David Chappell

Understanding NoSQL on Microsoft Azure

Sponsored by Microsoft Corporation

Contents

Relational technology has been the dominant approach to working with data for decades. Typically accessed using Structured Query Language (SQL), relational databases are incredibly useful. And as their popularity suggests, they can be applied in many different situations.

But relational technology isn't always the best approach. Suppose you need to work with very large amounts of data, for example, too much to store on a single machine. Scaling relational technology to work effectively across many servers (physical or virtual) can be challenging. Or suppose your application works with data that's not a natural fit for relational systems, such as JavaScript Object Notation (JSON) documents. Shoehorning the data into relational tables is possible, but a storage technology expressly designed to work with this kind of information might be simpler.

NoSQL technologies have been created to address problems like these. As the name suggests, the label encompasses a variety of storage technologies that don't use the familiar relational model. Yet because they can provide greater scalability, alternative data formats, and other advantages, NoSQL options can sometimes be the right choice. Relational databases have a good future, and they're still the best choice in many situations. But NoSQL databases get more important every day.

Microsoft Azure supports a variety of NoSQL technologies. This guide walks through the options, explaining what each one provides.

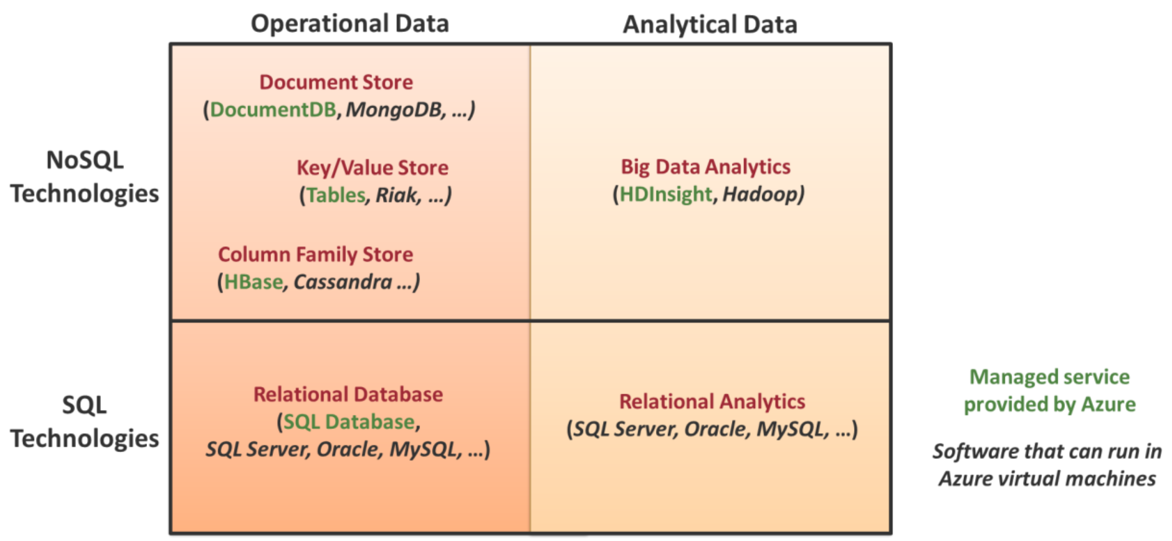
Data on Azure: The Big Picture

One way to think about data is to divide it into two broad categories:

- *Operational* data that's read and written by applications to carry out their ordinary functions. Examples include shopping cart data in a web commerce application, information about employees in a human resources system, and buy/sell prices in a stock-trading application.

- *Analytical* data that's used to provide business intelligence (BI). This data is often created by storing the operational data used by applications over time, and it's commonly read-only. For example, an organization might record all of the purchase data from its web commerce application or store all buy/sell prices for stock trades, then analyze this data to learn about customer buying habits or market trends. Because these analytical datasets provide a historical record, they're commonly much bigger than an application's current operational data.

Although the line between operational and analytical data can sometimes be blurry, different kinds of technologies are commonly used to work with these two kinds of information. Those technologies can be either relational—they use SQL—or non-relational. Figure 1 uses these two dimensions of operational/analytical and SQL/NoSQL to categorize the data technologies that you can use on Azure today.



Operational Data
Analytical Data
NoSQL Technologies
SQL Technologies
Document Store (DocumentDB, MongoDB, …)
Key/Value Store (Tables, Riak, …)
Column Family Store (HBase, Cassandra …)
Big Data Analytics (HDInsight, Hadoop)
Relational Database (SQL Database, SQL Server, Oracle, MySQL, …)
Relational Analytics (SQL Server, Oracle, MySQL, …)
Managed service provided by Azure
Software that can run in Azure virtual machines

Figure 1: Azure data technologies can be organized into four quadrants.

As the figure shows, Azure provides a group of managed services (shown in **green**) for working with relational and non-relational data. It also lets you use other data technologies (shown in ***black italics***) by running them in Azure virtual machines (VMs).

The two quadrants in the bottom row of the table illustrate the SQL technologies that can be used on Azure. They are:

- **Relational databases**, including the managed Azure service provided by **SQL Database** and the ability to run other database systems, such as ***SQL Server***, ***Oracle***, and ***MySQL***, in Azure VMs.
- **Relational analytics**, which can be done using ***SQL Server***, ***Oracle***, ***MySQL***, or another relational database system running in Azure VMs.

The two quadrants in the top row of Figure 1 illustrate the NoSQL technologies that can be used on Azure. As the diagram shows, it's common to group these technologies into a few different categories. The options include the following:

- **Document stores**, including the managed Azure service provided by **DocumentDB**. You can also run other document stores in Azure VMs, such as ***MongoDB***.
- **Key/value stores**, including the managed Azure service provided by **Tables**. You can also run other key/value stores in Azure VMs, such as ***Riak***.
- **Column family stores**, including a managed Azure service that implements **HBase**. You can also run other column family stores in Azure VMs, such as ***Cassandra***.
- **Big data analytics**, including the managed service provided by Azure **HDInsight**. This service implements Hadoop, and it's also possible to run ***Hadoop*** yourself in Azure VMs.

This guide focuses on Azure's four NoSQL managed services: DocumentDB, Tables, HBase, and HDInsight.

Relational Technology: A Quick Review

Before diving into the NoSQL world, it's worth starting with a quick look at relational technology. As just described, Azure offers two options for working with relational data: using SQL Database or running a relational database system in an Azure VM. While the two differ in important ways, they both use the same relational model for data. Figure 2 shows a concrete example of this approach.

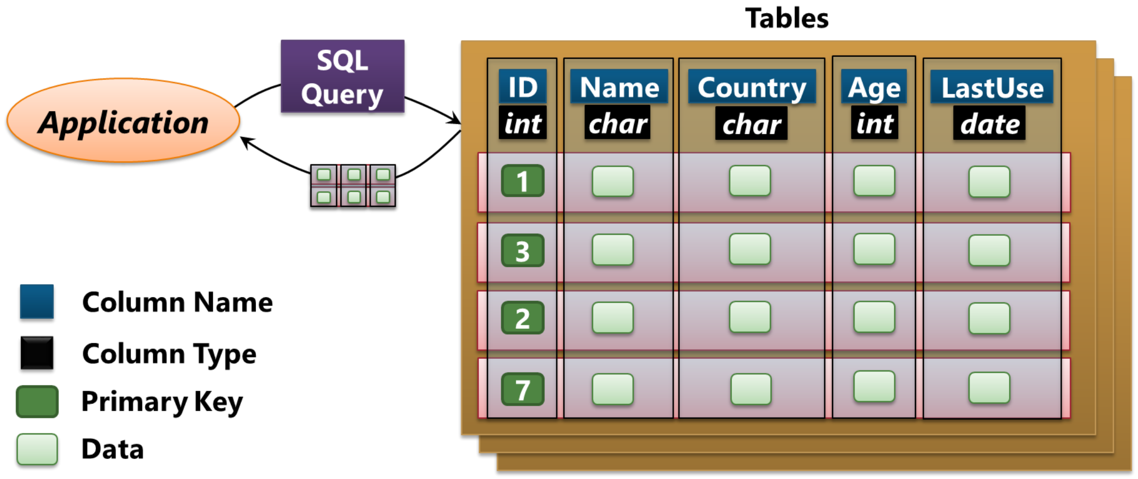


Figure 2: The relational model organizes data into tables, with columns defined by a schema.

A relational database stores data in *tables*. (More formally, a table is a *relation*, which is where the technology's name comes from.) A table contains some number of *columns*, each of a specific type, such as character, integer, or date. A *schema* describes the columns each table can have, and every table's data is stored in one or more *rows*. Each row contains a value for every column in that table, and the rows aren't kept in any particular order.

For example, suppose the table in Figure 2 is used to store data about the users of a web application. Each user has a unique identifier along with a name, a country, an age, and the date this user last accessed the application. Each row describes one user, and every row contains the same fields, one for each column in the table.

An application can issue a *SQL query* against one or more tables. The result is a relation containing values grouped into rows. An application can also atomically update or add data in one or more tables using a *transaction*. When changes are wrapped in a transaction, either all of them happen (if the transaction succeeds) or none of them happen (if it fails). To write logic that runs inside the database system itself, a developer can create *stored procedures* or *triggers*. In SQL Database, this logic is written in T-SQL, a language specifically designed for this purpose.

One or more columns in each table are designated as the *primary key*. In Figure 2, for instance, the unique identifier created for each user serves this purpose. The system automatically creates an *index* containing the values in this column, which speeds up searching for data using that key. It's also possible to create *secondary indexes* on other columns in a table. These indexes speed up the execution of queries that access data using column values other than the primary key. For example, if an application often needed to search for users by the date of their last use, a relational database could create an index on the example table's LastUse column.

The relational model is a beautiful thing. Schemas help avoid errors, since an attempt to write the wrong type of data into a particular column can be blocked by the database system. Transactions free developers from worrying about inconsistent data caused by failures during updates, even when the updates span more than one table. Secondary indexes let applications access data efficiently using different keys, giving developers more flexibility. All of these are good things.

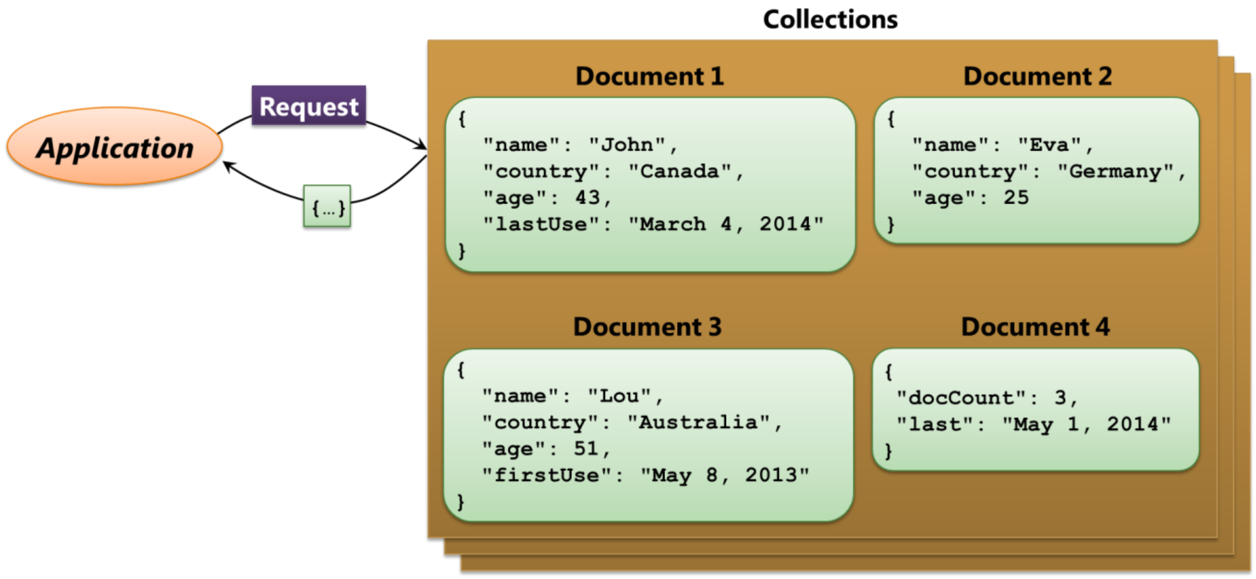
But these benefits come at a cost. For example, it's hard to spread data across many servers—something that's required at large scale—and still provide all of these features. It can also sometimes be challenging for an application developer to map the objects in her application to relational tables. And schemas can make it difficult to deal with changing data. In situations like these, the people creating an application might instead choose to use a NoSQL solution.

Azure NoSQL Technologies

Saying that something is a NoSQL technology tells you what it's not—it's not relational. This label doesn't tell you what the technology is, however, because quite different approaches are lumped together under this broad umbrella. For operational data, these approaches are commonly grouped into the three categories shown in the upper left quadrant of Figure 1: document stores, key/value stores, and column family stores. This section looks at the managed services that Azure offers in each of these categories.

Document Store: DocumentDB

The rows and columns in a relational table provide structure for data. But what if that structure doesn't match the data your application is working with? For an application working with JSON data, for example, a storage technology designed for JSON may well be a better fit. For situations like this, and plenty of others, Azure provides DocumentDB. Figure 3 provides a simple illustration of this cloud-based document store.



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

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

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

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

Figure 3: DocumentDB, a document store, lets an application work with data as JSON documents.

As the figure shows, a DocumentDB database contains a *collection* of JSON *documents.* Don't be confused, though: The word "document" here has nothing to do with, say, Microsoft Word documents. A JSON document is just a bunch of JSON text. JSON is derived from the syntax of JavaScript, which makes it easy for JavaScript developers to use. But JSON support is available for multiple languages—it's not only for JavaScript. Because it provides a simple and compact way to represent data, JSON has become a popular way to represent information for transmission and storage.

The example in Figure 3 once again holds data about users of a web application. This data is stored quite differently than in the relational table shown in Figure 2, however. Rather than storing each user's data in a row in some table, DocumentDB instead stores each user's data in a separate JSON document. For example, Document 1 contains the name, country, age, and last use date for a user named "John". This is the same information that a row of the table shown in Figure 2 might contain, but unlike a relational table, DocumentDB stores everything as text, using the conventions and data types defined by JSON.

DocumentDB differs from relational tables in other ways, too. As described earlier, relational tables have a fixed schema, with each row containing a value for all of the table's columns. This isn't true for documents in DocumentDB. In Figure 3, for example, Document 2 omits the lastUse element, while Document 3 replaces lastUse with something completely different, the date that a user first began using the application. And look at Document 4—it doesn't contain information about a user at all. Instead, this document contains a count of the number of user documents in this collection and the date the last one was added. This is all perfectly legal. As with most NoSQL stores, DocumentDB has no fixed schema. Each document in a collection can look any way its creator wants it to look.

To access and modify data, an application makes calls to DocumentDB's RESTful interface. Among other things, these calls can submit queries against one or more documents in a collection using a SQL-based query language. It's also possible to create stored procedures and triggers that run inside DocumentDB itself. Written in JavaScript, a stored procedure is always wrapped in an atomic transaction, so all of the changes it makes to documents in a collection succeed or fail as a group. And to make access fast, DocumentDB creates an index for every JSON element in every document. These are like the secondary indexes in a relational system, but they're built automatically by the system.

Like most NoSQL technologies, DocumentDB is designed to support very large amounts of data, up to hundreds of terabytes in a single database. To allow this, the collections in a DocumentDB database can be stored on different machines within the service. While this helps with scale, it also brings some constraints. Each query can target only one collection, for example, and so data that's often accessed together should be kept in the same collection. Transactions also can't span collections. While it's possible to do atomic updates on documents within a collection, it's up to the application to ensure correctness for updates across collections.

To make sure that a single server failure won't make a collection's documents unavailable, DocumentDB stores multiple copies of each collection on different machines. But replicating data like this brings another challenge: What happens when data is changed? If an application reading that changed data waits until all of the replicas are updated, it's guaranteed to see correct data, but the read will be relatively slow. If an application chooses instead to read from any available replica without waiting for a change to propagate to all of them, the read will be fast, but the data it returns might be wrong—what if this replica hasn't been updated yet? Different applications have different requirements—their creators want to trade off performance and data consistency in different ways—and so DocumentDB provides four consistency options:

- *Strong*, which is the slowest of the four, but is guaranteed to always return correct data.
- *Bounded staleness*, which ensures that an application will see changes in the order in which they were made. This option does allow an application to see out-of-date data, but only within a specified window, e.g., 500 milliseconds.
- *Session*, which ensures that an application always sees its own writes correctly, but allows access to potentially out-of-date or out-of-order data written by other applications.
- *Eventual*, which provides the fastest access, but also has the highest chance of returning out-of-date data.

DocumentDB is a managed service, so developers can create new databases and collections quickly. There's no need to install or manage servers. The service is multi-tenant, however, which means that it's used simultaneously by multiple applications. How can the system make sure that no single user gets more than its share of the service's resources? The answer is that each DocumentDB users buys a specific number of Capacity Units (CUs). Each CU includes a specific amount of storage and reserved throughput, which lets developers know that they'll get the performance they're paying for.

It's fair to say that document stores are the most widely used NoSQL option today. With DocumentDB, Microsoft aims at making it simpler and faster for developers to build applications that use this approach for their data.

NoSQL in the Azure Store

Microsoft provides a variety of NoSQL technologies as managed services on Azure. But this cloud platform also includes the Azure Store, through which developers can purchase other managed NoSQL offerings. Among the choices are:

- Managed MongoDB services provided by MongoLabs or the company called MongoDB.
- RavenHQ, a managed document store based on RavenDB.
- Redis Cloud, a managed key/value store based on Redis.

And as always, you're free to install and run any NoSQL database you like in Azure VMs. The reality is that there are lots of different ways to use NoSQL technologies on Microsoft Azure.

Key/Value Store: Tables

Suppose your application needs very fast access to large amounts of data. Maybe you're creating an ecommerce website, for instance, that maintains a large number of online shopping carts. The data is relatively simple: It's just information about the items a customer is interested in purchasing. The operations the application performs on this data are also relatively simple: read and write using a unique key for each shopping cart.

This scenario doesn't need the power of a relational database. And since that power comes with a cost, using a relational system would likely limit the number of simultaneous users your application can support. For carrying out lots of operations on large amounts of simply structured data, a key/value store such as Azure Tables can be a better choice. Figure 4 illustrates the basics of this technology.

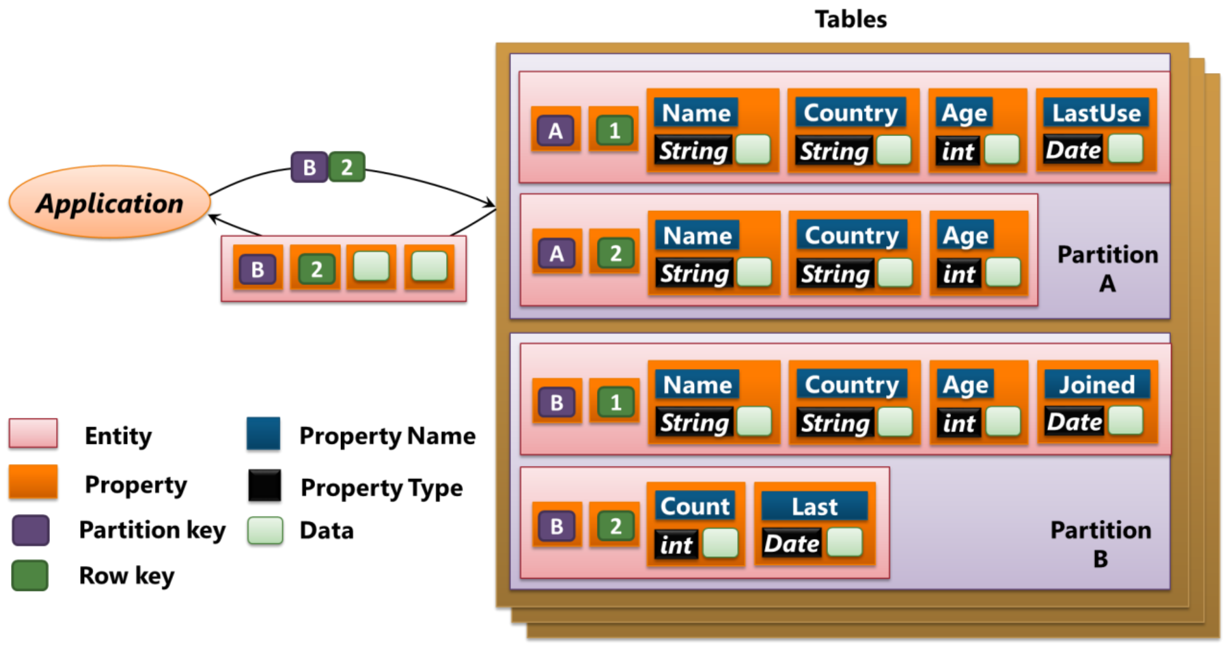

Tables
Application
B 2
B 2
A 1 Name String Country String Age int LastUse Date
A 2 Name String Country String Age int
Partition A
B 1 Name String Country String Age int Joined Date
B 2 Count int Last Date
Partition B
Entity
Property
Partition key
Row key
Property Name
Property Type
Data

Figure 4: Azure Tables, a key/value store, lets an application provide a key and get back values associated with that key.

The basic idea of a key/value store is simple. An application gives the store a unique key, then gets back one or more values associated with that key, as the figure shows.

In Azure Tables, data is held in *tables*, which are split into *partitions*. Each partition holds some number of *entities*, and each entity contains *properties*. Each property has a *name* and a *type*, such as integer or character string or date, and each one holds a value. One property in each entity is designated as the *partition key*, and it contains the same value for all entities in a particular partition. A second property in each entity is designated as the *row key*, and it contains a value that's unique within its partition.

To retrieve an entity, an application provides the partition key and the row key for that entity. What comes back is the entity this key pair identifies, including some or all of its properties. This response can be formatted as either XML or JSON, depending on what the client specifies. It's also possible to access a range of entities in a single request. Azure Tables has no support for secondary indexes, however—access is via partition and row keys.

Like other NoSQL technologies, Azure Tables has no notion of schema. Each entity in a partition can contain different properties with different data types if that's what makes the most sense for an application. In the example shown in Figure 4, for instance, the entity with the key A1 might hold the same information as Document 1 in Figure 3: the name of a user, her country, her age, and the date she last used the system. Entity A2 might hold the same information as Document 2 in Figure 3: just a user's name, country, and age. Entity B1 might hold the same information as Document 3 in the earlier example, while entity B2 again holds a count of users and the date on which the last user was added. There's no required structure.

Whatever the data looks like, Tables allows very fast access to entities. This access is relatively simple, however. There's no real query language, although some operations defined by OData are supported, and the service has no support for stored procedures or triggers. A group of updates can be wrapped in an atomic transaction as long as all of the entities involved are in the same partition of the same table.

Like DocumentDB, Azure Tables stores multiple replicas of each partition on different servers, so a single machine failure won't make that data unavailable. Unlike DocumentDB, however, Azure Tables always provides strong consistency, which means that reads always return the latest data. Tables also offers the option of geo-redundant storage, with a copy of your data stored in two different Azure datacenters. Changes are asynchronously updated across these copies.

One more attractive aspect of Tables is their low price. While the details vary depending on the options you choose—geo-redundant storage costs more—this NoSQL service is less expensive than DocumentDB. This is partly because you pay only for storage; there's no guaranteed CPU capacity, which means that application performance might vary. Still, a key/value store is the right choice in many situations. Its simplicity, scalability, and low cost make it a good match for quite a few applications.

Column Family Store: HBase

Suppose the data you're working with fits well into traditional tables—rows and columns are a good approach—but it's too big to use a relational database. Suppose further that your tables are sparse—many of the cells in each table don't have a value. For situations like this, you might be happiest with a column family store.

For example, imagine creating a table with information about every web page on the Internet. Each row could describe a page, while each column describes some aspect of that page. You'd have lots of rows—the web is big—and because a webpage can potentially have many different aspects, you'd also have lots of columns. But many of the cells in this table would be empty, because most pages will have only a subset of possible attributes.

HBase is a good fit for problems like this. Figure 5 illustrates the basics of the HBase data model.

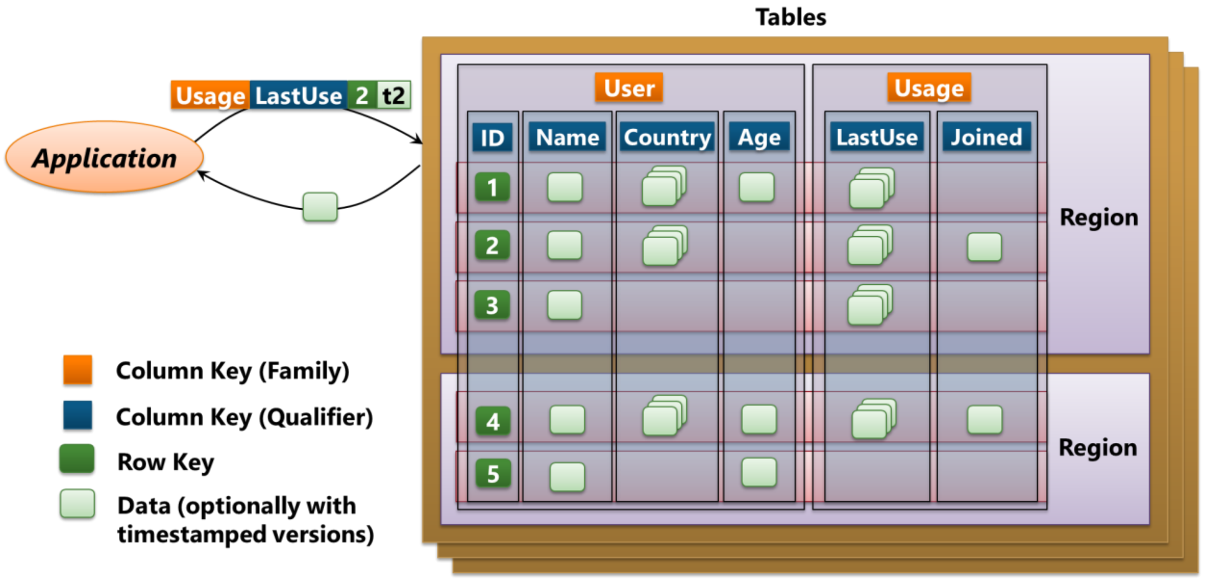


Figure 5: HBase, a column family store, provides tables whose columns are grouped into column families.

As the figure shows, HBase tables have rows and columns. The columns are grouped into *column families*, however, which is different from relational tables. The column families in a table must be defined up front, so HBase does have a little schema, but the columns in a particular column family are not. You can add new ones at any time.

The example shown here once again stores information about the users of a web application. The column family User has a column that holds a unique key for each row, along with columns for the user's name, country, and age. The column family Usage has columns that can store the date a user last used the application and the date he first used it.

Notice that unlike a relational table, not all rows have a value for all columns. This relaxed view of rows is how HBase deals with diverse data. Rather than holding documents or entities with different values of different types, as in DocumentDB or Tables, an HBase user just adds a column for whatever types a row might be required to store. Tables can be big—they can have millions of columns and billions of rows—yet most of the cells in the table might be empty.

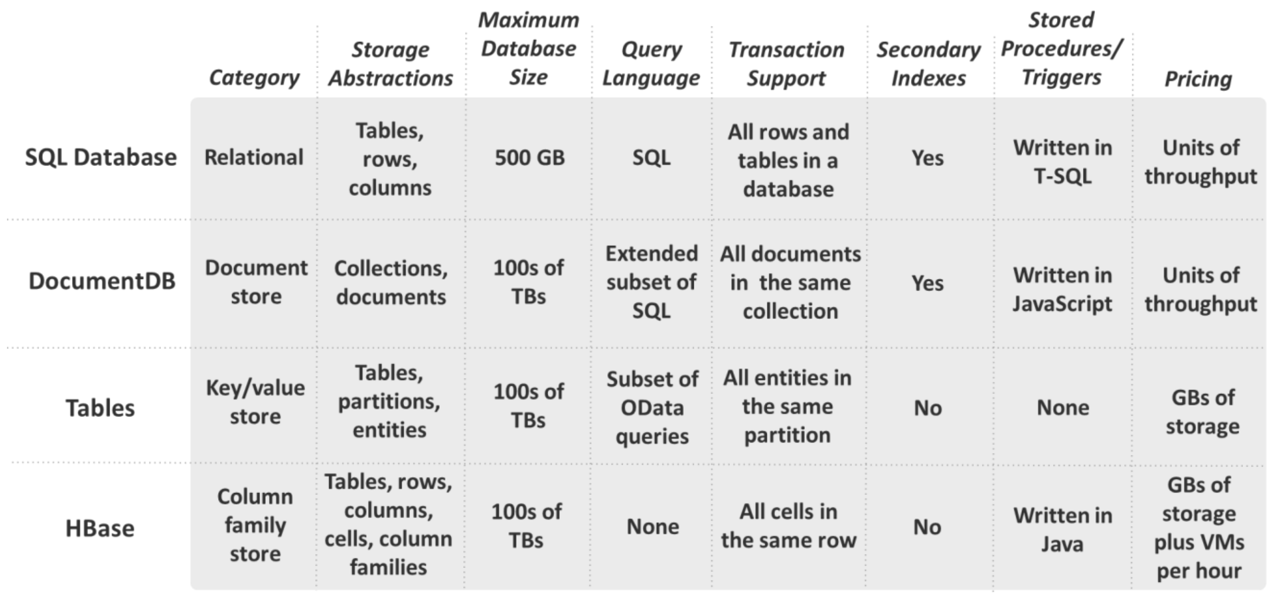
HBase tables also differ from relational tables in other interesting ways. There are no data types, for instance. Every table holds nothing but bytestrings. And each cell can contain multiple time-stamped versions of a value, which is useful when applications might need to access older information. Also, as the figure shows, the rows in a table are broken into *regions*, each of which can be stored on a different machine. This is how an HBase table scales to hold more data than will fit on a single machine. A developer using HBase doesn't see regions, however. Unlike DocumentDB containers and partitions in Azure Tables, both of which must be defined by the creator of the database, HBase automatically partitions data across machines.

HBase doesn't provide a query language. Instead, an application can access the value of a particular cell by providing three things: a column family, a column qualifier (i.e., a column name), and a row key. The request can also specify the timestamp for a specific version of a cell's data if desired. And because the rows are sorted by their keys, there's no need for the client to know which region contains a particular row; HBase can figure this out by itself. This approach is similar to a key/value store, and it provides fast access. Like Azure Tables, however, HBase doesn't support secondary indexes; an application needs to know how to reference the data it needs. HBase also supports creating stored procedures in Java with an option called *coprocessors*.

For updates, HBase provides transactions, but only for changes to cells in the same row of a single table. And like most NoSQL technologies, HBase replicates data, storing the same information on multiple servers to provide fault tolerance. This once again raises the question of consistency, which HBase answers in a straightforward way: It always provides strong consistency.

HBase is part of the Hadoop family of technologies, and on Azure, it's provided as part of the HDInsight service. Rather than being offered as a fully managed service, like DocumentDB or Tables, HBase requires a user to specify how many instances (i.e., virtual machines) he wants for his HBase cluster. HDInsight then automatically creates and manages this cluster. HBase data is stored in Azure Blobs, and so pricing for this service has two parts: a per-hour charge for each instance in the HBase cluster, along with a per-gigabyte per-month charge for the data stored in Blobs.

Like other NoSQL technologies, HBase can be just the right choice for an application. But keeping all of the options straight can be challenging—there's lots to remember. To help you do this, Figure 6 summarizes some of the key aspects of the four managed services that Azure provides for operational data today: SQL Database, DocumentDB, Tables, and HBase.



	Category	*Storage Abstractions*	*Maximum Database Size*	*Query Language*	*Transaction Support*	*Secondary Indexes*	*Stored Procedures/ Triggers*	*Pricing*
SQL Database	Relational	Tables, rows, columns	500 GB	SQL	All rows and tables in a database	Yes	Written in T-SQL	Units of throughput
DocumentDB	Document store	Collections, documents	100s of TBs	Extended subset of SQL	All documents in the same collection	Yes	Written in JavaScript	Units of throughput
Tables	Key/value store	Tables, partitions, entities	100s of TBs	Subset of OData queries	All entities in the same partition	No	None	GBs of storage
HBase	Column family store	Tables, rows, columns, cells, column families	100s of TBs	None	All cells in the same row	No	Written in Java	GBs of storage plus VMs per hour

Figure 6: Azure provides a range of relational and NoSQL managed services for operational data.

Another Option: Graph Databases

There's one more NoSQL category that's often talked about: graph databases. As the name suggests, this approach represents data as a graph, and it's especially useful when the connections between data items are as important as the data items themselves. Think about a social graph, for example, where the relationships between people are a fundamental part of what you're storing. For situations like this, a graph database can be useful.

Graph databases are less widely used today than the other three NoSQL categories, however, and Azure doesn't provide a managed data service that supports this approach. You are free, however, to install and run existing graph database technologies, such as Neo4J, in Azure virtual machines. For some kinds of problems, they're exactly the right solution.

Big Data Analytics: HDInsight

Applications use operational data to carry out their everyday functions: letting customers make purchases, onboarding new employees, updating a shared leaderboard for a mobile game, and lots more. But storing this operational data as it changes over time, then analyzing it for patterns, trends, and other information can have huge value. Because of this, many organizations have long turned operational data into analytical data by creating data warehouses, then applying standard BI tools. All of this has conventionally used relational technology such as SQL Server.

Today, though, there are plenty of situations where the analytical data we'd like to examine doesn't fit well in the relational world. More and more data isn't relational to start with; it might come from sensors or imaging devices or an operational NoSQL database like DocumentDB, Tables, or HBase. It might also be too big to store in relational systems. Whatever the situation, we need a way to work with analytical data that doesn't fit in the classic model of relational BI.

Remarkably, our industry has settled on a single technology for doing this: Hadoop. The term "Hadoop" is commonly used to refer to a group of open source technologies. Exactly what this group includes is a little fuzzy—different people use it in different ways—but the core technologies are:

- The Hadoop Distributed File System (HDFS), which provides a way to store and access very large binary files across a cluster of commodity servers and disk drives.
- Hadoop MapReduce, supporting the creation of applications that process large amounts of analytical data in parallel. That data is commonly stored in HDFS.
- Hive, a Hadoop-based framework for querying and analyzing data. Among other things, it provides HiveQL, a SQL-like language that can generate MapReduce jobs.
- Pig, another Hadoop-based framework for working with data. It provides a language called Pig Latin for creating MapReduce jobs.

As mentioned earlier, the Hadoop family also includes HBase, a column family store. Like Hadoop MapReduce, HBase is built on HDFS, although it's intended for operational rather than analytical data. Azure supports all of these with its HDInsight service.

Hadoop MapReduce, the main focus here, is designed to run applications that analyze large amounts of unstructured data. With these kinds of big data problems, the data you're analyzing can't all fit on one machine. (If it can, you probably don't have a big data problem.) To analyze your data effectively, you'd like to break your application into pieces, then run each of those pieces on the machine where the chunk of data it's analyzing is stored.

Doing this requires a way to store and access data across all of the machines in a cluster. And since writing distributed software can be challenging, it would also be nice to have a framework for creating and running the applications that analyze this data. These two things are provided by HDFS and Hadoop MapReduce, respectively. Figure 7 shows how these look in HDInsight.

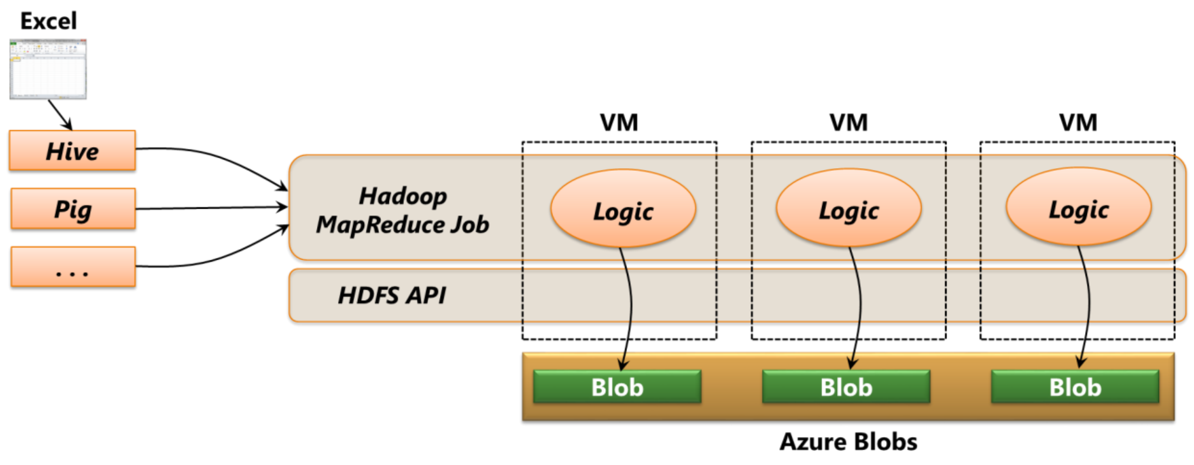


Figure 7: HDInsight supports Hadoop MapReduce applications running across multiple servers that process distributed data stored in Azure Blobs.

To use HDInsight, you start by asking it to create a Hadoop cluster, specifying the number of Azure virtual machines you need. Once the cluster exists, you can run MapReduce applications, commonly called *jobs*, on it. The logic of a MapReduce job is spread across multiple VMs, as the figure shows, with each VM processing a specific part of the data being analyzed. The more VMs you have, the faster your application can complete its task.

With HDInsight, a MapReduce job accesses data through the standard API provided by HDFS. The data this job works on is stored in Azure Blobs, however, a low-cost way to store unstructured data. Yet because this data is exposed through the HDFS API, existing MapReduce applications can run unchanged. And because it makes sense in some situations, HDInsight also allows copying data from Blobs into HDFS running in VMs.

This kind of parallel data analysys is a perfect fit for a public cloud platform like Azure. Rather than buying and maintaining a fleet of on-premises servers that might sit unused much of the time, running Hadoop in the cloud lets you pay only for the VMs you need when you need them. This can be significantly less expensive than buying and maintaining an on-premises cluster, and it's one of the most attractive aspects of using HDInsight.

HDInsight allows creating MapReduce jobs in various languages, including Java, C#, F#, and JavaScript. As the figure shows, it's also possible to use Hive, Pig, and other tools to create those jobs. And to make life easier for less technical data analysts, Microsoft provides a way to create MapReduce jobs using Excel by submitting HiveQL queries. The goal is to make this increasingly popular approach to analyzing unstructured data as easy to use as possible.

Running Your Own Hadoop Cluster

HDInsight makes it straightforward to create a Hadoop cluster, and it also takes care of cluster management. If you choose to, though, you can instead create a cluster yourself by installing a Hadoop distribution in Azure virtual machines. You might want to move an existing Hadoop cluster into the cloud, for instance, or perhaps you want to use a different Hadoop distribution than the Hortonworks technology that HDInsight uses.

Most often, you're likely to be happier using HDInsight. Not only does it save the time and expense of building and maintaining a cluster, it also avoids the need to have people available who know how to do these things. Hadoop administrative skills are in high demand today, which makes these people scarce and expensive. Why not let a cloud platform do this work for you?

Conclusion

Relational databases are familiar and easy to use. But they're not the best choice for an increasing number of applications. Whether it's because the application needs more scalability than a relational system can provide or because the data the application works with isn't a good fit for relations or because relational storage is just too expensive, a NoSQL solution is often better. Recognizing this reality, Azure provides both relational and NoSQL options.

Like all technologies, NoSQL stores have pros and cons. Most likely, relational storage will continue to be the right choice for many of your new applications. But when a SQL-based option isn't appropriate, don't be afraid to go with NoSQL. Azure supports every major NoSQL approach today, either as a managed service or by running it in VMs. As the applications we build continue to evolve, expect NoSQL technologies to get more and more popular.

About the Author

David Chappell is Principal of Chappell & Associates (http://www.davidchappell.com) in San Francisco, California. Through his speaking, writing, and consulting, he helps people around the world understand, use, and make better decisions about new technologies.