Microsoft

# Cosmos DB: Blazing Fast Planet-Scale NoSQL

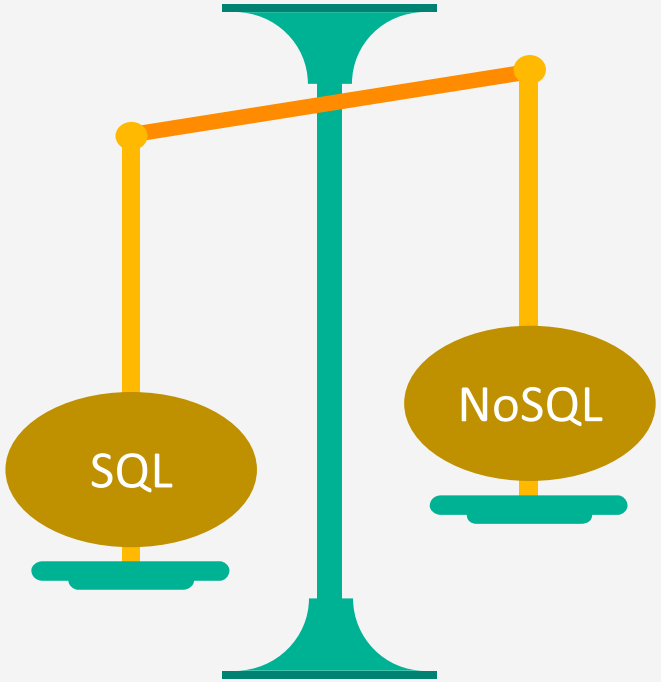
Manjunath Suryanarayana

masuryan@microsoft.com

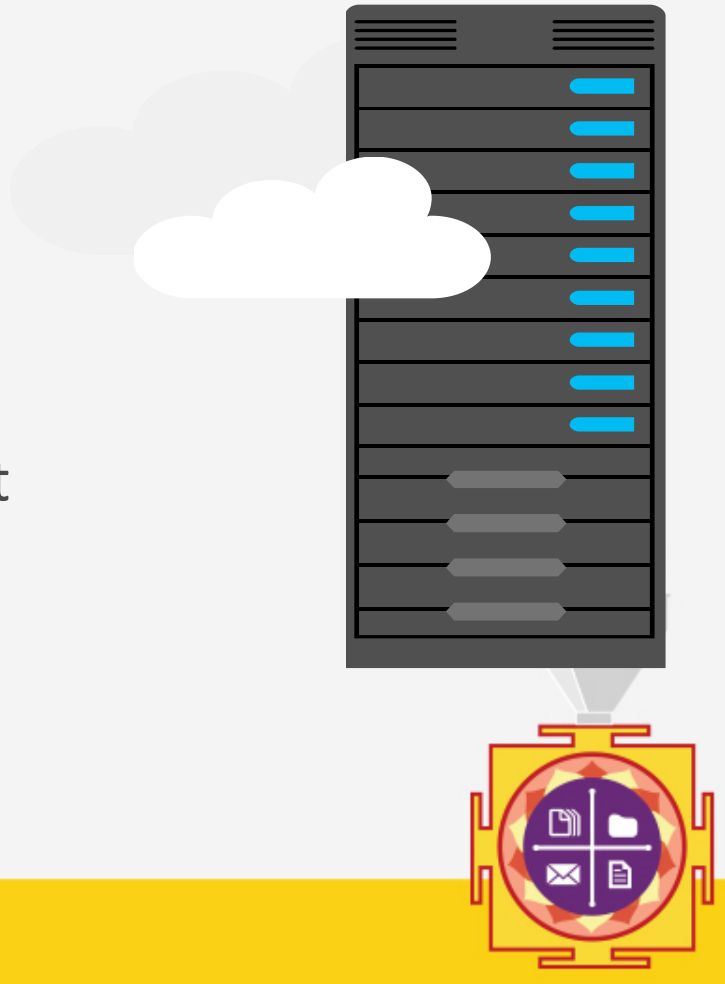
Microsoft

# What is NoSQL?

A database solution designed to compensate for the technical limitations of SQL



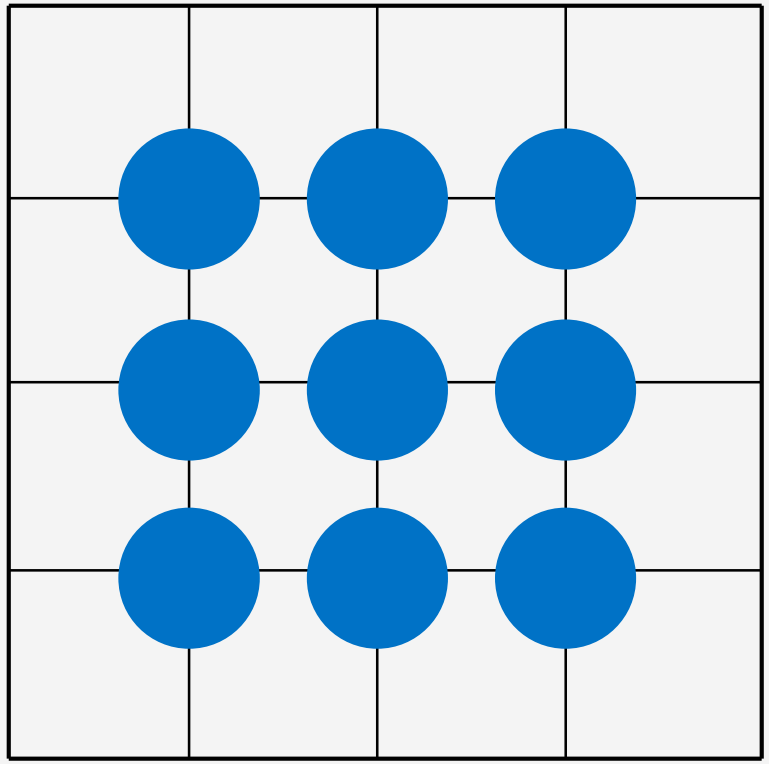

Choose the store that best fits your needs



# Traditional approach: relational stores

Data is stored in tables that comprise:

- Schemas
- Columns
- Rows



Microsoft

# NoSQL approach: various types of stores

Key value

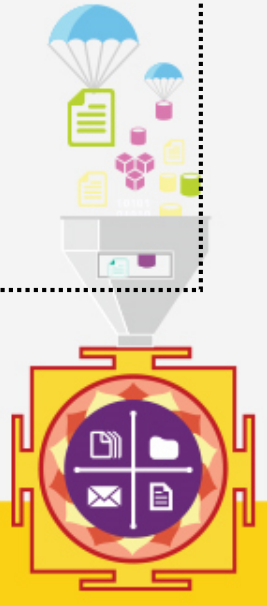
Wide column

Document

Graph

## Azure Cosmos DB

Uses all including graph category

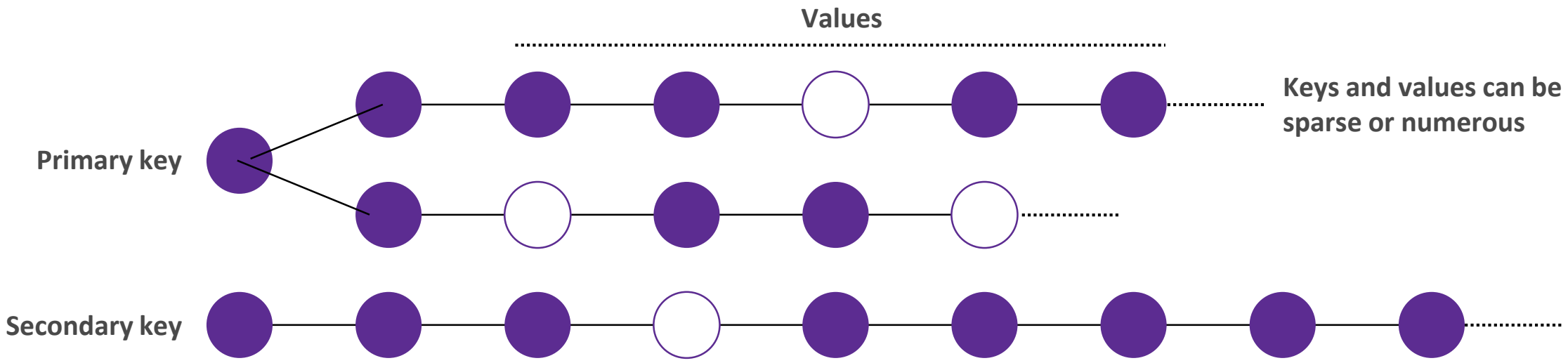


# Key-value stores

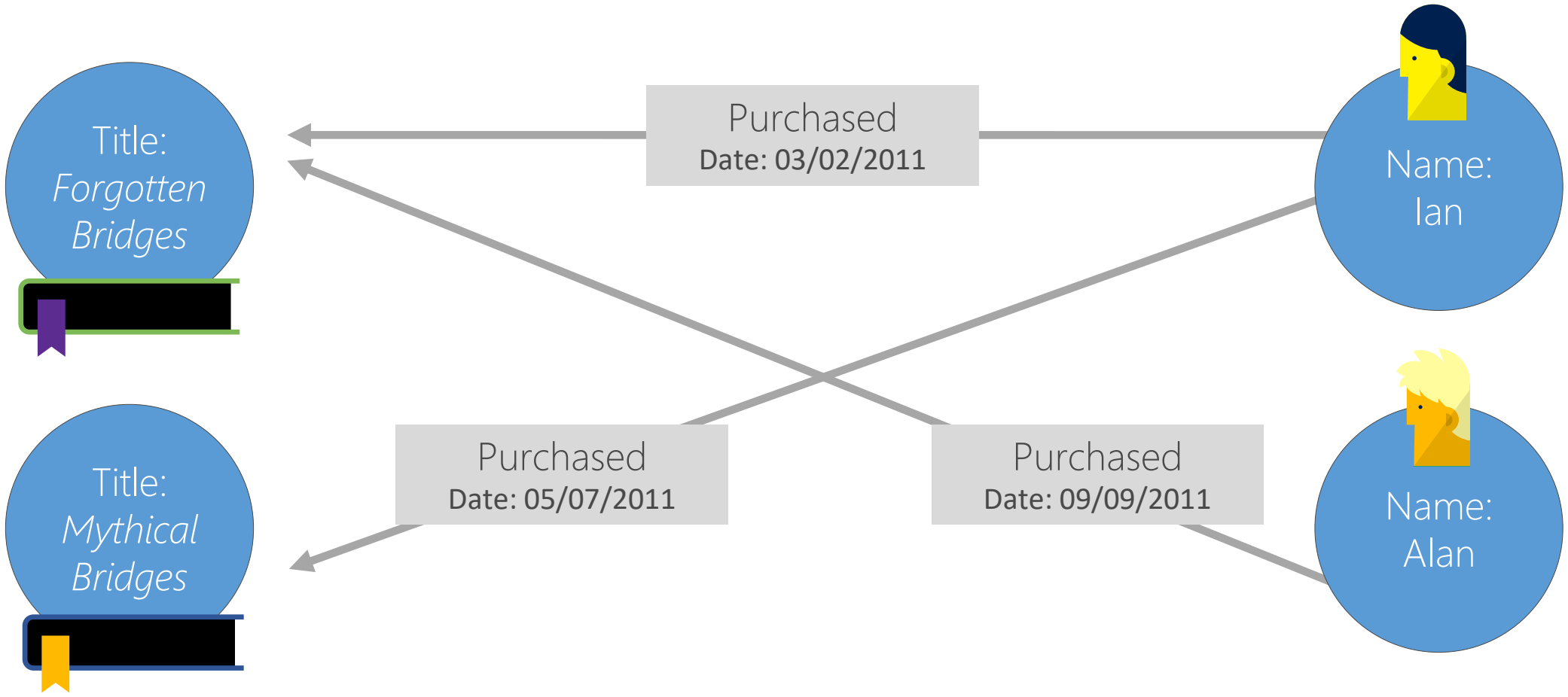
Key-value stores offer high speed through the least-complicated data model—anything can be stored as a value, as long as each value is associated with a key or name.

# Wide-column stores

Wide-column stores are fast and can be almost as simple as key-value stores. They include a primary key, an optional secondary key, and anything stored as a value.



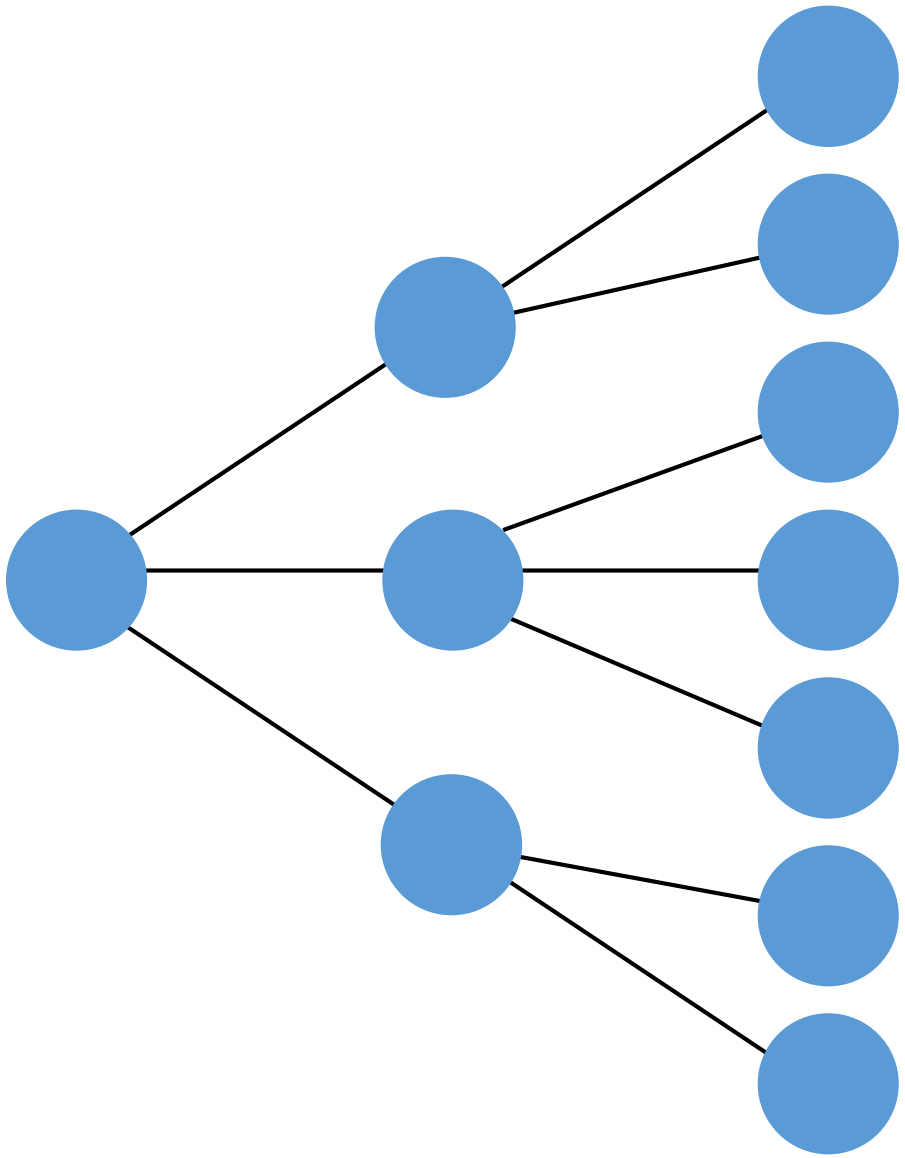
# Graph databases



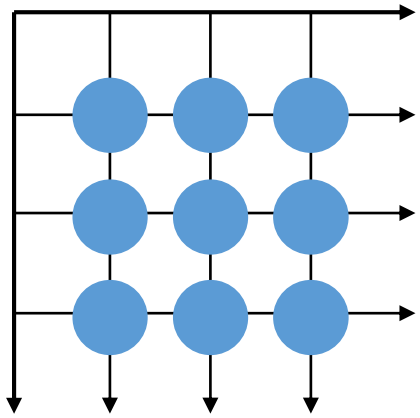
# Document stores

Document stores contain data objects that are inherently hierarchical, tree-like structures (most notably JavaScript Object Notation [JSON] or Extensible Markup Language [XML]).

Note that these are not Microsoft Word documents!

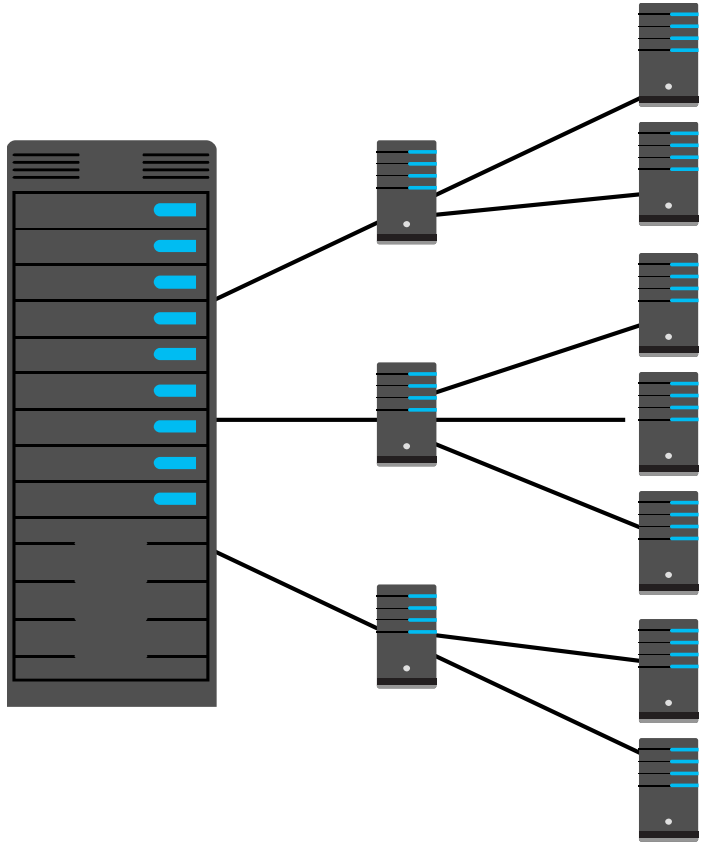


# Why NoSQL evolved



## Drivers

- Operational efficiency
- Resiliency
- Scalability
- Speed and variety of cloud-based data



Microsoft

# Azure Cosmos DB

## Globally distributed, multi-model database service

# Developing planet-scale apps comes with planet-scale challenges

- **Write accurate, globally distributed apps**
- **Managing and versioning complex schemas**
- **Scaling both throughput and storage based on global demand**
- **Balancing the needs for strong and eventual consistency**
- **Delivering highly-responsive experiences**
- **Ensuring an always-on system**

Microsoft

# Introducing Azure Cosmos DB

## A globally distributed, massively scalable, multi-model database service

### Global distribution

Automatically replicate all your data around the world – across more regions than Amazon and Google combined

Microsoft

# Introducing Azure Cosmos DB

## A globally distributed, massively scalable, multi-model database service

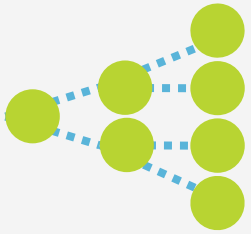
Global distribution

**Multi-model + multi API**

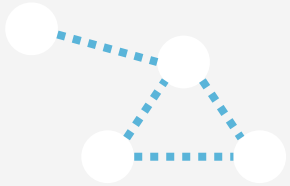
Use key-value, graph, and document with a schema-agnostic service that doesn't require any schema or secondary indexes

KEY-VALUE

COLUMN-FAMILY



DOCUMENT



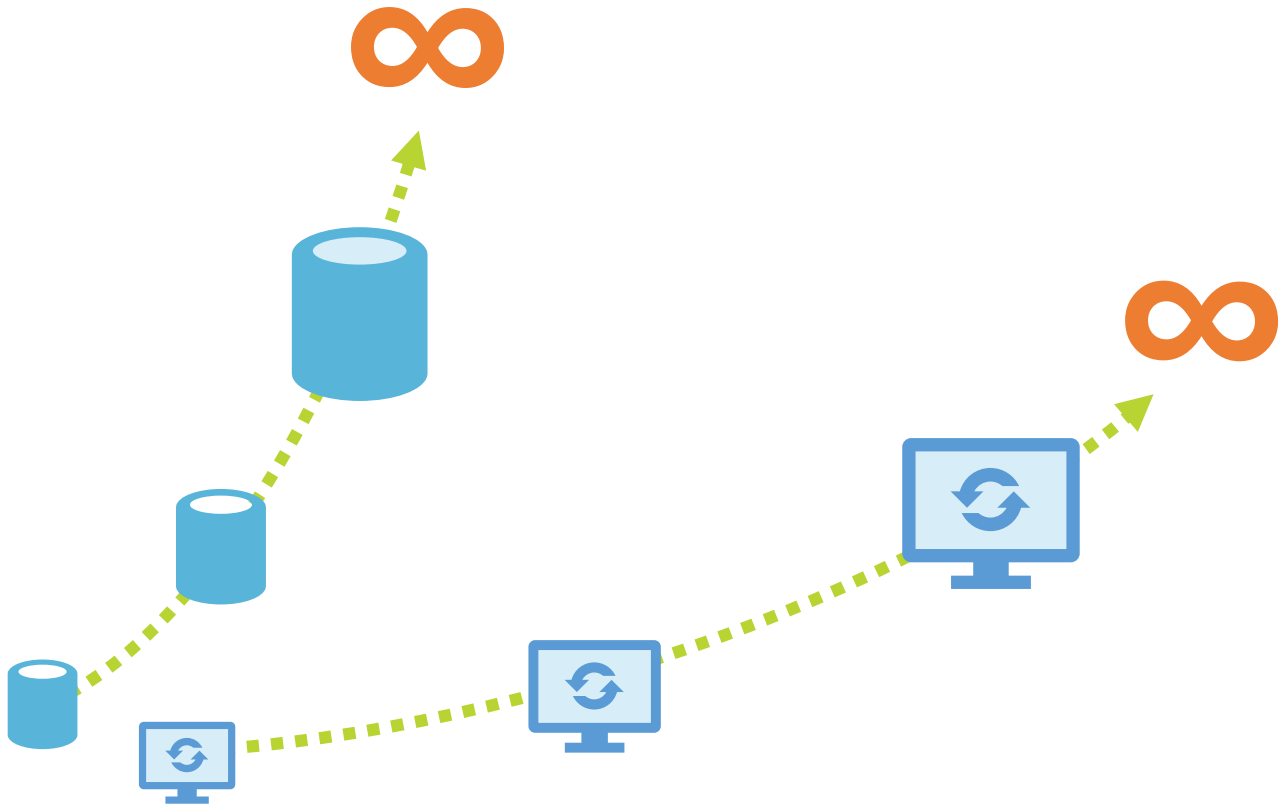
GRAPH

# Introducing Azure Cosmos DB

## A globally distributed, massively scalable, multi-model database service

- Global distribution
- Multi-model + multi API
- **Elastic scale-out**

Independently and elastically scale storage and throughput across regions

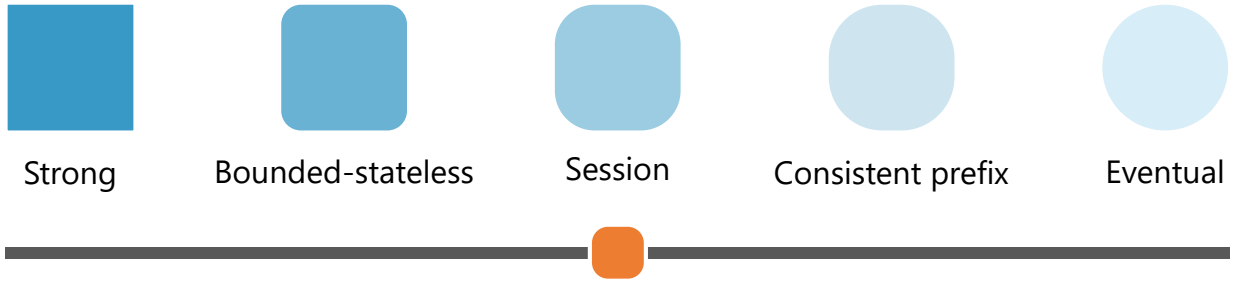


# Introducing Azure Cosmos DB

## A globally distributed, massively scalable, multi-model database service

- Global distribution
- Multi-model + multi API
- Elastic scale-out
- **Choice of consistency**

Choose from five defined consistency levels for low latency and high availability



# Introducing Azure Cosmos DB

## A globally distributed, massively scalable, multi-model database service

- Global distribution
- Multi-model + multi API
- Elastic scale-out
- Choice of consistency
- **Guaranteed single-digit latency**

Serve <10 ms read and <15 ms write requests at the 99th percentile from the nearest region while delivering data globally

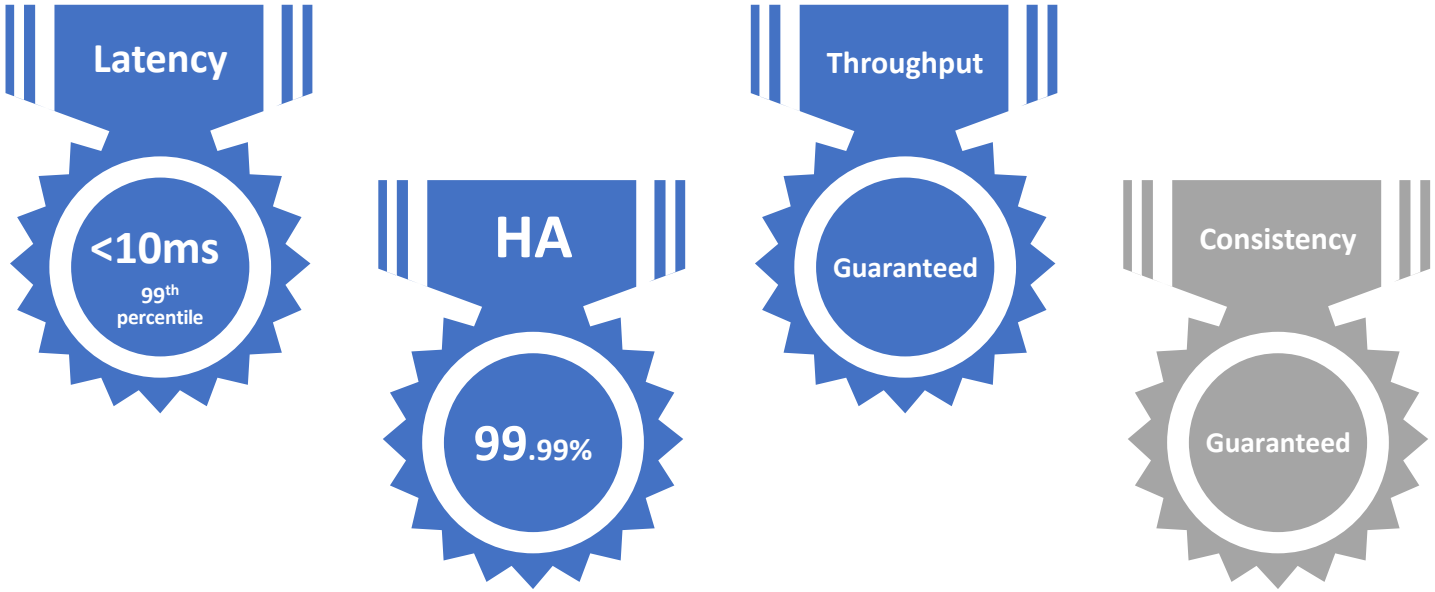
Guaranteed global millisecond latency at the 99th percentile

# Introducing Azure Cosmos DB

## A globally distributed, massively scalable, multi-model database service

- Global distribution
- Multi-model + multi API
- Elastic scale-out
- Choice of consistency
- Guaranteed single-digit latency
- **Enterprise-level SLAs**

Only service with financially-backed SLAs for millisecond latency at the 99th percentile, 99.99% HA and guaranteed throughput and consistency



# Introducing Azure Cosmos DB

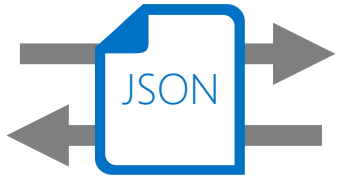
A globally distributed, massively scalable, multi-model database service

- **Global distribution**
- **Multi-model + multi API**
- **Elastic scale-out**
- **Choice of consistency**
- **Guaranteed single-digit latency**
- **Enterprise-level SLAs**

# Azure Cosmos DB

## A NoSQL document database-as-a-service, fully managed by Azure

**Application**




**SQL query**

**Azure Cosmos DB Document store**

DOCUMENT 1

```
{
  "name": "John",
  "country": "Canada",
  "age": 43,
  "lastUse": "March 4, 2014"
}
```

DOCUMENT 2

```
{
  "name": "Eva",
  "country": "Germany",
  "age": 25
}
```

DOCUMENT 3

```
{
  "name": "Lou",
  "country": "Australia",
  "age": 51,
  "firstUse": "May 8, 2013"
}
```

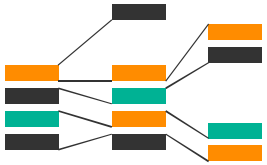
DOCUMENT 4

```
{
 "docCount": 3,
 "last": "May 1, 2014"
}
```

Perfect for cloud **architects and developers** who need an enterprise-ready NoSQL document database

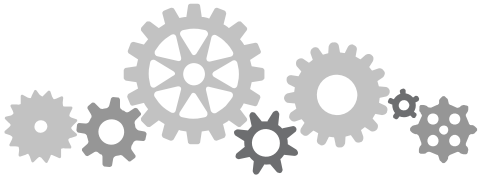
# Azure Cosmos DB details

Ideal for apps designed for the cloud when the following are high priorities:



## RichQuery and transactions over JSON data

- **Query JSON data** with no secondary indices



## Reliable and predictable performance

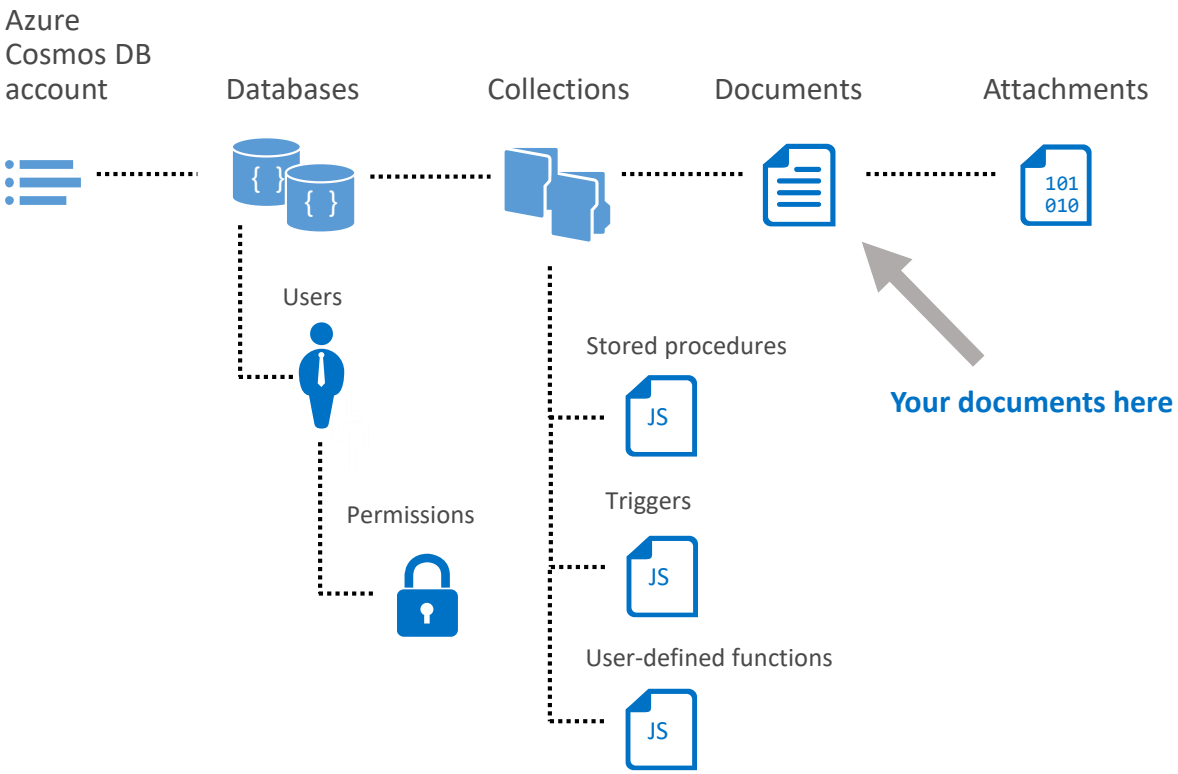
- Tunable consistency
- Elastic scale

## Rapid development

- **Build with familiar tools**—REST, JSON, JavaScript

Native support for JavaScript, SQL query, and transactions over JSON documents

# Azure Cosmos DB basics




## Resource model

- Entities addressable by logical Uniform Resource Identifier (URI)
- Partitioned for scale out
- Replicated for high availability
- Entities represented as JSON
- Accounts scale out by moving a slider

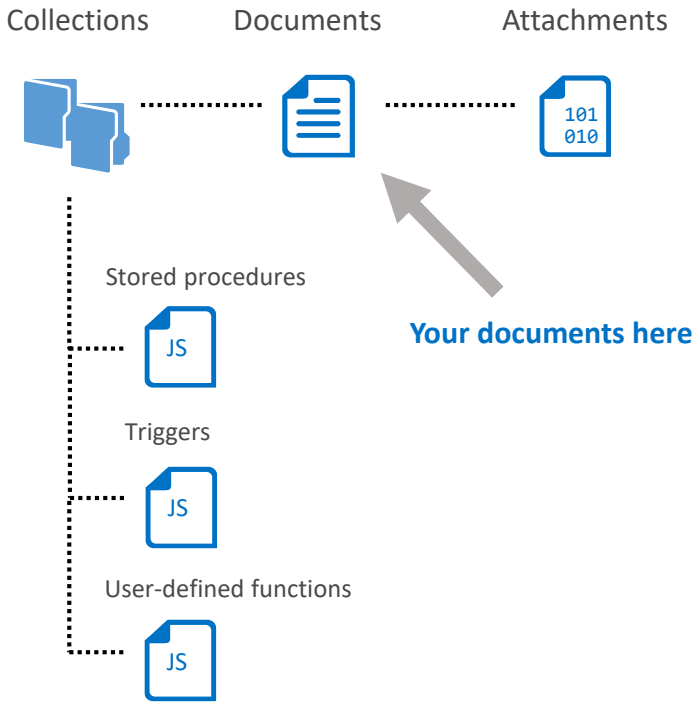
## Interaction model

- RESTful interaction over HTTPS
- HTTPS and TCP connectivity
- Standard HTTPS verbs and semantics

## Development

- .NET, Node.js, Python, Java, and JavaScript clients
- SQL for query expression, .NET LINQ
- JavaScript for server-side app logic

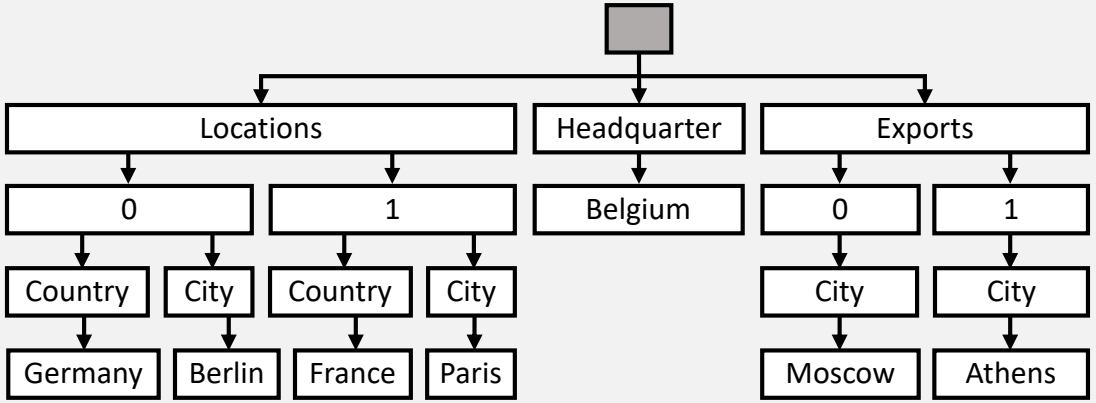
# Azure Cosmos DB collections




- Collections != tables
- Unit of partitioning
- Transaction boundary
- No enforced schema, flexible
- Queried or updated stay together in one collection
- Elasticity to 10 GB
- RUs evenly distributed across partitions

# Cosmos DB JSON documents

```
{
    “locations”:
    [
        {“country”: “Germany”, “city”: “Berlin”},
        {“country”: “France”, “city”: “Paris”},
    ],
    “headquarter”: “Belgium”,
    “exports”:[{“city”: “Moscow”},{“city”: “Athens”}]
}
```




JSON document as tree

## JSON

Intersection of most modern type systems

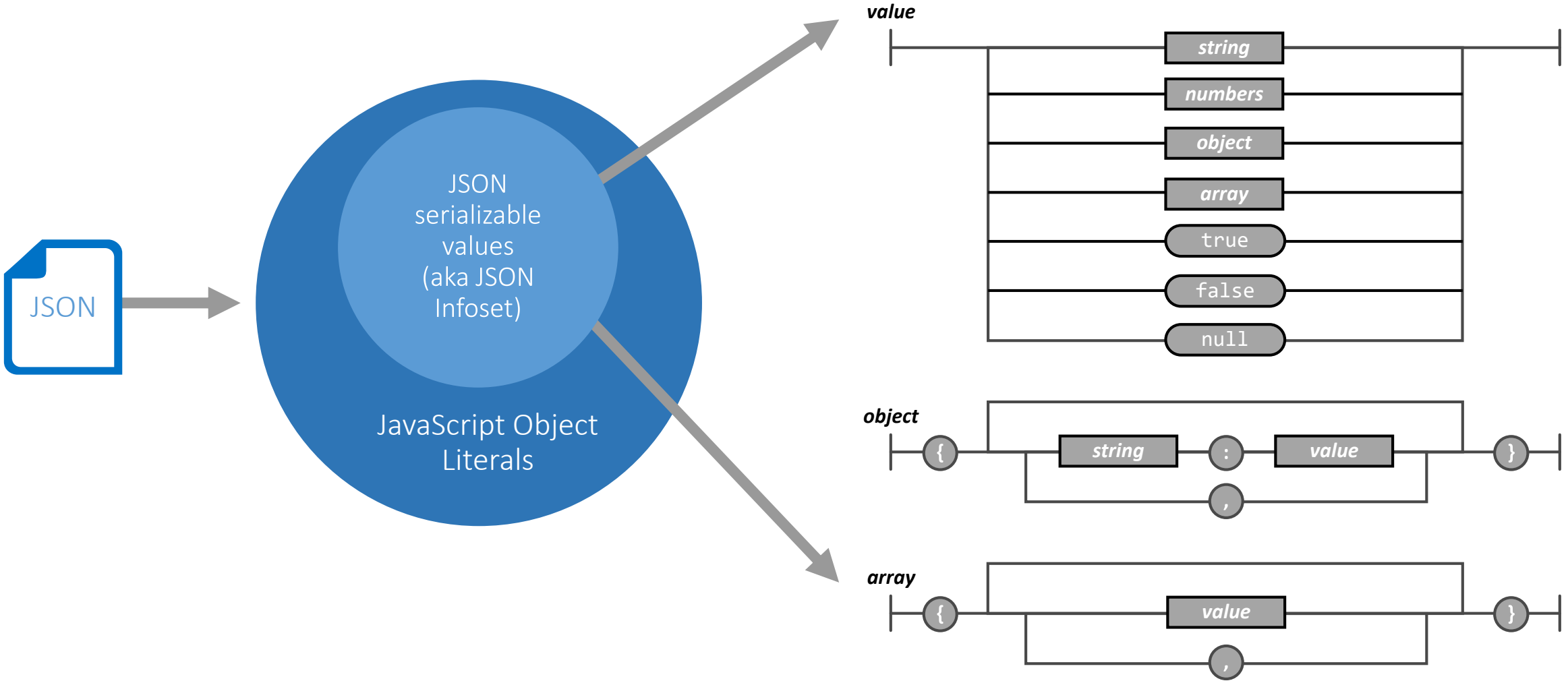
## JSON values

Self-describable, self-contained *values*

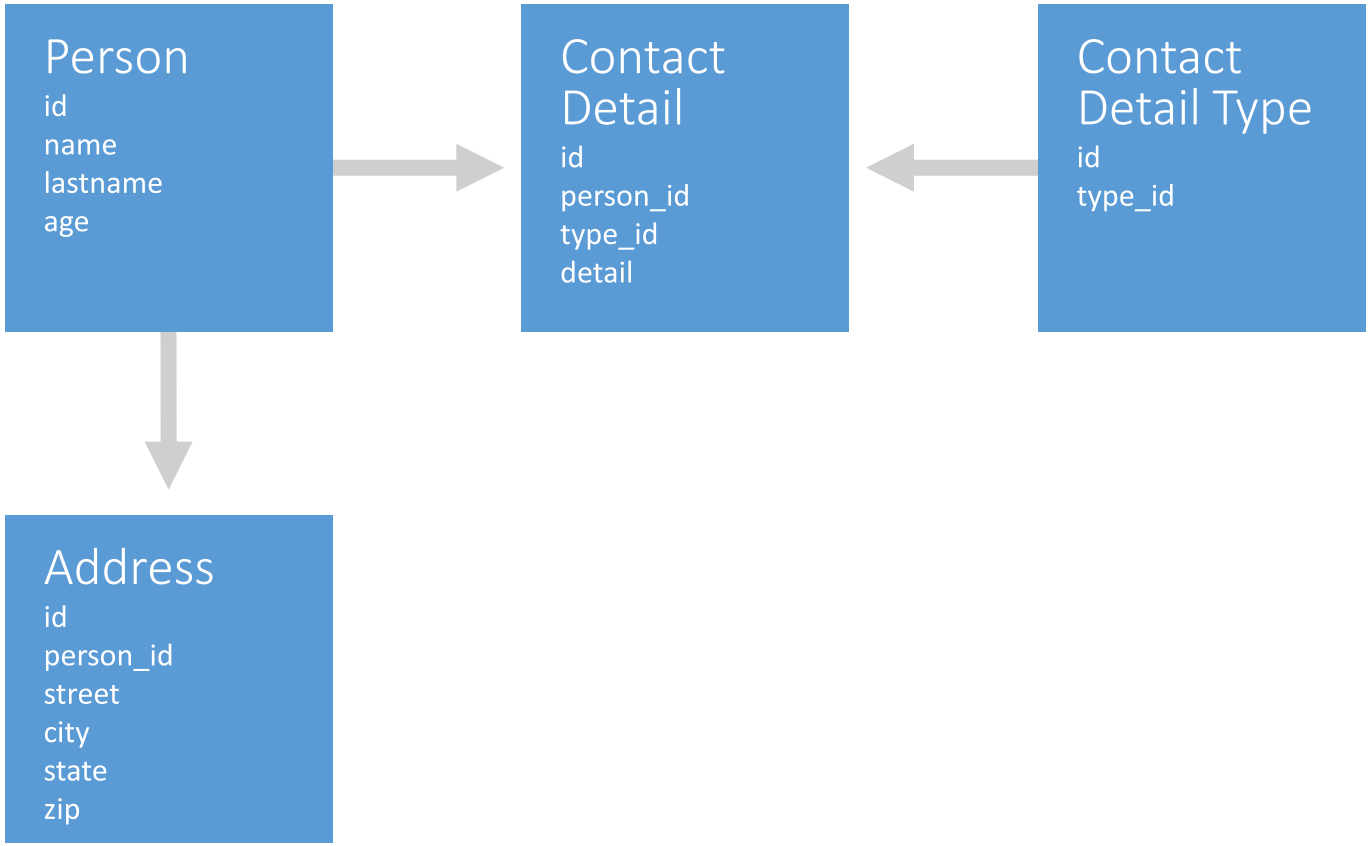
Are trivially serialized to/from text

## Cosmos DB makes a deep commitment to JSON for storage, indexing, query, and JavaScript execution

# Cosmos DB JSON documents



# Data modeling with RDBMS




Doing it the RDBMS way: normalize, normalize, normalize…

Joins needed to query for Person with related tables

```
SELECT p.name, p.lastName, p.age,
    cd.detail , cdt.type, a.street, a.city,
    a.state, a.zip
FROM Person p
  INNER JOIN Address a
    ON a.person_id = p.id
  INNER JOIN ContactDetail cd
    ON cd.person_id = p.id
  INNER JOIN ContactDetailType cdt
    ON cd.type_id = cdt.id
```

Updating Person, ContactDetail, and Address requires updates to many tables

# Data modeling with denormalization

```
{
   id_: <ObjectId1>,
   username: "123xyz",
   contact:
   {
      phone: "555-121-1212",
      email: "xyx@abc.com"
   },
   access:
   {
      level: 5,
      group: "dev"
   }
}
```

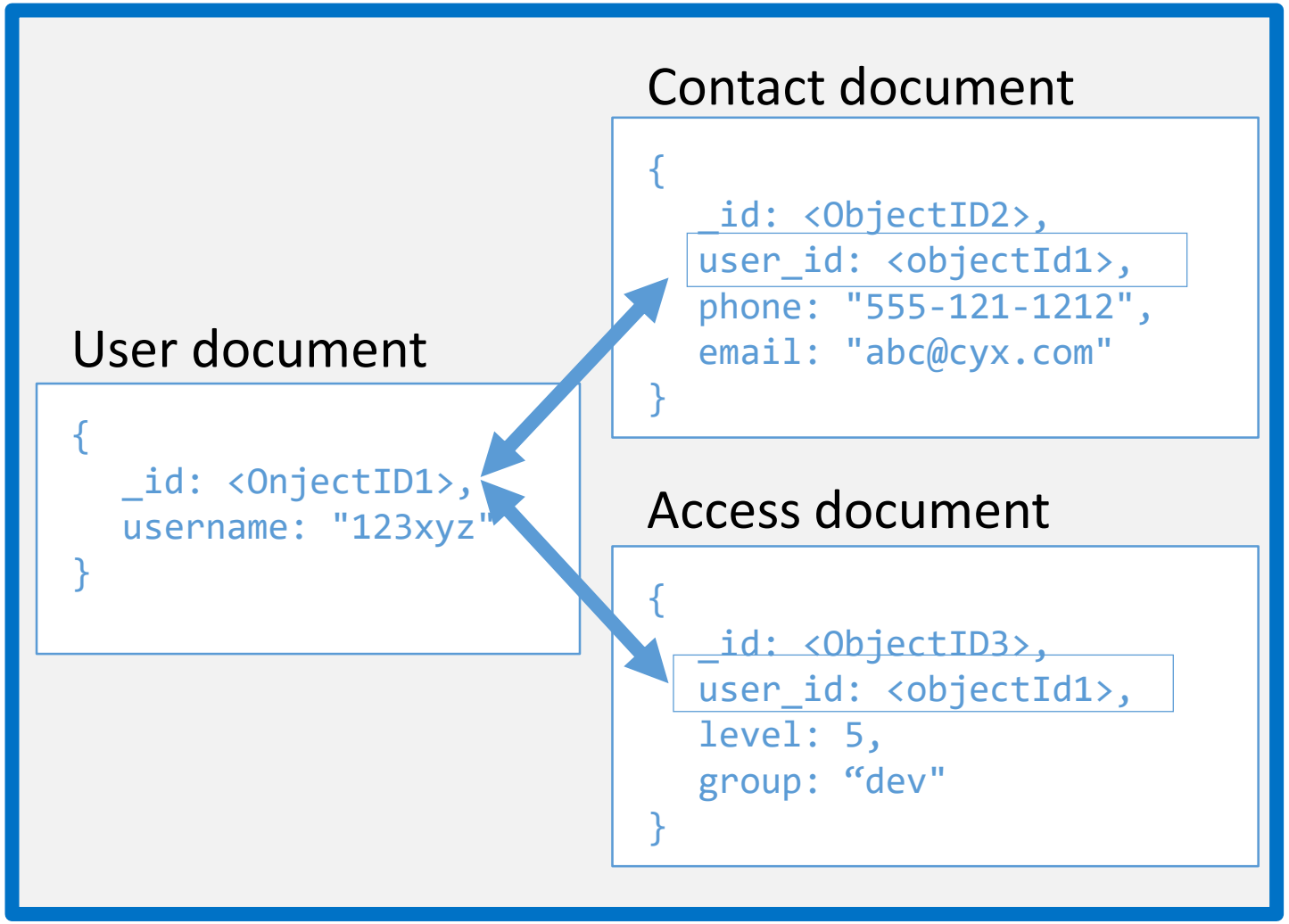
Applications may need to issue fewer queries and updates

Generally, use embedded data models when:

- There are "**contains**" relationships between entities
- There are **one-to-few** relationships between entities
- Embedded data **changes infrequently**
- Embedded data **won't grow** without bound
- Embedded data is **integral** to data in a document

Denormalizing typically provides better **read** performance

# Data modeling with referencing




In general, use normalized data models when:

- Write performance for duplication is more important
- Representing one-to-many relationships
- Representing many-to-many relationships
- Related data changes frequently

Provides more flexibility than embedding

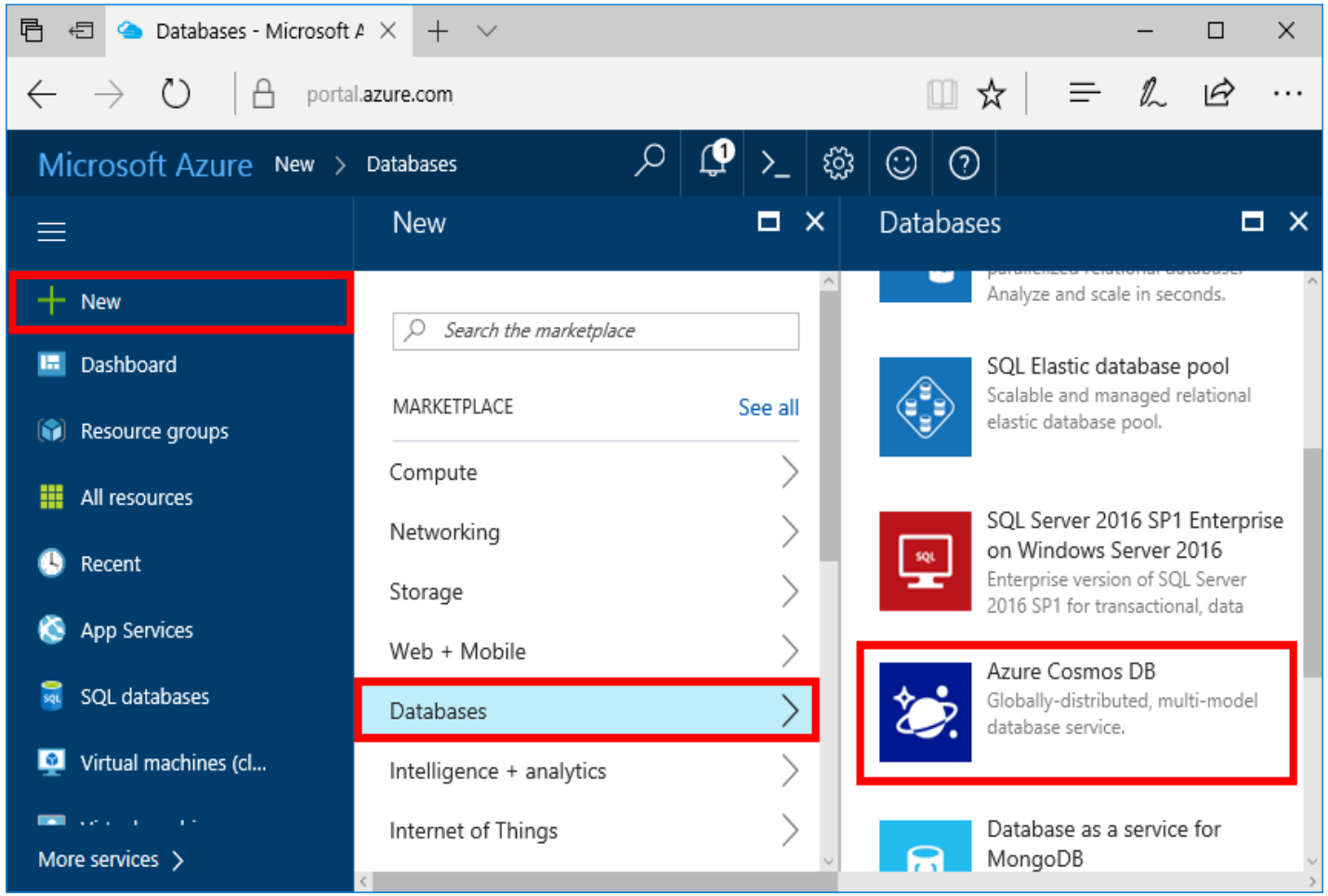
More round trips

Normalizing typically provides better write performance

# Account creation and portal experience

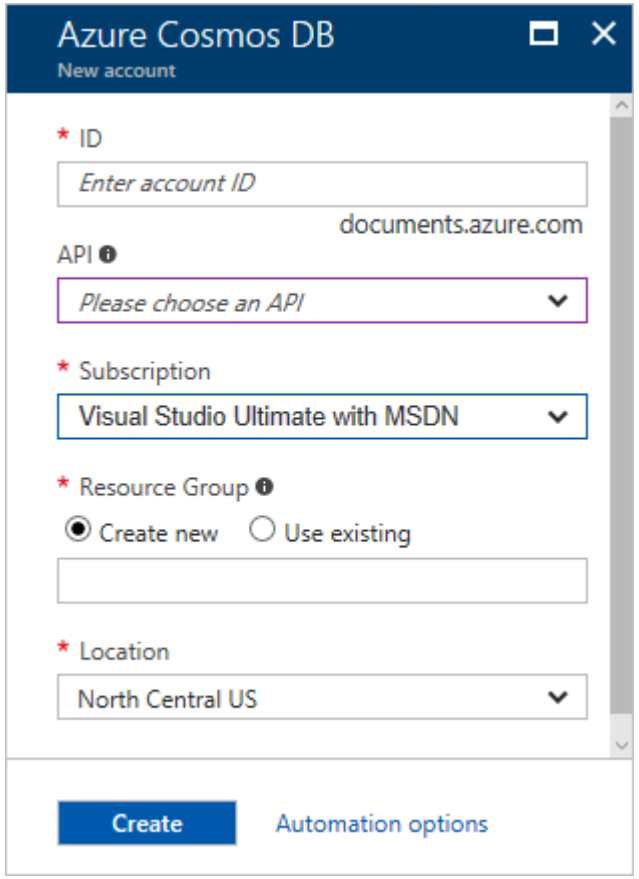
Demo

# Create a new account




Start from the Azure portal to create a Cosmos DB service under your Azure subscription.

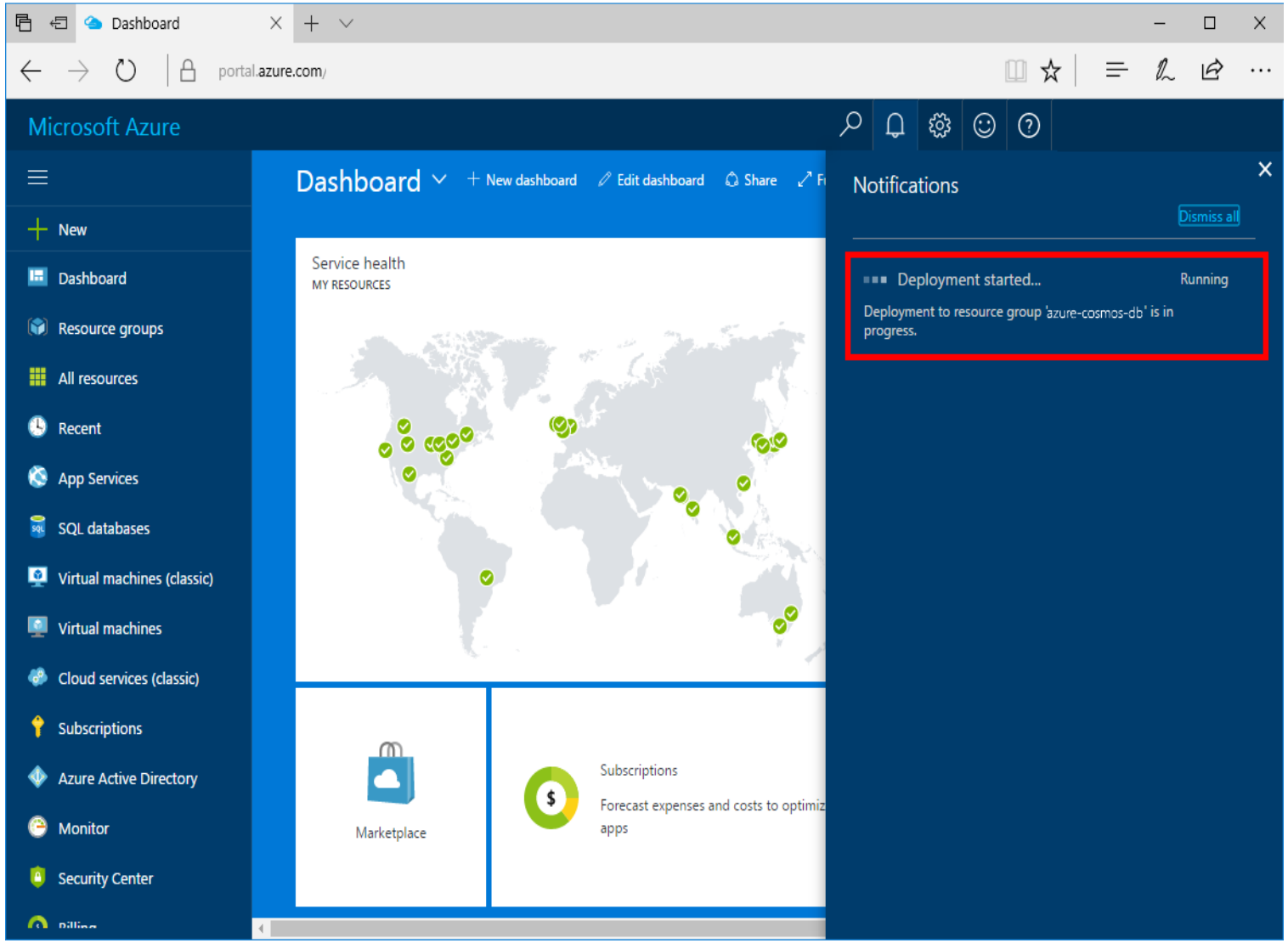
# Provide the service name




Provide a name for your service that will be used as part of the endpoint URI.

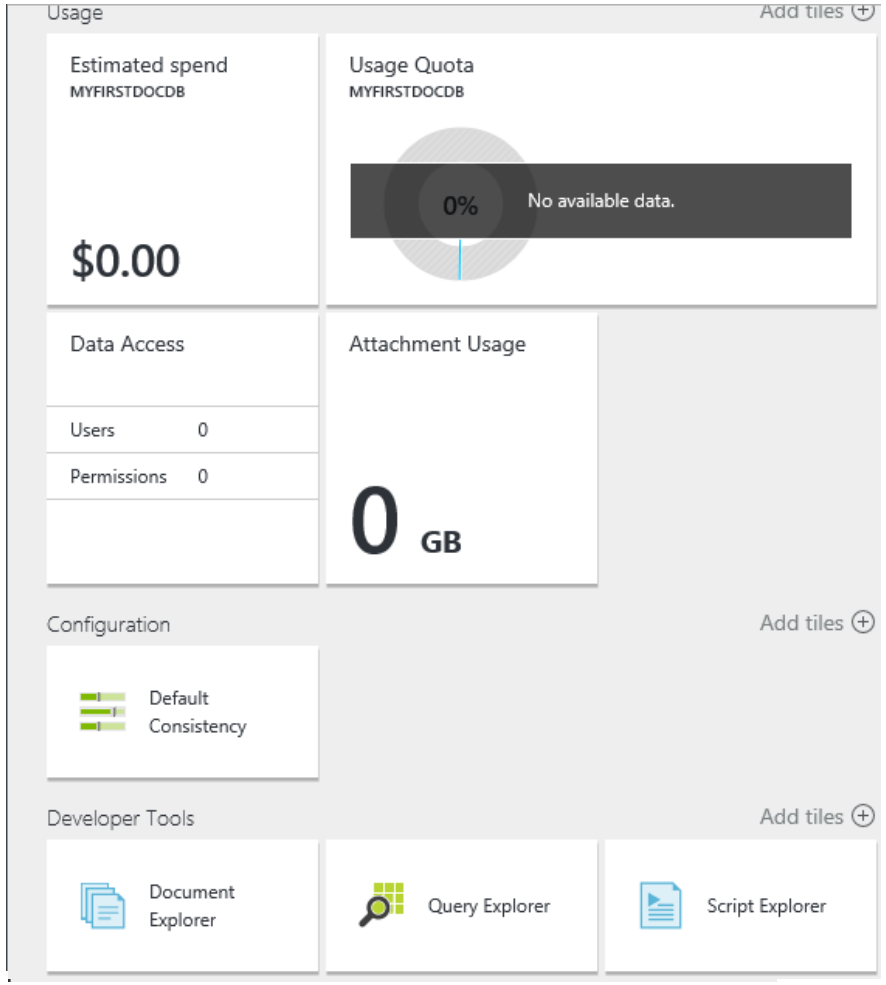
You can also define resource groups, capacity units, and service location.

# Complete provisioning




The Azure portal provides notifications to let you know the provisioning status.

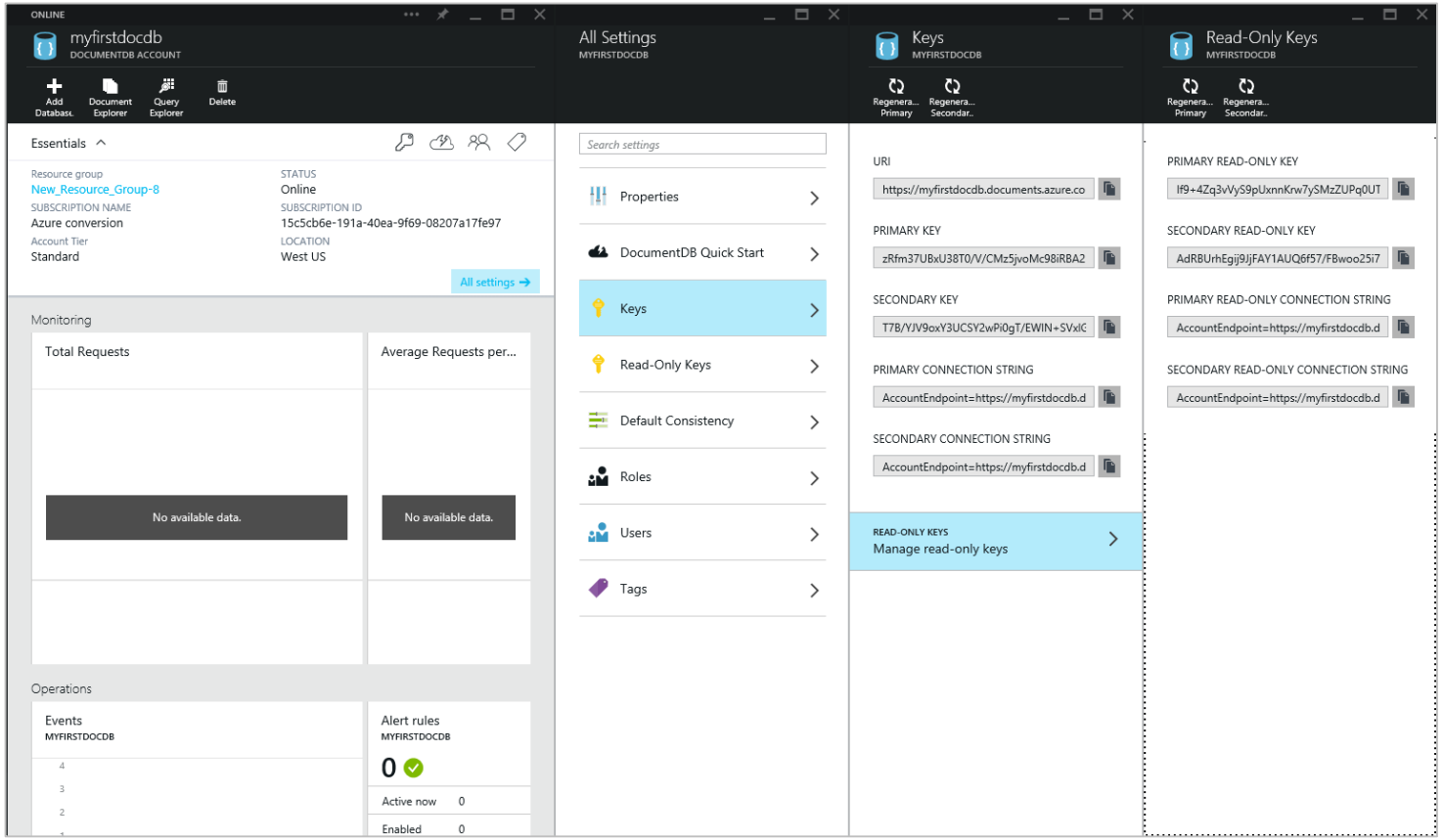
# Cosmos DB account blade




The account blade provides you with essential information about your service.

Check out **Cosmos DB Quick Start** by clicking on **All settings** in the Essentials section of the blade.

# Viewing service keys




Click **All settings** to display the **Keys** and **Read-Only Keys** blades.

Use the URI and service keys in your application for the endpoint and authentication.

Like other services, Cosmos DB includes primary and secondary keys for rolling key updates.

# Rich query over JSON data

Build modern, scalable apps with robust transactional querying and data processing on JSON documents. Unlike other document-database options, Azure Cosmos DB provides a full-featured NoSQL document database service with transactional processing over multiple documents by using SQL-like query grammar and native JavaScript support.

**Query JSON data without specifying secondary indices or constructing views**

Having no forced, pre-defined indices allows for differentiated querying

**Native JavaScript transactional processing**

A native JSON data model enables easy integration with web platforms and tools, making JavaScript the language of Azure Cosmos DB, just as T-SQL is the language of SQL Server

**Familiar SQL-based query language**

Query over multiple documents by using familiar commands

**Parameterized SQL queries**

Now supported in the Azure Cosmos DB REST API and SDKs

# SQL grammar examples

```
-- Nested lookup against index
SELECT B.Author
FROM Books B
WHERE B.Author.Name = "Leo Tolstoy"

-- Transformation, Filters, Array access
SELECT { Name: B.Title, Author: B.Author.Name }
FROM Books B
WHERE B.Price > 10 AND B.Language[0] = "English"

-- Joins, User Defined Functions (UDF)
SELECT CalculateRegionalTax(B.Price, "USA", "WA")
FROM Books B
JOIN L IN B.Languages
WHERE L.Language = "Russian"
```

## Query over heterogeneous documents

Query arbitrary paths, properties, and values without specifying secondary indexes or indexing hints

Execute queries with consistent results in the face of sustained writes

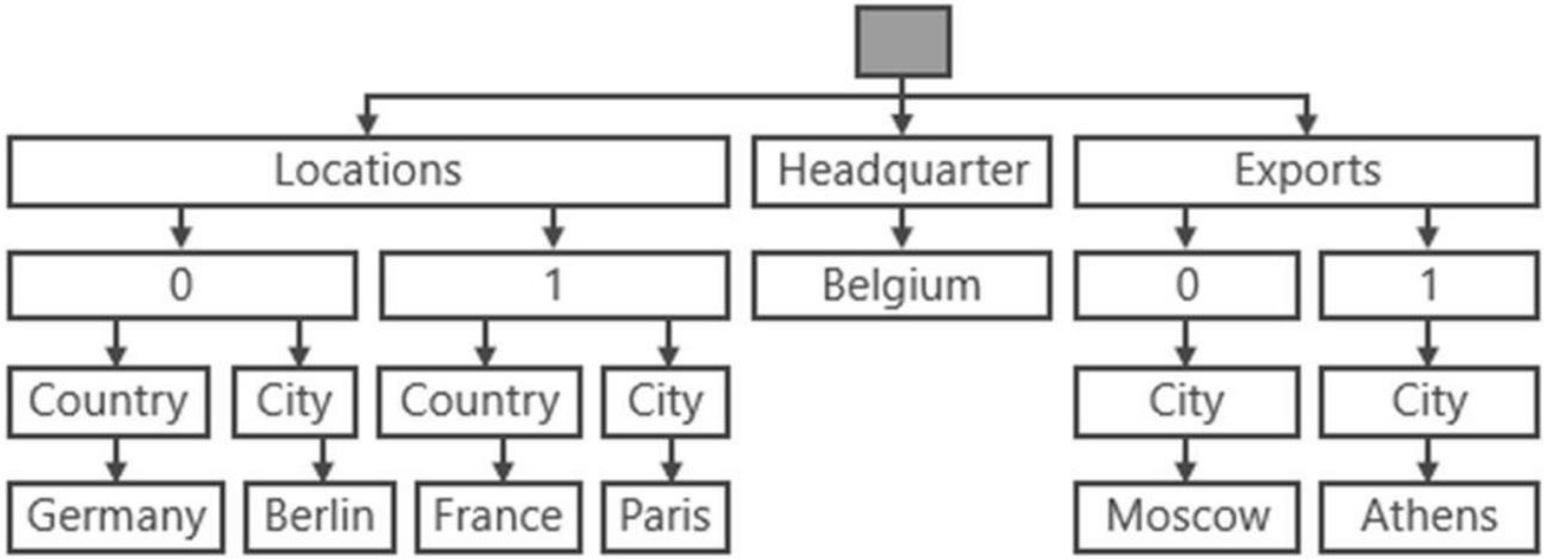
Query through fluent language integration, including LINQ for .NET developers and a “document oriented“ SQL grammar for traditional SQL developers

Extend query execution through application-supplied JavaScript UDFs

Supported SQL features; predicates, iterations (arrays), sub-queries, logical operators, UDFs, intra-document JOINs, or JSON transforms, order by, top, geospatial querying

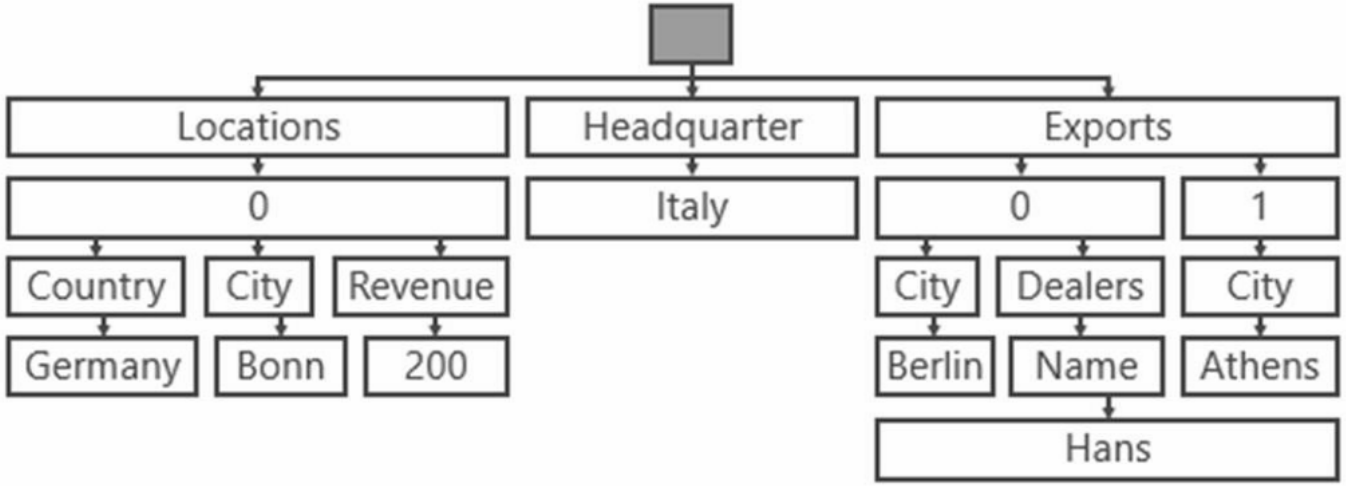
# Query Code

```
var company1 = {
    "locations": [
        {
            "country": "Germany",
            "city": "Berlin"
        },
        {
            "country": "France",
            "city": "Paris"
        }
    ],
    "headquarters": "Belgium",
    "exports": [
        {
            "city": "Moscow"
        },
        {
            "city": "Athens"
        }
    ]
}
```

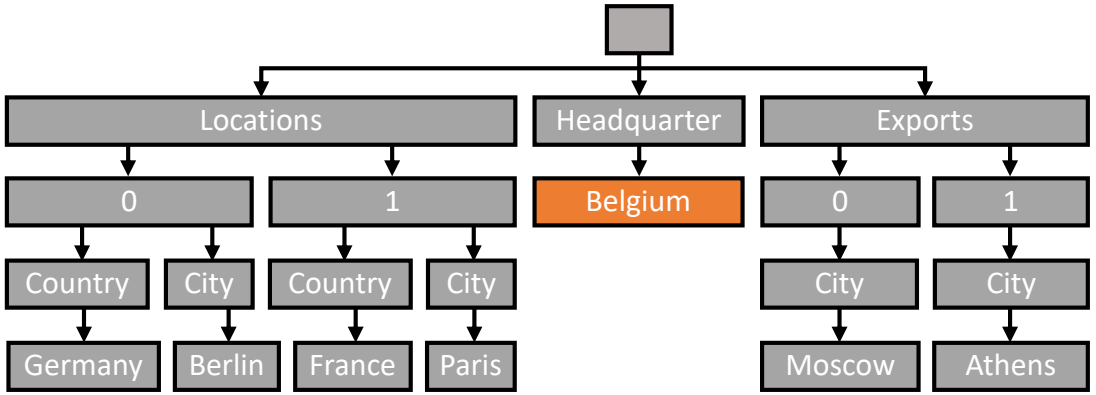


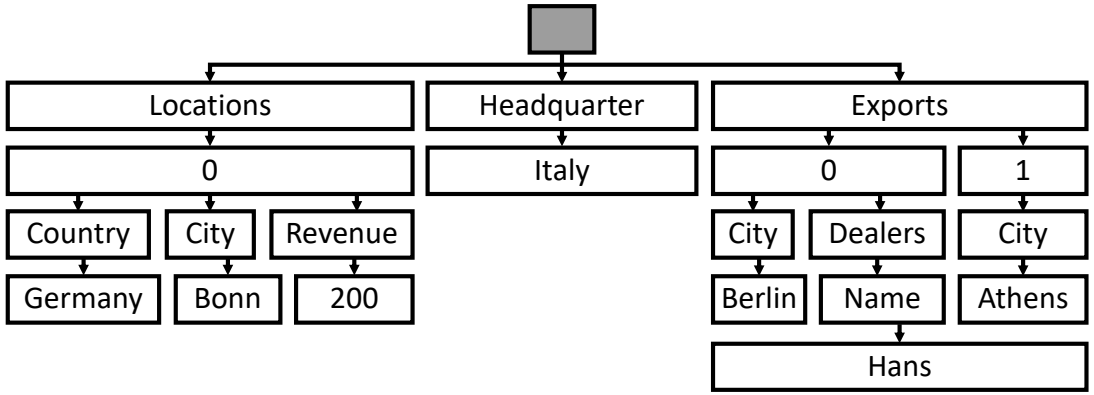
# Query Code

```
var company2 = {
    "locations": [
        {
            "country": "Germany",
            "city": "Bonn",
            "revenue": 200
        }
    ],
    "headquarters": "Italy",
    "exports": [
        {
            "city": "Berlin",
            "dealers": [
                {
                    "name": "Hans"
                }
            ]
        },
        {
            "city": "Athens"
         }
    ]
}
```



# Simple predicate query



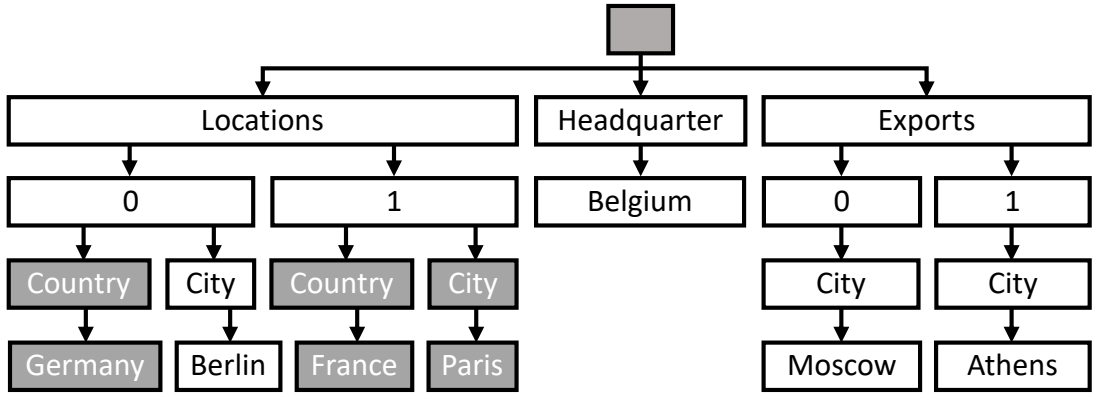

**SQL**

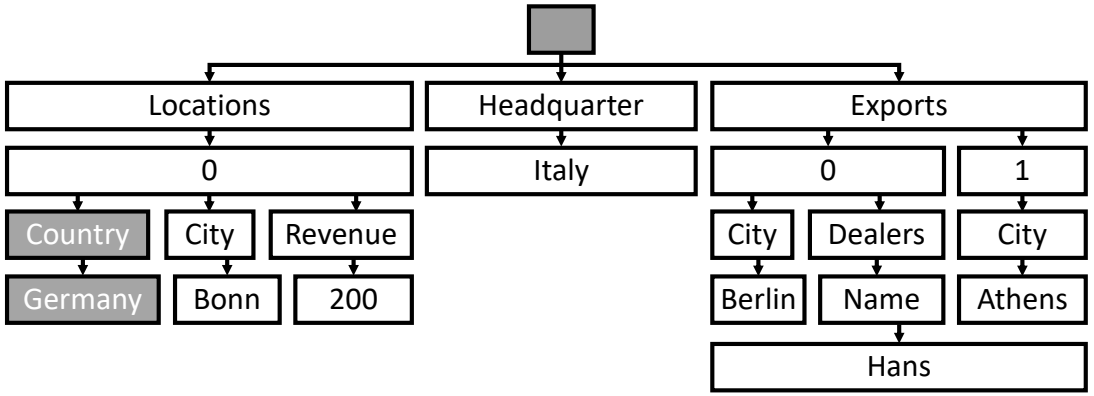
```
SELECT *
FROM company1 C
WHERE C.headquarter = 'Belgium'
```

**Results**

```
[  {
      "locations": [{ "country": "Germany", "city": "Berlin" },{ "country": "France", "city": "Paris" }],
      "headquarter": "Belgium",
      "exports": [{ "city": "Moscow" },{ "city": "Athens" } ]
  }]
```

# Iteration query



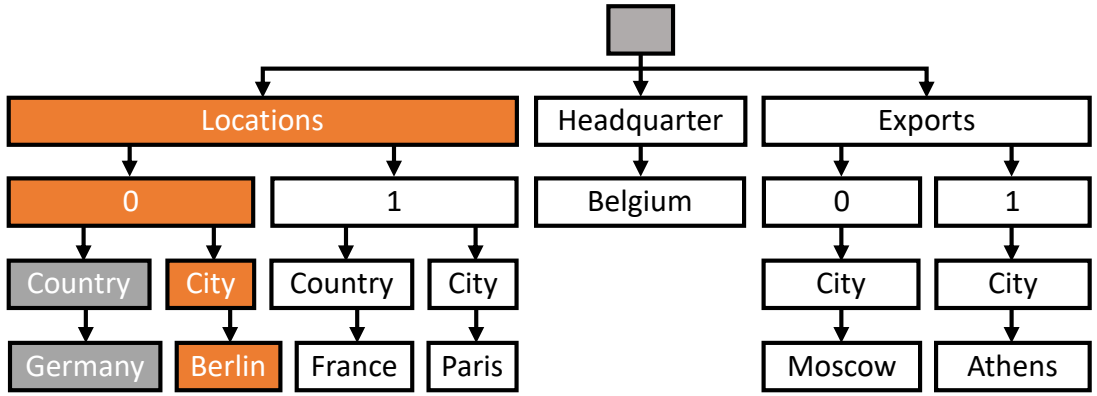

**SQL**

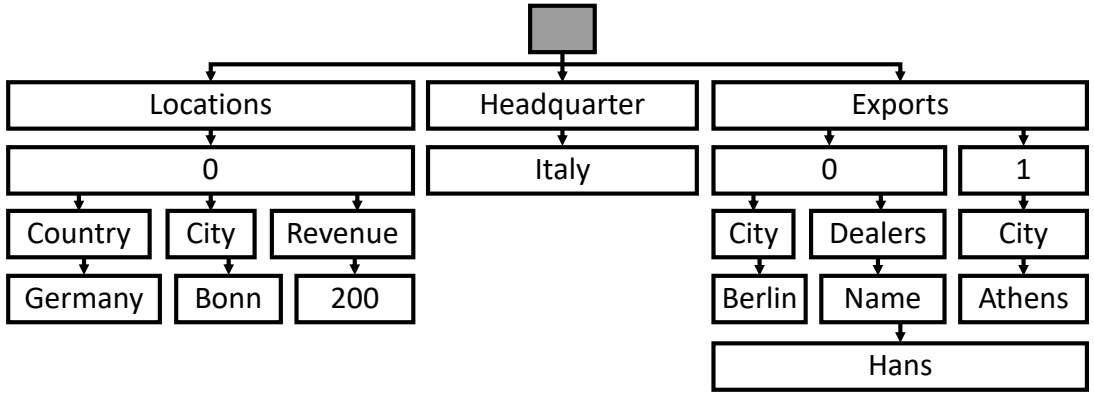
```
SELECT *
FROM company1 C
WHERE C.headquarter = 'Belgium'
```

**Results**

```
[
   {"country": "Germany"},
   {"country": "France"},
   {"country": "Germany"}
]
```

# Querying array with predicates



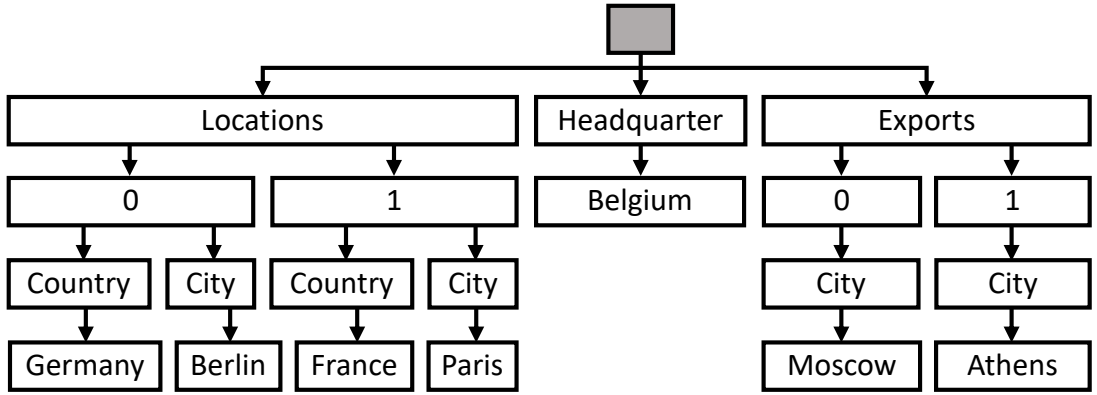

**SQL**

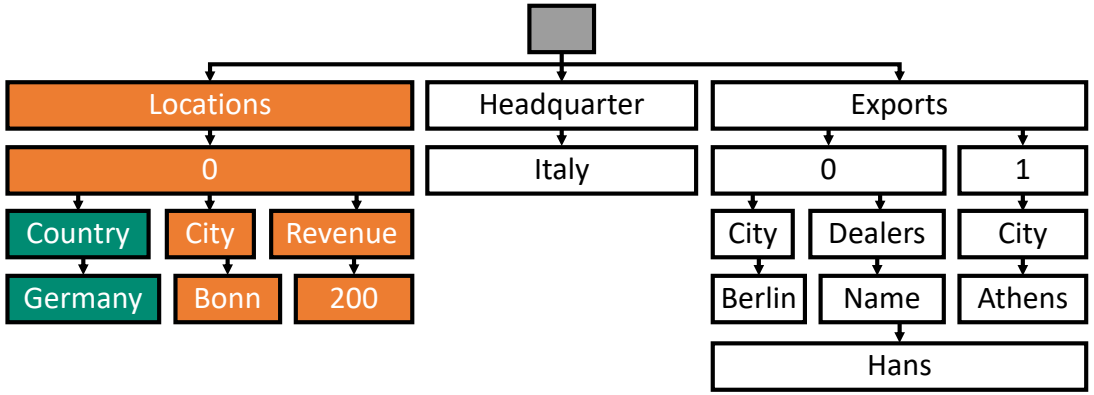
```
SELECT locations.country FROM company1 C JOIN locations IN company2.locations
WHERE locations.city = 'Berlin'
```

**Results**

```
[
   {"country": "Germany"}
]
```

# Cosmos DB query with logical operators



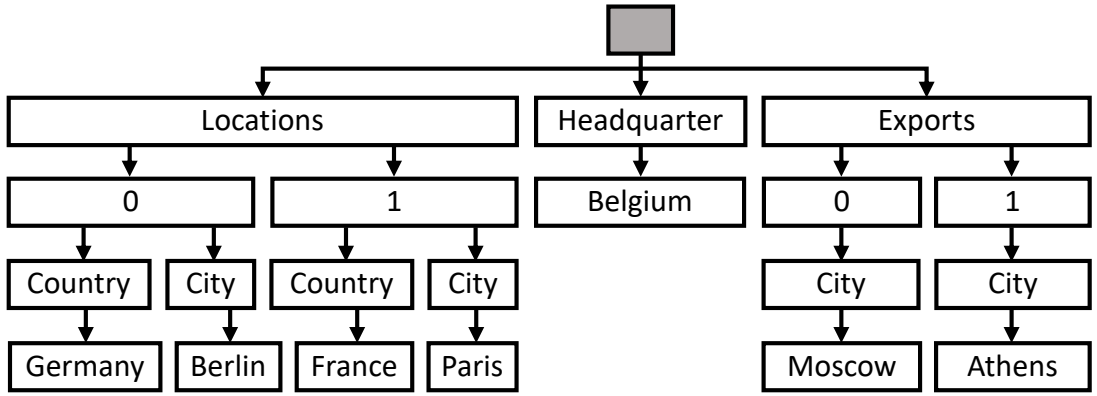

**SQL**

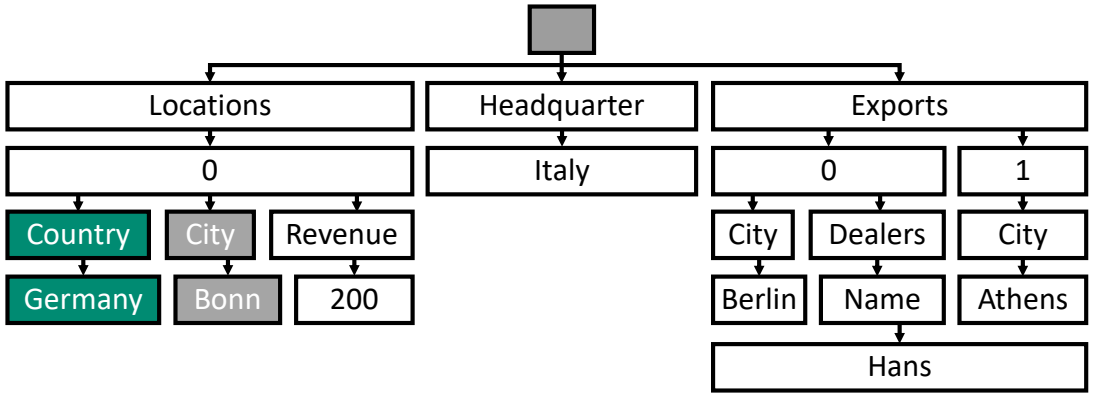
```
SELECT locations.country
FROM locations IN company2.locations
WHERE locations.city='Bonn' AND locations.revenue >100
```

**Results**

```
[
   {"country": "Germany"}
]
```

# Query with UDFs




**SQL**

```
SELECT GermanTax(locations) FROM locations in company2.locations
```

```
function GermanTax(income) {
   if(income < 1000)   return income * 0.1;
   else if(income < 10000) return income * 0.2;
   return income * 0.4;
}
```

**Results**

```
[
   {"country": "Germany", "city":"Bonn", "tax":50},
   {"country": "Germany", "city":"Berlin", "tax":0},
]
```

# Query performance – pagination

When performing a bulk read of documents or issuing a query, the server returns results in a segmented fashion if the result set is too large. In order to reduce round-trips, clients may override the page size.

- Bounded execution time of 5 seconds per batch
- Default page size is 100 (server returns results in chunks of at most 100)
- Desired page size can be specified:
  - Use x-ms-max-item-count request header
  - Use FeedOptions.MaxItemCount property

# Queries, transactions, indexing, and security

# Query with user-defined function

```
// User Defined Function
function tax(doc) {
// Use simple formula to compute the tax: use income multiplied by factor based on
country of headquarters.
    var factor =
        doc.headquarters == "USA" ? 0.35 :
        doc.headquarters == "Germany" ? 0.3 :
        doc.headquarters == "Russia" ? 0.2 :
        0;

    // Check for bad data.
    if (factor == 0)
    {
        throw new Error("Unsupported country: " +
                        doc.headquarters);
    }

    // Use simple formula and return.
    return doc.income * factor;
}

// Execute UDF with additional condition
var results = client.CreateDocumentQuery<dynamic>(colSelfLink, string.Format("SELECT
r.name AS company, Tax(r) AS tax FROM root r WHERE r.type='Company'", udfId));
```

## The complexity of a query impacts the request units consumed for an operation:

### Number of predicates

In general, more predicates result in a larger request charge.

Additional predicates can help if they result in narrowing the overall result set.

### Use of user-defined functions (UDFs)

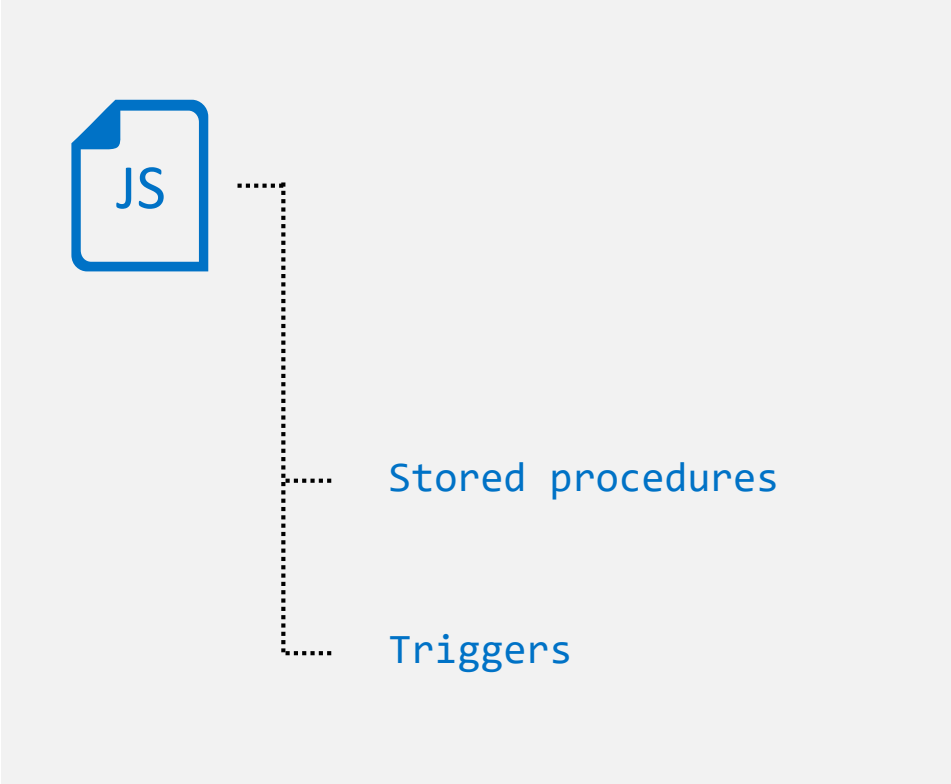
To take advantage of indexing, try and have at least one filter against an indexed property when leveraging a UDF.

Tip: For all operations, the cost of any given request can be inferred by inspecting the x-ms-request-charge response header.

# JavaScript transactions

Transactionally process multiple documents with application-defined stored procedures and triggers

- JavaScript as the procedural language
- Language integrated
- Execution wrapped in an implicit transaction
- Preregistered and scoped to a collection
- Performed with ACID guarantees
- Triggers invoked as pre- or post-operations



# JavaScript transactions

```
client.executeStoredProcedureAsync
 ("procs/1234", "MasterChief", "SolidSnake")
 .then(function (response) {
    console.log("success!");
  }, function (err) {
   console.log("Failed to swap!", error);
 }
);
```

Client

```
function(playerId1, playerId2) {
    var playersToSwap = __.filter()(function (document) {
        return (document.id == playerId1 || document.id == playerId2);
    });
    var player1 = playersToSwap[0], player2 = playersToSwap[1];

    var player1ItemTemp = player1.item;
    player1.item = player2.item;
    player2.item = player1ItemTemp;
    __.replaceDocument(player1)
        .then(function() { return collection.replaceDocument(player2); })
        .fail(function(error){ throw 'Unable to update players, abort';
});
}
```

Database

## Stored procedures and triggers

- Familiar programming model constructs for executing application logic
- Registered as named, URI addressable, durable resources
- Scoped to a Cosmos DB collection
- JavaScript as a procedural language to express business logic

## Language integration

- JavaScript throw statement aborts the transaction

## Execution

- JavaScript runtime is hosted on each replica
- Pre-compiled on registration
- Entire procedure is wrapped in an implicit database transaction
- Execution is fully resource governed and sandboxed

# JavaScript transaction tips

Understand transaction costs, and mind the 5 second rule: batched execution with continuation

Test with expected data volumes

Develop a strategy for versioning and code management

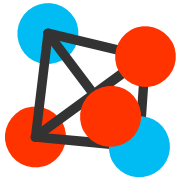
# Four consistency levels

## Lower consistency level on read operations

```
Document myDoc = await
client.ReadDocumentAsync(documentLink, new
RequestOptions { ConsistencyLevel =
ConsistencyLevel.Eventual });
```

Strong

Session

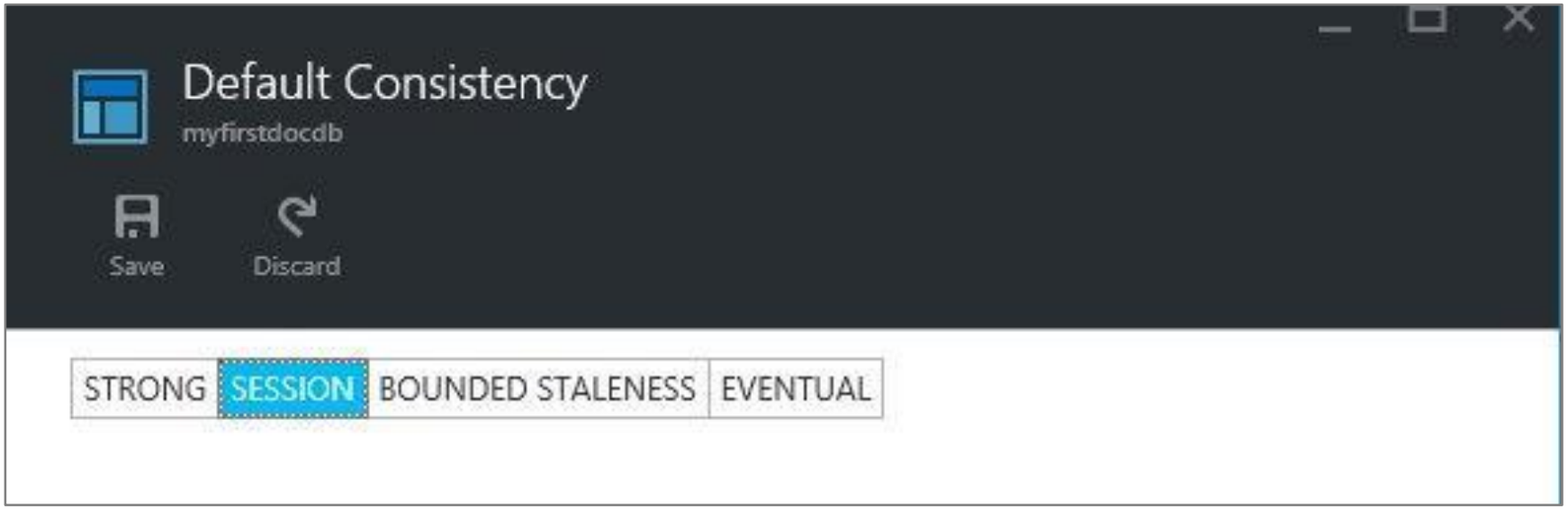


Bounded Staleness


Eventual

# Consistency levels enable guarantees

Choose your consistency level and make predictable trade-offs between consistency, availability, and performance




Choose your level

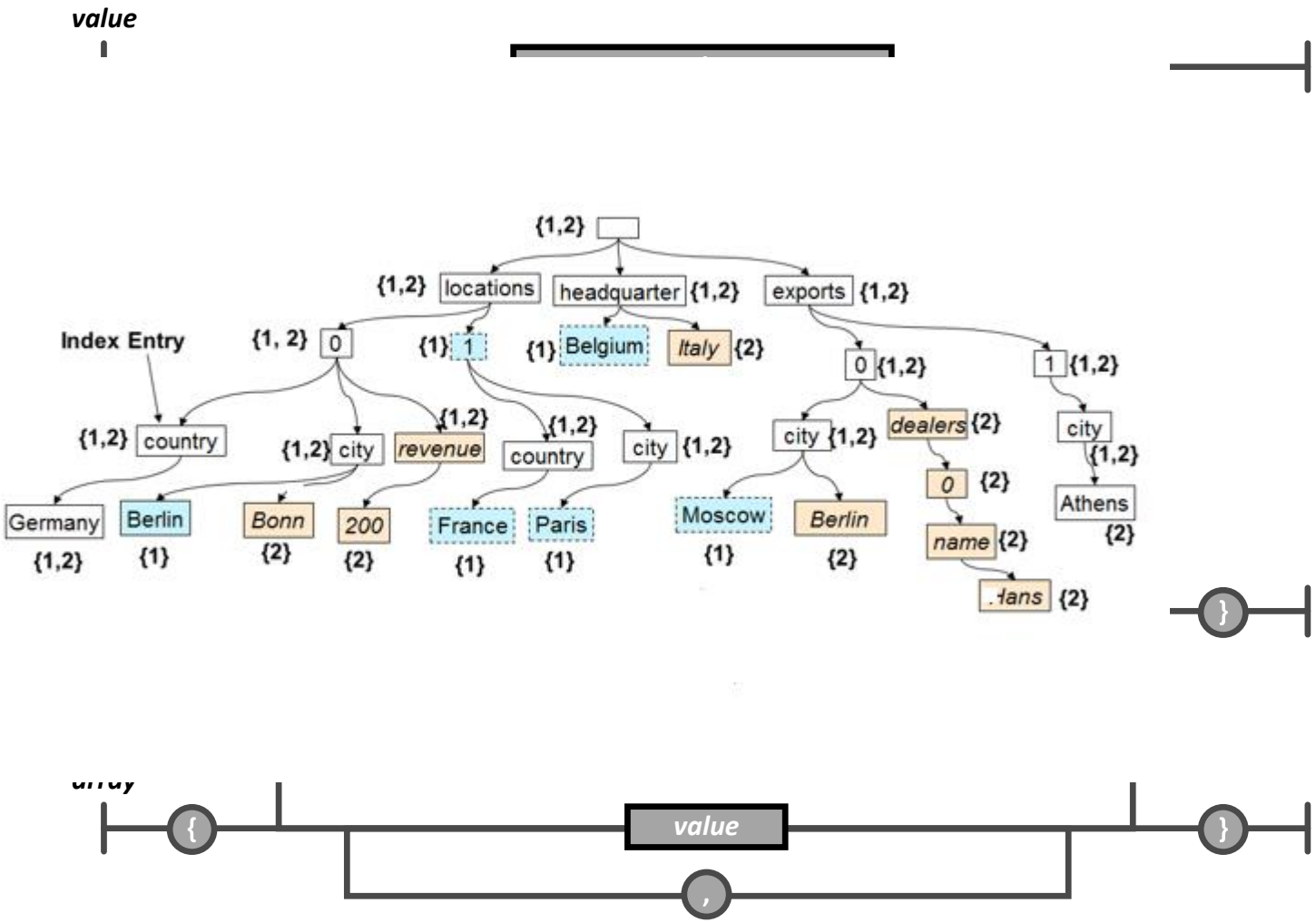
**Strong**
Data consistency

**Session**
Monotonic reads (on explicit read requests) and writes

**Bounded Staleness**
Total order of propagation of writes

**Eventual**
Lowest latency for reads and writes

# Indexing




## The “write” index for consistent queries

- Highly concurrent, lock-free, log-structured indexing technology developed with Microsoft Research
- Optimized for solid-state drives (SSDs)
- Resource-governed for tenant isolation

## Automatic indexing of JSON documents without requiring schema or secondary indices, but configurable via:

- Modes
- Paths
- Types

# Indexing – modes and policies

## Set indexing mode

```
var collection = new DocumentCollection
            {
                Id = “lazyCollection”
            };

            collection.IndexingPolicy.IndexingMode = IndexingMode.Lazy;
            collection = await
client.CreateDocumentCollectionAsync(databaseLink, collection);
```

## Set indexing policy

```
var collection = new DocumentCollection
            {
                Id = “manualCollection”
            };

            collection.IndexingPolicy.Automatic = false;

            collection = await
client.CreateDocumentCollectionAsync(databaseLink, collection);
```

## Indexing modes

- Consistent
  - Default mode
  - Index updated synchronously on writes
- Lazy
  - Useful for bulk ingestion scenarios

## Indexing policies

- Automatic
  - Default
- Manual
  - Can choose to index documents via RequestOptions
    - IndexingDirective = IndexingDirective.Include
  - Can read non-indexed documents via selflink

# Indexing – paths and types

## Setting paths, types, and precision

```
var collection = new DocumentCollection
            {
                Id =  “Orders”
            };

            collection.IndexingPolicy.IncludedPaths.Add(new IndexingPath
            {
                IndexType = IndexType.Hash,
                Precision = -1
                Path = "/",
            });

            collection.IndexingPolicy.IncludedPaths.Add(new IndexingPath
            {
                IndexType = IndexType.Range,
                Path = @"/""shippedTimestamp""/?",
                NumericPrecision = 7
            });

            collection.IndexingPolicy.ExcludedPaths.Add("/\"metaData\"/*");

            collection = await client.CreateDocumentCollectionAsync(databaseLink, collection);
```

### Index paths

Included

Excluded

### Index types

Hash

Supported for strings and numbers

Optimized for equality matches

### Range

Supported for numbers

Optimized for comparison queries

### Index precision

String precision

Default is 3

### Numeric precision

Default is 3

Increase for larger number fields (epoch timestamps)

# Query and indexing tips

Consider query needs and index policies (index policies are immutable, for now)

Understand query costs and limits, and avoid scans

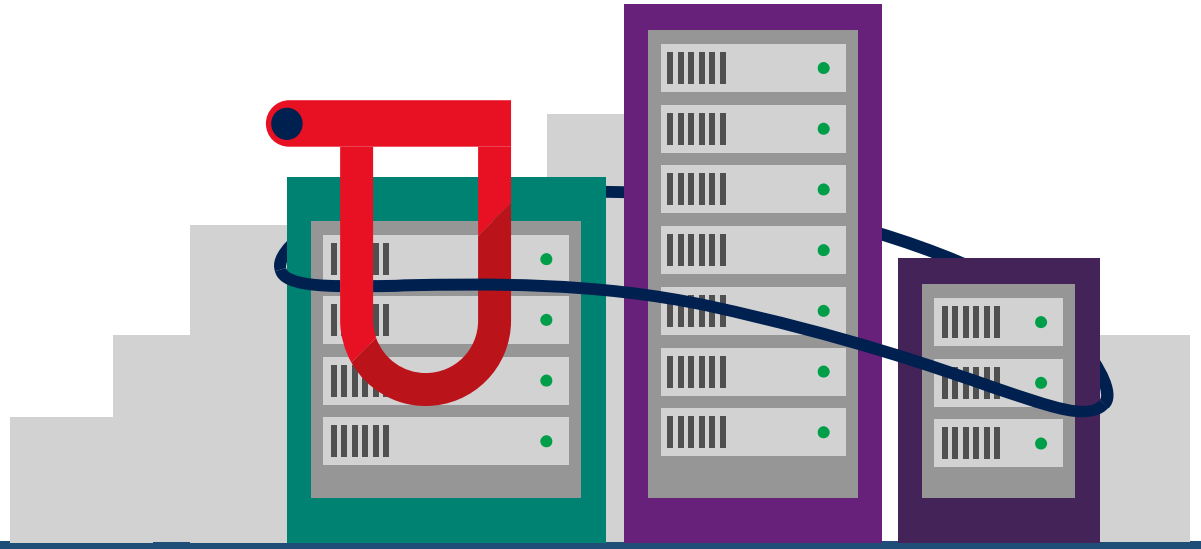
Pre-aggregate where possible

You should use the default indexing policy in most scenarios unless you can see a significant difference in RUs for your workload

# Security model

Azure Document DB is designed to be secure with:

- Master key
- Access control on resources
- User operations
- Permission operations
- Code execution



# Reliable and predictable performance

Cosmos DB is born in the cloud to achieve fast, predictable performance with reserved resources to deliver on your throughput needs. Benefit from reliable, tunable consistency to increase performance based on application needs.

### Fast, predictable performance

Defined throughput levels that scale linearly with application needs

### Tunable consistency

Tune and trade off consistency and performance through well-defined levels to suit application scenario needs—from eventual to strong

### Elastic scale

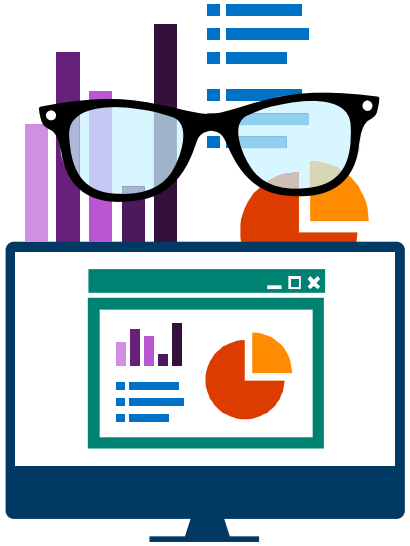
Enterprise-tested by a large internal consumer-facing service

# Management

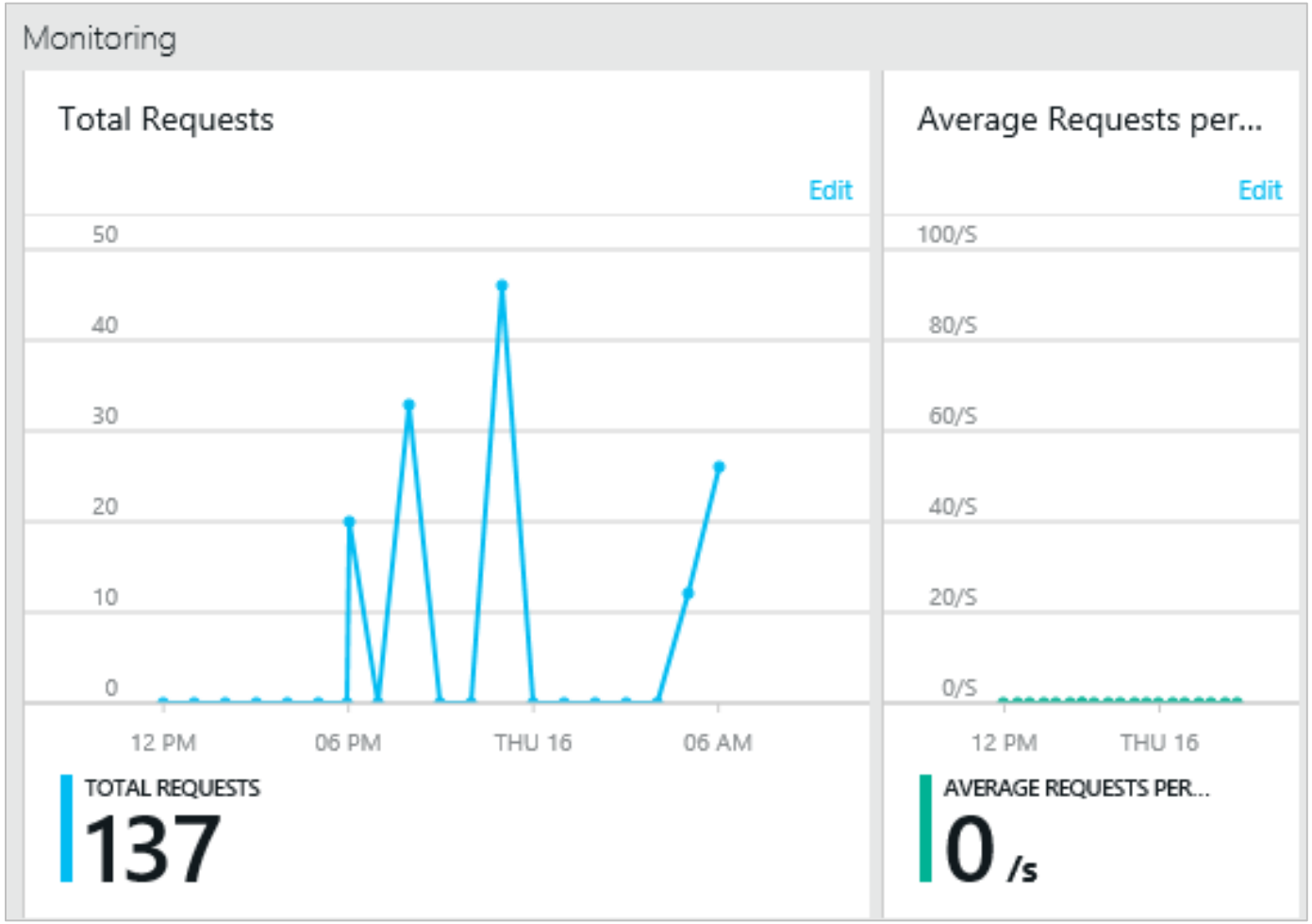
# Monitor an account

How to:

- View performance metrics for a Cosmos DB account
- Customize performance metric views for a Cosmos DB account
- Create side-by-side performance metric charts
- View usage metrics for a Cosmos DB account
- Set up performance metric alerts for a Cosmos DB account



# View performance metrics



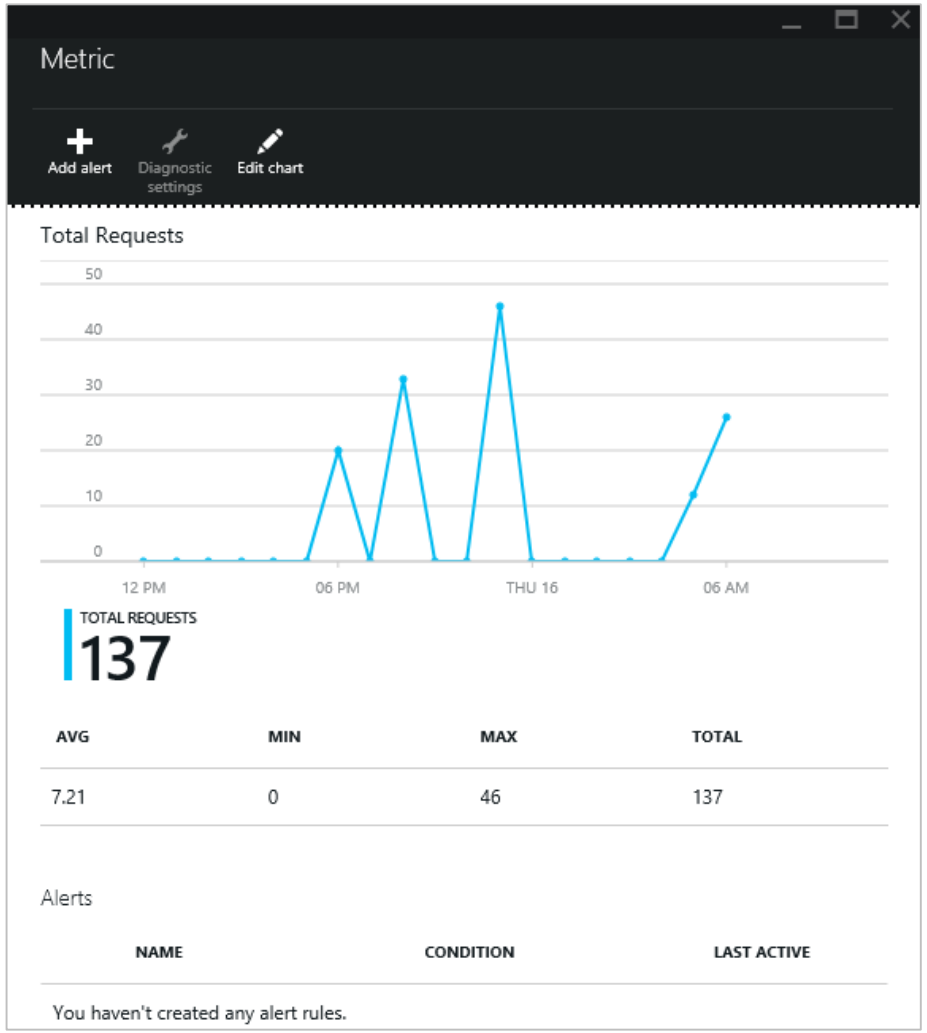

In the Azure Portal, click **Browse**, **Cosmos DB Accounts**, and then click the name of the Cosmos DB account for which you would like to view performance metrics.

Within the Monitoring lens you can, by default, see:

Total requests for the current day

Average requests per second for the current day

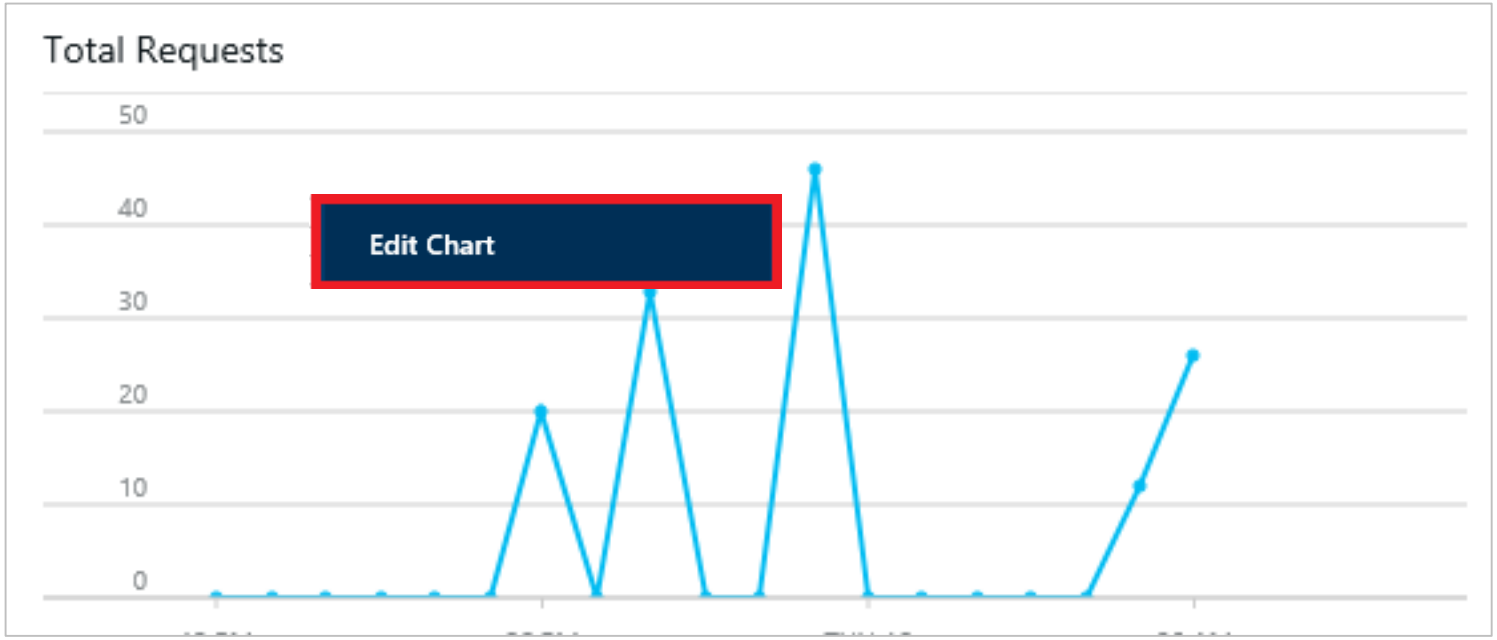
# View total requests




To open a detailed **Metric** blade, click **Total Requests** or **Average Requests per Second**.

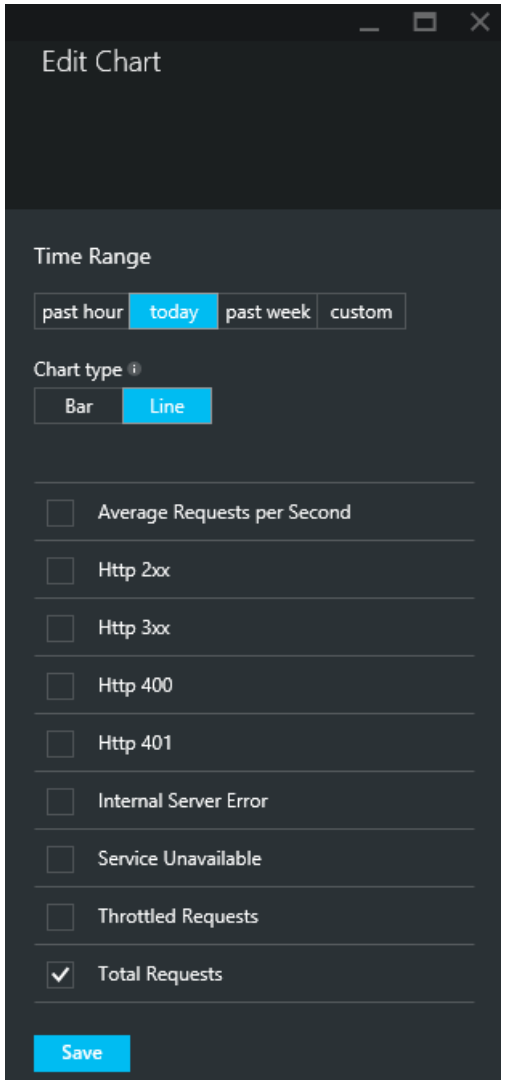
The **Metric** blade shows you details about the metrics that you have selected. At the top of the blade is a graph, and below that is a table that shows aggregation values of the selected metrics, such as average, minimum, and maximum. The metric blade also shows the list of alerts that have been defined, filtered to the metrics that appear on the current metric blade (that way, if you have a number of alerts, you'll only see the relevant ones presented here).

# Customize performance metric views



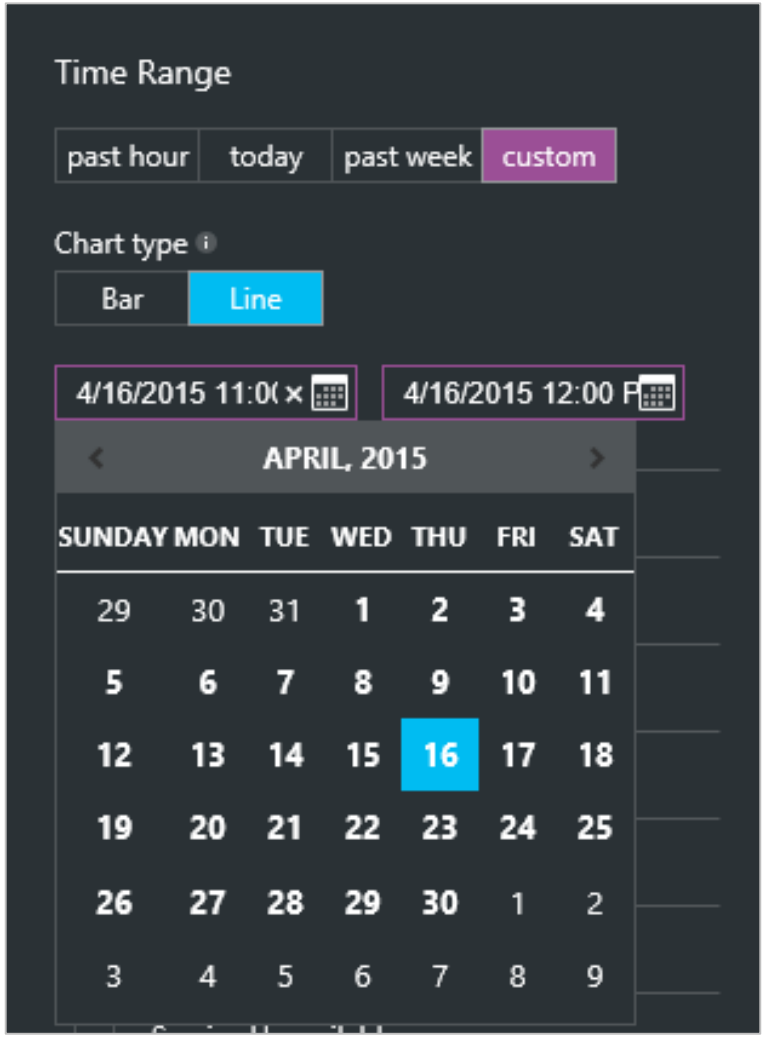

To customize the metrics that display in a particular part, right-click the metric chart, and then select **Edit Chart**.

# Customize performance metric views




On the **Edit Chart** blade, there are options to modify the metrics that display in the part, as well as their time range.

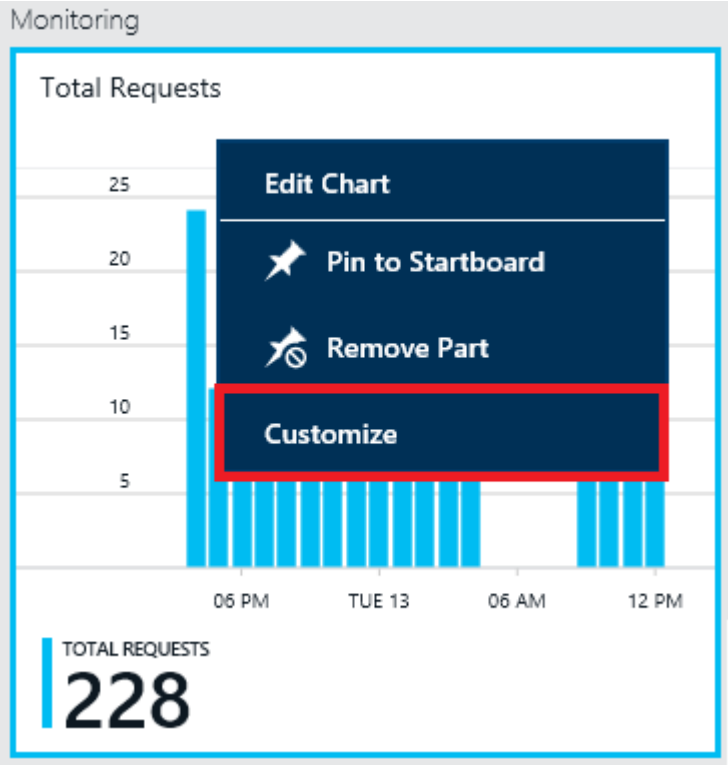
# Customize performance metric views




To change the metrics displayed in the part, simply select or clear the available performance metrics, and then click **Save** at the bottom of the blade.

To change the time range, choose a different range (for example, **Past Hour**), and then click **Save** at the bottom of the blade.

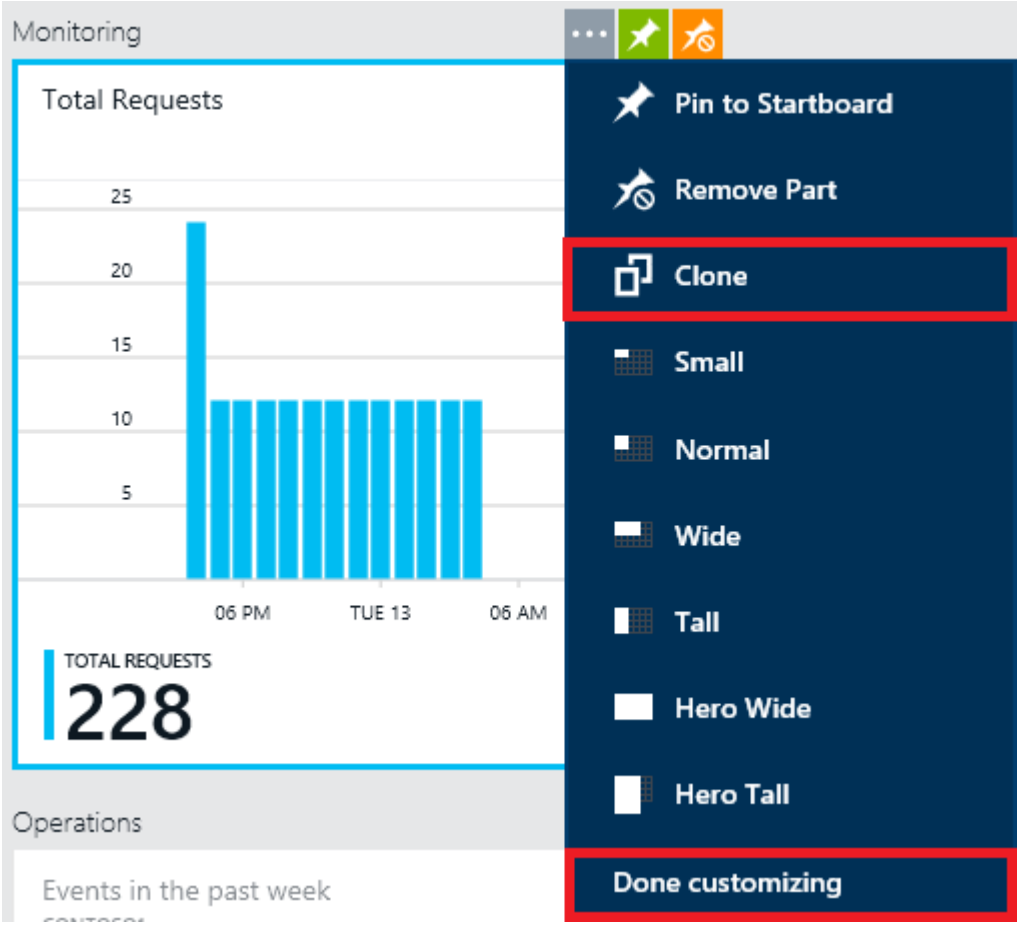
# Create side-by-side performance metric charts




The Azure Portal allows you to create side-by-side metric charts.

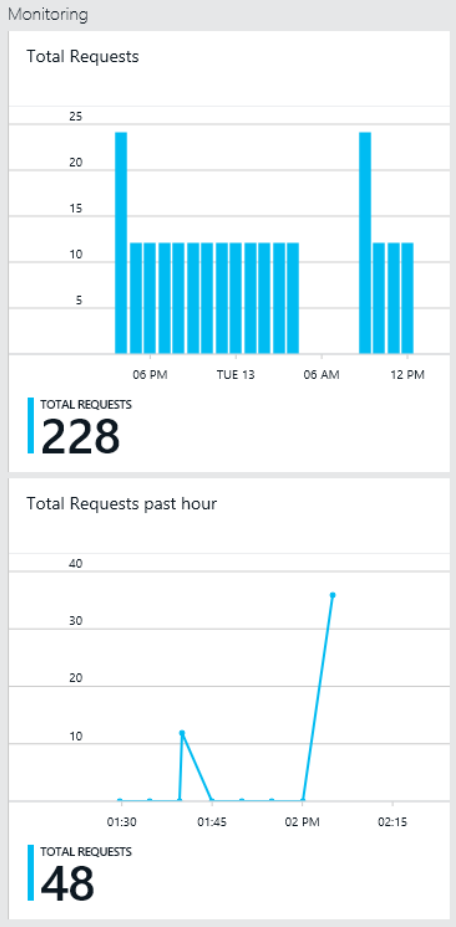
Right-click on the chart you want to clone or modify and then click **Customize**.

# Create side-by-side performance metric charts



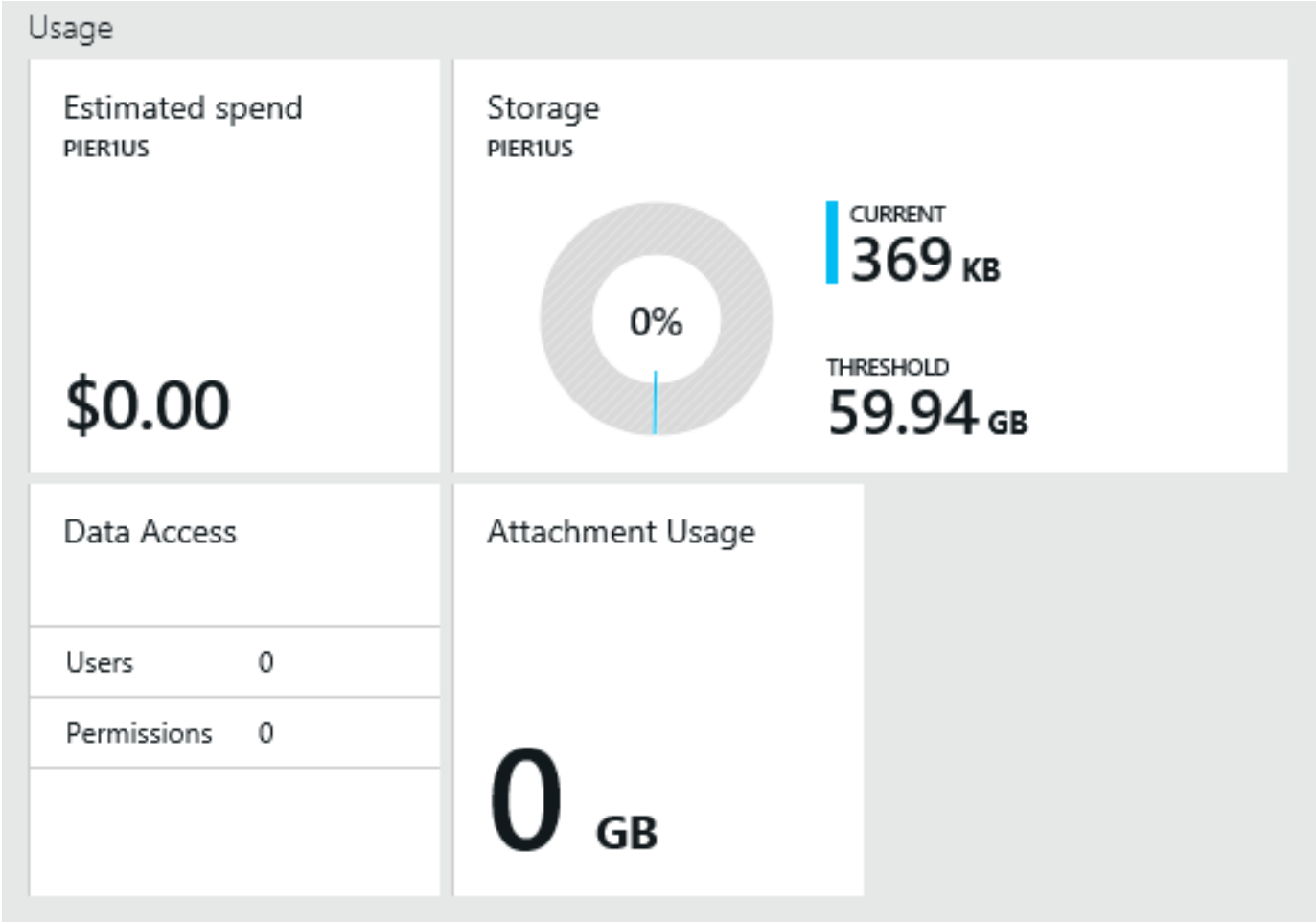

Click **Clone** on the menu to copy the part, and then click **Done customizing**.

# Create side-by-side performance metric charts




You may now treat this part as any other metric part, customizing the metrics and time range displayed in the part. This allows you to see two different metrics charts side-by-side at the same time.

# View usage metrics




In the Azure Portal, click **Browse**, **Cosmos DB Accounts**, and then click the name of the Cosmos DB account for which you would like to see usage metrics.

Within the **Usage** lens you can view the following by default:

Estimated cost

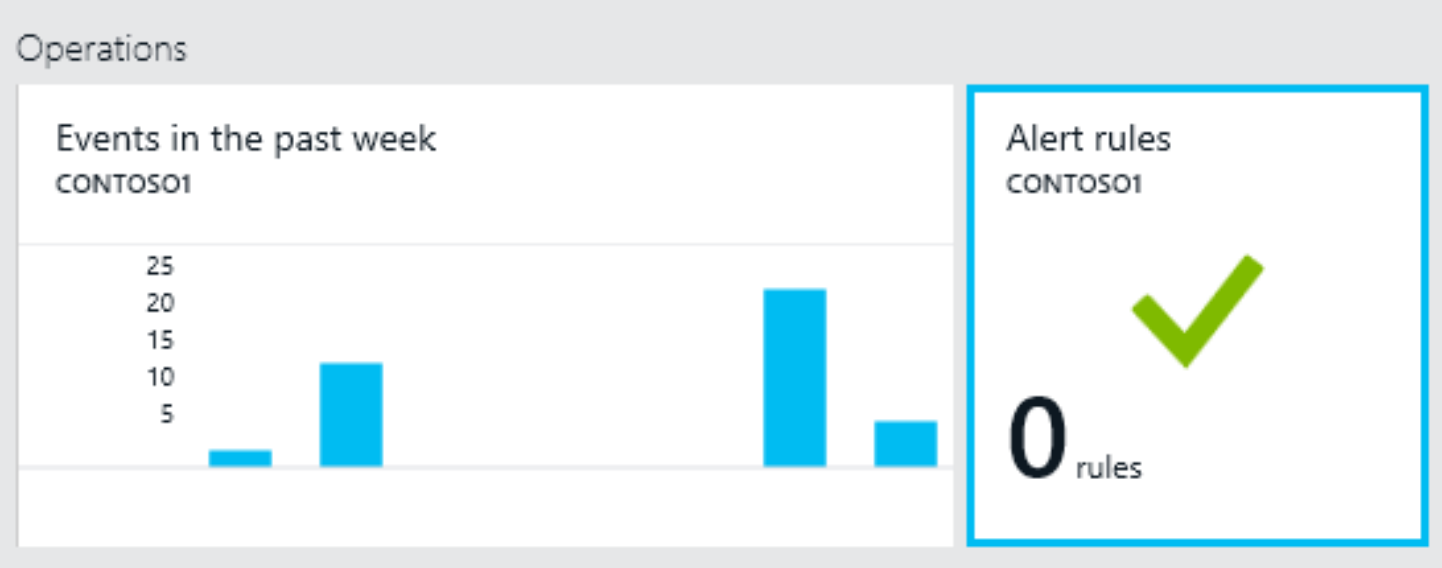
Storage consumed within the account

Maximum available storage of the account

User information

Attachment usage

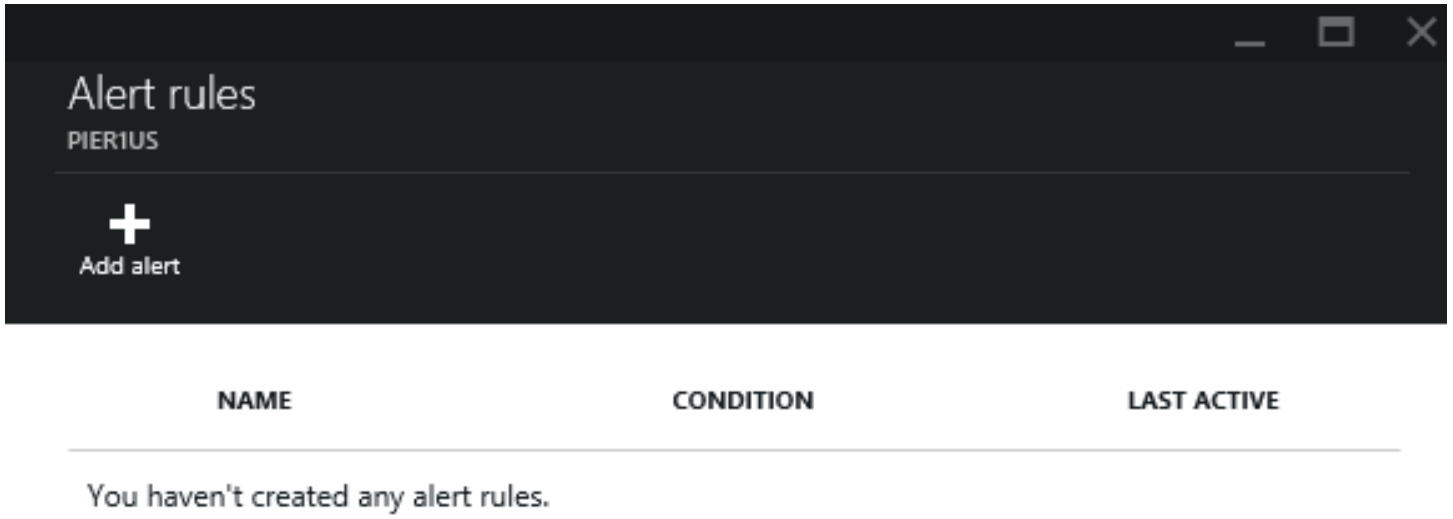
# Set up performance metric alerts




In the Azure Portal, click **Browse**, **Cosmos DB Accounts**, and then click the name of the Cosmos DB account for which you would like to set up performance metric alerts.

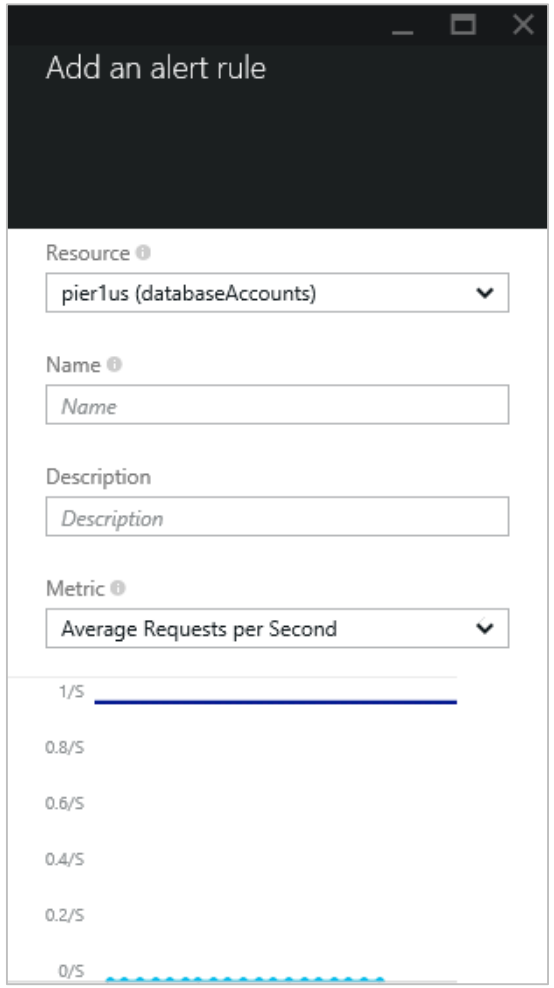
Within the **Operations** lens, click the **Alert rules** part.

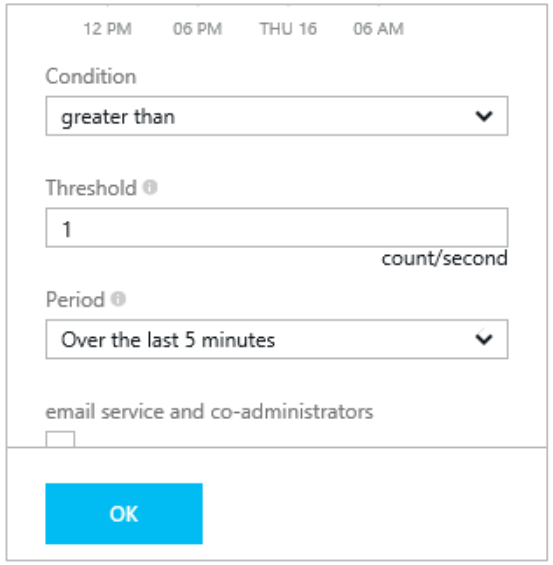
# Set up performance metric alerts




In the **Alert rules** blade, click **Add Alert**.

# Set up performance metric alerts




## In the **Add an alert rule** blade, specify:

The name of the alert rule you are setting up.

A description of the new alert rule.

The metric for the alert rule.

The condition, threshold, and period that determine when the alert activates. For example, a server error count greater than 5 over the last 15 minutes.

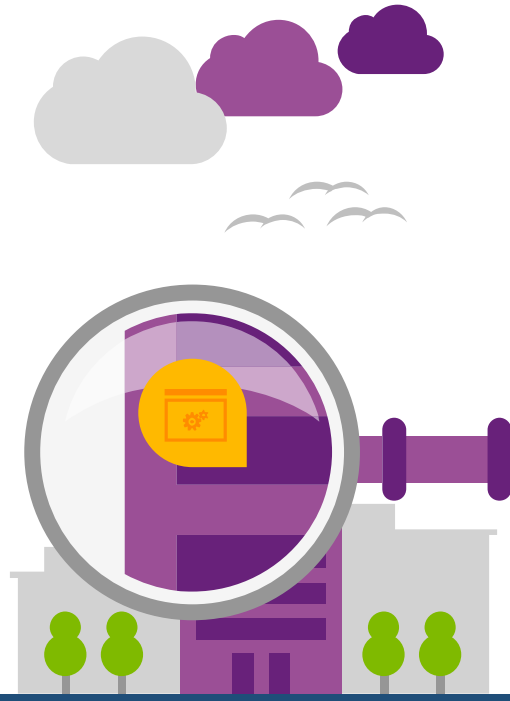
Whether the service administrator and co-administrators are emailed when the alert fires.

Additional email addresses for alert notifications.

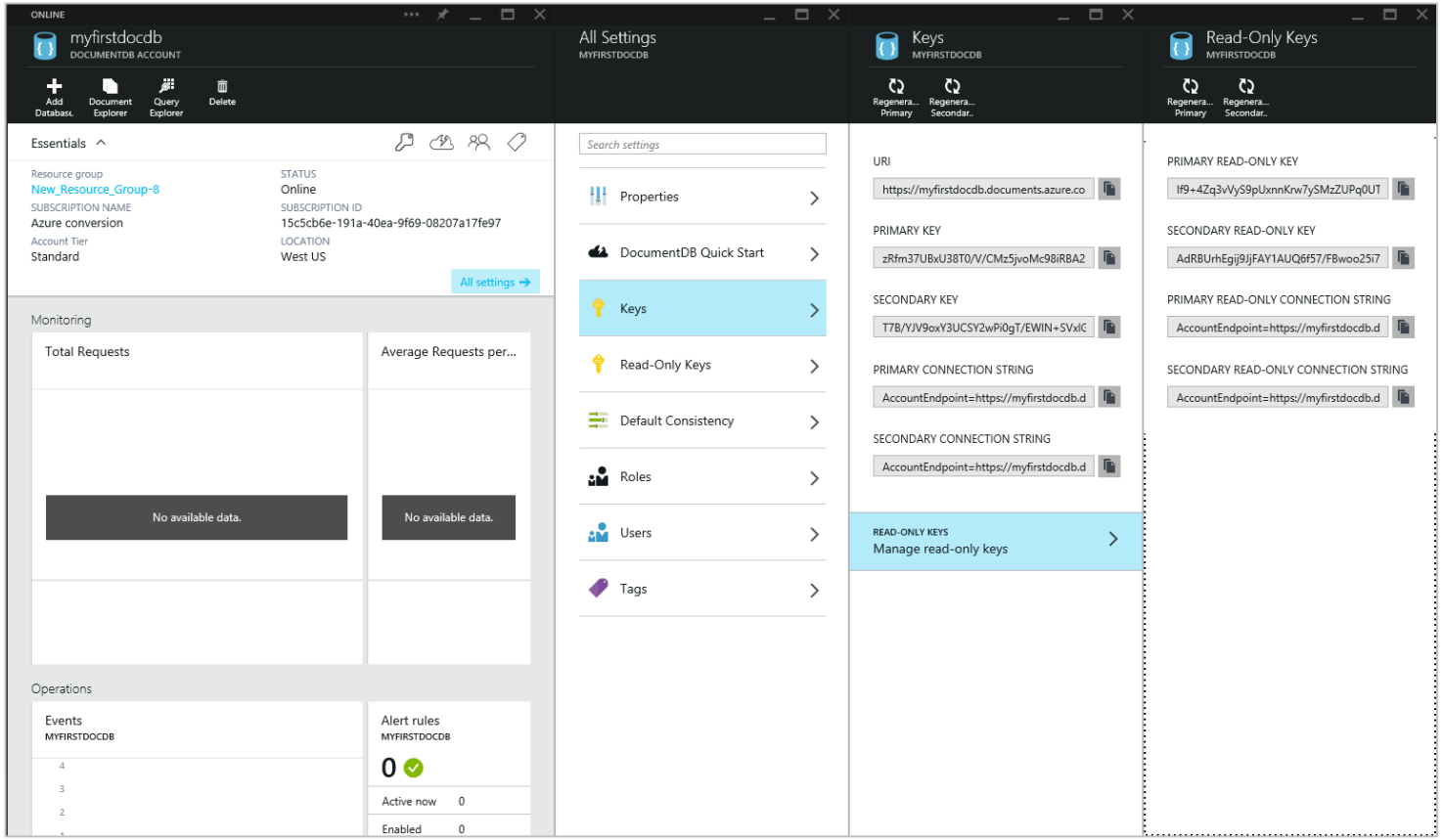
# Manage an account

How to:

- View, copy, and regenerate Cosmos DB access keys
- Manage Cosmos DB consistency settings
- Manage Cosmos DB capacity settings
- Delete a Cosmos DB account



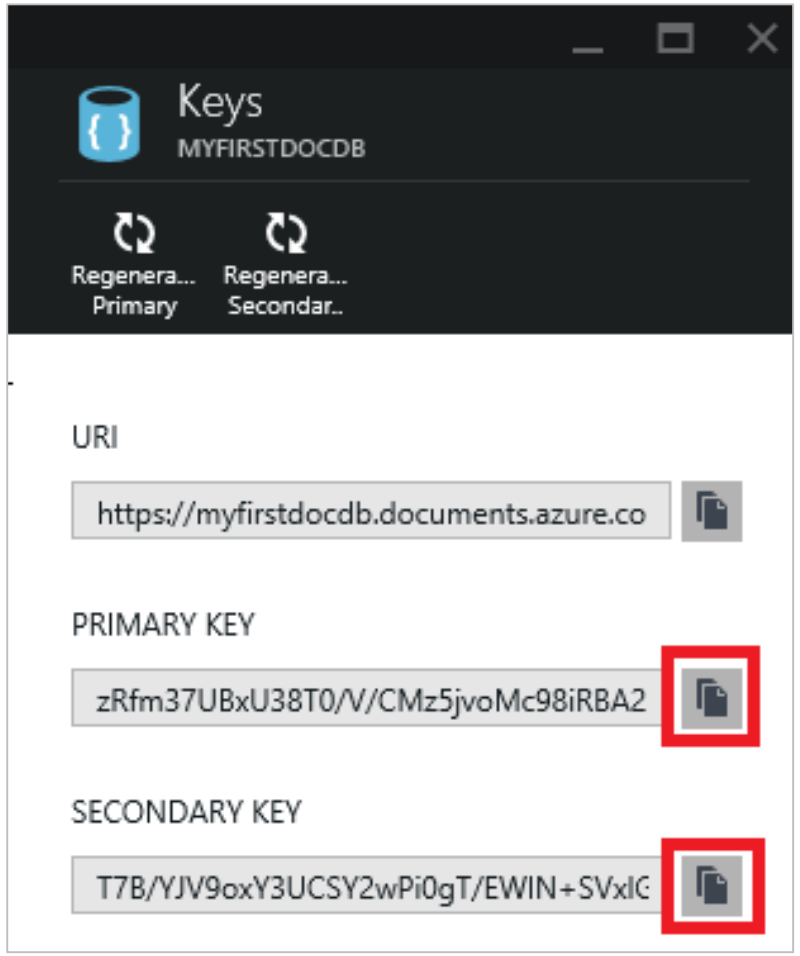
# View, copy, and regenerate access keys




When you create a Cosmos DB account, the service generates two master access keys that can be used for authentication when the Cosmos DB account is accessed. By providing two access keys, Cosmos DB enables you to regenerate the keys with no interruption to your Cosmos DB account.

In the Azure management preview portal, access the **Keys** part from your **Cosmos DB Account** blade **All settings** command to view, copy, and regenerate the access keys that are used to access your Cosmos DB account.

# View and copy an access key

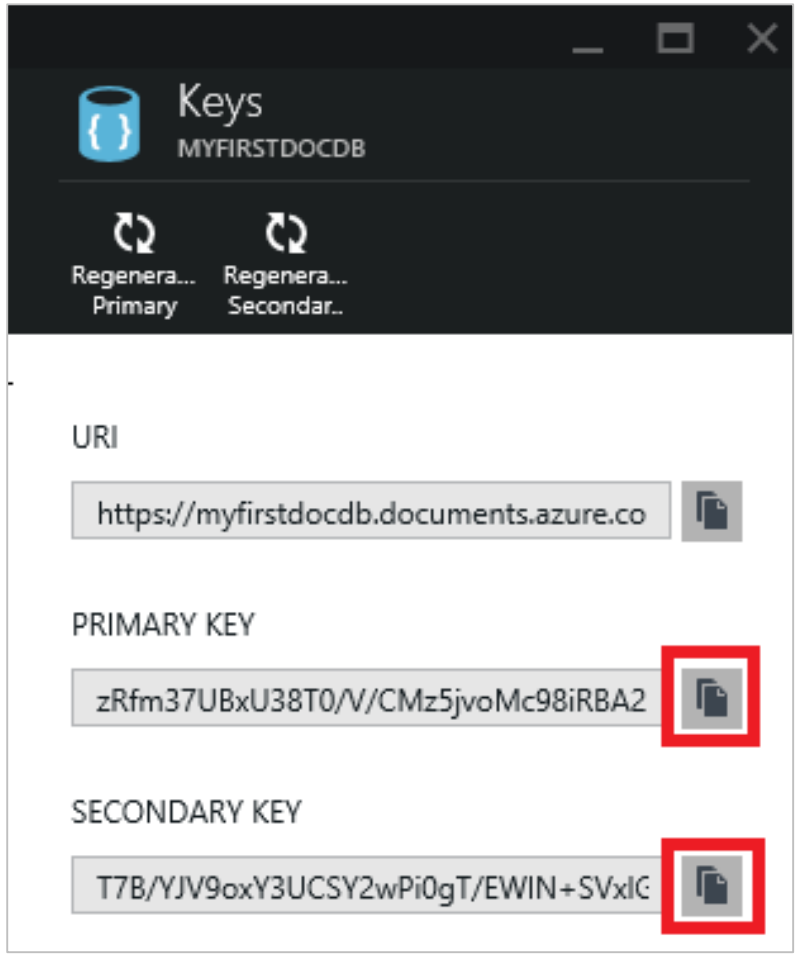



In the Azure Portal, access your Cosmos DB account.

In the **Summary** lens, click **Keys**.

On the **Keys** blade, click the **Copy** button to the right of the key you wish to copy.

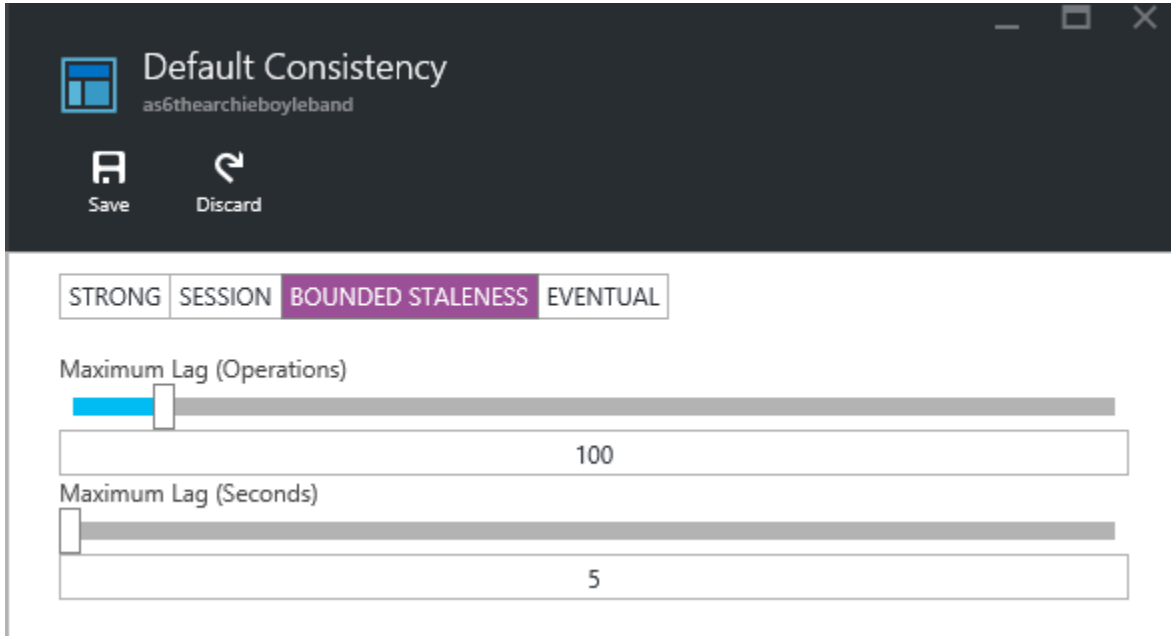
# Regenerate access keys




Change the access keys to your Cosmos DB account periodically to help keep your connections more secure. Two access keys are assigned to enable you to maintain connections to the Cosmos DB account using one access key while you regenerate the other access key.

If you have applications or cloud services using your Cosmos DB account, you will lose the connections if you regenerate keys, unless you roll your keys.

# Specify the default consistency




In the Azure management preview portal, access your Cosmos DB account.

In the **Configuration** lens, click **Default Consistency**.

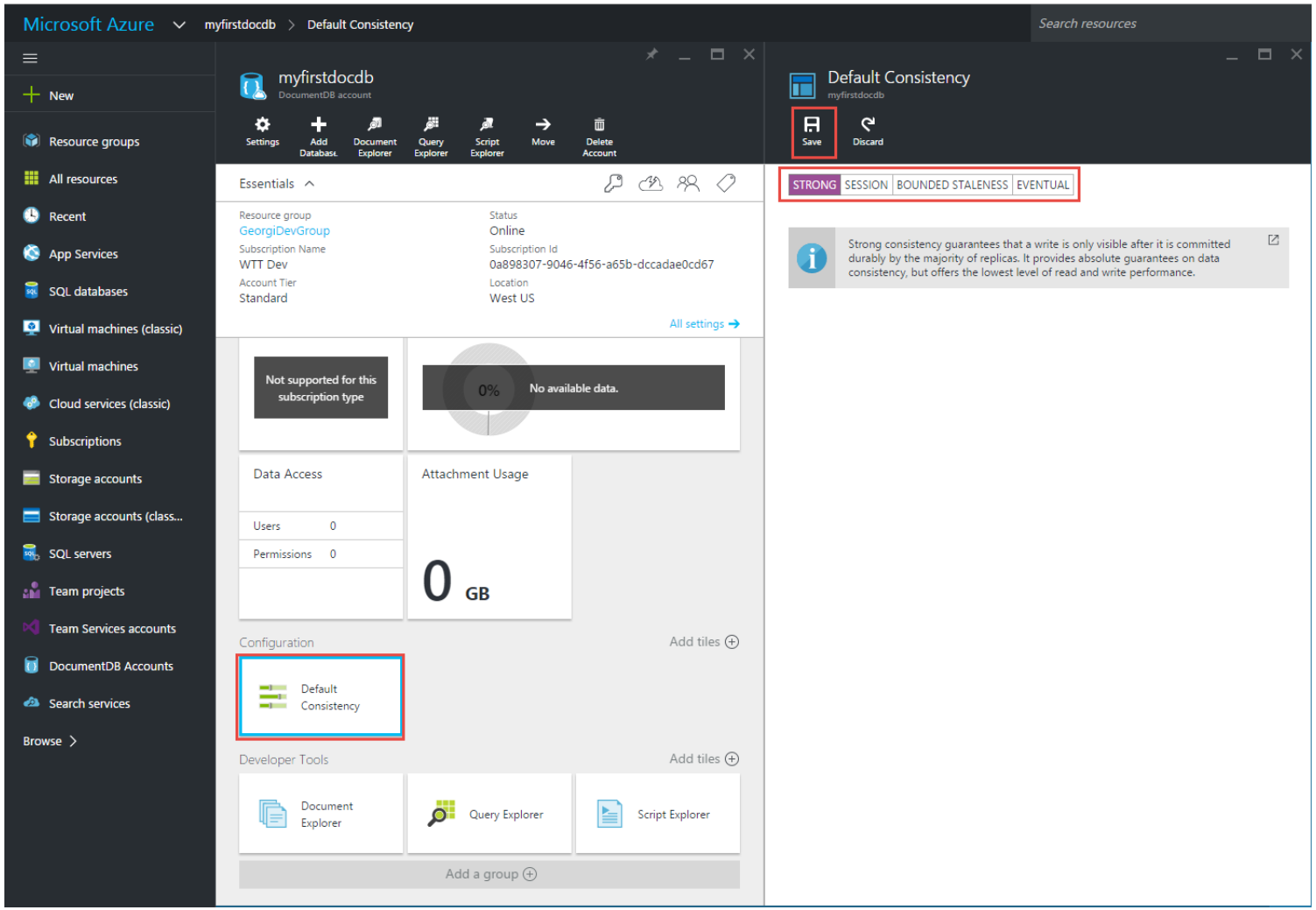
On the **Default Consistency** blade, select the default consistency level you want for your Cosmos DB account.

Click **Save**.

Monitor the progress of the operation via the Azure management preview portal Notifications hub.

Note that it can take several minutes before a change to the default consistency setting takes affect across your Cosmos DB account.

# Specify the indexing policy




In the Azure management portal, access your Azure Cosmos DB account.

In the **Configuration** lens, click **Default Consistency**.

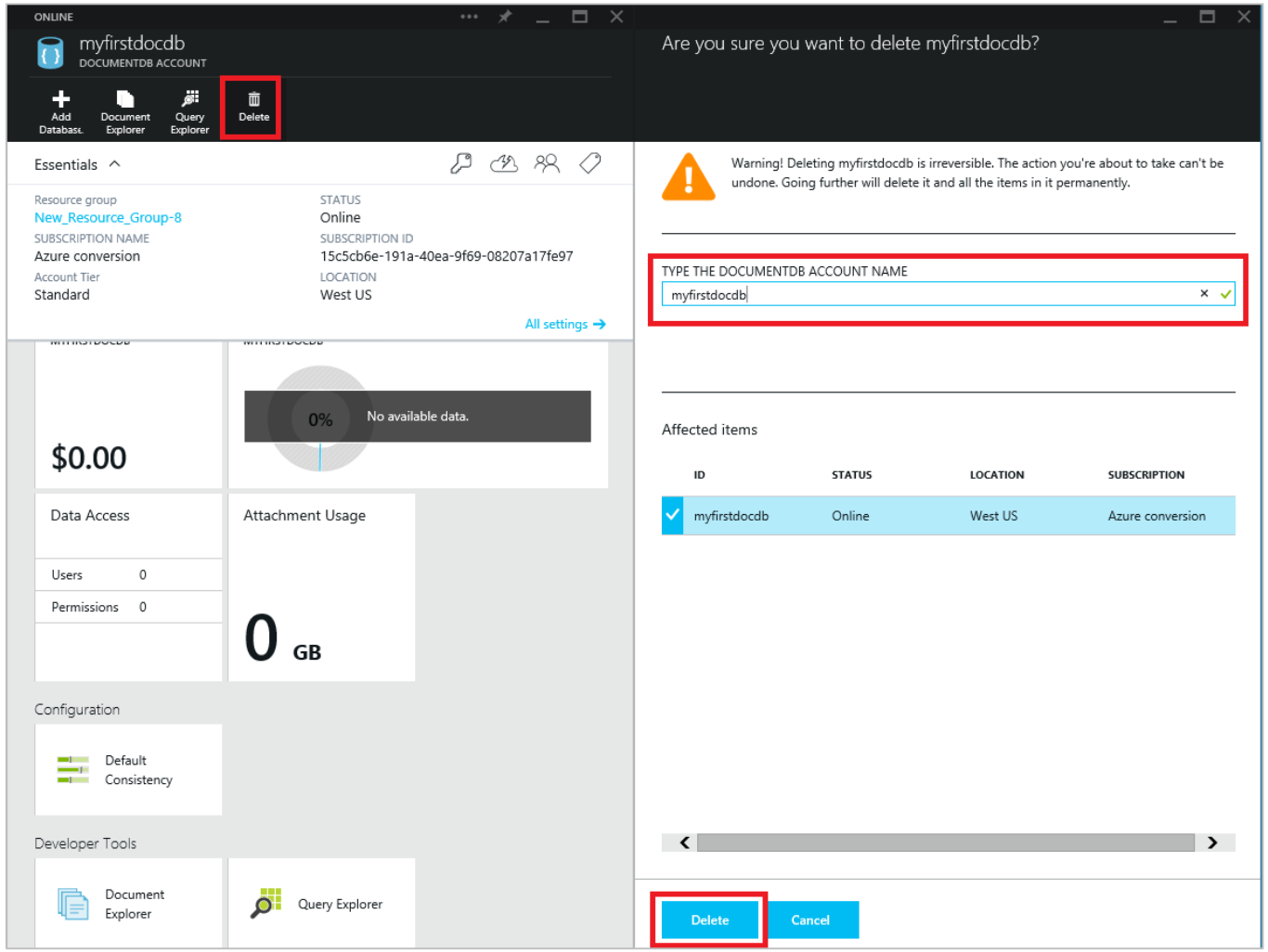
On the **Default Consistency** blade, select the default consistency level that you want for your Azure Cosmos DB account.

Click **Save**.

Monitor the progress of the operation via the Azure management portal **Notifications** hub.

Note that it can take several minutes before a change to the default consistency setting takes effect across your Azure Cosmos DB account.

# Delete a Cosmos DB account




In the Azure management preview portal, access the Cosmos DB account you wish to delete.

On the **Cosmos DB Account** blade, click **Delete**.

On the resulting confirmation blade, type the Cosmos DB account name to confirm that you want to delete the account.

On the confirmation blade, click **Delete**.

Warning: There is no way to restore the content from a deleted Cosmos DB account.

Deleting a Cosmos DB account will delete all of the account's resources, including databases, collections, documents, and attachments.

# Rapid development

Reduce development friction & complexity when building new business-class applications by leveraging familiar tools and industry standard platforms. Combine Cosmos DB with a portfolio of complementary cloud services on the Azure platform, such as the HDInsight Connector and Search Indexer

### Build with familiar tools – REST, JSON, JavaScript

Develop with a broad choice of popular platforms and technologies using industry standard protocols.

### Easy to start and fully-manage

Fully managed service eliminates the burden of machine, software, and cluster administration.

### Enterprise-grade Azure platform

Access a portfolio of complementary cloud services and receive support and assistance from a single vendor.

# Create, Get, Replace, Delete overview

# Create a document

```
POST
https://contosomarketing.documents.azure.com/dbs/ehs
zAA==/colls/ehszALxRRgA=/docs HTTP/1.1
Host: contosomarketing.documents.azure.com

{"id":"Book2","Title":"About
Seattle","Language":{"id":"English"},"Author":{"id":
"Fred","Location":{"City":"Seattle","Country":"Unite
d States"}},"Synopsis":"Seattle, the largest city in
the U.S. Pacific
Northwest...","Pages":400,"Topics":[{"Title":"Histor
y of Seattle"},{"Title":"Places to see in in
Seattle"}]}
```

A new document can be created by executing an HTTPS POST request against the URI resource path docs.

There is a document size limit to observe. For information on the allowable document size limit, please see the Cosmos DB Limits and Quotas article.

By default, a document is automatically indexed.

# Get a document

```
GET
https://contosomarketing.documents.azure.com/dbs/-
yI8AA==/colls/-yI8AKNuyAA=/docs/-
yI8AKNuyAANAAAAAAAAAA== HTTP/1.1
Host: contosomarketing.documents.azure.com

RESPONSE

{"Title":"About St.
Louis","Language":{"id":"English"},"id":"ISBN0-8800-
1599-1","_rid":"-
yI8AKNuyAANAAAAAAAAAA==","_ts":1408333905,"_self":"d
bs\/-yI8AA==\/colls\/-yI8AKNuyAA=\/docs\/-
yI8AKNuyAANAAAAAAAAAA==\/","_etag":"00005c00-0000-
0000-0000-
53f178510000","_attachments":"attachments\/"}
```

Performing a GET on a specific document resource will retrieve the user-defined JSON elements and system properties of the document.

While consistency level is defined at the database account level during account creation, read consistency can be overridden to meet the needs of the application. The override is set per GET operation by setting the x-ms-consistency-level header to the desired level.

The rule of thumb is that consistency override can only be the same or weaker than the level that was set during account creation.

# Replace a document

```
PUT
https://contosomarketing.documents.azure.com/dbs/ehszAA==/colls/ehszALxRRgA=/docs/XP0mAJ3H-AACAAAAAAAAAA== HTTP/1.1


{"id":"ISBDN 0-1231-1231-1","Title":"About Seattle","Language":{"id":"English"},"Author":{"id":"Fred","Location":{"City":"Seattle","Country":"United States"}},"Synopsis":"Seattle, the largest city in the U.S. Pacific Northwest...","Pages":400,"Topics":[{"Title":"History of Seattle"},{"Title":"Places to see in Seattle"}]}
```

Performing a PUT on a specific document resource will replace the entire document resource.

All user settable properties, including the id and the user-defined JSON elements, must be submitted in the body to perform the replacement.

Index updates are incremental, but data updates are full replace operations.

The x-ms-indexing-directive header can be set and submitted for the operation to include or exclude the document from being indexed.

# Delete a document

```
DELETE
https://contosomarketing.documents.azure.com/dbs/XP0
mAA==/colls/XP0mAJ3H-AA=/docs/XP0mAJ3H-
AAFAAAAAAAAAA== HTTP/1.1
Accept: application/json
Host: contosomarketing.documents.azure.com

RESPONSE

HTTP/1.1 204 No Content
Content-Length: 0
Content-Type: application/json
```

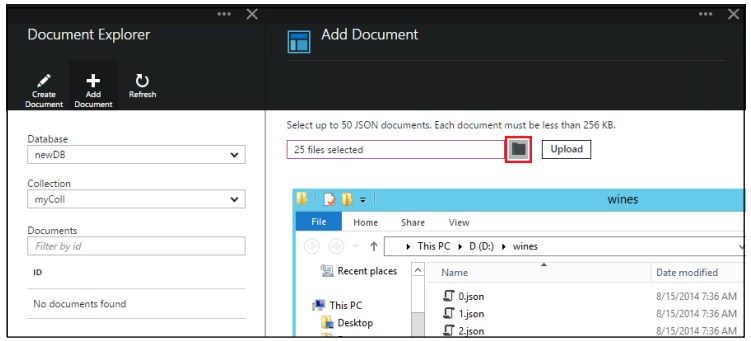
Performing a DELETE on a specific document resource will delete the document resource from the collection.

204 is returned when the delete operation is successful.

404 is returned when the document no longer exists (that is, the document may have already been deleted).

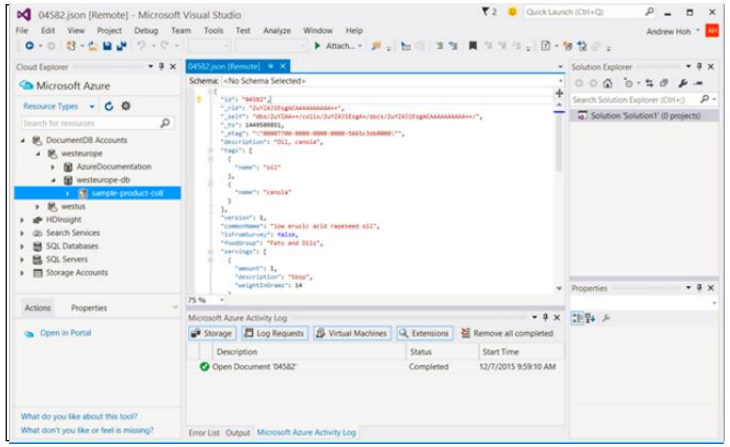
# Tools

## Document Explorer in Azure portal



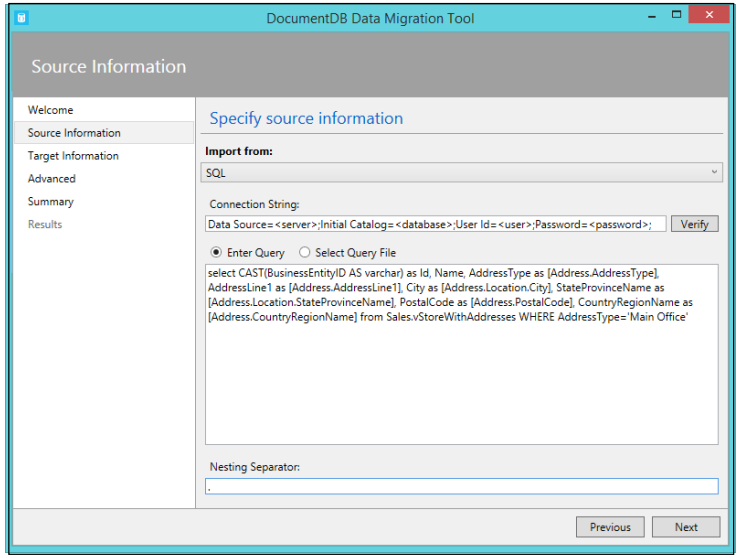

http://portal.azure.com

## Microsoft Cloud Explorer for Visual Studio




https://azure.microsoft.com/en-us/blog/exploring-azure-Cosmos DB-in-visual-studio/

## Azure Cosmos DB data-migration tool




https://azure.microsoft.com/en-us/documentation/articles/Cosmos DB-import-data/

# Cosmos DB is particularly suited for web and mobile applications

- Catalog data
- Preferences and state
- Event store
- User generated content
- Data exchange
- Gaming

# Azure Cosmos DB service summary

## Unique among NoSQL stores:

- Developed for the cloud and for delivery as a service
- Truly query-able JSON store
- Transactional processing through language-integrated JavaScript
- Predictable performance and tunable consistency

# Development scenarios

## Consider Azure Cosmos DB when you need:

- To build new web and mobile cloud-based applications
- Rapid development and high-scalability requirements
- Query and processing of user- and device-generated data
- More query and processing support for your key-value stores
- To run a document store in virtual machines
- A managed service model