David Chappell

Microsoft Azure Data Technologies: An Overview

Sponsored by Microsoft Corporation

Copyright © 2014 Chappell & Associates

Contents

Blobs 3
Running a DBMS in a Virtual Machine 4
SQL Database 5
DocumentDB 7
Tables 8
HDInsight 10
Hadoop MapReduce 10
Hadoop HBase 12
Search 13
Conclusion 15
About the Author 15

Managing and analyzing data in the cloud is just as important as it is anywhere else. To help you do this, Microsoft Azure provides a range of technologies for working with relational and non-relational data. This paper introduces some of the most important options.

Blobs

The word “blob” is short for “Binary Large OBject”, and it describes exactly what a blob is: a collection of binary information. Yet even though blobs are simple, they’re quite useful. Figure 1 illustrates the basics of Azure Blobs, a fundamental storage service provided by this cloud platform.

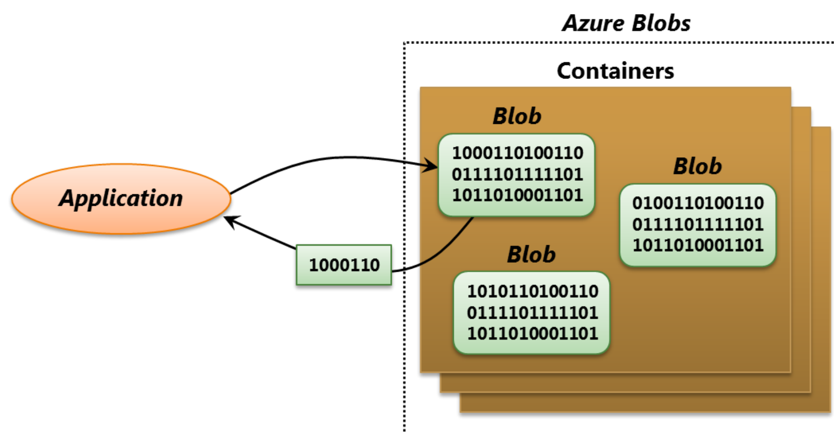

Figure 1: Azure Blobs stores binary data—blobs—in containers.

To use blobs, you first create an Azure *storage account*. As part of this, you specify the Azure datacenter that will store the things you create using this account. You next create one or more *containers* in your storage account, then create blobs in those containers.

Azure provides two different kinds of blobs. The choices are:

- *Block* blobs, each of which can contain up to 200 gigabytes of data. As its name suggests, a block blob is subdivided into some number of blocks. If a failure occurs while transferring a block blob, retransmission can resume with the most recent block rather than sending the entire blob again.

- *Page* blobs, which can be as large at one terabyte each. Page blobs are designed for random access, and so each one is divided into some number of pages. An application is free to read and write individual pages at random in the blob. For example, Azure’s infrastructure as a service (IaaS) technology, called *Azure Virtual Machines*, uses page blobs as persistent storage for virtual machines.

Whether you choose block blobs or page blobs, applications can access blob data directly through a RESTful interface or using the Azure Storage Client library, which provides a more developer-friendly wrapper for the raw RESTful services. Blobs are also used behind the scenes in other technologies. One example is Microsoft StorSimple, an on-premises appliance that automatically moves data between local disk and Azure Blobs based on how that data is used. Other Azure data technologies also rely on Blobs, as described later.

To guard against hardware failures and improve availability, every blob is replicated across three physical servers in an Azure datacenter. A write to a blob updates all three copies before it returns, so later reads won't see inconsistent results. Put more formally, blobs provide what's known as *strong consistency*. You can also specify that a blob's data should be copied to another Azure datacenter in the same geographic region but at least 500 miles away. This copying, called *geo-replication*, happens within a few minutes of an update to the blob, and it's useful for disaster recovery. Data in blobs can also be made available via the Azure *Content Delivery Network (CDN)*. By caching copies of blob data at dozens of servers around the world, the CDN can speed up access to information that's accessed repeatedly.

Simple as they are, blobs are the right choice in many situations. Storing and streaming video and audio are obvious examples, as are backups and other kinds of data archiving. Developers can also use blobs to hold any kind of unstructured data they like. Having a simple, low-cost way to store and access binary data turns out to be really useful.

Running a DBMS in a Virtual Machine

Many applications today rely on some kind of database management system (DBMS). Relational systems such as SQL Server are the most frequently used choice, but non-relational approaches, commonly lumped together under the banner of *NoSQL*, get more popular every day. To let cloud applications use these data management options, Azure allows you to run a DBMS (relational or NoSQL) in an IaaS VM. Figure 2 shows how this looks with SQL Server.

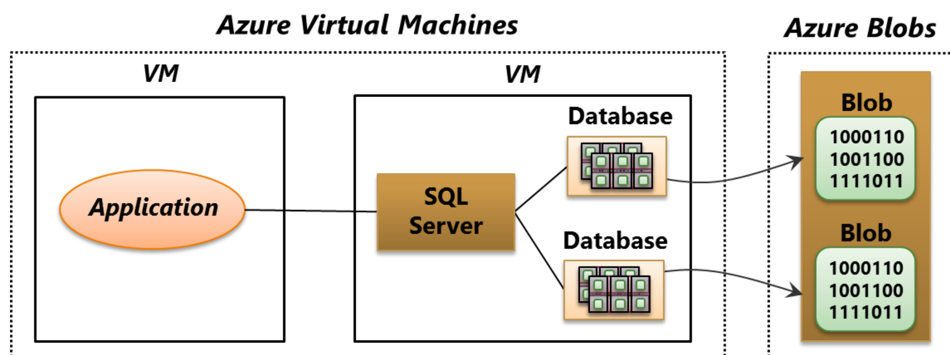

Figure 2: Azure Virtual Machines allows running a DBMS in a VM, with persistence provided by blobs.

To both developers and database administrators, this scenario looks much like running the same software in their own datacenter. In the example shown here, for instance, nearly all of SQL Server's capabilities can be used, and you have full administrative access to the system. You also have the responsibility of managing the database server, just as if it were running locally.

As the figure shows, your databases appear to be stored on the local disk of the VM the server runs in. Under the covers, however, each of those disks is written to an Azure blob. (It's similar to using a SAN in your own datacenter, with a blob acting much like a LUN.) As with any Azure blob, the data it contains is replicated three times within a datacenter and, if you request it, geo-replicated to another datacenter in the same region. It's also possible to use options such as SQL Server database mirroring for improved reliability.

A common scenario where running a DBMS in a VM makes sense is when an organization moves an on-premises application to Azure. Rather than change to a different data technology, it's simpler to stick with what already works. For new applications created on Azure, however, using one of the data technologies described next might be a better solution.

SQL Database

For many people, running a DBMS in a VM is the first option that comes to mind for managing data in the cloud. It's not the only choice, though, nor is it always the best approach. In some cases, managing data using a *Platform as a Service (PaaS)* technology makes more sense. A PaaS solution is managed for you by Azure, and so it's typically faster to set up, less work to administer, and cheaper to manage.

Azure's managed service for relational data is SQL Database. Figure 3 illustrates the idea.

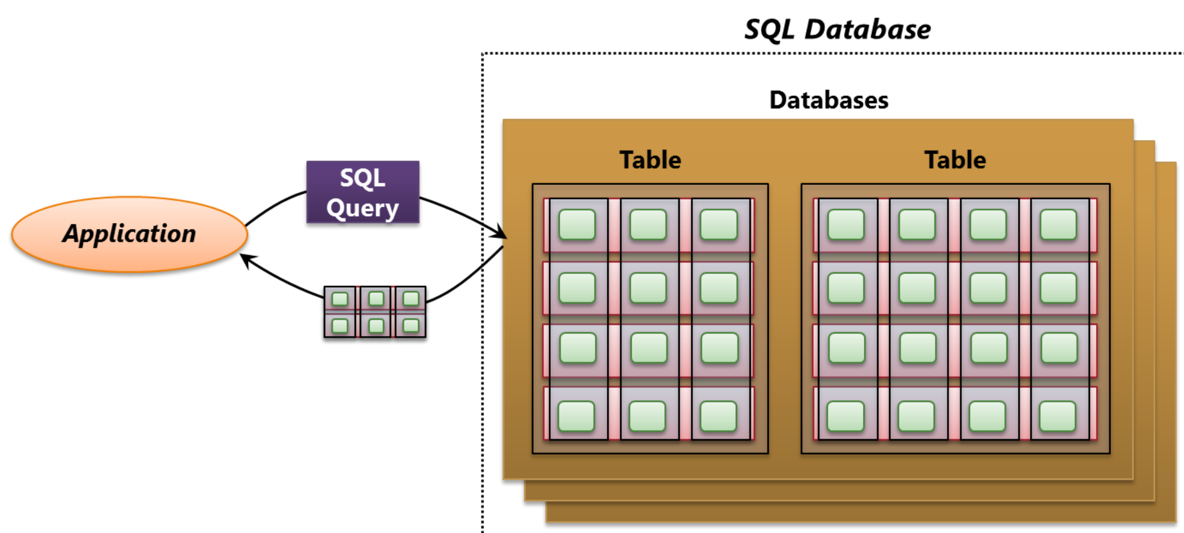

Figure 3: SQL Database provides a managed service for relational data.

SQL Database doesn't give each customer its own physical instance of SQL Server. Instead, each customer gets a logical SQL Database server that she can use to create databases and relational tables. As with Blobs, all data in SQL Database is replicated on three separate physical servers within an Azure datacenter, giving your databases built-in high availability.

To an application, SQL Database looks much like SQL Server. Applications can issue SQL queries against relational tables, use T-SQL stored procedures, and execute transactions across multiple tables in a database. And because applications access SQL Database using the Tabular Data Stream (TDS) protocol, the same approach used to access SQL Server, they can work with data using Entity Framework, ADO.NET, JDBC, and other familiar data access interfaces.

Yet because SQL Database is a cloud service running in Azure data centers, you don't need to manage any of the system's physical aspects, such as disk usage. You also don't need to worry about updating software or handling

other low-level administrative tasks. Each customer organization still controls its own databases, of course, including their schemas and user logins, but many of the mundane administrative tasks are done for you.

Using SQL Database is different from running your own SQL Server instance in another way, too. Because SQL Database is a shared service, it's being used simultaneously by lots of different applications owned by many different organizations. How can you be sure that you'll get the performance you need from the service when you can't predict how other applications will use it? The answer is that users of SQL Database pay for a specific number of *database throughput units (DTUs)*, each of which provides a specific amount of throughput. By buying the DTUs your application requires, you can guarantee predictable performance from this shared service.

Different applications need different levels of throughput, and so SQL Database offers three tiers of service, each with different numbers of DTUs:

- Basic, intended for applications with a relatively light transactional workload.
- Standard, designed to support mainstream business applications.
- Premium, intended to support large-scale, mission-critical databases.

SQL Database also provides other services, including:

- Geo-replication, which lets you keep replicas of a database in other Azure datacenters, then have committed transactions copied asynchronously to those replicas. While only the primary replica is read/write, you can choose to make the others readable with an option called *active geo-replication*. Among other things, this is useful for disaster recovery; if the datacenter holding the primary replica fails, a correct copy of the data is still available in another replica.
- Backup and restore, with SQL Database automatically creating backup copies of your databases. You can then use these backups to restore a database to its state at a specified point in time. The service also lets you store backups in a different Azure datacenter, helping make sure they'll be available even if the primary datacenter is down.
- Auditing, which tracks reads, writes, and other access to your databases. This kind of audit trail can be essential for meeting some compliance requirements, and it's also useful in other ways. For example, having a detailed view of how applications are using your data can help you detect security violations or understand when and where a database should be replicated.
- SQL Data Sync, which lets you synchronize data between SQL Database and another database in either SQL Database or SQL Server. Suppose you're running an application in multiple Azure datacenters, for instance, with data stored in SQL Database, and you need all copies of the data to be read/write. You can use SQL Data Sync to keep that data synchronized (although this synchronization isn't atomic across the copies—there's some delay). You can also use SQL Data Sync to synchronize data between a local SQL Server database used by on-premises applications and a cloud copy in SQL Database that's used by applications running on Azure.

The goal is to support a range of scenarios with a variety of requirements. And because SQL Database's PaaS approach makes managing data simpler and less expensive, it can be the right choice in many situations.

DocumentDB

Relational data is certainly useful, but it's not the only game in town. NoSQL solutions get more popular every day. One reason for this is that relational databases can be hard to scale, so if you're building an application that must handle lots of users and lots of data, you might look elsewhere. Developers also choose NoSQL databases because mapping their application's data to and from relational tables can be problematic—they'd like a simpler solution—or because they'd like to avoid the constraints of a fixed schema or for other reasons.

Document databases, one of the primary NoSQL categories, can address all of these concerns. As with a relational DBMS, you're free to run a document database yourself in Azure VMs; plenty of people do this today using MongoDB or another choice. To make it easier to use this approach, however, Azure provides DocumentDB, a PaaS document database service. Figure 4 illustrates the basics of this managed service.

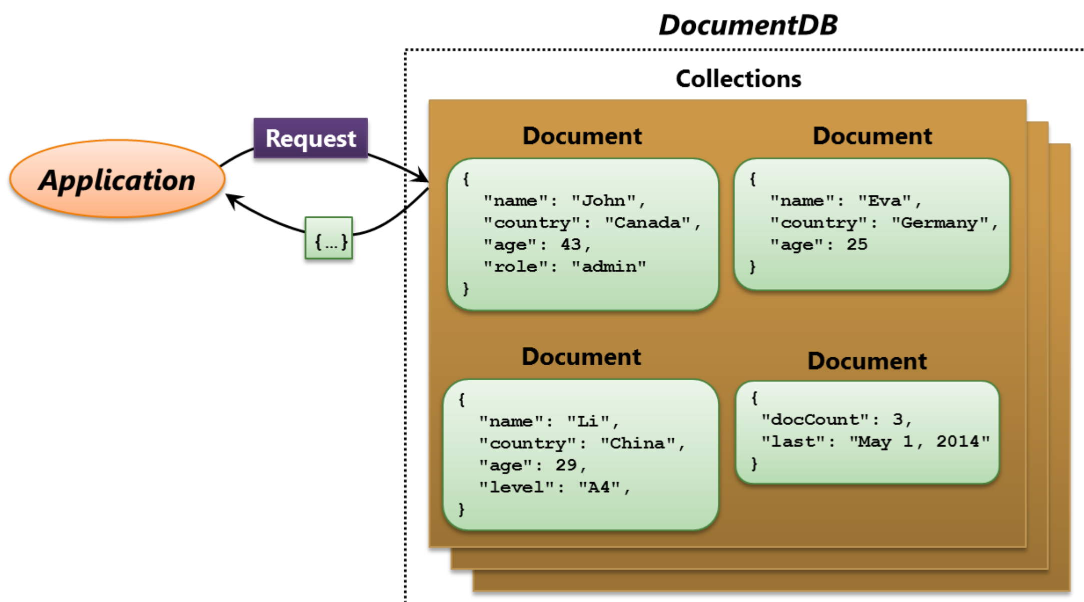

Figure 4: DocumentDB provides a managed service that stores JSON documents in collections.

In DocumentDB, all of an application's data is stored in *documents* grouped into *collections*. Each document contains data described in JavaScript Object Notation (JSON), which is a simple text-based format. As the figure shows, a collection can contain documents containing similar data or completely different data—there's no fixed schema. And because DocumentDB is a managed service, similar to SQL Database, there's no need to install or manage DBMS infrastructure.

To work with this data, applications have a few choices, all of which return JSON. The options are the following:

- Applications can use DocumentDB's RESTful interface, either directly or via the client libraries provided for .NET, JavaScript, Python, and other environments.
- The service provides a query language called *DocumentDB SQL*. As the name suggests, this language's syntax is based on SQL, today's most widely used query language.

- DocumentDB supports stored procedures and triggers, both written in JavaScript, that run within the service itself.

Document DB is designed to allow fast access to data. One way it does this is by automatically indexing all of the data in every JSON document. For example, in the documents shown in Figure 4, the system would automatically create indexes for name, country, age, role, level, and every other JSON element in each document. Rather than make developers decide up front which data they'd like fast access to, and thus which indexes they should create, DocumentDB just indexes everything.

A single database can contain many collections spread across many different servers. As a result, each DocumentDB database can hold hundreds of terabytes of data. But dividing data in this way brings some constraints. While it's possible to wrap multiple updates into a transaction, for example, those updates must all be to documents in the same collection—a transaction can't span collections. Also, each request an application makes must target a specific collection, so an application needs to have some idea where the data it's looking for is stored.

Like other Azure data services, DocumentDB maintains multiple replicas of each collection on different machines. This improves both the performance and the reliability of the service. But DocumentDB doesn't mandate strong consistency. Strong consistency guarantees that applications won't see old data or out-of-order updates, but it can also slow things down. Yet some applications can accept seeing older data on occasion if the result is better performance, and so while DocumentDB does offer strong consistency, it also provides other options. The goal is to let developers make the tradeoff between performance and consistency that's best for their application.

Document databases are rapidly gaining in popularity. Developers like the simple data model, the flexibility of having no required schema, the scalability, and more. The goal of DocumentDB is to combine these benefits with the simplicity of a managed service, while still providing traditional database features like a SQL-based query language and stored procedures[1].

Tables

NoSQL databases come in several different flavors. One popular approach today is what's known as a *key/value store*. If your application needs fast, simple access to large amounts of loosely structured data, this NoSQL style might be your best option.

Azure Tables provides a PaaS key/value store. In other words, it offers this NoSQL style as a managed service. Don't be confused, though: despite its name, Tables doesn't support standard relational tables. Instead, it associates a set of data with a particular key, then lets an application access that data by providing the key. Figure 5 illustrates the idea.

---

[1] For more on this topic, see Introducing DocumentDB: A NoSQL Database for Microsoft Azure.

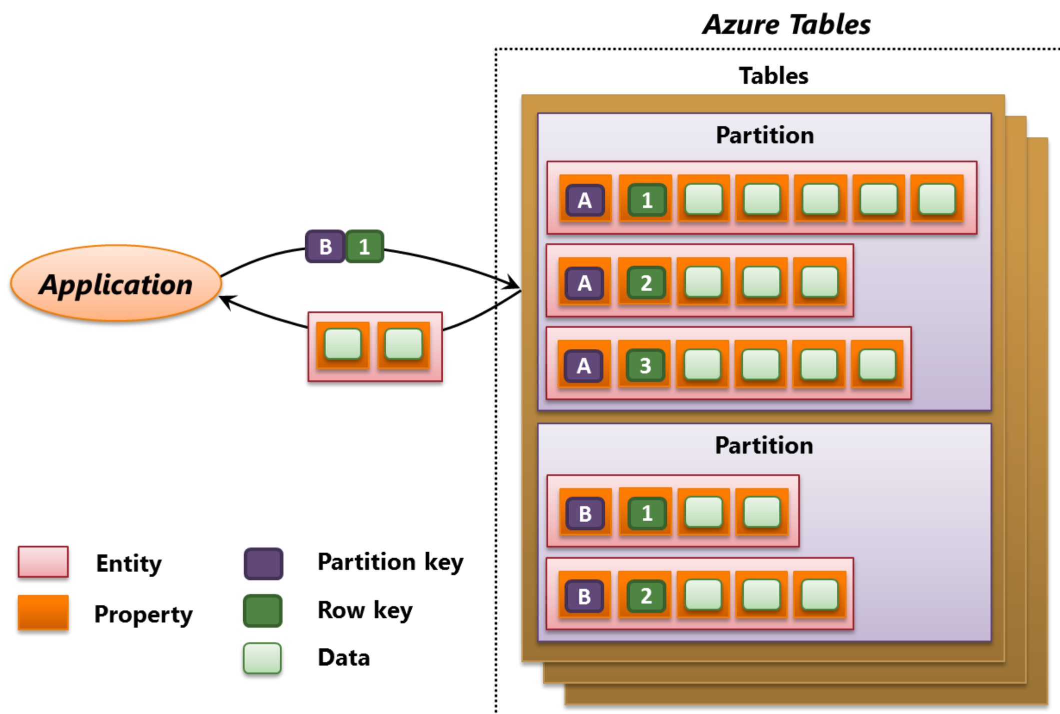

Figure 5: Azure Tables is a managed service that provides fast, simple access to large amounts of data.

As the figure shows, each table is divided into some number of *partitions*, each of which can be stored on a separate machine. Each partition is replicated on multiple machines, and Tables provides strong consistency for reads and writes. Like Azure Blobs, applications can access a table directly through its RESTful interface or using the wrappers provided by the Azure Storage Client library.

Every partition in a table holds a group of *entities*, each containing some number of *properties*. Every property has a name, a type (such as Binary, Bool, DateTime, Int, or String), and a value. These tables have no fixed schema, and so different entities in the same table can contain properties with different types. One entity might have just a String property containing a name, for example, while another entity in the same table has two Int properties containing a customer ID number and a credit rating.

To identify a particular entity within a table, an application provides that entity's key. The key has two parts: a *partition key* that identifies a specific partition and a *row key* that identifies an entity within that partition. In Figure 5, for example, the client requests the entity with partition key B and row key 1. Azure Tables returns this entity, including all of the properties it contains.

This structure lets tables be very large, and it allows fast access to the data they contain. It also brings limitations, however. For example, there's no support for transactional updates that span tables or even partitions in a single table. A set of updates to a table can only be grouped into an atomic transaction if all of the entities involved are in the same partition. There's also no way to create secondary indexes (i.e., indexes on properties other than the key), nor is there support for joins across multiple tables. And unlike traditional relational databases, Azure Tables has no support for stored procedures or triggers.

Azure Tables is a good choice for applications that need fast, cheap access to large amounts of loosely structured data. For example, an Internet application that stores profile information for lots of users might use tables. Fast access is important in this situation, and the application probably doesn't need the full power of a relational database. Giving up this functionality to gain speed and size might make sense, and so Tables can be just the right solution in cases like this.

HDInsight

The way we think about data analysis has changed significantly in the last few years. A vast amount of data is now available, and given the ever-dropping cost of storage, why not save it? And why bother to transform that data into the relational format used by traditional data warehouses? Why not instead just leave the data in its unstructured form, then create applications that know how to work with it intelligently?

Storing all of this data and building applications to analyze it requires a new kind of software foundation. Remarkably, our industry has largely settled on a single approach for doing this: an open source technology called *Hadoop*. Even though it has one name, Hadoop is actually a family of technologies designed to run on a cluster of servers. Figure 6 shows the basics.

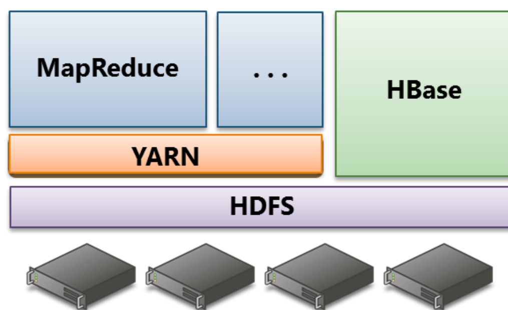

Figure 6: Hadoop is a group of technologies that runs on a cluster of servers.

To store lots of data cheaply, Hadoop includes the *Hadoop Distributed File System (HDFS)*. HDFS provides a way to store very large files using low-cost commodity disks. On top of this, Hadoop includes *Yet Another Resource Negotiator (YARN)*. YARN is cluster management software, and it allows running a variety of other technologies on the cluster. For example, it's common to run *MapReduce* in a Hadoop cluster, software that supports creating distributed applications for analyzing big data. As the figure shows, the Hadoop family also includes *HBase*, a NoSQL database that runs directly on top of HDFS.

Azure HDInsight is a managed service that supports all of these technologies. The rest of this section looks a little more closely at the two most visible aspects of HDInsight: MapReduce and HBase.

Hadoop MapReduce

Organizations have been building data warehouses for decades. These collections of information, traditionally stored in relational tables, let people work with and learn from data in many different ways. With SQL Server, for instance, it's common to use tools such as SQL Server Analysis Services to do this.

But suppose you want to do analysis on non-relational data. Your data might take many forms: information from sensors or RFID tags, log files in server farms, clickstream data produced by web applications, images from medical diagnostic devices, and more. Along with being non-relational, this data might also be too big to be used with a traditional data warehouse. Big data problems like this, rare just a few years ago, have now become quite common, and they're exactly what Hadoop MapReduce is designed to address.

MapReduce provides a framework for running a distributed application across a cluster of machines. In general, each piece of the application works with the data stored on the machine that piece is running on. Because data is processed in parallel, even large data sets can be analyzed in a reasonable amount of time. The more machines a MapReduce application has to use, the faster it can complete whatever work it's doing.

This kind of problem is a natural fit for the public cloud. Rather than maintaining an army of on-premises servers that might sit idle much of the time, running Hadoop in the cloud lets you create—and pay for—VMs only when you need them. Even better, more and more of the big data that you want to analyze with Hadoop MapReduce is created in the cloud, saving you the trouble of moving it around. Figure 7 illustrates HDInsight MapReduce on Azure.

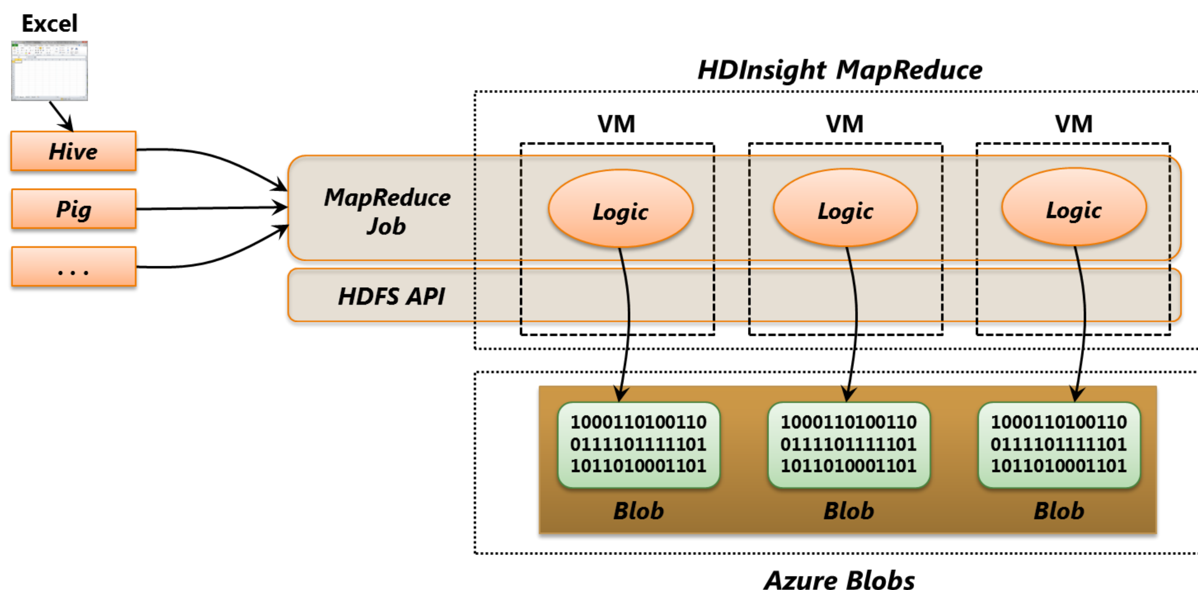

Figure 7: Azure HDInsight runs MapReduce jobs that process data in parallel using multiple virtual machines.

To use HDInsight, you first ask this managed service to create a Hadoop cluster, specifying the number of VMs you need. Setting up a Hadoop cluster yourself is a non-trivial task, and so letting Azure do it for you makes sense. Once the cluster is running, you can submit a MapReduce application, commonly called a *job*. As the figure shows, this job runs simultaneously across multiple VMs.

Rather than using HDFS, however, HDInsight relies on Azure Blobs. Blobs are similar to HDFS in some ways; both replicate data across multiple physical servers, for example. Rather than duplicate this functionality, HDInsight instead exposes Blobs through the HDFS API, as the figure shows. While the logic in a MapReduce job thinks it's accessing ordinary HDFS files, the job is in fact working with data stored in blobs. And to support the case where

multiple jobs are run over the same data, HDInsight also allow copying data from blobs into full HDFS running in the VMs.

MapReduce jobs are commonly written in Java today, an approach that HDInsight supports. Microsoft has also added support for creating MapReduce jobs in other languages, including C#, F#, and JavaScript. The goal is to make this big data technology more accessible to a larger group of developers.

Still, writing MapReduce jobs is non-trivial—it's a fairly specialized skill. To make this easier, people often use other technologies that let them analyze data without writing a MapReduce job themselves. For example, Pig is a high-level language designed for analyzing big data, while Hive offers a SQL-like language called HiveQL. Both Pig and Hive can actually generate MapReduce jobs, but they hide this complexity from their users.

HDInsight provides both Pig and Hive. Microsoft also offers an Excel add-in that lets business analysts create and run MapReduce jobs directly from Excel, then process and visualize the results using PowerPivot and other Excel tools. The intent is to make using this distributed data analysis platform available to a broad set of users.

Hadoop HBase

It's fair to say that MapReduce is the most visible member of the Hadoop family. When people say "Hadoop", they're often referring solely to its role as a platform for parallel analysis of unstructured data. Yet Hadoop offers more than this. For example, it also includes HBase.

HBase is a NoSQL database. Even though it runs on top of HDFS, it's intended to support applications that read and write data—it's not focused on data analysis. Instead, think of HBase as an alternative to DocumentDB or Tables, but with its own way of organizing data. Figure 8 illustrates how HBase does this.

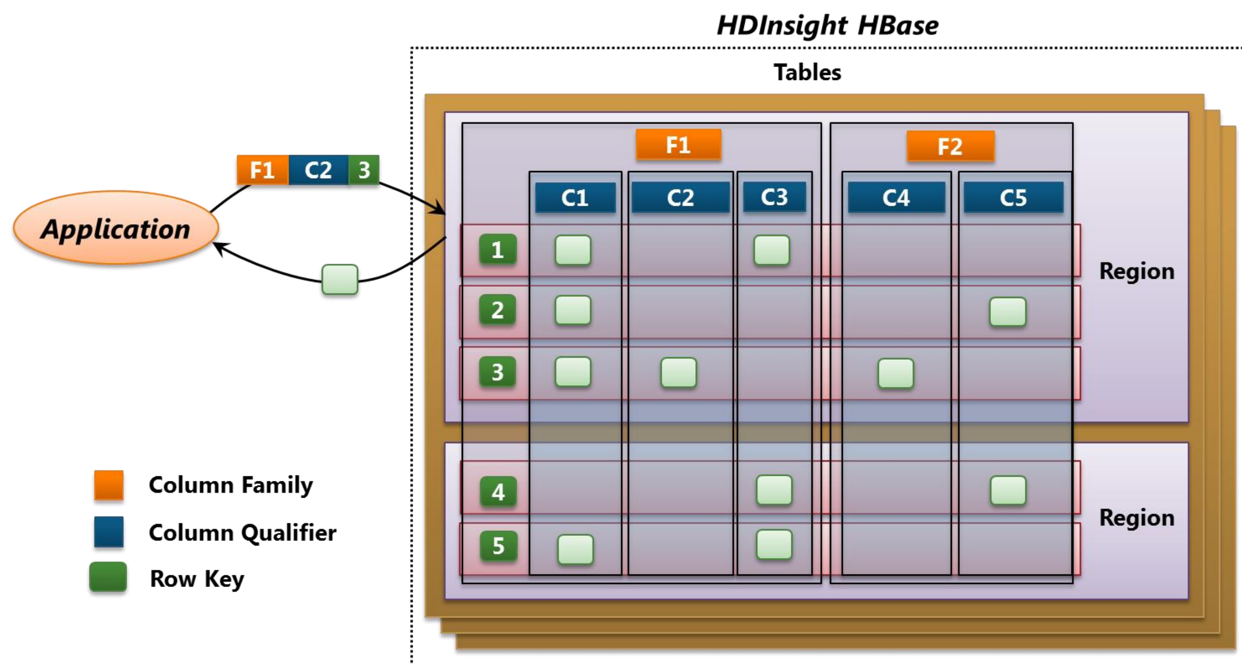

Figure 8: Azure HBase supports large, sparse tables with many rows and columns.

HBase is an example of the NoSQL style called a *column-family store*. As the figure shows, it stores data in tables with rows and columns. The columns are grouped into families, and so an application can access data by identifying a specific combination of column family, column qualifier, and row key. In this example, the application requests the data in column family F1, column C2, row 3. And although it's not shown, a cell in an HBase table can contain multiple time-stamped versions of data, letting an application specify which version it wants.

HBase divides a table into regions, as the figure shows, with different regions stored on different servers. As with DocumentDB and Azure Tables, spreading data across machines like this makes the system more scalable. In fact, a single HBase table can have millions of columns, billions of rows, and hundreds of terabytes of data. It's common, though, for an HBase table to be quite sparse—the system's physical storage is designed to handle this efficiently. Multiple replicas of each region are stored on different machines in an Azure datacenter, improving availability and performance. Like Azure Tables, HBase provides strong consistency, so an application won't see out-of-date or out-of-order updates to a table.

As usual with NoSQL technologies, HDInsight HBase lacks several things that are standard in relational databases. There's no query language, for example, and no way to create indexes other than on the row key. Transactions are possible, but they're limited: all of the data in a transaction must be contained within a single row of the same table. And HBase has no notion of data types—it stores everything as a string of bytes.

To work with HBase on Azure, you can use HDInsight to create a Hadoop cluster, then ask it to deploy HBase on this cluster. As with MapReduce, HDInsight HBase stores data in Azure blobs rather than HDFS. For applications that need to work with very large tables, HDInsight HBase can be just the right solution.

Search

On the Internet, everybody uses search. This style of interacting with data is both simple and powerful: we can do a lot without much effort. Yet behind the scenes, there's a lot going on. Implementing search well isn't simple.

Given how much people like search, it would be great if custom applications could also support this style of working with data. It's not reasonable to expect every developer to deploy his own search engine, however—it's too much work. What's needed is a general search service that can be used by many different applications. This is exactly what Azure Search provides.

Suppose you're creating an online shopping site, for example. Your customers will surely expect to search the site for the things they're interested in. And thanks to their experience with Internet search engines like Google and Bing, they'll expect your site's search to do things such as provide suggestions for search terms as they type. You can use Azure Search, a managed service, to provide all of this. Figure 9 illustrates the basics.

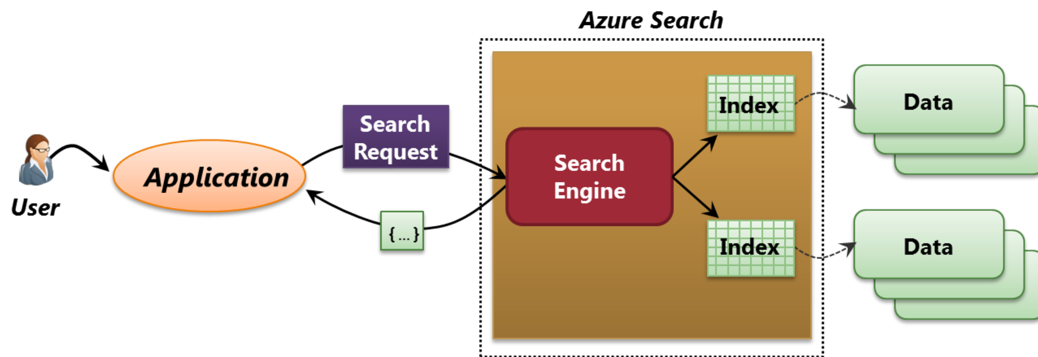

Figure 9: Azure Search allows creating applications that give users a traditional search experience.

Azure Search provides a search engine that lets your application submit a search request made by a user. The search engine searches an index to find all relevant data that matches this request, then returns a list of results to your application as JSON text. Notice that the user interface is defined entirely by your application; Azure Search has no search UI of its own. Instead, it exposes a RESTful interface for use by other software.

But why does your application need its own search function? Bing and Google both let your application expose searches that are scoped only to your site. Wouldn't it be easier just to use this? The answer is yes—it would be easier. But you also give up a lot by going this route. You can't see what searches your customers are doing, for example, nor can you tell when they search for things that you don't have. Even more important, using an external search service doesn't let you control the order of search results. Maybe you want to list the items that give you the highest profit margins first, for instance, or perhaps rank them by their current availability in your warehouse.

Azure Search is designed to support options like this. Each index contains some number of fields containing searchable data. For example, the index for an online shopping site would likely contain fields such as product name, product type, product description, brands, and other things that customers are likely to search on. It can also contain other fields that customers can't search on, however, such as profit margin. You can tell Search how to use these to influence the ordering of search results.

Even though ecommerce sites are perhaps the most obvious example, Azure Search is useful for more than this. Think about a site with user-generated content, such as a discussion site for movie buffs, where users want to find what they're interested in through search. This site might have advertisers who'd like you to order search results based on business requirements, which Azure Search allows. An internally facing enterprise application might also use Azure Search, especially if it contains lots of diverse data that's most easily accessed via search. Users like this style of access, and so supporting search in your next line-of-business application might help smooth its adoption by the people in your company.

While not every situation requires search, users will more and more expect to interact with applications in this way. Because doing this effectively requires specialized technology, Microsoft's cloud platform provides Azure Search.

Conclusion

Data is important. In fact, the field we work in, now called *information technology*, was originally known as *data processing*, an indication of how central data is to what we do. After decades in which relational systems ruled, the data world has expanded. With the rise of NoSQL, big data analytics, search, and more, we're able to work with data in new ways.

This is why Azure provides such a broad range of data management technologies: Blobs, support for running a DBMS in a VM, SQL Database, DocumentDB, Tables, HDInsight (including Hadoop and HBase), and Search. Whatever application you're trying to create, it's likely that you'll find something in this cloud platform that will work for you.

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.