**Explore concepts of non-relational data**

# Introduction

Data comes in all shapes and sizes, and can be used for many purposes. Many organizations use relational databases to store this data. However, the relational model might not be the most appropriate structure. The format of the data might be too varied to easily model as a set of relational tables. For example, ***the data might contain items such as video, audio, images, temporal information, large volumes of free text, or other types of data that aren't inherently relational. Additionally, the data processing requirements might not be best suited by attempting to convert this data into the relational format. In these situations, it may be better to use non-relational repositories that can store data in its original format, but that allow fast storage and retrieval access to this data.***

**Case Study**

**Suppose you're a data engineer working at Contoso, an organization with a large manufacturing operation. The organization has to gather and store information from a range of sources, such as real-time data monitoring the status of production line machinery, product quality control data, historical production logs, product volumes in stock, and raw materials inventory data. This information is critical to the operation of the organization. You've been asked to determine how best to store this information, so that it can be stored quickly, and queried easily.**

# Explore characteristics of non-relational data

1. Relational databases are an excellent tool for storing and retrieving data that has a well-known structure, containing fields that you can define in advance.

In some situations, you might not have the required knowledge of the structure of your data, in advance of it arriving in your database, to record it as a neat set of rows and columns in a tabular format. This is a common scenario in systems that consume data from a wide variety of sources, such **as data ingestion pipelines**. In these situations, a non-relational database can prove extremely useful.

## **What are the characteristics of non-relational data?**

You use a database to model some aspect of the real-world. Entities in the real-world often have highly variable structures. For example, in an ecommerce database that stores information about customers, how many telephone numbers does a customer have? A customer might have a landline and a mobile number, but some customers might have a business number, an additional home number, and maybe several mobile numbers. Similarly, the addresses of customers might not always follow the same format; addresses for customers in different states and regions might contain different elements, such as zip codes or postal codes.

In another scenario, if you are ingesting data rapidly, you want to capture the data and save it very quickly. Processing the data and manipulating it into a set of rows in different tables in a relational database might not be appropriate at this point; you can perform these tasks at a later date. At the time of ingestion, you simply need to store the data in its original state and format.

**A key aspect of non-relational databases is that they enable you to store data in a very flexible manner. Non-relational databases don't impose a schema on data**. Instead, they focus on the data itself rather than how to structure it. This approach means that you can store information in a natural format, that mirrors the way in which you would consume, query and use it.

**In a non-relational system, you store the information for entities in collections or containers rather than relational tables.** Two entities in the same collection can have a different set of fields rather than a regular set of columns found in a relational table. The lack of a fixed schema means that each entity must be self-describing. Often this is achieved by labeling each field with the name of the data that it represents. For example, a non-relational collection of customer entities might look like this:

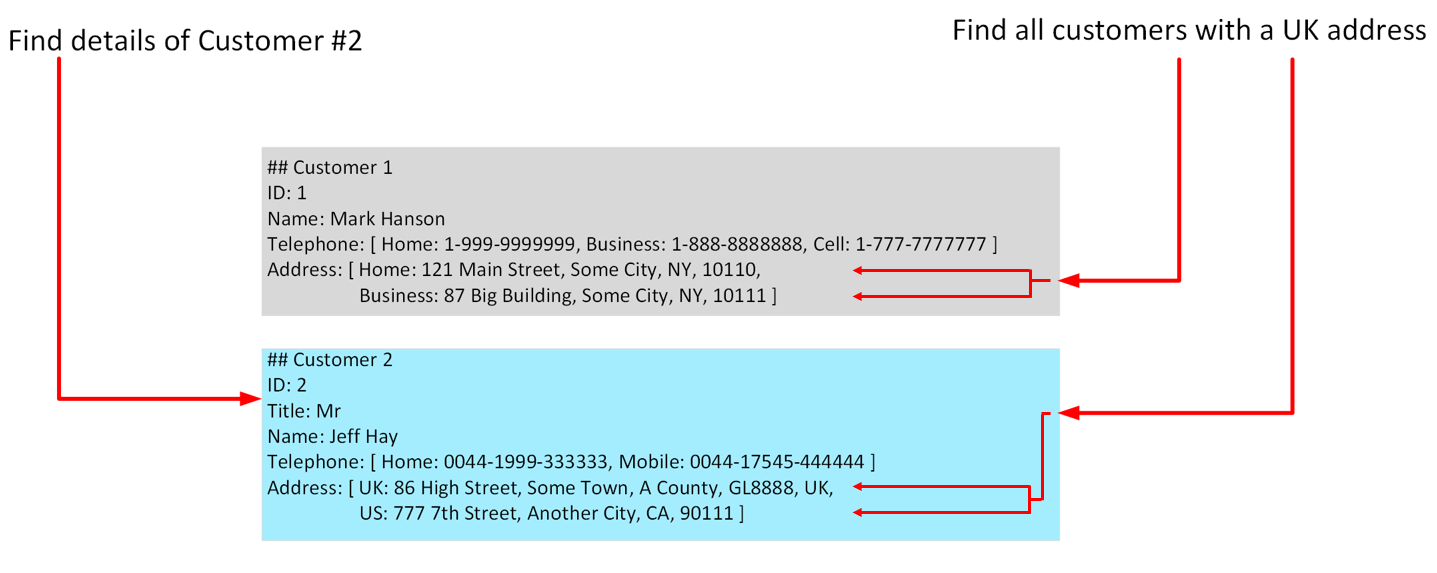
|  |
| --- |
| ## Customer 1  ID: 1  Name: Mark Hanson  Telephone: [ Home: 1-999-9999999, Business: 1-888-8888888, Cell: 1-777-7777777 ]  Address: [ Home: 121 Main Street, Some City, NY, 10110,  Business: 87 Big Building, Some City, NY, 10111 ]  ## Customer 2  ID: 2  Title: Mr  Name: Jeff Hay  Telephone: [ Home: 0044-1999-333333, Mobile: 0044-17545-444444 ]  Address: [ UK: 86 High Street, Some Town, A County, GL8888, UK,  US: 777 7th Street, Another City, CA, 90111 ] |

In this example, fields are prefixed with a name. Fields might also have multiple subfields, also with names. In the example, multiple subfields are denoted by enclosing them between square brackets.

Adding a new customer is a matter of inserting an entity with its fields labeled in a meaningful way. An application that queries this data must be prepared to parse the information in the entity that it retrieves.

The data retrieval capabilities of a non-relational database can vary

. Each entity should have a unique key value. The entities in a collection are usually stored in key-value order. In the example above, the unique key is the ID field. The simplest type of non-relational database enables an application to either specify the unique key, or a range of keys as query criteria. In the customers example, the database would enable an application to query customers by ID only. Filtering data on other fields would require scanning the entire collection of entities, parsing each entity in turn, and then applying any query criteria to each entity to find any matches. In the example below, a query that fetches the details of a customer by ID can quickly identify which entity to retrieve. A query that attempts to find all customers with a UK address would have to iterate through every entity, and for each entity examine each field in turn. If the database contains many millions of entities, this query could take a considerable time to run.



More advanced non-relational systems support indexing, in a similar manner to an index in a relational database. Queries can then use the index to identify and fetch data based on non-key fields. Non-relational systems such as Azure Cosmos DB (a non-relational database management system available in Azure), support indexing even when the structure of the indexed data can vary from record to record. For more information, read [Indexing in Azure Cosmos DB - Overview](https://docs.microsoft.com/en-us/azure/cosmos-db/index-overview).

When you design a non-relational database, it's important to understand the capabilities of the database management system and the types of query it will have to support.

**Note**

**Non-relational databases often provide their own proprietary language for managing and querying data. This language may be procedural (follow a set of commands, in order), or it may be similar to SQL; it depends on how the database is implemented by the database management system.**

## **Identify non-relational database use cases**

Non-relational databases are highly suitable for the following scenarios:

* **IoT and telematics**. These systems typically ingest large amounts of data in frequent bursts of activity. Non-relational databases can store this information very quickly. The data can then be used by analytics services such as Azure Machine Learning, Azure HDInsight, and Microsoft Power BI. Additionally, you can process the data in real-time using Azure Functions that are triggered as data arrives in the database.
* **Retail and marketing.** Microsoft uses CosmosDB for its own ecommerce platforms that run as part of Windows Store and Xbox Live. It's also used in the retail industry for storing catalog data and for event sourcing in order processing pipelines.
* **Gaming**. The database tier is a crucial component of gaming applications. Modern games perform graphical processing on mobile/console clients, but rely on the cloud to deliver customized and personalized content like in-game stats, social media integration, and high-score leaderboards. Games often require single-millisecond latencies for reads and write to provide an engaging in-game experience. A game database needs to be fast and be able to handle massive spikes in request rates during new game launches and feature updates.
* **Web and mobile applications.** A non-relational database such as Azure Cosmos DB is commonly used within web and mobile applications, and is well suited for modeling social interactions, integrating with third-party services, and for building rich personalized experiences. The Cosmos DB SDKs (software development kits) can be used to build rich iOS and Android applications using the popular Xamarin framework.

A relational database restructures the data into a fixed format that is designed to answer specific queries. When data needs to be ingested very quickly, or the query is unknown and **unconstrained**, a relational database can be less suitable than a non-relational database.

# Describe types of non-relational data

Non-relational data generally falls into two categories; **semi-structured and non-structured**.

## **What is semi-structured data?**

Semi-structured data is data that contains fields. The fields don't have to be the same in every entity. You only define the fields that you need on a per-entity basis. The Customer entities shown in the previous unit are examples of semi-structured data. The data must be formatted in such a way that an application can parse and process it. One common way of doing this is to store the data for each entity as a **JSON document**. The term JSON stands for JavaScript Object Notation; it's the format used by JavaScript applications to store data in memory, but can also be used to read and write documents to and from files.

A JSON document is enclosed in curly brackets ({ and }). Each field has a name (a label), followed by a colon, and then the value of the field. Fields can contain simple values, or subdocuments (each starting and ending with curly brackets). Fields can also have multiple values, held as arrays and surrounded with square brackets ([ and ]). Literals, or fixed values, in a field are enclosed in quotes, and fields are separated with commas.

The example below shows the customers from the previous unit, formatted as JSON documents:

JSONCopy

|  |
| --- |
| {  "ID": "1",  "Name": "Mark Hanson",  "Telephone": [  { "Home": "1-999-9999999" },  { "Business": "1-888-8888888" },  { "Cell": "1-777-7777777" }  ],  "Address": [  { "Home": [  { "StreetAddress": "121 Main Street" },  { "City": "Some City" },  { "State": "NY" },  { "Zip": "10110" }  ] },  { "Business": [  { "StreetAddress": "87 Big Building" },  { "City": "Some City" },  { "State": "NY" },  { "Zip": "10111" }  ] }  ]  }  {  "ID": "2",  "Title": "Mr",  "Name": "Jeff Hay",  "Telephone": [  { "Home": "0044-1999-333333" },  { "Mobile": "0044-17545-444444" }  ],  "Address": [  { "UK": [  { "StreetAddress": "86 High Street" },  { "Town": "Some Town" },  { "County": "A County" },  { "Postcode": "GL8888" },  { "Region": "UK" }  ] },  { "US": [  { "StreetAddress": "777 7th Street" },  { "City": "Another City" },  { "State": "CA" },  { "Zip": "90111" }  ] }  ]  } |

You're free to define whatever fields you like. The important point is that the data follows the JSON grammar. When an application reads a document, it can use a JSON parser to break up the document into its component fields and extract the individual pieces of data.

Other formats you might see include **Avro, ORC, and Parquet:**

* Avro is a row-based format. It was created by Apache. Each record contains a header that describes the structure of the data in the record. This header is stored as JSON. The data is stored as binary information. An application uses the information in the header to parse the binary data and extract the fields it contains. Avro is a very good format for compressing data and minimizing storage and network bandwidth requirements. This example is a subset of the header information for the previous example, formatted as Avro:

AvroCopy

|  |
| --- |
| * { * "type": "record", * "name": "contact\_schema", * "fields": [ * { * "name": "id", * "type": "int", * "doc": "ID of the contact" * }, * { * "name": "name", * "type": "string", * "doc": "Name of the contact" * }, * { * "name": "telephone", * "type": [ * "null", * { * "type": "array", * "items": { * "type": "record", * "name": "contact\_schema.telephone", * "fields": [ * { * "name": "phoneid", * "type": "int" * }, * { * "name": "phonetype", * "type": [ "null", "string" ] * } * ] * } * } * ] * } * ] * } |

* **ORC (Optimized Row Columnar format**) organizes data into columns rather than rows. It was developed by **HortonWorks** for optimizing read and write operations in Apache Hive. Hive is a data warehouse system that supports fast data summarization and querying over very large datasets. Hive supports SQL-like queries over unstructured data. An ORC file contains stripes of data. Each stripe holds the data for a column or set of columns. A stripe contains an index into the rows in the stripe, the data for each row, and a footer that holds statistical information (count, sum, max, min, and so on) for each column.
* **Parquet** is another columnar data format. It was created by **Cloudera** and **Twitter**. A Parquet file contains row groups. Data for each column is stored together in the same row group. Each row group contains one or more chunks of data. A Parquet file includes metadata that describes the set of rows found in each chunk. An application can use this metadata to quickly locate the correct chunk for a given set of rows, and retrieve the data in the specified columns for these rows. Parquet specializes in storing and processing nested data types efficiently. It supports very efficient compression and encoding schemes.

## **What is unstructured data?**

Unstructured data is data that doesn't naturally contain fields. Examples include video, audio, and other media streams. Each item is an amorphous blob of binary data. You can't search for specific elements in this data.

You might choose to store data such as this in storage that is specifically designed for the purpose. In Azure, you would probably store video and audio data as block blobs in an Azure Storage account. (The term blob stands for Binary Large Object\*). A block blob only supports basic read and write operations.

You could also consider files as a form of unstructured data, although in some cases a file might include metadata that indicates what type of file it is (photograph, Word document, Excel spreadsheet, and so on), owner, and other elements that could be stored as fields. However, the main content of the file is unstructured.

# Describe types of non-relational and NoSQL databases

Non-relational data is an all-encompassing term that means anything not structured as a set of tables. There are many different types of non-structured data, and the information is used for a wide variety of purposes. Consequently, there are many different types of non-relational database management systems, each oriented towards a specific set of scenarios.

## **What is NoSQL?**

You might see the term NoSQL when reading about non-relational databases. NoSQL is a rather loose term that simply means non-relational. There's some debate about whether it's intended to imply Not SQL, or Not Only SQL; some non-relational databases support a version of SQL adapted for documents rather than tables (examples include Azure Cosmos DB).

NoSQL (non-relational) databases generally fall into four categories:

1. key-value stores,
2. document databases,
3. column family databases,
4. and graph databases.

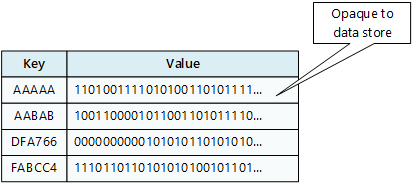
The following sections discuss these types of NoSQL databases.

## **What is a key-value store?**

A key-value store is the simplest (and often quickest) type of NoSQL database for inserting and querying data. Each data item in a key-value store has two elements, a key and a value. The key uniquely identifies the item, and the value holds the data for the item. The value is opaque to the database management system. Items are stored in key order.

**Note**

The term **opaque** means that the database management system just sees the value as an unstructured block. Only the application understands how the data in the value is structured and what fields it contains. The opposite of opaque is **transparent**. If the data is transparent, the database management system understands how the fields in the data are organized. A relational table is an example of a transparent structure.



A query specifies the keys to identify the items to be retrieved. You can't search on values. An application that retrieves data from a key-value store is responsible for parsing the contents of the values returned.

Write operations are restricted to inserts and deletes. If you need to update an item, you must retrieve the item, modify it in memory (in the application), and then write it back to the database, overwriting the original (effectively a delete and an insert).

The focus of a key-value store is the ability to read and write data very quickly. Search capabilities are secondary. A key-value store is an excellent choice for data ingestion, when a large volume of data arrives as a continual stream and must be stored immediately.

**Azure Table storage** is an example of a **key-value store**. Cosmos DB also implements a key-value store using the [**Table API**](https://docs.microsoft.com/en-us/azure/cosmos-db/table-introduction).

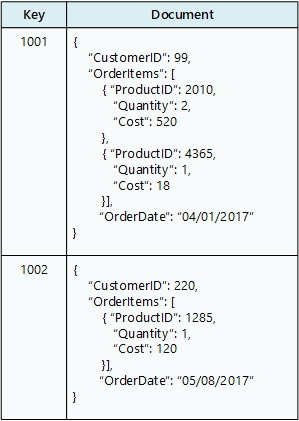
## **What is a document database?**

A document database represents the opposite end of the NoSQL spectrum from a key-value store. In a document database, each document has a unique ID, but the fields in the documents are transparent to the database management system. Document databases typically store data **in JSON format,** as described in the previous unit, or they could be encoded using other formats such as **XML, YAML, JSON, BSON**. Documents could even be stored as **plain text**.

**The fields in documents are exposed to the storage management system, enabling an application to query and filter data by using the values in these fields.**

Typically, a document contains the entire data for an entity. What items constitute an entity are application-specific. For example, an entity could contain the details of a customer, an order, or a combination of both. A single document may contain information that would be spread across several relational tables in an **RDBMS (relational database management system).**

A document store does not require that all documents have the same structure. This free-form approach provides a great deal of flexibility. Applications can store different data in documents as business requirements change.



An application can retrieve documents by using the document key. The key is a unique identifier for the document. Some document databases create the document key automatically. Others enable you to specify an attribute of the document to use as the key. The application can also query documents based on the value of one or more fields. Some document databases support indexing to facilitate fast lookup of documents based on one or more indexed fields.

Some document database management systems support in-place updates, enabling an application to modify the values of specific fields in a document without rewriting the entire document. Other document database management systems (such as Cosmos DB) can only read and write entire documents. In these cases, an update replaces the entire document with a new version. This approach helps to reduce fragmentation in the database, which can, in turn, improve performance.

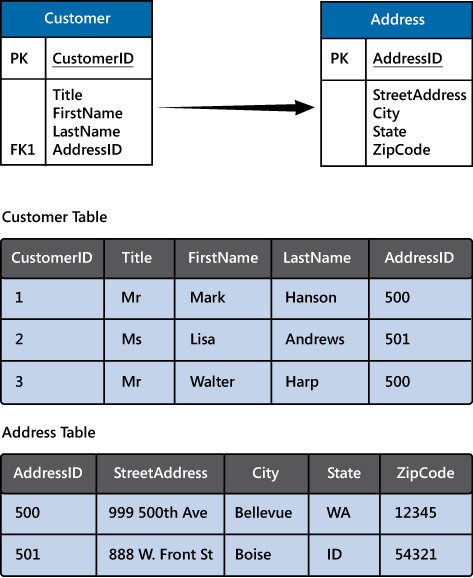
Most document databases will ingest large volumes of data more rapidly than a relational database, but aren't as optimal as a key-value store for this type of processing. The focus of a document database is its query capabilities.

**Azure Cosmos DB implements a document database approach in its Core (SQL) API.**

## **What is a column family database?**

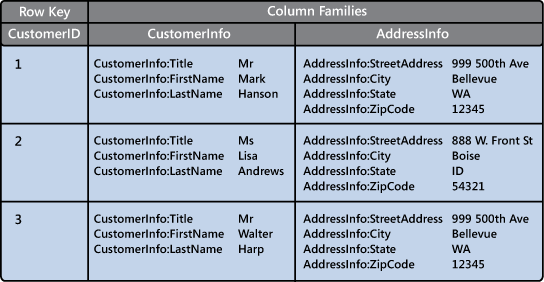
A column family database organizes data into rows and columns. Examples of this structure include ORC and Parquet files, described in the previous unit.

In its simplest form, a column family database can appear very similar to a relational database, at least conceptually. The real power of a column family database lies in its denormalized approach to structuring sparse data.

For example, if you need to store information about customers and their addresses in a relational database (ignoring the need to maintain historical data as described in the previous section), you might design a schema similar to that shown below. This diagram also shows some sample data. In this example, customer 1 and customer 3 share the same address, and the schema ensures that this address information is not duplicated. This is a standard way of implementing a one-to-many relationship.

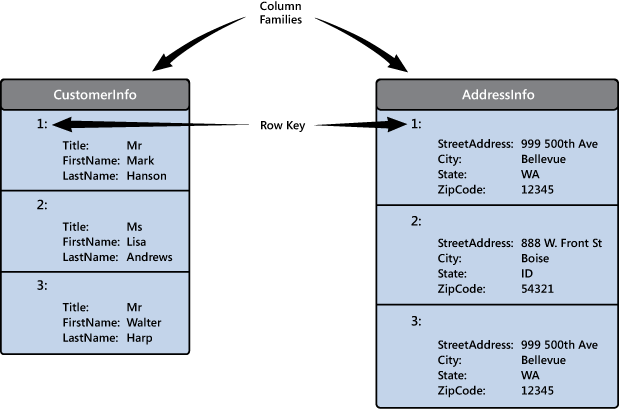
The relational model supports a very generalized approach to implementing this type of relationship, but to find the address of any given customer an application needs to run a query that joins two tables. If this is the most common query performed by the application, then the overhead associated with performing this join operation can quickly become significant if there are a large number of requests and the tables themselves are large.

The purpose of a column family database is to efficiently handle situations such as this. You can think of a column family database as holding tabular data comprising rows and columns, but you can divide the columns into groups known as column-families. Each column family holds a set of columns that are logically related together. The image below shows one way of structuring the same information as the previous image, by using a column family database to group the data into two column-families holding the customer name and address information. Other ways of organizing the columns are possible, but you should implement your column-families to optimize the most common queries that your application performs. In this case, queries that retrieve the addresses of customers can fetch the data with fewer reads than would be required in the corresponding relational database; these queries can fetch the data directly from the **AddressInfo** column family.



The illustration above is conceptual rather than physical, and is intended to show the logical structure of the data rather than how it might be physically organized. Each row in a column family database contains a key, and you can fetch the data for a row by using this key.

In most column family databases, the column-families are stored separately. In the previous example, the CustomerInfo column family might be held in one area of physical storage and the AddressInfo column family in another, in a simple form of vertical partitioning. You should really think of the structure in terms of column-families rather than rows. The data for a single entity that spans multiple column-families will have the same row key in each column family. As an alternative to the conceptual layout shown previously, you can visualize the data shown as the following pair of physical structures.



The most widely used column family database management system is **Apache Cassandra**.

Azure Cosmos DB supports the column-familiy approach through the **Cassandra API**.

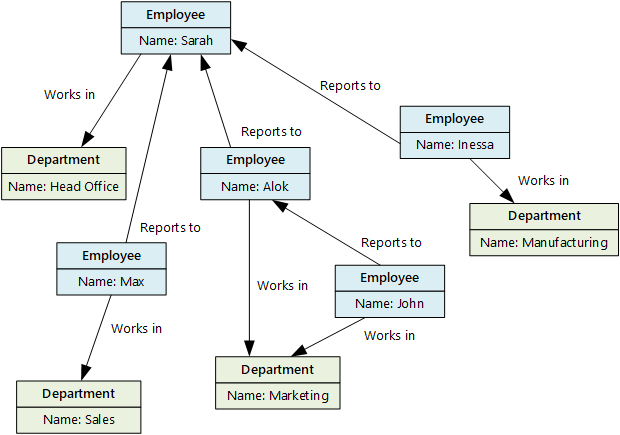
## **What is a graph database?**

Graph databases enable you to store entities, but the main focus is on the relationships that these entities have with each other. A graph database stores two types of information:

* 1. **nodes** that you can think of as instances of entities, and
  2. edges, which specify the relationships between nodes.

**Nodes and edges can both have properties that provide information about that node or edge (like columns in a table). Additionally, edges can have a direction indicating the nature of the relationship.**

The purpose of a graph database is to enable an application to efficiently perform queries that traverse the network of nodes and edges, and to analyze the relationships between entities. The image below shows an organization's personnel database structured as a graph. The entities are the employees and the departments in the organization, and the edges indicate reporting lines and the department in which employees work. In this graph, the arrows on the edges show the direction of the relationships.



A structure such as this makes it straightforward to conduct inquiries such as "Find all employees who directly or indirectly work for Sarah" or "Who works in the same department as John?" For large graphs with lots of entities and relationships, you can perform very complex analyses very quickly, and many graph databases provide a query language that you can use to traverse a network of relationships efficiently. You can often store the same information in a relational database, but the SQL required to query this information might require many expensive recursive join operations and nested subqueries.

**Azure Cosmos DB supports graph databases using the**[**Gremlin API**](https://docs.microsoft.com/en-us/azure/cosmos-db/graph-introduction)**. The Gremlin API is a standard language for creating and querying graphs.**

Microsoft Azure provides a variety of technologies for storing non-relational data. Each technology has its own strengths, and is suited to specific scenarios.

* [Choose the right data store](https://docs.microsoft.com/en-us/azure/architecture/guide/technology-choices/data-store-overview)
* [Welcome to Azure Cosmos DB](https://docs.microsoft.com/en-us/azure/cosmos-db/introduction)
* [Indexing in Azure Cosmos DB - Overview](https://docs.microsoft.com/en-us/azure/cosmos-db/index-overview)
* [Introduction to Azure Cosmos DB: Table API](https://docs.microsoft.com/en-us/azure/cosmos-db/table-introduction)
* [Introduction to Azure Cosmos DB: Gremlin API](https://docs.microsoft.com/en-us/azure/cosmos-db/graph-introduction)
* [Introduction to Azure Blob storage](https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction)