David Chappell

Understanding NoSQL Technologies on Windows 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 both on premises and in the cloud.

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 independent 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 or graphs. 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 still have a good future, and they're still best in many situations, but NoSQL databases are also important.

This is especially true for applications running in the public cloud. A public cloud platform such as Windows Azure provides enough compute power to run very scalable applications with lots of simultaneous users. But at this scale, relational technology often breaks down—there's too much data. This makes NoSQL technologies especially important for cloud applications, and so Windows Azure supports a variety of NoSQL alternatives. This guide walks through the options, explaining what each one provides and why you might want to use it.

Data on Windows 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 they provide a historical record, analytical datasets are 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 can be used on Windows Azure today.

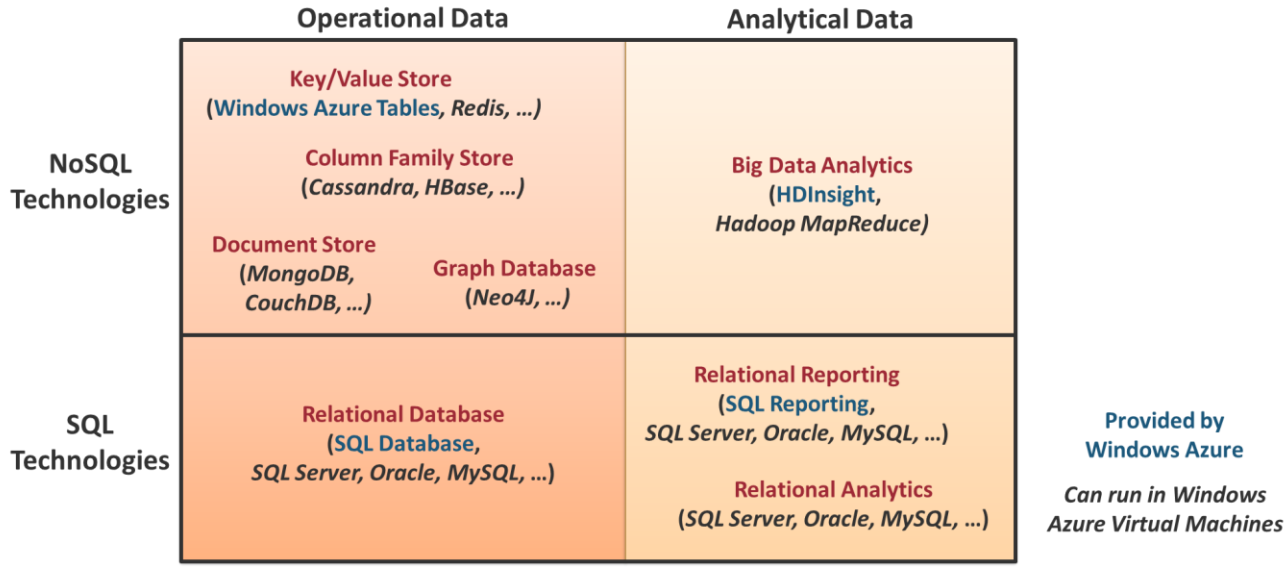

Figure 1: Data technologies on Windows Azure can be organized into four quadrants.

As the figure shows, Windows Azure provides some built-in services for working with relational and non-relational data. It also lets you use other data technologies by running them in Windows Azure Virtual Machines, which provide what's known as *Infrastructure as a Service (IaaS)*[1].

The two quadrants in the bottom row of the table describe the SQL technologies available on Windows Azure. They are:

- *Relational databases*, including the managed service provided by Windows Azure SQL Database and the ability to run other database systems, such as Microsoft SQL Server, Oracle, and MySQL, in Windows Azure Virtual Machines.
- *Relational reporting*, such as the managed service provided by Windows Azure SQL Reporting. You're also free to run other relational database systems in Windows Azure VMs, then use the reporting services they provide.
- *Relational analytics*, which can be performed by SQL Server, Oracle, MySQL, or another relational database system running in Windows Azure VMs.

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

[1] Windows Azure also provides *Platform as a Service (PaaS)* options. PaaS is similar to IaaS in some ways—applications still run in virtual machines—but PaaS technologies take on more of the management burden, such as applying operating system updates.

- *Key/value stores*, including the managed service provided by Windows Azure Tables. You can also run other key/value stores in Windows Azure VMs, such as Redis. For more on key/value stores, see page 7.
- *Column family stores*, which can run in Windows Azure VMs. The options include HBase, which is part of the Hadoop technology family, and Cassandra. Column family stores are described starting on page 10.
- *Document stores*, which can run in Windows Azure VMs. The document store choices include the two most visible options in this category: MongoDB and CouchDB. For a closer look at this technology, go to page 12.
- *Graph databases*, which can run in Windows VMs. The most well-known technology in this category is Neo4J, but other options are also available. Graph databases are described beginning on page 14.
- *Big data analytics*, including the managed service provided by Windows Azure HDInsight. This service implements Hadoop MapReduce, and it's also possible to create and run your own Hadoop cluster in Windows Azure VMs. For more detail on big data analytics, see page 16.

These NoSQL technologies are the topic of this guide. What follows describes each of these options.

NoSQL on Linux on Windows Azure

Some NoSQL technologies can run on Windows. It's much more common today to run them on Linux, however. And even though Microsoft's cloud platform is called "Windows Azure", it also runs Linux VMs, with support for Ubuntu, CentOS, and SUSE. In fact, if you choose to run a NoSQL technology in Windows Azure VMs, you're probably better off running it in Linux even if the technology also runs on Windows. The reason is simple: Since far more people are running NoSQL on Linux, the community support for this combination is significantly stronger than for running it on Windows. Don't be confused by the name (or corporate ownership) of Windows Azure. Running a NoSQL technology in Linux VMs is certainly a viable option.

Windows Azure Relational Technologies: A Quick Look

Before diving into the NoSQL world, it's worth starting with a quick look at the relational technologies that Windows Azure supports. As just described, this cloud platform offers two approaches for working with relational data: using SQL Database and running SQL Server or another relational database in a Windows Azure VM. While the two differ in important ways, they both use the same relational model, which is summarized in Figure 2.

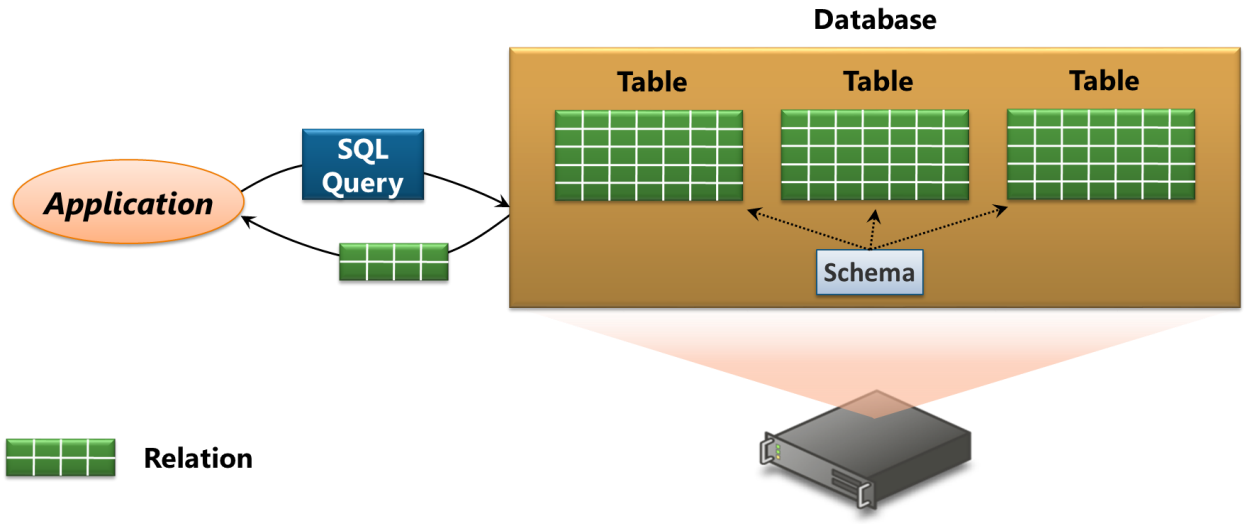

Figure 2: SQL Database and SQL Server in a Windows Azure VM both use the traditional relational model.

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 Boolean. A *schema* describes the columns each table can have, and every table's data is stored in one or more *rows*. Each row contains some value for every column in that table.

An application can issue a *SQL query* against one or more tables. The result is a relation containing values grouped into rows. It's common to design the tables in a relational database so that data is *normalized*—every piece of data is stored just once. To retrieve data using more than one table, an application issues a kind of SQL query called a *join*. An application can also atomically update data contained in one or more tables using a *transaction*.

One or more columns in each table can be designated as the *primary key*. 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. And to help combine data stored in different tables, a table can contain *foreign keys*, which are the primary keys of some other table.

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. Normalization makes life simpler for application developers, since there's no need to find and update multiple instances of the same data. 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 and still provide all of these features. Especially for applications that need to work with very large amounts of data—more than will fit on a single server—it can be better to leave behind the familiar relational world. Those applications and others might instead need to enter the land of NoSQL.

Windows Azure NoSQL Technologies: Operational Data

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 four categories shown in the upper left quadrant of Figure 1: key/value stores, column family stores, document stores, and graph databases. This section looks at each one.

Key/Value Stores

Relational technologies impose a significant amount of structure on data. But what if an application needs almost no structure? There are plenty of situations where all that's required is fast access to large amounts of simply structured information. In situations like this, a key/value store can be a good choice.

Technology Basics

Suppose you wish to create a shopping cart in a web commerce application. 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 slows things down, using one 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 can be a better choice. Figure 3 illustrates the basics of this technology using Windows Azure Tables as a concrete example.

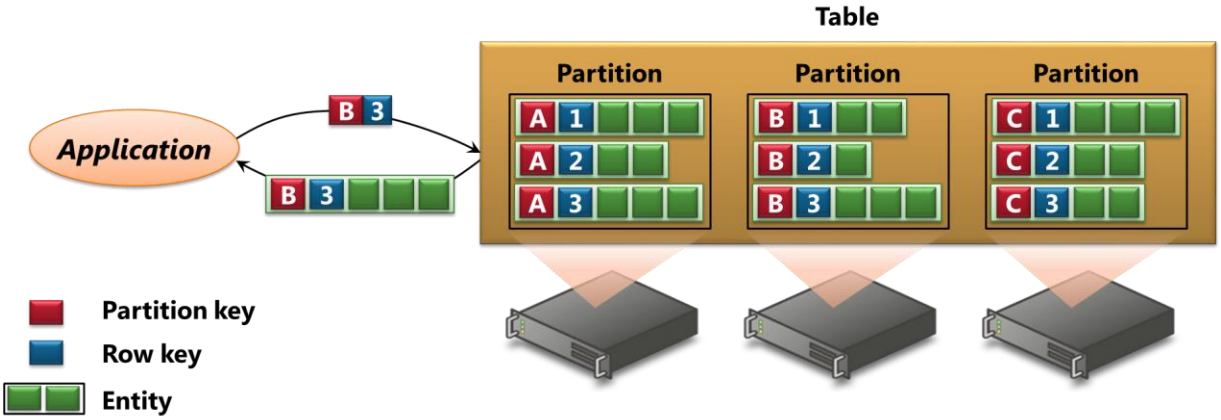

Figure 3: Windows Azure Tables, a key/value store, lets an application supply a two-part key and get back the value associated with that key.

The details vary across different technologies, as do how the components are named, but the basic idea of a key/value store remains the same: an application gives the store a unique key, then gets back one or more values associated with that key. In Windows Azure Tables, for example, data is held in *tables*, which despite their name are nothing like relational tables. Each table consists of one or more *partitions*. Each partition holds some number of *entities*, with each entity containing *properties*.

Each property has a *type*, such as integer or character string or something else, and each one holds a single 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. Unlike a relational database (but like other key/value stores), Windows Azure Tables has no notion of schema. Each entity in a partition can contain different kinds of properties if that's what makes the most sense for an application.

To retrieve an entity, an application provides both 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. It's also possible to do other kinds of queries, such as requesting properties whose keys fall within a specified range of values. Windows Azure Tables has no support for secondary indexes, however, which is typical of key/value stores.

Like many NoSQL technologies, key/value stores are designed to support very large amounts of data. With Windows Azure Tables, the partitions in a single table can be spread across multiple machines. This is different from a typical relational system, where an entire database is usually stored on a single machine. Commonly known as *sharding*, this approach lets databases be much bigger than they would be if all of their data were restricted to a single server. To make life simpler for developers, Windows Azure Tables silently shards data as it's added, letting a table scale automatically. Sharding also brings some constraints, however. Each query can target only one partition, for example—there's no concept of joins—and so data that's frequently accessed together should be kept in the same partition. Transactions also can't span partitions, so while it's possible to do atomic updates on data within a single partition, it's up to the application to ensure correctness for updates that span partitions.

Like other key/value stores, Windows Azure Tables keeps multiple copies of its data on different servers, so a single machine failure won't make that data unavailable. But think about what happens when an application writes

data. Since data is replicated across multiple servers, a write to one copy of the data takes time to propagate to all of the others. While this propagation is happening, it's possible that the same data is read by this or another application. But what does that application see? In general, there are several possibilities. If the system provides *strong consistency*, as does Windows Azure Tables, all applications are guaranteed to see the same value, even if they read this data immediately after it's written. If the system provides *eventual consistency*, the data will eventually be the same across all servers, but it's possible for the application to read an old copy of the data if its read is handled by a server whose copy of the data hasn't yet been updated.

When to Use It

To decide whether a key/value store is the best option for your Windows Azure application, consider three things:

- Does the application work with relatively simple data? Applications that need the powerful query capabilities of a relational database aren't a particularly good fit for key/value stores, while applications that use, say, JSON documents are likely to work better with a document store. For applications in neither category, however—and there are plenty of them—a key/value store can be the right choice.
- Does the application need to work with large amounts of data? Like most NoSQL technologies, key/value stores are typically designed to scale well beyond the limits of a relational database.
- Do you want a low-cost managed service? For example, Windows Azure Tables are significantly cheaper than SQL Database.

All of these things make key/value stores a good choice for any application that needs to store lots of relatively simple data at low cost. As with every NoSQL technology, however, choosing a key/value store rather than a relational database means giving up some things. There's no required schema, so application developers should be careful about types when accessing data. There's likely to be no normalization—the same information might be stored in more than one entity to improve performance—so updating data might require finding and changing multiple copies. Relational tables can also make it simpler to share data across multiple applications, because schemas are commonly designed for this purpose. A key/value store, however, is likely to be more focused on supporting a single application, with its data grouped together based on how it's typically accessed. This makes access for those queries very fast, at the cost of making other kinds of queries (perhaps from different applications) slower. And since the majority of business intelligence tools today are designed to work with relational data, using a key/value store means that those tools can't be used.

Despite these limitations, a key/value store is the right choice in plenty of situations. Its simplicity, low cost, and scalability make it a good match for a number of applications running on Windows Azure and elsewhere.

Column Family Stores

If all you need to do is access simple data quickly, a key/value store can be the right option. But suppose you want a little more structure in your data—maybe a simple set of keys and values isn't enough. In a case like this, you might be better off with a column family store.

Technology Basics

A column family store is similar to a key/value store in that an application supplies a unique key and gets back a set of values. Those values are organized in a somewhat more complex fashion than with a key/value store, however. Figure 4 illustrates the idea, using Cassandra as a concrete example.

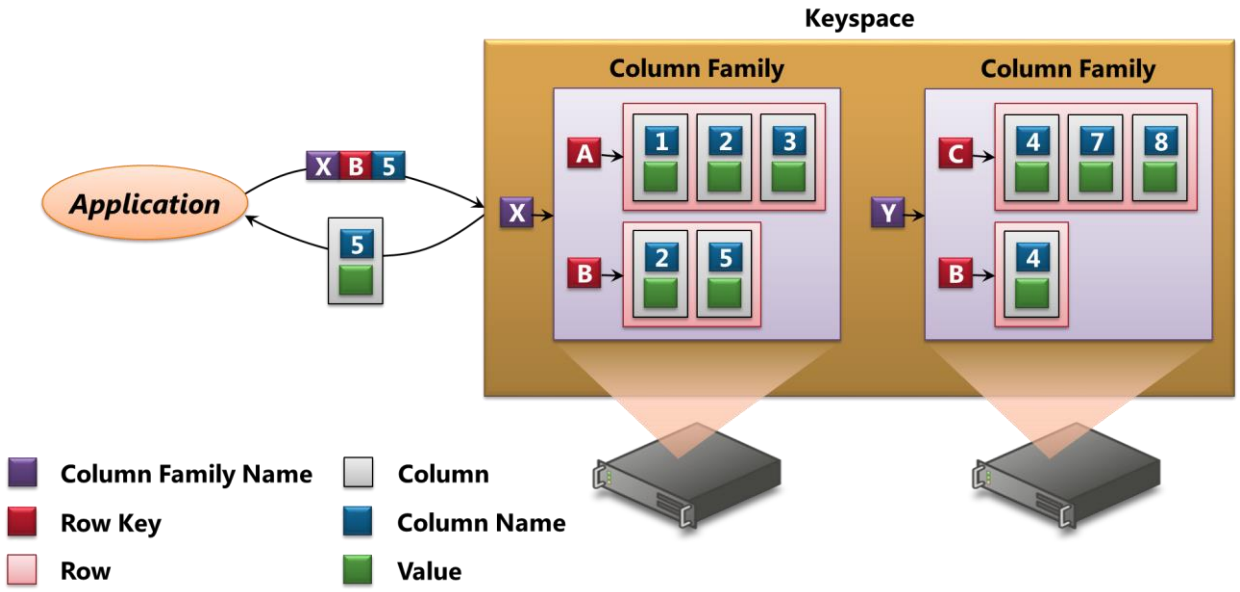

Figure 4: Cassandra, a column family store, groups key/value pairs (called columns) into rows, then groups rows into column families.

As the figure shows, Cassandra stores *column families* grouped into a *keyspace*. Each column family contains some number of *rows*, and each row holds some number of *columns*. These columns are quite unlike columns in a relational table, however. In fact, what Cassandra calls a column is really just a key/value pair.

Different rows in the same column family are free to contain different types of columns—this isn't constrained by schema as it would be in a relational table. The column families in a keyspace must be defined up front, so there is a little schema, but the columns in a particular column family are not—you can add new ones at any time. And the name and value in a column are both just arrays of bytes, so they can be strings or UUIDs or binary values or whatever else you like. To access the value of a particular column, an application can provide a unique key with three parts: column family, row key, and column name. It's much like a slightly fancier key/value store.

But there's more. Every value in every column is versioned, for example, which means that writes don't replace old values. By default, the latest version is returned, but it's possible to explicitly request an earlier version of a column's value. Also, rows and columns are sorted by their keys, which allows fast access. For example, suppose a row contains a million columns, each with a unique column name. Because those columns are sorted by their names, finding even a column somewhere in the middle can be quite speedy. (As this example suggests, this class of NoSQL technology allows very wide columns, which is why column family stores are sometimes called *wide-column* stores.) It's also possible to group column families into *super columns*, providing yet another level of structure for data.

Columns in a single column family are stored together, so it's common to organize data that's accessed together into the same column family. Notice how different this is from a relational database, where the logical and physical structure of data are completely independent. Because NoSQL stores are commonly striving for fast access to lots of data, designing an effective data model typically requires understanding how things are physically stored. The data model should reflect the most common queries that your application will perform. Getting the best performance requires knowing what kinds of questions your application will be asking most often.

Like most NoSQL technologies, Cassandra replicates data, storing the same information on multiple servers to provide fault tolerance. As with key/value stores, this raises the question of consistency: eventual or strong? Cassandra's answer is interesting; an application can decide what level of consistency it wants on a per-request basis. And the choices aren't limited to just eventual or strong. The system also provides various in-between options, letting an application make fine-grained trade-offs between consistency and performance.

When to Use It

When is a column family store the best option for a Windows Azure application? Here are some questions to ask to help make the decision:

- Does the application need to work with large amounts of data? Like key/value stores, column family stores such as Cassandra and HBase are designed to scale well beyond the limits of a relational database.
- Does the application work with data that's too simple to need a relational database but could benefit from more structure than a key/value store provides? There's a sweet spot between these two where column family stores can be the best choice.

A traditional example of when a column family store makes sense is storing information about many web pages. The table's row key could be the page's URL, while the row contains a column for each type of content on this page. The columns can be grouped together into column families to associate related types of data.

As usual with NoSQL approaches, choosing a column family store rather than a relational database means giving up some things. There's virtually no schema and typically no normalization. It might also be hard to share data across applications, since the connection between logical and physical organization implies structuring your data to effectively answer the specific questions posed by a particular application. And because most BI tools today are designed to work with relational data, choosing a column family store means that those tools can't be used. Like other NoSQL technologies, though, a column family store running in Windows Azure VMs can be just the right solution for some scenarios.

Document Stores

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. This is a primary motivation for the creation of document stores.

Technology Basics

Suppose you're creating an application that sends and receives JSON data. With JSON, information is represented as character strings organized into name/value pairs called *properties*. For example, information about an order might be described like this:

```
{
   "Order": {
     "OrderID": 5630,
     "Status": "Awaiting delivery",
     "CustID": 8499734 }
}
```

As the name suggests, JSON is derived from the syntax of JavaScript, which makes it easy for JavaScript developers to use. JSON support is available for multiple languages, however—it's not only for JavaScript—and because it provides a simple and compact way to represent data, it's become a popular way to serialize information for transmission and storage.

It's certainly possible to convert JSON's text-based format into the rows and columns of a relational table. But why do this? If an application is sending and receiving JSON data, why not just store it as JSON? Especially if the application is written in JavaScript, using a native JSON storage technology—a document store—can make life simpler. If that storage technology is very scalable, all the better. Figure 5 illustrates the basics of this idea using MongoDB as an example.

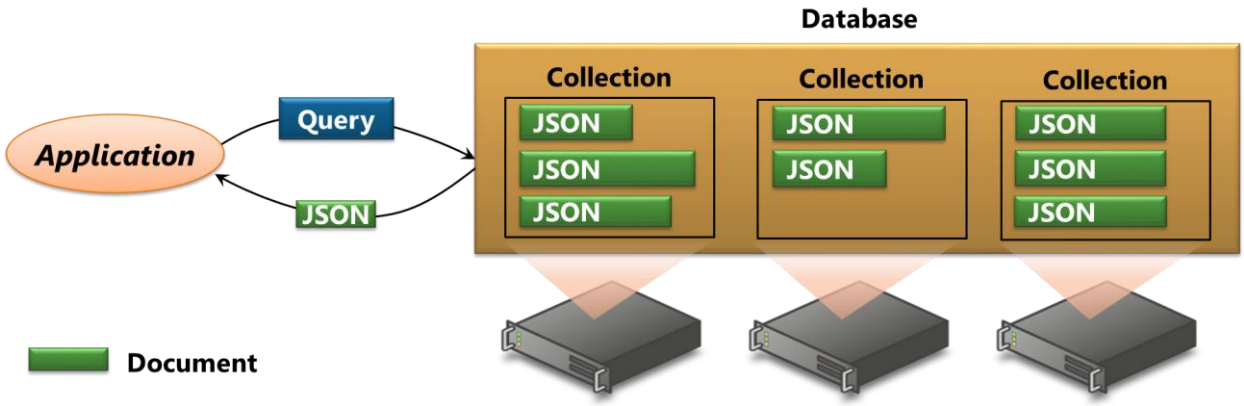

Figure 5: MongoDB, a document store, lets an application issue queries on JSON documents.

In MongoDB, a *database* contains a *collection* of JSON *documents*[2]. (More precisely, MongoDB stores its data in a binary version of JSON called *BSON*.) The documents in a particular collection might all look the same, with each one containing, say, the information for a specific order in the format shown earlier. This isn't required, however; it's entirely legal for a collection to contain a group of documents where each one has a different structure.

[2] Don't be confused: The word "document" here has nothing to do with, say, Microsoft Word documents. A JSON document is just a bunch of JSON text as shown earlier.

MongoDB enforces no schema, and so unlike a relational table, where each row holds data in a fixed set of columns, a document in a collection can contain whatever the developer likes.

To access data, an application issues queries, each of which returns JSON data. Each document has a unique identifier, so it's possible to treat the database like a key/value store: the unique identifier is the key, while the document itself is the value returned. MongoDB also allows creating secondary indexes on specific properties, such as OrderID in the JSON example shown earlier, allowing more fine-grained queries.

Like most other NoSQL technologies, MongoDB is designed to support very large amounts of data. As Figure 5 shows, a database's collections can be spread across multiple machines. While this helps with scale, it also brings some constraints. Each query can target only one collection, for example—there's no concept of joins—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.

MongoDB also stores multiple copies of each collection on different systems, so a single machine failure won't make a collection's documents unavailable. For writes, MongoDB supports both eventual and strong consistency.

On Windows Azure, you're free to install MongoDB in Linux VMs and manage it yourself. If you'd rather use a managed service, however, you can instead use the MongoDB offering provided by MongoLabs. Available in the Windows Azure Store, this service is analogous to Windows Azure SQL Database—you can get access to a functioning database in just a few minutes—but the underlying database is based on MongoDB rather than SQL Server.

When to Use It

To decide whether a document store is the right choice for your Windows Azure application, you should ask two main questions:

- Does the application work with JSON data? If so, a document database is likely to be a natural choice. This is especially true if the data is highly structured, e.g., it contains nested documents.
- Does the application need to work with large amounts of data? Because document stores are designed for scale in a way that relational databases are not, they can be a good option for storing and accessing large data sets. (In fact, the name "Mongo" is derived from the word "humongous".)

For example, think about creating a consumer application on Windows Azure. The clients, which probably include mobile devices, are likely to send and receive JSON data, and the application itself might even be written in JavaScript. (Windows Azure provides direct support for Node.js and other server-side JavaScript options.) If you're successful, the application will have lots of users, and so designing for scale makes sense. As always, choosing a document store rather than a relational system means giving up some things, such as the comfort of schema, atomic transactions that span collections, and the use of familiar BI tools. But given the increasingly widespread use of JSON and the growing need for scale, a document store can make very good sense.

Use a Managed Service? Or Run Your Own Database Servers?

Once you've chosen to use a NoSQL technology on Windows Azure, you next need to decide whether to use a managed service or to run your own database servers. Managed NoSQL services, such as the key/value store provided by Windows Azure Tables and the MongoLabs document store, require essentially no set up or administration—your application can begin using them immediately. Running your own servers requires significantly more work, since you must set up the software and manage its operation. But relying on a managed service brings some constraints, while controlling your own storage technology gives you more options. And not all NoSQL technologies are available as Windows Azure managed services today, so you might have no choice but to run your own. In general, though, it's a good idea to use a managed service whenever it works for your application. Why do extra administrative work when somebody else will do it for you?

Graph Databases

The one thing that all NoSQL technologies have in common is that they don't rely on the relational data model. Key/value stores hold just simple properties, column family stores organize properties in a somewhat more structured way, and document stores group their information into collections of JSON documents. The fourth category of NoSQL technologies also stores data in a non-relational way, but it takes a very different approach. Rather than aggregating data, a graph database spreads it out into a graph.

Technology Basics

Data items, things like customer, order, and web page, are clearly important. But how important are the relationships among these data items? The answer depends on what your application needs to do with that data.

In some situations, relationships among data items don't matter much at all, and so a simple key/value store is fine. In others, representing relationships with foreign keys works well, and so using a relational database and joins can be a good solution. But think about the case where the relationships between data items are as important as the items themselves. If the relationships matter so much, why make your application reconstruct them from the data using joins? Why not embed the relationships in the data itself?

This is exactly what a graph database does. Figure 6 illustrates the idea, using Neo4J as an example.

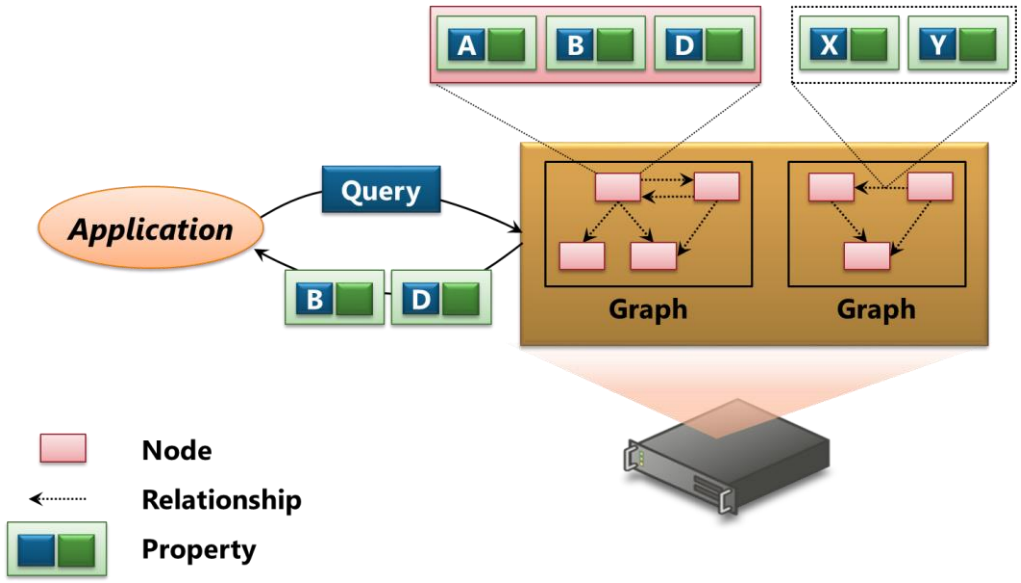

Figure 6: Neo4J, a graph database, stores graphs where both nodes and the relationships between nodes can have properties.

As its name suggests, a graph database represents data as a graph. In Neo4J, the data items are called *nodes*, while the edges that connect those nodes are called *relationships*. And as the figure shows, both nodes and relationships can have *properties*, which are just name/value pairs. Neo4J doesn't enforce schema, so these properties can contain whatever information is most useful. A typical query indicates where to start, such as a particular node in the graph, what data to match, and which nodes, relationships, and/or properties to return.

A familiar example of graph data today is the social graph maintained by various web applications. The people are important—they're the nodes in the graph—but the relationships among these people are at least as important. Representing this kind of information as a graph is entirely natural. This approach also makes some kinds of common queries significantly faster. If Ann is represented by a node in the graph, for example, finding who her friends are can be simple: just look at the Friend relationships her node has with other people. To find the friends of Ann's friends, do this one more time, looking at the Friend relationships of her friends. Answering questions like this can be much faster with a graph database than with traditional relational technology. Rather than working out the connections between nodes at run time using joins and foreign keys, those relationships are right there in the data.

Social graphs aren't the only example; other kinds of data can also be represented as a graph. Geospatial data might be modeled in this way, for instance, with each node representing a location and each relationship a route between locations. Doing this could make calculating routes between locations easier, and it also mirrors the way people naturally think about this kind of information. Even data like a user's order history could be modeled as a graph, with orders over time connected into a graph.

Like every other technology, graph databases come with trade-offs. For example, while queries that match the graph structure can be blazingly fast, those that don't can be much slower than with a relational database. Also, splitting a graph across multiple servers is challenging. This means that providing scalability by sharding data, as is commonly done in other NoSQL technologies, is uncommon in graph databases.

When to Use It

To decide whether a graph database is the right choice for your Windows Azure application, consider two main issues:

- Is a graph the most natural way to work with your data? As usual with NoSQL solutions, you use them most often when a standard relational database won't work well enough. A graph database is a good choice when it's hard to work with your data in any other form than as a graph.
- Can your application live within the scale constraints imposed by a graph database? If it's running on a good-sized server, a graph database can certainly handle lots of data. But it's not designed for the massive datasets supported by key/value stores or column family stores or document stores. If your application needs operational access to very large amounts of data, another NoSQL technology is probably a better choice.

As with other NoSQL technologies, choosing a graph database rather than a relational system means giving up some things, such as schema and the use of familiar BI tools. But if storing and working with your data as a graph is the best option for your Windows Azure application, you should definitely consider using a graph database.

Windows Azure NoSQL Technologies: Analytical Data

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. Storing this operational data as it changes over time, then analyzing it for patterns, trends, and other information can also 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. It might also be too big or too diverse 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 MapReduce. Windows Azure supports this technology in two different ways: you can run it yourself in Windows Azure Virtual Machines or you can use the managed service provided by Windows Azure HDInsight. This section looks at both options.

Hadoop MapReduce

The term "Hadoop" is commonly used to refer to a group of open source technologies. Exactly what this group includes is a little unclear—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.

The Hadoop technology family also includes other components, including:

- HBase, the column family store mentioned earlier. HBase is built on HDFS, but it's designed for operational data rather than analytical data.
- 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.

All of these (and more) can be run in Windows Azure Virtual Machines.

Technology Basics

How can an application analyze very large amounts of data? One approach is to run the application on a very big computer. But if the data can be broken into pieces, with each piece analyzed independently, there's another solution. Rather than use one giant machine, you can instead spread the data across many smaller computers, then run a copy of the application on each of those machines. Creating a cluster of commodity servers is cheaper than buying one really big server, and it can also expand to handle datasets too big for any single machine.

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, as Figure 7 shows.

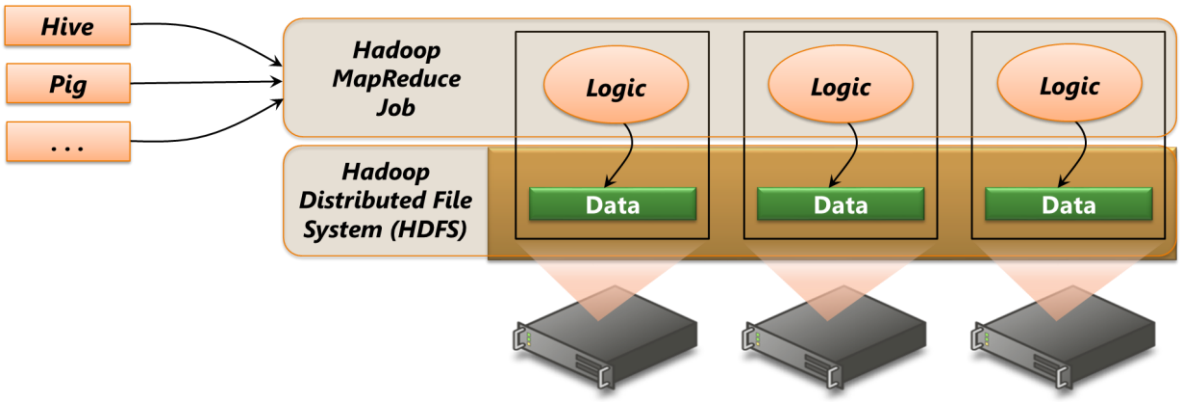

Figure 7: Hadoop MapReduce supports parallel applications running across multiple servers that process distributed data stored in HDFS.

A MapReduce application, commonly called a *job*, is divided into multiple distinct instances. Each instance runs on a server that stores the data this instance is processing. This is an important idea: with MapReduce, the logic is typically moved to the data rather than the other way around. The more servers you have, the faster your application can complete its task.

This kind of problem is a perfect fit for a public cloud platform like Windows 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. To do this, you can create your own Hadoop cluster

using Windows Azure Virtual Machines, then create and run MapReduce jobs on that cluster. As the figure shows, it's also possible to use Hive, Pig, and other tools to create those jobs. Windows Azure VMs are just ordinary virtual machines, so this open source software will happily run in them.

When to Use It

Hadoop MapReduce has become the default technology for processing big data. But "big data" is hard to define clearly—opinions vary. One way to think about it is to consider using Hadoop MapReduce whenever traditional relational BI won't work, such as when data is too big or too unstructured. It's also become common to support cloud-based applications with an operational NoSQL store, such as Windows Azure Tables, HBase, or something else, then analyze the data it contains using Hadoop MapReduce. If the data you're analyzing is created on Windows Azure—it's born in the cloud—why not do the analysis on Windows Azure as well?

But when should you create your own Hadoop cluster on Windows Azure? Creating and managing a cluster isn't simple, and as described next, Windows Azure HDInsight will do this for you. There are a few situations, though, where it might make sense for you to run your own cluster on Windows Azure. They include these:

- You might have an existing Hadoop cluster that you want to lift and shift onto Windows Azure. Doing this would let you expand the cluster at will with more pay-as-you-go VMs.
- Windows Azure HDInsight uses the Hortonworks Hadoop distribution. If you want your Hadoop cluster to use a different distribution while still retaining the flexibility of Hadoop in the cloud, creating a cluster on Windows Azure VMs might make sense.
- You might want to use some aspect of Windows Azure that's not supported by HDInsight. For example, Windows Azure Virtual Network lets you configure a virtual private network (VPN) connection between your on-premises network and a group of Windows Azure VMs. HDInsight doesn't support this today, but you can create your own Hadoop cluster on Windows Azure to which your users can have VPN access.

Even though there are some situations where creating your own Hadoop cluster on Windows Azure makes sense, it's probably fair to say that using HDInsight will more often be a better solution. How this looks is described next.

HDInsight

Windows Azure SQL Database provides a managed relational database service in the cloud. MongoLabs provides a managed MongoDB service in the cloud. In an analogous way, Windows Azure HDInsight provides a managed Hadoop MapReduce service in the cloud.

Technology Basics

To use HDInsight, you start by asking it to create a Hadoop cluster, specifying the number of VMs you need. Rather than deal with this complexity yourself, you can let Windows Azure to do it for you. Just as important, you can shut the cluster down—and stop paying for its VMs—when you're done using it. 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 provides a Hadoop environment in the cloud. Figure 8 shows the main components.

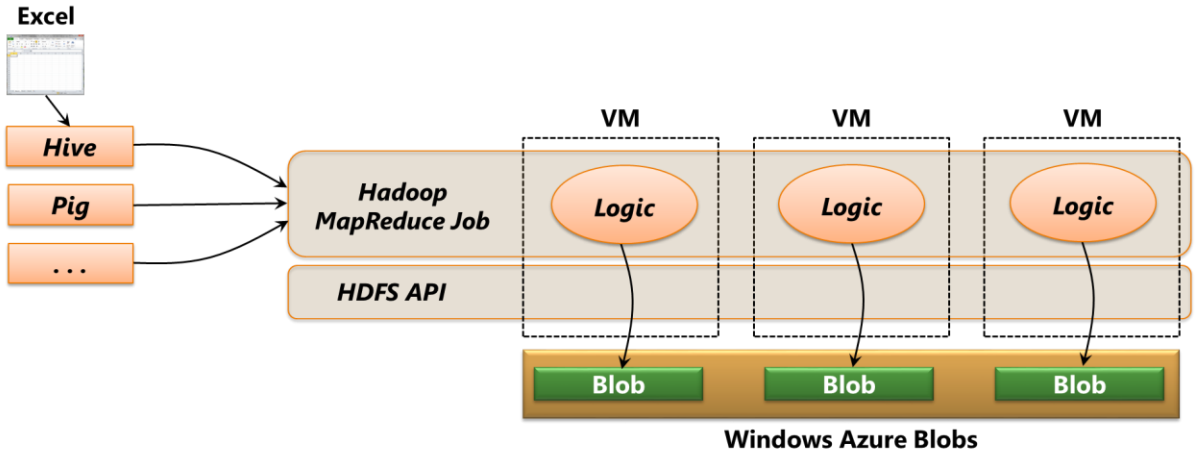

Figure 8: Windows Azure HDInsight implements Hadoop MapReduce on top of Windows Azure Blobs.

As the figure shows, your MapReduce job runs in VMs, and it accesses data through the standard API provided by HDFS. The data this job works on isn't stored persistently in HDFS, however. Instead, it's kept in Windows Azure Blobs, a low-cost way to store unstructured data. Because this data is exposed through the HDFS API, however, existing MapReduce applications can run unchanged. (To let multiple jobs run over the same data, HDInsight also allows copying data from Blobs into HDFS running in VMs.)

HDInsight supports MapReduce jobs written in various languages: Java, C#, F#, and even JavaScript. It also allows using Pig Latin and HiveQL, both of which make it easier to create MapReduce jobs. And as Figure 8 suggests, Microsoft provides a way for users to analyze Hadoop data using Excel.

When to Use It

There are situations where creating your own Hadoop cluster on Windows Azure VMs can make sense. Most of the time, though, 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 Windows Azure 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, Windows Azure provides both relational and NoSQL options.

Like all technologies, NoSQL offerings have pros and cons. Most likely, relational storage will continue to be the right choice for a majority of your new applications. But when a SQL-based option isn't appropriate, don't be afraid to go with NoSQL. Windows Azure supports every major NoSQL approach today, either as a native 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.