, enabling you to implement machine learning at scale across fast-changing, high-volume, globally distributed data. The Spark to Azure Cosmos DB connector lets Azure Cosmos DB act as an input source or output sink for Spark jobs and can even push down predicate filtering to indexes within Azure Cosmos DB to improve the efficiency of Spark jobs.
- **Migration of existing NoSQL workloads to the cloud.** Azure Cosmos DB makes it easy to migrate existing NoSQL workloads to the cloud—in many cases, with no more than a change to a connection string in your app. With the Azure Cosmos DB Mongo DB and Cassandra APIs, you can migrate on-premises MongoDB and Cassandra databases to Azure Cosmos DB, respectively, then continue to use your existing tools, drivers, libraries, and SDKs. You won't need to spend any more time managing an on-premises database and will benefit from all that Azure Cosmos DB brings to the table. The videos on the Azure Cosmos DB YouTube channel can help you get started.

On the following pages, we take a deeper look at these and other key capabilities of Azure Cosmos DB, including how they work and how to put them to use. We're confident that, by the time you finish reading, you'll be ready to choose an API and go hands-on with Azure Cosmos DB. Or, if you prefer to learn by doing, you can skip forward to Choosing a data model and API, get started with your chosen API, and refer to the Key Concepts section of this e-book on an as-needed basis.

If you'd prefer to watch a video, many of these same concepts are also covered in the first webinar in the Azure Cosmos DB Technical Training series.

Core concepts and considerations

Azure Cosmos DB accounts

To begin using Azure Cosmos DB, you'll need to create an Azure Cosmos DB account under your Azure subscription. If you don't have an Azure subscription, you can sign up for a free one. You can also try Azure Cosmos DB for free without an Azure subscription, without any charges or commitments.

An Azure Cosmos DB account is the fundamental unit of global distribution and high availability for Azure Cosmos DB. To globally distribute your data and throughput across multiple Azure regions, you can add Azure regions to your Azure Cosmos DB account at any time. When an Azure Cosmos DB account is associated with more than one region, you can configure the account to have one write region or multiple write regions (i.e., multi-master). Each Azure Cosmos DB account can also be configured with a default consistency level, which can be overridden on an as-needed basis.

Currently, you can create a maximum of 100 Azure Cosmos DB accounts under one Azure subscription. Each Azure Cosmos DB account is identified by a unique DNS name and supports a single Azure Cosmos DB API.

Manage an Azure Cosmos DB account provides detailed instructions on how to create an account, add or remove regions, configure multiple write regions, enable automatic failover, set failover priorities, and perform a manual failover. Most Azure Cosmos DB account management tasks can be performed by using the Azure portal, Azure CLI, or Azure PowerShell.

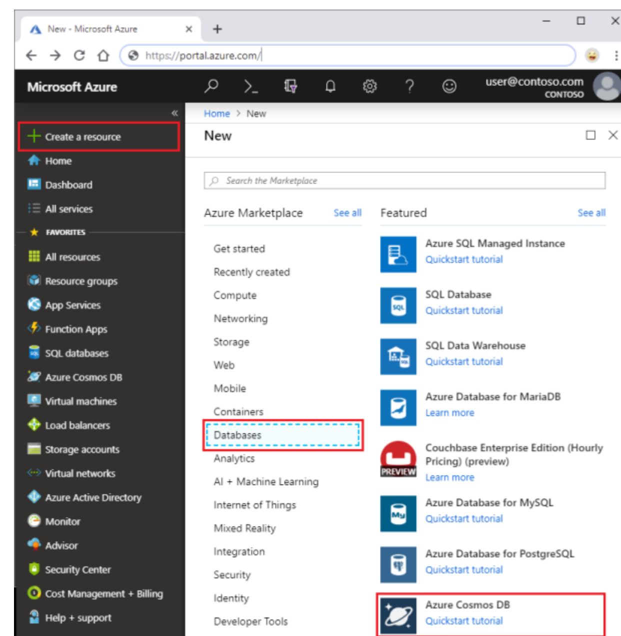


Figure 3. You can create an Azure Cosmos DB account in just a few minutes.

Resource model: Databases, containers, and items

Developers can start using Azure Cosmos DB by provisioning an Azure Cosmos DB account, which includes choosing an API. Entities under the Azure Cosmos DB account, called resources, are uniquely identified by a stable and logical URI and are represented as a JSON document. The overall resource model of an app using Azure Cosmos DB is a hierarchical overlay of the resources rooted under the Azure Cosmos DB account and can be navigated using hyperlinks.

An Azure Cosmos DB account manages one or more *databases*, which in turn manage *users*, *permissions*, and *containers*. Containers are schema agnostic, containing items (i.e., your data), stored procedures, triggers, and user-defined functions (UDFs). If your Azure Cosmos DB account is associated with multiple regions, then the containers within it will also contain merge procedures and conflicts.

Depending on the API selected when creating the Azure Cosmos DB account, container and item resources are projected in different ways. For example:

- With the SQL and MongoDB (document-oriented) APIs, containers are projected as *containers* and *collections* respectively and items are projected as *documents* for both.
- With the Gremlin API, containers are projected as *graphs* and items are projected as *nodes* and *edges*. (Using the multi-model capabilities of Azure Cosmos DB, you can also query the nodes and edges as documents using the SQL API.)
- With the Table (key-value) API, containers are projected as *tables* and items are projected as *rows*.
- With the Cassandra (columnar) API, containers are projected as *key-spaces* and items are projected as *rows*.

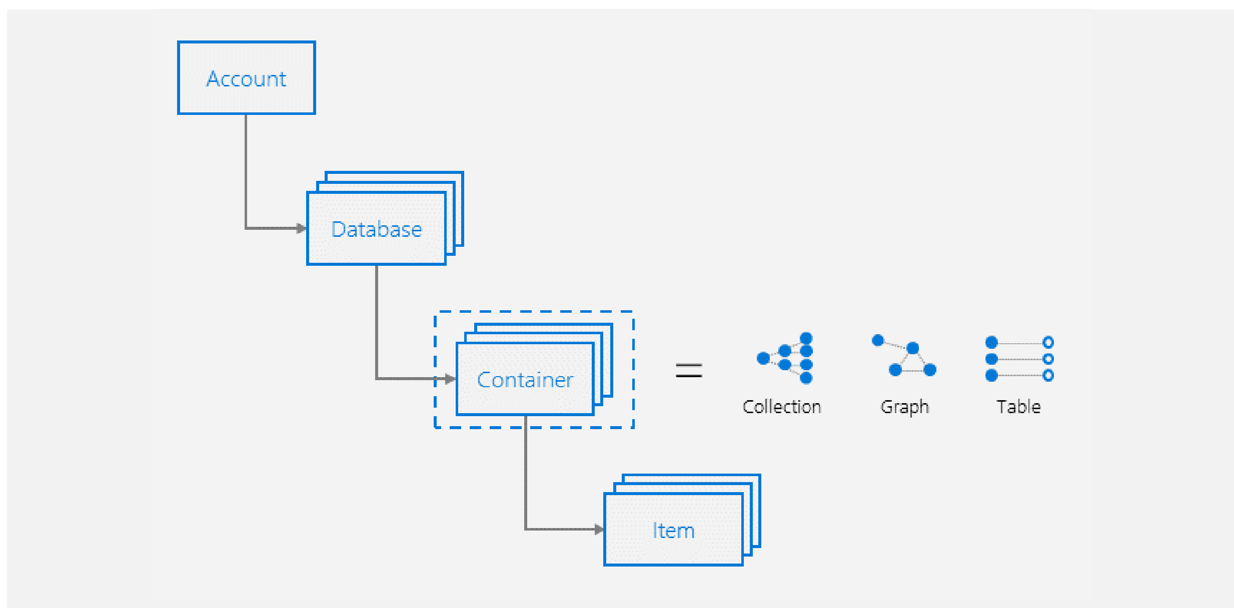


Figure 4. The Azure Cosmos DB resource model.

The Azure Cosmos DB documentation provides more information on databases, containers, and items.

Partitioning and horizontal scalability

Azure Cosmos DB containers can provide virtually unlimited storage and throughput. They can scale from a few requests per second into the millions, and from a few gigabytes of data to several petabytes. But just how does Azure Cosmos DB achieve this? If you've ever sharded a database, you probably have an idea—including how complex it can get. With Azure Cosmos DB, the service does 99 percent of the work for you, as long as you choose a good partition key (more on this later).

Here's how it works: In Azure Cosmos DB, there are two types of partitioning: physical and logical. Physical partitioning, which is how Azure Cosmos DB delivers virtually unlimited storage and throughput, is built into Azure Cosmos DB and is transparent to you as a developer. Containers, which are logical resources, can span one or more physical partition sets (composed of replicas or servers), each of which has a fixed amount of reserved, SSD-backed storage. The number of physical partition sets across which a container is distributed is determined internally by Azure Cosmos DB based on the storage size and the throughput you've provisioned for the container.

All physical partition management, including that required to support scaling, also is fully managed by Azure Cosmos DB; when a container meets the partitioning prerequisites, the act of partitioning is transparent to your app, shielding you from a great deal of complexity. The service handles the distribution of data across physical and logical partitions and the routing of query requests to the right partition—without compromising the availability, consistency, latency, or throughput of an Azure Cosmos DB container. Again, you don't need to worry about physical partitioning—it's handled at runtime automatically by the service.

You should, however, be aware of how logical partitioning works in Azure Cosmos DB, including the importance of choosing a good partition key at design time. Here's why: the data within a container is horizontally distributed and transparently managed through the use of *logical resource partitions,* which are also known as a customer-specified *partition keys,* each of which is limited to 10 GB. For example, in the following diagram, a single container has three physical partition sets, each of which stores the data for one or more partition keys (in this example, LAX, AMS, and MEL). Each of the LAX, AMS, and MEL partition keys can't grow beyond the maximum logical partition limit of 10 GB.

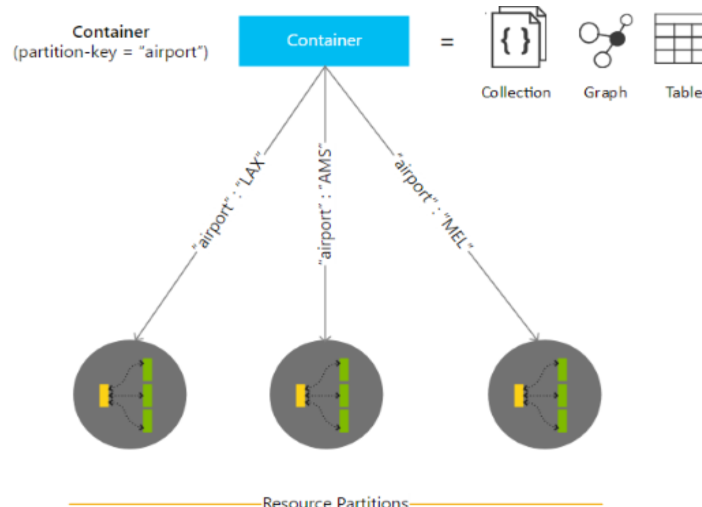


Figure 5. A single Azure Cosmos DB container distributed over three logical resource partitions.

Many partition keys can be co-located on a single physical partition set and are automatically redistributed by the service as needed to accommodate growth in traffic, storage, or both. Because of this, you don't need to worry about having too many partition key values. In fact, having more partition key values is generally preferred to fewer partition key values. This is because a single partition key value will never span multiple physical partition sets.

The Azure Cosmos DB documentation provides an overview of partitioning in Azure Cosmos DB, as well as additional detail on physical and logical partitions.

Choosing a good partition key

Choosing a good partition key is ***a critical design decision,*** as it's really the only factor that can limit horizontal scalability. To scale effectively with Azure Cosmos DB, you'll need to pick a good partition key when you create your container. You should choose a partition key such that:

- The storage distribution is even across all the keys.
- The volume distribution of requests at a given point in time is even across all the keys.
- Queries that are invoked with high concurrency can be efficiently routed by including the partition key in the filter predicate.

In general, a partition key with higher cardinality is preferred because it typically yields better distribution and scalability. The Azure Cosmos DB documentation provides additional guidance for choosing a partition key. The first webinar in our Azure Cosmos DB Technical Training series also covers how partitioning works and how to choose a good partition key.

In selecting a partition key, you may want to consider whether to use unique keys to add a layer of data integrity to your database. By creating a unique key policy when a container is created, you ensure the uniqueness of one or more values per partition key. When a container is created with a unique key policy, it prevents the creation of any new or updated items with values that duplicate values specified by the unique key constraint. For example, in building a social app, you could make the user's email address a unique key—thereby ensuring that each record has a unique email address and no new records can be created with duplicate email addresses.

You may also want to consider the use of synthetic partition keys. Here's why: As stated above, it's considered a best practice to have a partition key with many distinct values, as a means of ensuring that your data and workload is distributed evenly across the items associated with those partition key values. If such a property doesn't exist in your data, you can construct a synthetic partition key in several ways, such as by concatenating multiple properties of an item, appending a random suffix (like a random number) at the end of a partition key value, or by using a pre-calculated suffix based on something you want to query.

Request units and provisioned throughput

Request units per second (RU/s)—often represented in the plural form RUs—are the "throughput currency" of Azure Cosmos DB. To establish the throughput you'll need, you reserve a number of RUs, which are guaranteed to be available to your app on a per-second basis. As your app runs, each operation in Azure Cosmos DB (such as writing a document, performing a query, and updating a document) consumes CPU, memory, and IOPS—a blended measure of compute resources used, which is expressed in RUs.

The number of RUs for an operation is deterministic. Azure Cosmos DB supports various APIs that have different operations, ranging from simple reads and writes to complex graph queries. Because not all requests are equal, requests are assigned a normalized quantity of RUs, based on the amount of computation required to serve the request.

RUs let you scale your app's throughput with one simple dimension, which is much easier than separately managing CPU, memory, and IOPS. You can dynamically dial RUs up and down by using the Azure portal or programmatically, enabling you to avoid paying for spare capacity that you don't need. For example, if your database traffic is heavy from 9 AM to 5 PM, you can scale up your RUs for those hours and then scale back down for the remaining 16 hours when database traffic is light.

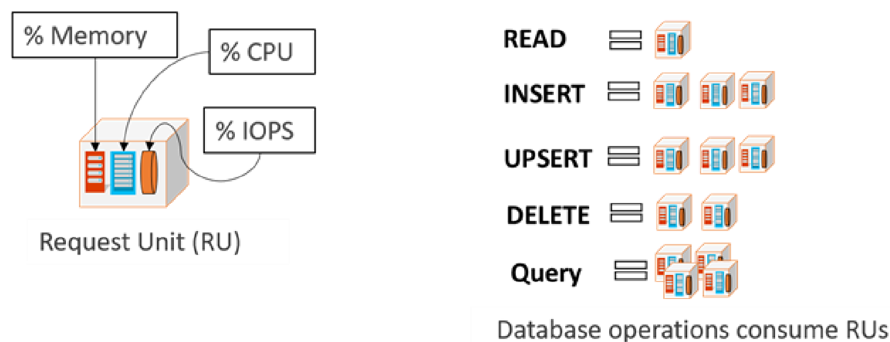


Figure 6. Request units are the normalized currency of throughput for various database operations.

The Azure Cosmos DB documentation provides [more information on RUs](), including how to specify RU capacity, variables to take into consideration when estimating the number of RUs to reserve, and what happens—and how to how to handle it—when your app exceeds reserved throughput.

Capacity planning made easy

For the developer, request units (RUs) immensely simplify capacity planning. Say you have an on-premises database, which you're running on a server that has a given amount of all three key system resources: CPU, memory, and I/O. Now say you want to scale that database from 1,000 requests per second to 10,000 requests per second. How much more RAM do you buy? How much CPU do you buy? Do you even know which is the bottleneck?

You might do some stress testing and find that RAM is indeed the bottleneck, but that doesn't necessarily tell you how much RAM translates to how many requests per second. Furthermore, as soon as you add some RAM, CPU might become the bottleneck. And as you add more processor cores, I/O might become the new bottleneck. Clearly, this approach gives you a very difficult set of dimensions to scale against—and that's for a single, monolithic server. Imagine doing this for a distributed database.

Azure Cosmos DB uses a machine-learning model to provide a predictable RU charge for each operation. So if you create a document today and it costs 5 RUs, then you can rest assured that the same request will cost you 5 RUs tomorrow and the day after—inclusive of all background processes. This lets you forecast required capacity with some basic "mental math," using one simple dimension.

For example, in the previous scenario, where you want to scale from 1,000 operations per second to 10,000, you'll need 10 times the number of RUs. So if it takes 5,000 RUs to support 1,000 writes per second, you can rest assured that you can support 10,000 writes per second with 50,000 RUs. It's really that simple. Just provision the RUs that you'll want, and Azure Cosmos DB will set aside the necessary system resources for you, abstracting all the complexity in terms of CPU, memory, and I/O.

Understanding throughput requirements

By understanding your app's throughput requirements and the factors that affect RU charges, you can run your app as cost-effectively as possible. In estimating the number of RUs to provision, it's important to consider the following variables:

- **Item size.** As size increases, the number of RUs consumed to read or write the data also increases.
- **Item property count.** Assuming default indexing of all properties, the RUs consumed to write a document, node, or entity increase as the property count increases.
- **Data consistency.** Data consistency levels like Strong or Bounded Staleness (discussed later under Consistency) consume more RUs than other consistency levels when reading items.
- **Indexed properties**. An index policy on each container determines which properties are indexed by default. You can reduce RU consumption for write operations by limiting the number of indexed properties or by enabling lazy indexing.
- **Query patterns.** The complexity of a query affects how many RUs are consumed for an operation. The number of query results, the number of predicates, the nature of the predicates, the number of user-defined functions, the size of the

source data, and projections all affect the cost of query operations.

- **Script usage**. As with queries, stored procedures and triggers consume RUs based on the complexity of the operations being performed. As you develop your app, inspect the request charge header to better understand how each operation consumes RU capacity.

One method for estimating the amount of reserved throughput required by your app is to record the RU charge associated with running typical operations against a representative item used by your app. Then, estimate the number of operations you anticipate performing each second. Be sure to also measure and include typical queries and Azure Cosmos DB script usage. For example, these are the steps you might take:

1. Record the RU charge for creating (inserting) a typical item.
2. Record the RU charge for reading a typical item.
3. Record the RU charge for updating a typical item.
4. Record the RU charge for typical, common item queries.
5. Record the RU charge for any custom scripts (stored procedures, triggers, or user-defined functions) that the app uses.
6. Calculate the required RUs given the estimated number of each of the above operations you anticipate runninhg in each second.

There's an Azure Cosmos DB capacity planner to help you estimate your throughout needs, and an article on finding the RU consumption for any operation executed against a container in Azure Cosmos DB.

Provisioning throughput on containers and databases

With Azure Cosmos DB, you can provision throughput at two granularities: at the container level, and at the database level. At both levels, provisioned throughput can be changed at any time.

When you provision throughput at the container level, the throughput you provision is reserved exclusively for that container and is uniformly distributed across all of its logical partitions. If the workload running on a logical partition consumes more than its share of overall provisioned throughput, your database operations against that logical partition are rate-limited, which may result in a "429" throttling exception response that includes the amount of time (in milliseconds) that the app must wait before retrying the request.

When this happens, for workloads that aren't sensitive to latency, in many cases, you can simply let the application handle it as part of normal operations. The native SDKs (.NET/.NET Core, Java, Node.js and Python) implicitly catch this response, respect the server-specified retry-after header, and retry the request. Unless your account is being accessed concurrently by multiple clients, the next retry will succeed. If retries become excessive, you can operationally scale the provisioned RUs for the container to support the capacity requirements of your app.

Provisioning throughput at the container level is the most frequently used option for large-scale production applications that require guaranteed performance.

When you provision throughput at the database level, the throughput is shared across all the containers in the database. Although this guarantees you'll receive the provisioned throughput for that database all the time, because all containers within the database share the provisioned throughput, it doesn't provide any predictable throughput

guarantees for any particular container. Instead, the portion of provisioned throughput that a specific container may receive is dependent on the number of containers, choice of partition keys for those containers, and the workload distribution across the various logical partitions of those containers.

Provisioning throughput at the database level is a good starting point for development efforts and small applications. Should the need arise, you can always adopt a per-container provisioning model later. You can even combine the two throughput provisioning models, in which case the throughput provisioned at the database level is shared among all containers in the database *for which throughput is not explicitly provisioned at the container level*.

The Azure Cosmos DB documentation includes how-to guides for provisioning throughput at the container level and at the database level.

Global distribution

As illustrated in the following diagram, with Azure Cosmos DB, a customer's resources can be distributed along two dimensions. Within a given region, all resources are horizontally partitioned using resource partitions—called local distribution. If you've set up more than one region, each resource partition is also replicated across geographical regions—called global distribution.

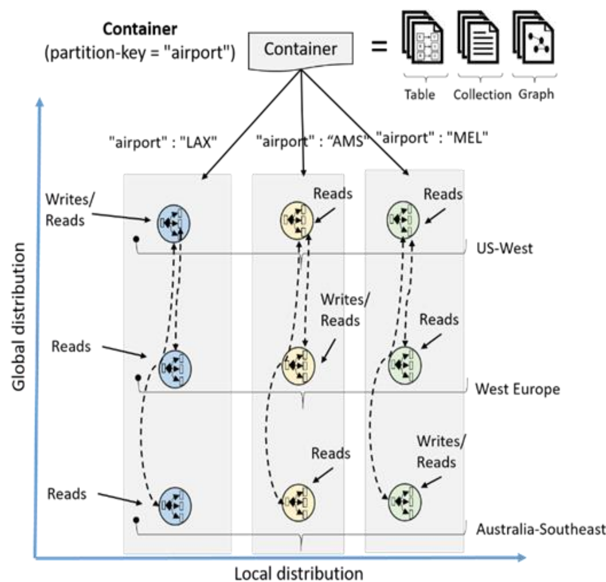


Figure 7. A container can be both locally and globally distributed.

Global distribution of resources in Azure Cosmos DB is turnkey. At any time, outside of geo-fencing restrictions (for example, China and Germany), with a few button clicks or a single API call, you can associate any number of geographical regions with your Azure Cosmos DB account. Regardless of the amount of data or number of regions, Azure Cosmos DB guarantees (at the ninety-ninth percentile) that each newly associated region will start processing client requests within 30 minutes for up to 100 TB of data, which is achieved by parallelizing the seeding and copying of data. You can also remove an existing region or take a region that was previously associated with your Azure Cosmos DB account "offline."

When you elastically scale throughput or storage for a globally distributed database, Azure Cosmos DB transparently performs the necessary partition management operations across all the regions, continuing to provide a single system image, independent of scale, distribution, or any failures.

Azure Cosmos DB supports both explicit and policy-driven failovers, allowing you to control the end-to-end system behavior in the event of failures. In the rare event of an Azure regional outage or datacenter outage, Azure Cosmos DB automatically triggers failovers of all Azure Cosmos DB accounts with a presence in the affected region. You can also manually trigger a failover, as may be required to validate the end-to-end availability of your app. Because both the safety and liveness properties of the failure detection and leader election are guaranteed, Azure Cosmos DB guarantees zero data-loss for a tenant-initiated, manual-failover operation. (Azure Cosmos DB guarantees an upper bound on data loss for a system-triggered, automatic failover due to a regional disaster.)

Upon regional failover, you won't need to redeploy your app. Azure Cosmos DB lets your app interact with resources using either logical (region-agnostic) or physical (region-specific) endpoints—the former ensuring that your app can transparently be multihomed in case of failover, and the latter providing fine-grained control for the app to redirect reads and writes to specific regions.

With a multi-region account, Azure Cosmos DB guarantees 99.999 percent availability, regardless of data volumes, specified throughput, the number of regions associated with your Azure Cosmos DB account, or the distance between the geographical regions associated with your database. The same holds true for SLAs around consistency, availability, and throughput.

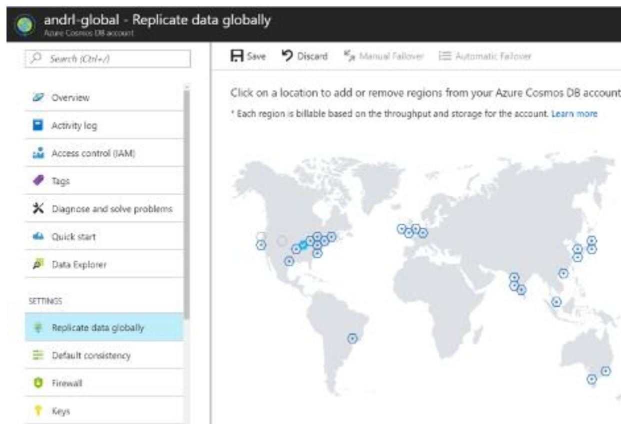


Figure 8. You can configure global distribution using the Azure portal, with just a few clicks.

Clearly, the global distribution enabled by Azure Cosmos DB can help ensure high availability. But there's a second very good reason to use it: app responsiveness. You can't break the speed of light, which means requesting data that's stored halfway across the globe is going to take a lot longer than data that resides significantly closer to you. Even under ideal network conditions, sending a packet halfway across the globe can take hundreds of milliseconds.

However, you can cheat the speed of light by using data locality, taking advantage of Azure Cosmos DB global replication to put copies of your data in strategic locations that are close to your users. Content delivery networks (CDNs) have employed this approach successfully for years when it comes to static content; now Azure Cosmos DB lets you do the same thing for dynamic content. Through such an approach, you can often achieve data retrieval times that are lower than 10 milliseconds, or an order of magnitude reduction. And if your app is making several round trips to the database, that can mean a big difference in the user experience.

The Azure Cosmos DB documentation provides more information on global distribution, including how to associate your Azure Cosmos DB account with any number of regions using the Azure portal or the Azure Cosmos DB resource provider's REST APIs.

Multi-master replication

Azure Cosmos DB supports multi-master replication, meaning that it supports multi-region writes. When you configure multi-region writes, you can write data to any region associated with your Azure Cosmos DB account and have those updates propagate asynchronously, enabling you to seamlessly scale both write and read throughput around the world—with single-digit millisecond write latencies at the ninety-ninth percentile and 99.999 percent write availability (compared to 99.99 percent write availability for single-region writes). To use the multi-master feature in your application, you'll need to enable multi-region writes and configure the multi-homing capability in Azure Cosmos DB.

When you use multi-region writes, you'll need to take into account update conflicts, which can occur when the same item is concurrently updated in multiple regions. Azure Cosmos DB provides a flexible means of dealing with such conflicts, allowing you to choose from two resolution policies:

- **Last write wins.** By default, this conflict resolution policy uses a system-defined timestamp property based on the time-synchronization clock protocol. If you're using the SQL API, you also can specify any other custom numerical property (e.g., your own notion of a timestamp) to be used for conflict resolution—sometimes referred to as the conflict resolution path.
- **Custom.** This resolution policy, which is available only for SQL API accounts, supports application-defined semantics for conflict reconciliation. When you set this policy, you'll also need to register a merge stored procedure, which is automatically invoked when conflicts are detected. If you don't register a merge procedure on the container or the merge procedure throws an exception at runtime (Azure Cosmos DB provides exactly once guarantee for the execution of a merge

procedure as part of the commitment protocol), the conflicts are written to the conflicts feed for manual resolution by your application.

The Azure Cosmos DB documentation provides more information on conflict types and resolution policies, as well as a how-to article on managing conflict resolution policies.

How provisioned throughout is distributed across multiple regions

Earlier in this e-book, we discussed how provisioned throughput is expressed in RU/s, or RUs, which measure the “cost” of both read and write operations against an Azure Cosmos DB container. If you provision 'R' RUs on a container (or database), Azure Cosmos DB ensures that 'R' RUs are available in each region associated with your Azure Cosmos DB account. Each time you add a new region to your account, Azure Cosmos DB automatically provisions 'R' RUs in the newly added region.

Assuming that a Cosmos container is configured with 'R' RUs and there are 'N' regions associated with the Cosmos account, then:

- If the Azure Cosmos DB account is configured with a single write region, the total RUs available globally on the container = R x N.
- If the Azure Cosmos DB account is configured with multiple write regions, the total RUs available globally on the container = R x (N+1). The additional R RUs (i.e., the “+1” part) are automatically provisioned to process cross-region update conflicts and anti-entropy traffic.

It’s also worth noting that your choice of consistency level (discussed next) also affects throughput. In general, you’ll get greater read throughout for the more relaxed consistency levels (e.g., session, consistent prefix and eventual consistency) compared to stronger consistency levels (e.g., bounded staleness or strong consistency).

Consistency

Azure Cosmos DB makes using geographic distribution to build low-latency global apps not only possible, but also easy. As a write-optimized database, it offers multiple, intuitive, tunable consistency levels to give you read predictability, let you make the right tradeoffs between consistency and latency, and help you correctly implement your geographically distributed app. But why is consistency so important, and why are the five consistency levels provided by Azure Cosmos DB better than what you can get with any other NoSQL database?

Most commercially available distributed database fall into two categories: either they don't offer well-defined, provable consistency choices, or they offer only the two extremes: strong concurrency, or eventual concurrency. Systems that fall into the first category burden developers with the minutiae of replication protocols and make difficult tradeoffs between consistency, availability, latency, and throughput. Systems in the second category force developers to choose between the two extremes, neither of which is optimally suited for many of the real-world scenarios that are driving the development of globally distributed apps.

For example, while the strong consistency level is the gold standard of programmability, it comes at the steep price of much higher latency (in steady state), reduced availability (in the face of failures), and lower read scalability. Conversely, with eventual concurrency, you'll get great performance, but a complete lack of predictability when it comes to whether you're getting the latest and greatest data. Despite an abundance of research and proposals, the distributed database community has not been able to commercialize consistency levels beyond strong and eventual consistency—that is, until now.

Azure Cosmos DB allows you to choose from five well-defined consistency levels, spanning the spectrum from strong to eventual with three intermediate levels: bounded-staleness, session, and consistent prefix. You can specify the default consistency level for an Azure Cosmos DB account, which will apply to all data within all partition sets across all regions. If you want, you can override the default consistency level on a per-request basis.

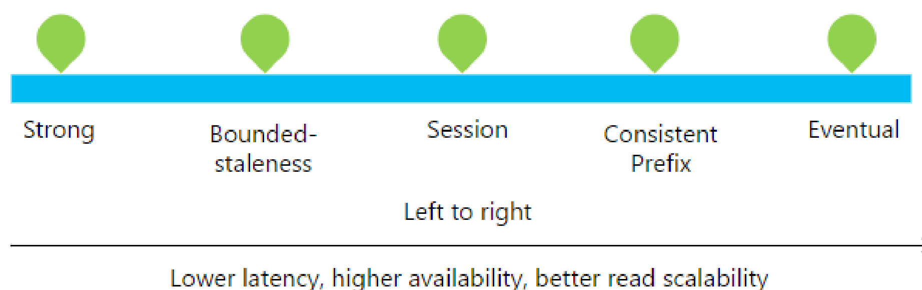


Figure 9. Azure Cosmos DB offers five well-defined consistency levels, enabling you to fine-tune the tradeoffs between consistency and latency for your app.

The following table captures the guarantees and characteristics associated with each consistency level.

Consistency level	Guarantees and characteristics
Strong	• Offers a linearizability guarantee, with reads guaranteed to return the most recent version of an item. • Guarantees that a write is only visible after it's committed durably by the majority quorum of replicas. A write is either synchronously committed durably by both the primary and the quorum of secondaries, or it's abandoned. A read is always acknowledged by the majority read quorum; a client can never see an uncommitted or partial write and is always guaranteed to read the latest acknowledged write. • The cost of a read operation (in terms of request units consumed) with strong consistency is higher than session and eventual, but the same as bounded staleness.
Bounded staleness	• Guarantees that the reads may lag behind writes by at most *K* versions or prefixes of an item or *t* time interval. (When choosing bounded staleness, the "staleness" can be configured in two ways: number of versions *K* of the item by which the reads lag behind the writes, and the time interval *t*.) • Offers total global order except within the "staleness window." The monotonic read guarantees exist within a region both inside and outside the "staleness window." • Provides a stronger consistency guarantee than session, consistent-prefix, or eventual consistency. For globally distributed apps, we recommend that you use bounded staleness for scenarios where you would like to have strong consistency but also want 99.999 percent availability and low latency. • The cost of a read operation (in terms of RUs consumed) with bounded staleness is higher than session and eventual consistency, but the same as strong consistency.
Session	• Unlike the global consistency offered by strong and bounded-staleness consistency levels, session consistency is scoped to a client session. • Ideal for all scenarios where a device or user session is involved because it guarantees monotonic reads, monotonic writes, and read your own writes (RYW) guarantees. • Provides predictable consistency for a session and maximum read throughput while offering the lowest latency writes and reads. • The cost of a read operation (in terms of RUs consumed) is less than strong and bounded staleness but more than eventual consistency.

Consistent prefix	• Guarantees that in the absence of any further writes, the replicas within the group eventually converge. • Guarantees that reads never see out-of-order writes. If writes were performed in the order *A, B, C,* then a client sees either *A; A, B;* or *A, B, C,* but never an order like *A, C* or *B, A, C.*
Eventual	• Guarantees that in the absence of any further writes, the replicas within the group eventually converge. • Is the weakest form of consistency, where a client might get values that are older than the ones it had seen before. • Provides the weakest read consistency but offers the lowest latency for both reads and writes. • The cost of a read operation (in terms of RUs consumed) is the lowest of all the Azure Cosmos DB consistency levels.

All consistency levels are supported by our consistency SLAs. To report any violations, we employ a linearizability checker, which continuously operates over our service telemetry. For bounded staleness, we monitor and report any violations to *K* and *t* bounds. For all four relaxed consistency levels, we track and report the probabilistic bounded-staleness (PBS) metric among other metrics.

The Azure Cosmos DB documentation provides more information on consistency levels, including how to configure the default consistency level for an Azure Cosmos DB account, guarantees associated with consistency levels, and consistency levels explained through an example based on baseball scores. The documentation also includes articles on:

- Choosing the right consistency level based on which API you're using, as well as practical considerations related to consistency guarantees.
- Mapping between Apache Cassandra or MongoDB and Azure Cosmos DB consistency levels (when using SQL API, Gremlin API, and Table API, the default consistency level configured on the Azure Cosmos DB account is used.)
- Consistency, availability, and performance tradeoffs—including those between consistency levels and latency, consistency levels and throughput, and consistency levels and data durability.

Choosing a consistency level

The following scenarios illustrate when each consistency level might be appropriate:

- **Strong consistency** ensures that you'll never see a stale read, making it a good fit for scenarios like transaction processing, such as updating the state of an order.
- **Bounded-staleness** gives you an SLA on "how eventual is eventual?" You can think of this as a window within which stale reads are possible, which you can configure in terms of time or number of operations. Outside of this window, strong consistency is guaranteed.
- **Session consistency** is the sweet spot for most apps; it provides a means of scoping strong consistency down to a single session, without paying the performance penalty associated with global strong consistency. Take the case of a user posting a comment on Facebook: when the page is refreshed, if the user doesn't see his or her post, that user might repeat the process—only to see multiple copies. With session concurrency, where reads follow your own writes within a session, you can avoid this.
- **Consistent prefix** is good when you can handle some latency, as long as you'll never see out-of-order updates. A group chat app is a good example. Say you have Alice and Bob organizing dinner, and saying "What time should we meet? How about 7:00? I'm busy then. How about 8:00? That's great—let's meet then." If Carol and Dan are also part of the group chat and see these messages out of order, they might not arrive at the proper time. With consistent prefix, you can ensure that the messages arrive in the correct order.
- **Eventual concurrency** is a good choice where low latency matters above all else. Again, let's use the example of a Facebook post. You want it to load quickly and aren't very concerned whether the number of "Likes" takes into account every Like by every user up to that moment in time, around the world.

About 73 percent of Azure Cosmos DB tenants use session consistency, while 20 percent prefer bounded staleness. Also, approximately 3 percent of customers initially experiment with various consistency levels before settling on a choice for their app, and only 2 percent of customers override consistency levels on a per-request basis.

For more information on consistency levels in Azure Cosmos DB, check out the interactive e-book.

Automatic indexing

Azure Cosmos DB provides automatic indexing, which you can tune and configure. It works across every data model, automatically indexing every property of every record by default. You won't need to define schemas and indexes up front or manage them over time, so you'll never have to do an alter table or create index operation. Automatic indexing is based on a latch-free data structure and is designed to run on the Azure Cosmos DB write-optimized database engine, enabling automatic indexing while sustaining high data-ingestion rates.

Automatic indexing is made possible through the use of an inverted index. Here's how it works, using the two JSON documents in Figures 10 and 11 as an example. Note that the two records have different schemas. Document 1 has a set of exports that have only a city property. Document 2, on the other hand, has a set of exports where some of the cities also have a set of dealers.

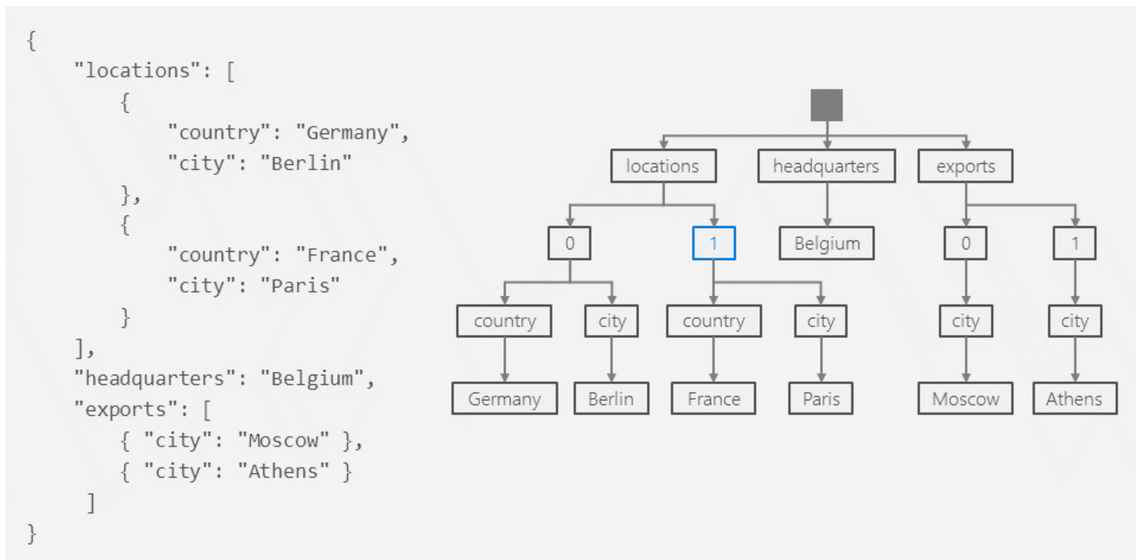


Figure 10. Sample JSON document 1.

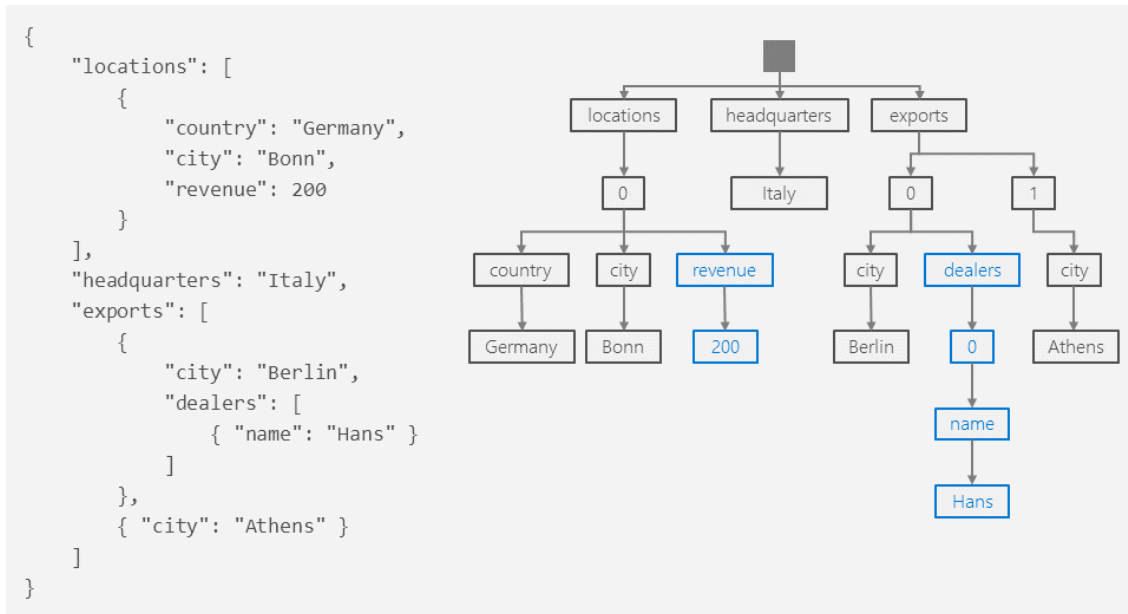


Figure 11. Sample JSON document 2.

Behind the scenes, using its ARS-based data model, Azure Cosmos DB models these records as trees: the root node is the document ID (or record ID), the properties under the root become the child nodes, and the instance values become the leaf nodes. The result is two different trees, each representing a different record. This schema never goes away and is always defined at a record level.

By merging these trees into an inverted index, we can establish pointers to the actual underlying set of records for our query results, as shown in Figure 12.

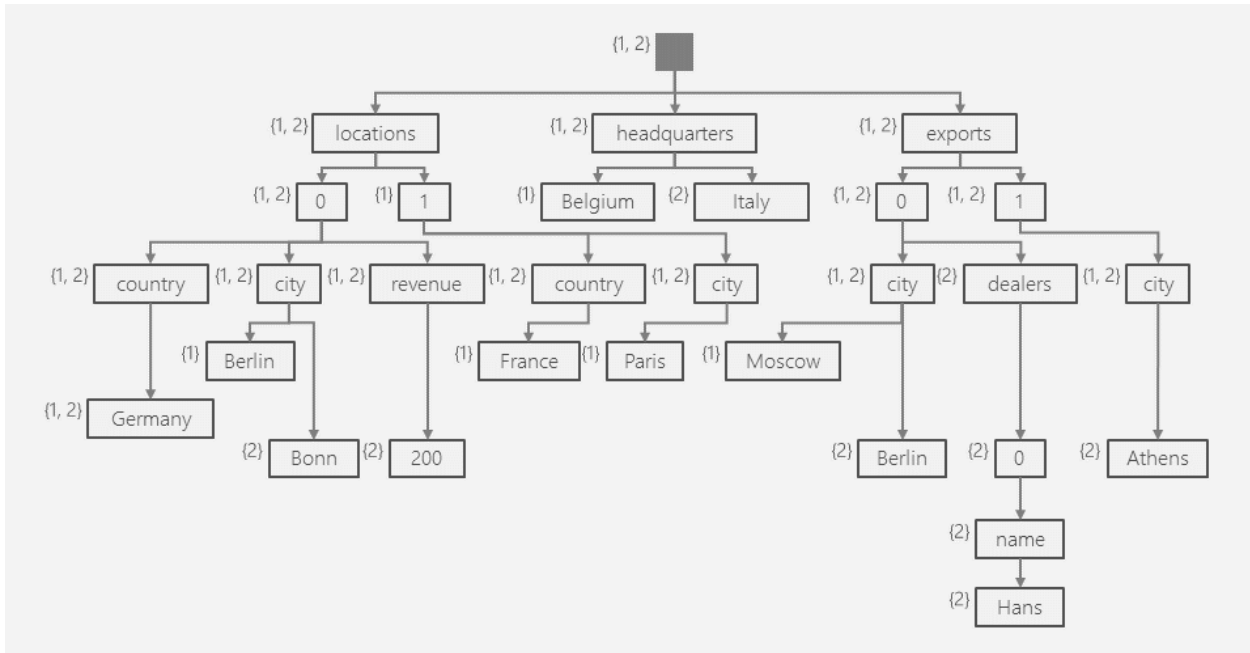


Figure 12. Inverted index based on sample documents 1 and 2.

Now, say we run a query on this container to find all records where the country location is Germany. By traversing the left side of the tree, we find that Germany has a pointer to records 1 and 2 and can efficiently return records 1 and 2 in the query result set. Similarly, by traversing the right side of the tree, we can determine that a dealer with the name Hans is unique to record 2.

One nice thing about this approach is that, because most of the paths or properties in a set of records will have a high degree of commonality, we can automatically index every property of every record while still achieving high compression rates to minimize storage overhead.

By default, Azure Cosmos DB indexes everything. However, you can specify a set of included paths and a set of excluded paths. So, we include the path /*. More specific paths will override entries with less specific paths, so you can always include /* and exclude a set of paths that you know you'll never need to query on—essentially, taking an opt-out approach. If you want to take a more traditional approach to indexing, you can define a set of paths to be included.

The Azure Cosmos DB documentation provides more information on indexing in Azure Cosmos DB and working with indexing policies.

Architectural considerations

Change feed

A common design pattern in many applications is to use changes to the data to trigger additional actions, with IoT, gaming, retail, and operational logging applications all being good examples. Change feed support in Azure Cosmos DB makes building such apps easy. It works by listening to an Azure Cosmos DB container for any changes and provides them as a sorted list of documents that were changed, in the order in which they were modified. Change feed is available for the container as a whole, or for each logical partition key within the container. Change feed output can be distributed across one or more consumers for parallel processing.

Following are just a few of the ways you can put change feed to use:

- Triggering a notification or a call to an API, when an item is inserted or updated.
- Real-time stream processing for IoT or real-time analytics processing on operational data.
- Additional data movement by either synchronizing with a cache or a search engine or a data warehouse or archiving data to cold storage.

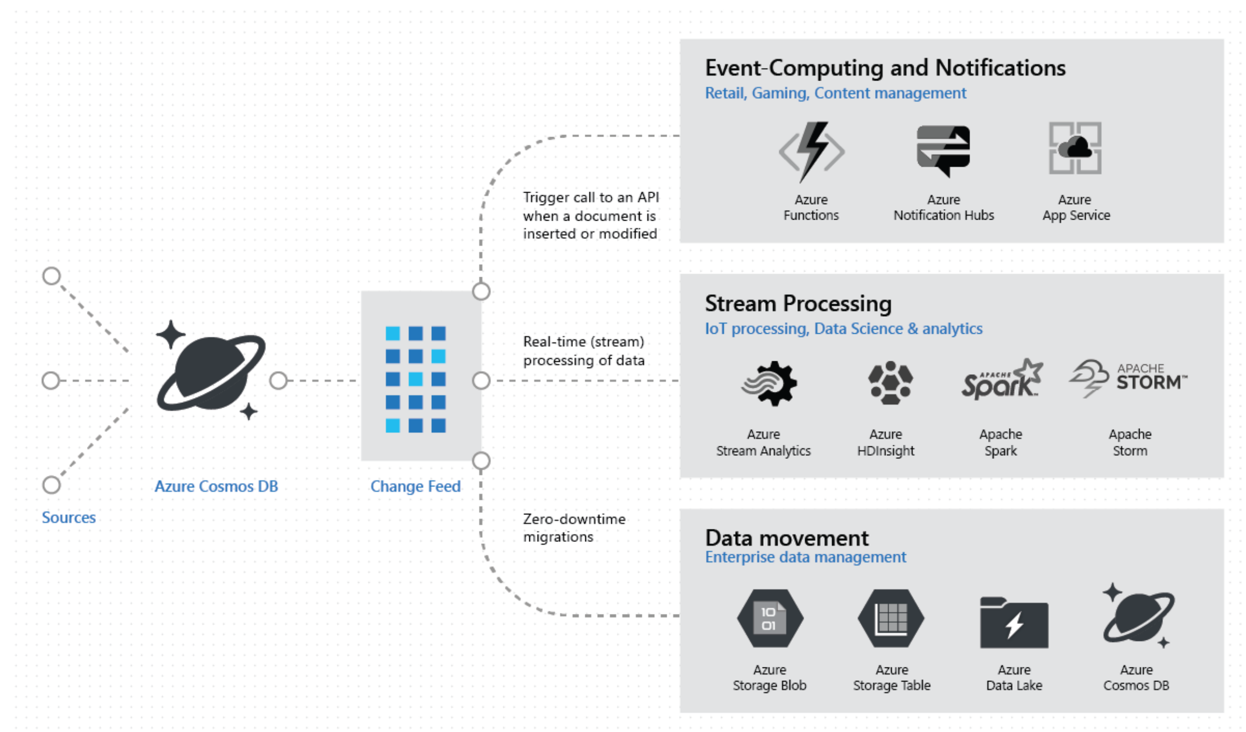


Figure 13. Azure Cosmos DB change feed support makes it easy to build applications that use changes to data to trigger additional actions.

You can work with change feed by using any of the following options.

- **Azure Functions.** This is the simplest (and recommended) option. When you create an Azure Cosmos DB trigger in an Azure Functions application, the Azure Function gets triggered whenever there's a change to the specified container. Building serverless apps, the next topic in this e-book, discusses using Azure Functions

together with Azure Cosmos DB in greater detail. *(NOTE: Currently, the Azure Cosmos DB trigger is supported for use with the core SQL API only. For all other Azure Cosmos DB APIs, you should access the database from your function by using the static client for your API.)*

- **Azure Cosmos DB SQL API SDK.** The Azure Cosmos DB change feed processor library within the Azure Cosmos DB SQL API SDK gives you complete, low-level control of the change feed while shielding you from excess complexity. It follows the observer pattern, where your processing function is called by the library. If you have a high-throughput change feed or want to distribute event processing across multiple consumers for other reasons, you can use the change feed processor library to automatically divide the load among the different clients—without you having to write that code. If you want to build your own load balancer, you can use the change feed processor library to implement a custom partition strategy. The SDK can be downloaded here. (Note the drop-down menu at the top of the page, which provides access to SDKs for other languages.)

The Azure Cosmos DB documentation provides more information on change feed, including how it works with different operations, use cases and scenarios, ways to work with change feed, and key features of change feed.

Building serverless apps with Azure Cosmos DB and Azure Functions

Serverless computing is all about the ability to focus on individual pieces of logic that are repeatable and stateless; they require no infrastructure management and they consume resources only for the seconds, or milliseconds, they run for. At the core of serverless computing are functions, which are made available in the Azure ecosystem by Azure Functions. With the native integration between Azure Cosmos DB and Azure Functions, you can create database triggers, input bindings, and output bindings directly from your Azure Cosmos DB account—making it easy to create and deploy event-driven serverless apps with low-latency access to rich data for a global user base.

Azure Cosmos DB and Azure Functions let you integrate your databases and serverless apps in the following ways:

- You can create an event-driven **Azure Cosmos DB trigger** in Azure Functions. This trigger relies on change-feed streams to monitor your Azure Cosmos DB container for changes. When any changes are made to a container, the change-feed stream is sent to the trigger, which invokes the function.
- Alternatively, you can bind a function to an Azure Cosmos DB container using an **input binding,** which reads data from a container when a function executes.
- You can also bind a function to an Azure Cosmos DB container using an **output binding,** which writes data to a container when a function completes.

The Azure Cosmos DB documentation provides more information on using Azure Functions to integrate your databases and serverless apps and on Azure Functions bindings.

Server-side programming

Azure Cosmos DB supports language-integrated, transactional execution of JavaScript when using the SQL API in Azure Cosmos DB. This allows you to write stored procedures, triggers, and user-defined functions (UDFs) in JavaScript, then have them execute within the database engine. You can create and execute triggers, stored procedures, and UDFs by using the Azure portal, the JavaScript query API in Azure Cosmos DB, or the Azure Cosmos DB SQL API client SDKs.

Stored procedures and triggers provide a means of executing multi-document transactions—a sequence of operations performed as a single logical unit of work. In Azure Cosmos DB, the JavaScript runtime is hosted inside the database engine, which means requests made within the stored procedures and the triggers execute in the same scope as the database session. This enables Azure Cosmos DB to guarantee ACID properties for all operations that are part of a stored procedure or a trigger.

The Azure Cosmos DB documentation provides more information on server-side programming with Azure Cosmos DB, including a discussion of benefits and additional detail on transactions, bounded execution, triggers, user-defined functions, and the JavaScript language integrated query API. There's a separate article that includes supported JavaScript functions in the JavaScript query API and a SQL to JavaScript cheat sheet.

Apache Spark to Azure Cosmos DB Connector

The Apache Spark to Azure Cosmos DB Connector lets you run Spark jobs on the data stored in Azure Cosmos DB. You can use the connector with Azure Databricks, Azure HDInsight, which provide managed Spark clusters on Azure. You can also use it with your own Spark deployment. The Apache Spark to Azure Cosmos DB Connector provides a low-latency data source for Spark that works for both batch and stream processing.

Built-in operational analytics with Apache Spark (in preview)

More recently, we announced a limited preview of built-in operational analytics in Azure Cosmos DB using Apache Spark. This allows you to run analytics from Apache Spark against data stored in an Azure Cosmos account *without a connector*, instead providing native support for Apache Spark jobs within Azure Cosmos DB. Capabilities also include built-in support for Jupyter notebooks, which run within Azure Cosmos DB accounts.

Built-in support for Apache Spark in Azure Cosmos DB will provide several advantages, beginning with the fastest time to insight for geographically distributed users and data. You can also simplify your analytics architecture and lower its TCO, as the system will have the least number of data processing components and avoid any unnecessary data movement among them. Scalability will be built-in, and you'll have a security, compliance, and auditing boundary that encompasses all the data under management. Finally, you'll be able to deliver highly available analytics backed by stringent SLAs.

The Azure Cosmos DB documentation provides more information on its built-in support for Apache Spark. Again, it's currently in limited preview, so if you want to try it out, you'll need to sign up for it here.

Operational considerations

Cost optimization with Azure Cosmos DB

The pricing model for Azure Cosmos DB simplifies cost management and planning, in that you pay only for the throughout you've provisioned (in RUs) and the storage that you consume. It's just one of the many reasons why Azure Cosmos DB delivers such a compelling total cost of ownership (TCO).

That said, just because Azure Cosmos DB delivers a great TCO, it doesn't mean that you shouldn't try to get the very most out of the resources you're paying for. The Azure Cosmos DB documentation includes numerous articles to help you optimize TCO—from understanding your bill to optimizing the cost of provisioned throughput. You'll also find articles on optimizing costs in relation to queries, storage, reads and writes, geographic distribution, development/test, and reserved capacity.

Security

Azure Cosmos DB includes numerous features and capabilities designed to help you prevent, detect, and respond to database breaches. That said, there are a few worth calling out here:

- **Data encryption.** All data is encrypted at rest and during transport, by default and at no additional cost.
- **Secure access.** With Azure Cosmos DB, data access is secured in several ways. Administrative resources (Azure Cosmos DB accounts, databases, users, and permissions) are secured using master keys. Application resources (containers, documents, attachments, stored procedures, triggers, and UDFs) are secured using resource tokens.
- **IP firewall.** By default, an Azure Cosmos DB account is accessible from the internet, as long as the request is accompanied by a valid authorization token. Configurable IP-based access controls in Azure Cosmos DB provide an additional layer of security, enabling access only from approved machines and/or cloud services (which still need a valid authorization token).
- **Access from virtual networks.** You can configure an Azure Cosmos DB account to allow access only from specified a specific subnet of a virtual network (Vnet). When you do this, only requests originating from those subnets will get a valid response; requests originating from any other source will receive a 403 (Forbidden) response.
- **Role-based access control.** Azure Cosmos DB provides built-in role-based access control (RBAC) for common management scenarios. An individual with a profile in Azure Active Directory can grant or deny access to resources (and operations on Azure Cosmos DB resources) by assigning these RBAC roles to users, groups, service principals, or managed identities. Role assignments are scoped to control-plane access only, which includes access to Azure Cosmos accounts, databases, containers, and offers (throughput).

Online backup and restore

Azure Cosmos DB automatically takes backups of your data at regular intervals, which is done without affecting the performance or availability of database operations. All backups are stored separately in Azure Blog storage, with those backups geographically replicated to protect against regional disasters. These automatic backups can be

helpful if you accidentally delete or update your Azure Cosmos account, database, or container and need to recover that data.

Azure Cosmos DB takes snapshots of your data every four hours. At any given time, only the last two snapshots are retained. However, if a container or database is deleted, Azure Cosmos DB retains existing snapshots of that container or database for 30 days.

With Azure Cosmos DB SQL API accounts, you can also maintain and manage your own backups. You can use Azure Data Factory to periodically output any data to any Azure Data Factory-supported storage destination, or you can use the Azure Cosmos DB change feed to read data periodically (for full backups and/or incremental changes) and store that data in an Azure Blob storage account.

The Azure Cosmos DB documentation provides more information on online backup and restore, including options to manage your own backups, backup retention, restoring data from online backups, and migrating restored data to the original Azure Cosmos DB account. (Although it's possible to use the restored account as the live account, it's not a recommended option for production workloads.)

Compliance

To help customers meet their own compliance obligations across regulated industries and markets worldwide, Azure maintains the largest compliance portfolio in the industry in terms of both breadth (total number of offerings) and depth (number of customer-facing services in assessment scope). These compliance offerings are grouped into four segments (globally applicable, US Government, industry specific, and region or country/region specific) and are based on various types of assurances, including formal certifications, attestations, validations, authorizations, and assessments produced by independent third-party auditing firms, as well as contractual amendments, self-assessments, and customer guidance documents produced by Microsoft. The Azure Cosmos DB documentation provides a comprehensive list of compliance certifications.

Building an app with Azure Cosmos DB

Choosing the right API

It's easy to get up and running with Azure Cosmos DB. If you haven't used it before, we provide a wealth of getting started resources on the following pages. Either way, the first thing you'll need to do is choose an API; each Azure Cosmos DB account supports one API, which you'll need to specify when creating an account, and each tutorial is specific to an API. Read on for advice on when to consider each API, followed by how to get started with the one you choose.

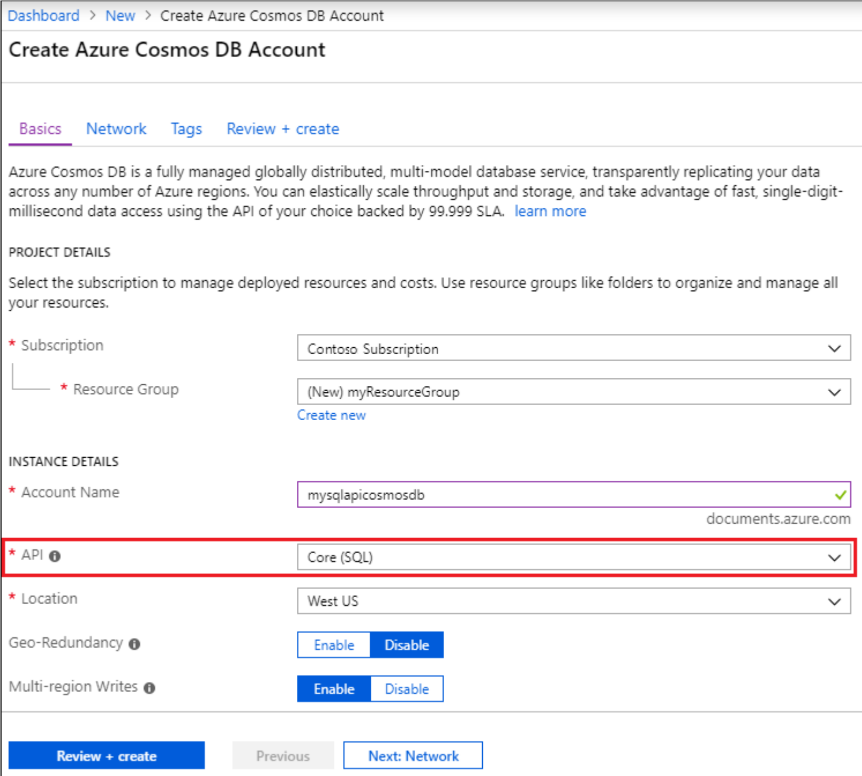


Figure 14. When you create an Azure Cosmos DB account, you'll need to choose an API.

Short on time? Try a 5-minute quickstart

Our 5-minute quickstarts can help you get started with Azure Cosmos DB in the time it takes to grab a cup of coffee. They're organized by API and programming language, so you shouldn't have trouble finding one that sparks your interest.

SQL API: .NET, .NET Preview, Java, Node.js, Python, Xamarin

MongoDB API: .NET, Java, Node.js, Python, Xamarin, Golang,

Gremlin API: .NET, Gremlin console, Java, Node.js, Python, PHP

Table API: .NET, Java, Node.js, Python

Cassandra API: .NET, Node.js, Java, Python

One database, multiple APIs

Azure Cosmos DB natively supports multiple data models and APIs, which we're continuing to add. It does this through an atom-record-sequence (ARS) based core type system, where atoms consist of a small set of primitive types (such as string, bool, and number), records are structs, and sequences are arrays consisting of atoms, records, or sequences.

The Azure Cosmos DB database engine translates and projects supported data models onto this core ARS-based data model, which is accessible from dynamically typed programming languages and can be exposed as is using JSON or other similar representations. The same design also enables native support for multiple APIs—enabling developers to build their apps using popular open-source APIs with all the benefits of an

enterprise-grade, fully managed, globally distributed database system.

Choosing an API

At this point, you might be thinking, "Multiple APIs give me options, but which do I choose?" The answer depends on your data, and what you want to do with it. Here are some general guidelines:

- **Document databases (supported by the SQL and MongoDB APIs)** store and retrieve documents, which are typically blocks of XML or JSON. Documents are self-describing (containing a description of the data type and a value for that description) and employ a hierarchical, tree-based structure, and they all don't have to be the same. At a high level, you can think of a document database as a key-value database where the value part (the document) can be examined via a query language.
- **Graph databases (supported by the Gremlin API)** allow you to store entities and the relationships between them. Entities (aka nodes) can have properties, just like the instance of an object in an app. Relations with other nodes (aka edges) can also have properties. With a graph database, you can store data once and then interpret the relationships within it in different ways very quickly and efficiently because the relationship between nodes is persisted instead of calculated at query time.
- **Column family databases (supported by the Cassandra API)** store data in groups of related information that that are often accessed together—such as a customer name, street address, and postal code. A column family is a set of rows (each with a row key) and an associated set of columns. Various rows in the column family don't need to contain the same columns, and columns can be added to one row without having to add them to other rows.
- **Key-value databases (supported via the Table API)** are relatively simple to use. From an API perspective, the database client can either Put, Get, or Delete values by keys. The value is a blob; the database doesn't care what's inside it. They're good for storing semi-structured data, providing a flexible data schema where the information in different rows can have different structures.

With five APIs to choose from, odds are that you'll have one optimized to the task at hand. For example, key-value databases are great for doing lookups with known values such as state code or postal code, or even for pulling user preferences for a signed-in user. Similarly, column databases are highly efficient at projections—that is, obtaining a few properties from a document that has many. Graph databases are helpful for visualizing both connected or networked data sets and hierarchical data relationships—family trees and flights between airports are good examples.

Get started with a free Azure subscription

To get started with Azure Cosmos DB, you need a Microsoft Azure subscription. If you don't have one, you can sign up for a free Azure subscription in just a few minutes. You can also try Azure Cosmos DB for free without an Azure subscription, free of any charges or commitments.

Getting Started with the SQL API

If you've chosen the SQL API, then you'll likely be modeling document data. And while schema-free databases like Azure Cosmos DB make it easy to change your data model, that doesn't mean you shouldn't spend some time thinking about it first. In doing so, you may want to ask yourself some basic questions: How will my data be stored? How will my app retrieve and query that data, and is the app read-heavy or write-heavy? Data modeling in Azure Cosmos DB provides a good overview—and some general rules-of-thumb—when it comes to modeling document data.

Quickstarts

Our 5-minute quickstarts can help you get started with the Azure Cosmos DB SQL API in the time it takes to walk down the hall for a cup of coffee. They're organized by programming language, so you shouldn't have trouble finding one that sparks your interest.

Language	Scenario
.NET	Create an Azure Cosmos DB SQL API account, document database, and container using the Azure portal, and then build and deploy a web app built on the SQL .NET API.
Java	Create an Azure Cosmos DB SQL API account using the Azure portal, and then create a Java app using the SQL Java SDK and add resources to your Azure Cosmos DB account by using the Java application.
Node.js	Create an Azure Cosmos DB SQL API account, document database, and container using the Azure portal, and then build and run a console app built on the SQL JavaScript SDK.
Python	Create an Azure Cosmos DB SQL API account, document database, and container using the Azure portal, and then build and run a console app built with the Python SDK for SQL API.
Xamarin	Create an Azure Cosmos DB SQL API account, document database, and container using the Azure portal, and then build and deploy a web app built on the SQL .NET API and Xamarin utilizing Xamarin.Forms and the MVVM architectural pattern.

Tutorials

When you're ready to dive deeper into the SQL API, the following tutorials are a great place to start. The tutorials listed under *Creating an Azure Cosmos DB* account assume you're starting from scratch and include instructions for creating an account. All of the other tutorials assume you already have an account.

Creating an Azure Cosmos DB account

The first thing you'll need to do is to create Azure Cosmos DB account and then a container. After that, you can connect to your new database and start using it. Any of the following tutorials will walk you through these basic concepts:

- Build a console app using .NET, Java, Async Java, or Node.js
- Build a web app using .NET, Java, Node.js, or Xamarin

Importing data

After you've familiarized yourself with Azure Cosmos DB, you might want to try migrating some of your own data into it. You can easily do this using the Data Migration tool, an open-source solution that imports data to Azure Cosmos DB from a variety of sources. Migrate your data to Azure Cosmos DB walks you through installing the Data Migration tool, importing data from different data sources, and exporting from Azure Cosmos DB to JSON. You can also import data programmatically using the Azure Cosmos DB bulk executor library.

Querying data

The Azure Cosmos DB SQL API supports querying documents using SQL. Querying data using the SQL API covers how to do this, including a sample document and two sample SQL queries.

Distributing data globally

After you have some data in your database, you might want to distribute it to additional Azure regions. Set up global distribution using the SQL API shows you how to set up Azure Cosmos DB global distribution using the Azure portal, and then connect to a preferred region using the SQL API.

How-to guides

The Azure Cosmos DB documentation includes how-to guides for common SQL API-related tasks, such as tuning query performance, server-side programming, and working with DateTime and geospatial data types. There are dozens of non-API-specific guides in the Azure Cosmos DB documentation, too, with the first one starting here and the rest listed below it in the left-hand navigation pane.

Additional resources

Here are some additional resources that you may find useful:

- **Sample applications** that show you how to work with Azure Cosmos DB, including performing CRUD operations and other common operations. They're organized by language: .NET, Java, Async Java, Node.js, Python, PowerShell, Azure CLI, and Azure Resource Manager.
- **Release notes and references** for various SDKs, libraries, resource providers, and so on. There are too many to list in this document, but you can find the first one here, with rest listed below it in the left-hand navigation pane.
- **Hands-on experience** working with Azure Cosmos DB using the SQL API, JavaScript, and the .NET Core SDK, which you can find at the Azure Cosmos DB workshop on GitHub.
- **Videos** on the Azure Cosmos DB YouTube channel, which cover a broad range of topics.

Getting Started with the Cassandra API

Like all supported APIs in Azure Cosmos DB, the Cassandra API is based on a native wire protocol implementation—that is, an implementation that does not use any Cassandra source code. This lets you easily migrate your Cassandra apps to Azure Cosmos DB while preserving significant portions of your application logic, enables you to keep your apps portable, and lets you continue to remain cloud vendor-agnostic.

The Azure Cosmos DB Cassandra API lets you interact with data stored in Azure Cosmos DB using the Cassandra Query Language (CQL), Cassandra-based tools (like cqlsh), and Cassandra client drivers that you're already familiar with. In many cases, you can switch from using Apache Cassandra to using the Azure Cosmos DB Cassandra API by merely changing a connection string—and realize all the benefits that using Azure Cosmos DB provides.

You can communicate with the Azure Cosmos DB Cassandra API through Cassandra Query Language (CQL) v4 wire protocol compliant open-source Cassandra client drivers. Our online documentation provides more information on supported CQL commands, tools, limitations, and exceptions.

Quickstarts

Our 5-minute quickstarts can help you get started with the Azure Cosmos DB MongoDB API in the time it takes to walk down the hall for a cup of coffee. They're organized by programming language, so you shouldn't have trouble finding one that sparks your interest.

Language	Scenario
.NET	Create an Azure Cosmos DB Cassandra API account, then build and review a sample .NET app by cloning an example from GitHub.
Java	Create an Azure Cosmos DB Cassandra API account, then build and review a sample Java app by cloning an example from GitHub.
Node.js	Create an Azure Cosmos DB Cassandra API account, then build and review a sample Node.js app by cloning an example from GitHub.
Python	Create an Azure Cosmos DB Cassandra API account, then build and review a sample Python app by cloning an example from GitHub.

Tutorials

When you're ready to dive deeper into the Cassandra API, the following tutorials are a great place to start. The first three tutorials listed below build on each other, so make sure to compete them in order. The tutorial on migrating data is self-contained.

Creating an Azure Cosmos DB account and managing data

Creating a Cassandra API account in Azure Cosmos DB describes how to create an Azure Cosmos DB Cassandra API account, and then use a Java sample project hosted on GitHub to create a project and dependencies, add a database and a table, and run the sample Java application.

Loading data

Next, you might want to try importing some of your own data. Loading sample data into a Cassandra API table in Azure Cosmos DB shows you how to load sample user data to a table in a Cassandra API account in Azure Cosmos DB by using a Java application.

Querying data

After you've imported your data, you'll be ready for our tutorial on querying data by using the Cassandra API.

Migrating Data

If you have existing Cassandra workloads that are running on-premises or in the cloud, and you want to migrate them to Azure, then you may want to peruse this tutorial on migrating data to an Azure Cosmos DB Cassandra API account. It covers your various options, which include using the cqlsh COPY command or using Apache Spark.

Cassandra and Spark

Apache Cassandra is often used together with Apache Spark as components of a comprehensive analytics stack; Cassandra stores the data, and Spark handles the data processing, including in-memory analytics. You can do the same thing on Azure—by using the Azure Cosmos DB Cassandra API as a data store and either Azure Databricks or Azure HDInsight for the analytics. Connecting to the Azure Cosmos DB Cassandra API from Spark provides guidance on how to do this—including connectivity dependencies, Spark connector throughout configuration parameters, Data Definition Language (DDL) operations, basic Data Manipulation Language (DML) operations, and more.

How-to guides

The Azure Cosmos DB documentation includes several how-to guides for common Cassandra API-related tasks, such as using Spring Data with Azure Cosmos DB, connecting to the Azure Cosmos DB Cassandra API from Spark, managing Azure Cosmos DB Cassandra API resources using Azure Resource Manager templates, and Azure PowerShell samples for the Azure Cosmos DB Cassandra API. There are dozens of non-API-specific guides in the Azure Cosmos DB documentation, too, with the first one starting here and the rest listed below it in the left-hand navigation pane.

There are also several videos on the Azure Cosmos DB YouTube channel, which cover a broad range of topics that you may find useful.

Getting Started with the Azure Cosmos DB for MongoDB API

Like all supported APIs in Azure Cosmos DB, the Azure Cosmos DB for MongoDB API is based on a *native* wire protocol implementation—that is, an implementation that does not use any MongoDB source code. This lets you easily migrate your MongoDB apps to Azure Cosmos DB while preserving significant portions of your application logic, enables you to keep your apps portable, and lets you continue to remain cloud vendor agnostic.

Because supported MongoDB wire protocol versions will change over time, it's best to refer to our online documentation for information on wire protocol compatibility and wire protocol support.

Quickstarts

Our 5-minute quickstarts can help you get started with the Azure Cosmos DB MongoDB API in the time it takes to walk down the hall for a cup of coffee. They're organized by programming language, so you shouldn't have trouble finding one that sparks your interest.

Language	Scenario
.NET	Create an the Azure Cosmos DB for MongoDB API account, a document database, and a collection using the Azure portal, and then build and deploy a web app based on the MongoDB .NET driver.
Java	Create an Azure Cosmos DB for MongoDB API account, a document database, and a collection using the Azure portal, and then build and deploy a web app based on the MongoDB Java driver.
Node.js	Connect an existing MongoDB app written in Node.js to an Azure Cosmos DB for MongoDB API account. When you're done, you will have a MEAN application (MongoDB, Express, Angular, and Node.js) running on Azure Cosmos DB.
Python	Build a simple Flask app by using the Azure Cosmos DB Emulator and the Azure Cosmos DB for MongoDB API.
Xamarin	Create an Azure Cosmos DB for MongoDB API account, a document database, and a collection using the Azure portal, and then build a Xamarin.Forms app by using the MongoDB .NET driver.
Golang	Create an Azure Cosmos DB for MongoDB API account and then connect to it using an existing MongoDB app written in Golang.

Tutorials

When you're ready to dive deeper into the Azure Cosmos DB for MongoDB API, the following tutorials are a great place to start. The tutorials listed under *Creating an Azure Cosmos DB account* assume you're starting from scratch and include instructions for creating an account. All of the other tutorials assume you already have an account.

Creating an Azure Cosmos DB account and managing data

The following tutorials walk you through creating an Azure Cosmos DB account and creating a collection, and then they show you how to connect to your new database and start using it.

Tutorial	Scenario
Node.js console app	Use a Node.js console app to connect to an Azure Cosmos DB for MongoDB API database
Create a MongoDB app with Angular	Create a MongoDB app with Express, Angular, and Node.js (the MEAN stack), and then connect it to Azure Cosmos DB. You'll create a Node.js Express app with the Angular CLI; build the UI with Angular; create an Azure Cosmos DB account using the Azure CLI; connect to Azure Cosmos DB using Mongoose; and add Post, Put, and Delete functions.
Create a MongoDB app with React	Similar to the above but uses React instead of Angular—and is a video tutorial.

Migrating data

After you've familiarized yourself with Azure Cosmos DB, you might want to try importing some of your own data. To migrate data into or out of Azure Cosmos DB for MongoDB API collections, you need to use Mongoimport.exe or Mongorestore.exe.

Querying data

Querying data using the Azure Cosmos DB for MongoDB API provides several example queries that show you how to query your data in Azure Cosmos DB using MongoDB shell.

Distributing data globally

After you have some data in your database, you might want to distribute it to additional Azure regions. Set up global distribution using the Azure Cosmos DB for MongoDB API shows you how to set up Azure Cosmos DB global distribution using the Azure portal, verify your regional setup, and then connect to a preferred region using the Azure Cosmos DB for MongoDB API ,

How-to guides

The Azure Cosmos DB documentation includes several how-to guides for common Azure Cosmos DB for MongoDB API-related tasks, such as getting the connection string, connecting using Studio 3T, distributing reads globally, using time-to-live (TTL) functionality to automatically expire data, and managing data indexing. There are dozens of non-API-

specific guides in the Azure Cosmos DB documentation, too, with the first one starting here and the rest listed below it in the left-hand navigation pane.

There are also several videos on the Azure Cosmos DB YouTube channel, which cover a broad range of topics that you may find useful.

Getting Started with the Gremlin API

Like all supported APIs in Azure Cosmos DB, the Gremlin API is based on a native wire protocol implementation—that is, an implementation that does not use any Gremlin source code. This lets you easily migrate your Gremlin apps to Azure Cosmos DB while preserving significant portions of your application logic, enables you to keep your apps portable, and lets you continue to remain cloud vendor-agnostic.

Azure Cosmos DB supports Apache TinkerPop's Gremlin graph traversal language, which you can use to create graph entities (vertices and edges), modify properties within those entities, perform queries and traversals, and delete entities. If you're not familiar with graph databases or Gremlin, the following resources may be helpful:

- Introduction to the Azure Cosmos DB Gremlin API provides an overview of graph databases and explains how you can use them to store massive graphs with billions of vertices and edges.
- Azure Cosmos DB Gremlin graph support provides a basic introduction to Gremlin, including examples, features, GraphSON (the Gremlin wire format), and the Gremlin steps supported by Azure Cosmos DB.

Quickstarts

Our 5-minute quickstarts can help you get started with the Azure Cosmos DB for MongoDB API in the time it takes to walk down the hall for a cup of coffee. They're organized by programming language, so you shouldn't have trouble finding one that sparks your interest.

Language	Scenario
Gremlin console	Create an Azure Cosmos DB Gremlin API account, database, and graph (container) using the Azure portal, and then use the Gremlin Console from Apache TinkerPop to work with Gremlin API data
.NET	Create an Azure Cosmos DB Gremlin API account, database, and graph (container) using the Azure portal, and then build and run a console app built using the open-source driver Gremlin.Net.
Java	Create a simple graph database using the Azure portal, and then create a Java console app using a Gremlin API database using the OSS Apache TinkerPop driver.
Node.js	Create an Azure Cosmos DB Gremlin API account, database, and graph using the Azure portal, and then use the open-source Gremlin Node.js driver to build and run a console app.
Python	Create an Azure Cosmos DB Gremlin API account by using the Azure portal, and then use Python to build a console app by cloning an example from GitHub.
PHP	Create an Azure Cosmos DB Gremlin API account by using the Azure portal, and then use PHP to build a console app by cloning an example from GitHub.

Tutorials

When you're ready to dive deeper into the Gremlin API, the following tutorials are a great place to start:

Migrating data

You can import data programmatically using the bulk executor library for the Gremlin API on GitHub. Our how-to guide for using the bulk executor library with the Gremlin API provides instructions for using it to import and update graph objects into an Azure Cosmos DB Gremlin API container.

Querying data

Querying data using the Gremlin API provides sample documents and queries to get you started with Gremlin queries.

How-to guides

The Azure Cosmos DB documentation includes several how-to guides for common Gremlin API-related tasks, such as using a partitioned graph in Azure Cosmos DB, optimizing Gremlin queries, and Azure PowerShell samples for the Azure Cosmos DB Gremlin API. There are dozens of non-API-specific guides in the Azure Cosmos DB documentation, too, with the first one starting here and the rest listed below it in the left-hand navigation pane.

There are also several videos on the Azure Cosmos DB YouTube channel, which cover a broad range of topics that you may find useful.

Getting Started with the Table API

The Azure Cosmos DB Table API supports applications that are written for Azure Table storage, augmenting them with premium capabilities such as turnkey global distribution, dedicated throughput worldwide, single-digit millisecond latencies at the 99th percentile, guaranteed high availability, and automatic secondary indexing. Our introduction to the Table API takes a closer look at the benefits of moving from Azure Table storage to the Azure Cosmos DB Table API.

Quickstarts

Our 5-minute quickstarts can help you get started with the Azure Cosmos DB for MongoDB API in the time it takes to walk down the hall for a cup of coffee. They're organized by programming language, so you shouldn't have trouble finding one that sparks your interest.

Language	Scenario
.NET	Use the Azure portal to create an Azure Cosmos DB Table API account, use Data Explorer to create tables and entities, and then build a .NET app that connects to the Table API by cloning an example from GitHub.
Java	Use the Azure portal to create an Azure Cosmos DB Table API account, use Data Explorer to create tables and entities, and then build a Java app that connects to the Table API by cloning an example from GitHub.
Node.js	Use the Azure portal to create an Azure Cosmos DB Table API account, use Data Explorer to create tables and entities, and then build a Node.js app that connects to the Table API by cloning an example from GitHub.
Python	Use the Azure portal to create an Azure Cosmos DB Table API account, use Data Explorer to create tables and entities, and then build a Python app that connects to the Table API by cloning an example from GitHub.

Tutorials

When you're ready to dive deeper into the Table API, the following tutorials are a great place to start. The tutorials listed under *Step 1: Creating an Azure Cosmos DB* account assume you're starting from scratch and include instructions for creating an account. All of the other tutorials assume you already have an account.

Step 1: Creating an Azure Cosmos DB account and managing data

Get started with Azure Cosmos DB Table API and Azure Table storage walks you through creating an Azure Cosmos DB account and creating a table, and then shows you how to connect to your new database and start using it. You learn how to enable functionality in the app.config file, create a table using the Table API, add an entity to a table, insert a batch of entities, retrieve a single entity, query entities

using automatic secondary indexes, replace an entity, delete an entity, and delete a table.

Step 2: Importing data

Migrate your data to an Azure Cosmos DB Table API account walks you through importing data for use with the Table API. If your data is in Azure Table storage, you can use either the Data Migration Tool or AzCopy to import it. If your data is in an Azure Cosmos DB Table API (preview) account, you'll need to use the Data Migration tool to import it. Both methods are addressed in the tutorial.

Step 3: Querying data

The Azure Cosmos DB Table API supports OData and LINQ queries against key/value (table) data. Both methods are covered in our tutorial on querying Azure Cosmos DB by using the Table API.

Step 4: Distributing data globally

After you have some data in your database, you might want to distribute it to additional Azure regions. Set up global distribution using the Table API walks you through replicating data to additional Azure regions by using the Azure portal, and then connecting to a preferred region by using the Table API.

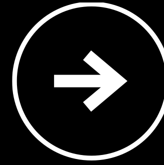
How-to guides

The Azure Cosmos DB documentation includes several how-to guides for common SQL API-related tasks, such as building apps with the Table API, guidance on Table storage design, and Azure PowerShell samples for the Azure Cosmos DB Table API. There are dozens of non-API-specific guides in the Azure Cosmos DB documentation, too, with the first one starting here and the rest listed below it in the left-hand navigation pane.

There are also several videos on the Azure Cosmos DB YouTube channel, which cover a broad range of topics that you may find useful.

Conclusion

If you're looking for a NoSQL database, you owe it to yourself to consider Azure Cosmos DB. No other multi-model distributed database offers turnkey global distribution; unlimited elastic scalability of storage and throughput; guaranteed single-digit-millisecond latency; five well-defined consistency levels; and comprehensive SLAs for availability, latency, throughput, and consistency. Regardless of what your next app is built to do, if it needs to do it with low latency at a global scale, Azure Cosmos DB can help you get there.



Sign up for a free Azure account and get started with Azure Cosmos DB today – or try Azure Cosmos DB for free without an Azure subscription.