Database Concepts

8th Edition

David M. Kroenke • David J. Auer

Scott L. Vandenberg • Robert C. Yoder

Online Appendix K

Big Data

VP Editorial Director: Andrew Gilfillan
Senior Portfolio Manager: Samantha Lewis
Content Development Team Lead: Laura Burgess
Program Monitor: Ann Pulido/SPi Global
Editorial Assistant: Madeline Houpt
Product Marketing Manager: Kaylee Carlson
Project Manager: Katrina Ostler/Cenveo® Publisher Services
Text Designer: Cenveo® Publisher Services
Interior design: Stock-Asso/Shutterstock; Faysal Shutterstock
Cover Designer: Brian Malloy/Cenveo® Publisher Services
Cover Art: Artwork by Donna R. Auer
Full-Service Project Management: Cenveo® Publisher Services
Composition: Cenveo® Publisher Services
Printer/Binder: Courier/Kendallville
Cover Printer: Lehigh-Phoenix Color/Hagerstown
Text Font: 10/12 Simoncini Garamond Std.

Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appear on the appropriate page within text.

Library of Congress Cataloging-in-Publication Data

Kroenke, David M., 1948- author. | Auer, David J., author.
Database concepts / David M. Kroenke, David J. Auer, Western Washington University, Scott L. Vandenberg, Siena College, Robert C. Yoder, Siena College.
Eighth edition. | Hoboken, New Jersey : Pearson, [2017] | Includes index.
LCCN 2016048321| ISBN 013460153X | ISBN 9780134601533
LCSH: Database management. | Relational databases.
LCC QA76.9.D3 K736 2017 | DDC 005.74--dc23
LC record available at https://lccn.loc.gov/2016048321

Appendix K — 10 9 8 7 6 5 4 3 2 1

Appendix Objectives

- Learn the basic concepts of Big Data
- Learn the basic concepts of nonrelational database management systems
- Learn about replicated, partitioned data stores for supporting clusters
- Understand the limitations and trade-offs of replicated, partitioned stores as indicated by the CAP theorem
- Learn the basic concepts of key-value databases
- Learn the basic concepts of document databases
- Learn the basic concepts of column family databases
- Learn the basic concepts of graph databases
- Understand the importance of XML
- Learn the elements of XML documents
- Understand how to describe and validate XML document structure using XML Schema
- Understand the role of XSLT in materializing XML documents
- Learn the basic concepts of JSON as a way of structuring nonrelational data
- Obtain a practical introduction to the Microsoft Azure cloud environment to set up an account, create SQL and NoSQL DocumentDB databases, and run simple queries.

What Is the Purpose of This Appendix?

In Chapter 8, we discussed Big Data, dimensional databases, and data warehouses in depth. We introduced **Not Only SQL (NoSQL)** nonrelational databases and DBMSs and learned that they can be categorized into four categories: **key-value**, **document**, **column family,** and **graph**. We then discussed column family database structure in detail, followed by a discussion of the MapReduce process.

This appendix takes a more thorough look at Big Data and nonrelational DBMSs. It recaps some of the material covered in Chapter 8 to provide a context for the new material, and it logically should be studied after the Chapter 8 sections on Big Data and the Not Only SQL Movement.

This appendix also covers cloud computing and two common nonrelational encodings of data in more depth: **XML (Extensible Markup Language)** and **JSON (JavaScript Object Notation)**, which were introduced briefly in Chapter 7. Both of these data storage frameworks are commonly associated with Big Data and NoSQL databases.

What is Big Data?

Big Data is a hot topic. *Big Data* is a "buzz word."

According to the Merriam-Webster dictionary, **Big Data** (aka **big data**) is "an accumulation of data that is too large and complex for processing by traditional database management tools."[1] And yet, as danah boyd (who prefers all lowercase letters in her name) and Kate Crawford[2] point out, United States census data sets that have been gathered over many decades are larger than some data sets now considered Big Data. They note the social phenomena aspects of Big Data and write that

> We define Big Data as a cultural, technological, and scholarly phenomenon that rests on the interplay of:
>
> (1) *Technology*: maximizing computation power and algorithmic accuracy to gather, analyze, link, and compare large data sets.
> (2) *Analysis*: drawing on large data sets to identify patterns in order to make economic, social, technical, and legal claims.
> (3) *Mythology*: the widespread belief that large data sets offer a higher form of intelligence and knowledge that can generate insights that were previously impossible, with the aura of truth, objectivity, and accuracy.[3]

Most users of Big Data would probably agree with the first two parts of this definition, while we should definitely be thinking about the third part and its implications.

The Three Vs

In early 2001, Doug Laney, working as a member of the META Group (which is now part of Gartner), needed a way to explain Big Data to his clients. He created and popularized the **3V framework** for discussing Big Data.[4]

---

[1] Merriam-Webster online dictionary at http://www.merriam-webster.com/dictionary/big%20data (accessed November 27, 2016).
[2] danah boyd and Kate Crawford, "Critical Questions for Big Data," in *Information, Communication and Society*, 15(5), 2012, 662–679, DOI: 10.1080/1369118X.2012.678878.
[3] Ibid., p. 663.
[4] Doug Laney, *3-D Data Management: Controlling Data Volume, Velocity and Variety* (Stamford, CT: META Group Inc., 2001. See Doug Laney's January 12, 2014, Gartner Blog Network article *Deja VVVu: Others Claiming Gartner's Construct for Big Data* (http://blogs.gartner.com/doug-laney/deja-vvvue-others-claiming-gartners-volume-velocity-variety-construct-for-big-data/ (accessed November 27, 2016)—there is a PDF version of the original paper available there.

The 3V framework discussed Big Data in terms of:

- **[Data] Volume**: the extremely large amount of data that needs to be stored.
- **Velocity**: the speed and continuous nature of data acquisition.
- **Variety**: the many different forms of data being acquired and stored.

Volume

In their 2012 book *Understanding Big Data*[5] (an e-book available from IBM and sponsored by IBM), Paul Zikopoulos and his colleagues note that in 2012 Twitter needed to store more than 7 terabytes of data per day and that Facebook was generating 10 terabytes daily. They projected that 35 zettabytes of data would need to be stored by 2020. IBM, in its illustration *The FOUR V's of Big Data,*[6] projects data growth of 40 zettabytes in 2020 and claims that 2.3 trillion gigabytes of data are currently created each day! Figure K-1 shows what these terms mean relative to each other, and remember that as this is being written in November 2016, 1-terabyte hard drives are a typical purchase for personal computers. IBM added *veracity*, having to do with the uncertainty of the quality and accuracy of data, to the list of V's.

Velocity

Saying that Twitter adds 7 terabytes (that is a one with 12 zeros!) per day not only is a measure of volume (the amount of data needing to be stored), it is also a measure of the speed at which the data arrives: 7 terabytes / *day* = over 290 gigabytes / *hour* = over 4.8 gigabytes / *minute* = over 81 megabtyes / *second*. The data arrives, and it must be stored and (hopefully) processed. And it doesn't stop—it keeps coming!

Variety

Twitter stores tweets (IBM's illustration claims 400 million tweets per day), which are short text messages of up to 140 characters (and stored in JSON format). YouTube stores and streams video content. Facebook stores text, pictures, "likes," and friend connection data. Instagram stores pictures, "likes," and comments (more text!). Pandora stores and streams music. The term *variety* reflects all the different types of content data that need to be stored.

[5] Paul Zikopoulos, Chris Eaton, Dirk deRoos, Thomas Deutsch, and George Lapis, *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data* (New York: McGraw Hill Bulk Sales, 2012). Available online at http://www.ibmbigdatahub.com/whitepaper/understanding-big-data-e-book (accessed November 27, 2016).
[6] Available at http://www.ibmbigdatahub.com/infographic/four-vs-big-data (accessed November 27, 2016)

Name	Symbol	Approximate Value for Reference	Actual Value
Byte			8 bits [Store one character]
Kilobyte	KB	About 10^3	2^{10} = 1,024 bytes
Megabyte	MB	About 10^6	2^{20} = 1,024 KB
Gigabyte	GB	About 10^9	2^{30} = 1,024 MB
Terabyte	TB	About 10^{12}	2^{40} = 1,024 GB
Petabyte	PB	About 10^{15}	2^{50} = 1,024 TB
Exabyte	EB	About 10^{18}	2^{60} = 1,024 PB
Zettabyte	ZB	About 10^{21}	2^{70} = 1,024 EB
Yottabyte	YB	About 10^{24}	2^{80} = 1,024 ZB

Figure K-1 — Storage Capacity Terms

Big Data and NoSQL Systems

We have used the relational database model and SQL throughout this book up until chapter 8. However, there is another school of thought that has led to what was originally known as the **NoSQL** movement but now is usually referred to as the **Not only SQL movement**.[7] It has been noted that most, but not all, DBMSs associated with the NoSQL movement are nonrelational DBMSs. A NoSQL DBMS is often a distributed, replicated (i.e., cloud) database, as described in Chapter 8, and is used where this type of a DBMS is needed to support large data sets. There have been several classification systems proposed for grouping and classifying NoSQL databases.

For our purposes, we will adopt and use a set of four categories of NoSQL databases:[8]

- **Key-Value**—Examples are DynamoDB and MemcacheDB
- **Document**—Examples are Couchbase Server, MongoDB, and Microsoft Azure DocumentDB
- **Column Family**—Examples are Apache Cassandra and HBase
- **Graph**—Examples are Neo4j and AllegroGraph

[7] For a good overview, see the Wikipedia article **NoSQL**.
[8] This set of categories corresponds to the four categories used in the Wikipedia article **NoSQL** as Wikipedia's taxonomy of NoSQL databases and is also used in Ian Robinson, Jim Webber, and Emil Eifrem, *Graph Databases* (Sebastopol, CA: O'Reilly Media, 2013).

NoSQL databases are used by widely recognized Web applications—for example, both Facebook and Twitter use the Apache Software Foundation's **Cassandra** database. In Chapter 8, we discussed column family DBMSs. In this appendix, we will discuss all four categories. We will include our discussion of column family DBMSs here for completeness and so that you will not have to refer to Chapter 8 while reading this material.

A major motivating factor for the development of NoSQL database systems was the need to rapidly process large data sets. One must either "scale up" or "scale out" to gain storage and processing capacity. Maintaining a DBMS on a single computer system makes it easier to maintain database consistency, but there is a limit to the number of processors and storage that can be attached to a single computer system. Scaling up might initially work but will usually become untenable as data storage and processing requirements grow rapidly. Many NoSQL database systems have built-in support for running on clusters of computers—scaling out—that allows capacity to be added fairly easily and cheaply. As mentioned in chapter 8, this capacity growth is accomplished by partitioning (**sharding**) and **replication**.

Sharding is a set of approaches for taking portions of a database and moving them to a different server in a cluster. In the relational world, we can move entire tables to other servers—for example, move customer orders as a separate shard—or we can extract sets of rows and move them to corresponding tables on another server—for example move customers starting with letters A–H, I–P, and Q–Z to separate CUSTOMER tables. Of course, there has to be DBMS or application support for updating the rows on the correct shard.

Replication is accomplished by making one or more copies of a set of data and maintaining it on a different server. Each copy is called a **replica set**. Replication increases the processing capacity of our system by allowing several servers to access the data simultaneously. It also can increase availability of our system since processing can continue in the event that a server fails. However, this can lead to consistency problems. If a data item is updated on one server, it will take some time before the update can propagate to the other replica sets on other servers. This is known as *eventual consistency*. Some applications may end up reading stale data. This may or may not be a problem, depending on the application. What if two different updates are made to the same data item at the same time on different servers? Again, the severity of the problem depends on the application, but there are ways to address this problem using *timestamps* and *quorums*. Quorums are briefly described below.

To increase the consistency of our database, we can request that database reads or writes be propagated to several of the replica sets before indicating to the application that the request has completed successfully. A typical scenario is to ensure that the read or write operation completes on a majority (quorum) of the servers. Increasing the number of replica sets that must respond increases consistency but makes processing slower. The quorum values for read operations can be different than for write operations. This availability versus consistency trade-off is one of many that must be taken in consideration when using clusters, which brings us to the CAP theorem.

The CAP Theorem

The CAP theorem originated in 2000, when Eric Brewer made a proposal in his keynote presentation at the Symposium on Principles of Distributed Computing (PODC). The proposal was formally proved as a theorem by Seth Gilbert and Nancy Lynch.[9] It is now known as the **CAP theorem**. There is currently an ongoing discussion of the CAP theorem, and Brewer himself has been involved in this discussion.[10]

Basically, the CAP theorem defines three properties of distributed database systems:

1. Consistency
2. Availability
3. Partition tolerance

Consistency means that all the database replicas see the same data at any given point in time. **Availability** means that every request received by a server will result in a response, as long as the network is available. **Partition tolerance** means that the distributed database can continue to operate even when the cluster is partitioned by network failures into two or more disconnected sections (partitions).

The CAP theorem, illustrated in Figure K-2, basically states that *all three of these properties cannot be guaranteed at the same time*—at least one of the properties will be at least somewhat compromised, as there may be several levels of these properties.

For example, allowing partition tolerance will degrade consistency if each partition independently allows updates, resulting in different values for the same data item in other replica sets, or we could choose to lose availability by only allowing reads or by shutting down the entire system in the event of a partition.

The CAP theorem gives us the basis for a comparison of relational DBMSs to nonrelational "NoSQL" DBMSs. Relational DBMSs can be said to favor consistency and availability (since they are often running on a single, non-partitioned database system), while nonrelational DBMS systems must be designed according to whether they want to emphasize consistency or availability in their design (since they are by definition designed for clustered, distributed, and therefore partitioned systems).

Aggregates

The first three types of NoSQL systems (Key-value, Document, and Column Family) can store "aggregates." By this we mean more complex data structures than a simple row in a relational table, as aggregates can represent composite data (e.g., an address consists of street, city, and country data items) as well as lists (e.g., a set of phone numbers for a customer) that would normally be represented in separate relational tables to store 1:N or 1:1 relationships. Thus, we have a collection of data items

[9] See the Wikipedia article **CAP theorem** and Seth Gilbert and Nancy Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services," *ACM SIGACT News 333*(2), 2002).
[10] See Eric Brewer, "CAP Twelve Years Later: How the "Rules" have Changed," on the InfoQ Web site at http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed (accessed November 27, 2016).

that are treated as a unit. Aggregate data can be described using the data modeling languages XML and JSON, which will be covered in the next section. Since relational databases are designed for consistency, there is support for "atomic" transactions that require all updates to complete successfully before the data is committed to the database (see Chapter 6). For aggregate NoSQL databases, we have to think of "transactions" in a different way. Although operations on a single aggregate are atomic, there is usually not built-in support for operations across more than one aggregate data object. Thus, the design of the aggregates and the application itself become important considerations when consistency is a goal.

Extensible Markup Language (XML)

Many NoSQL database systems store, process, and communicate data in the form of documents. XML and JSON are two common document formats used for this. This section describes XML—JSON is described later in this appendix.

Database processing and document processing need each other. This is particularly true in nonrelational databases. Database processing needs document processing for transmitting database views; document processing needs database processing for storing and manipulating data. However, even though these technologies need each other, it took the popularity of the Internet to make that need obvious.

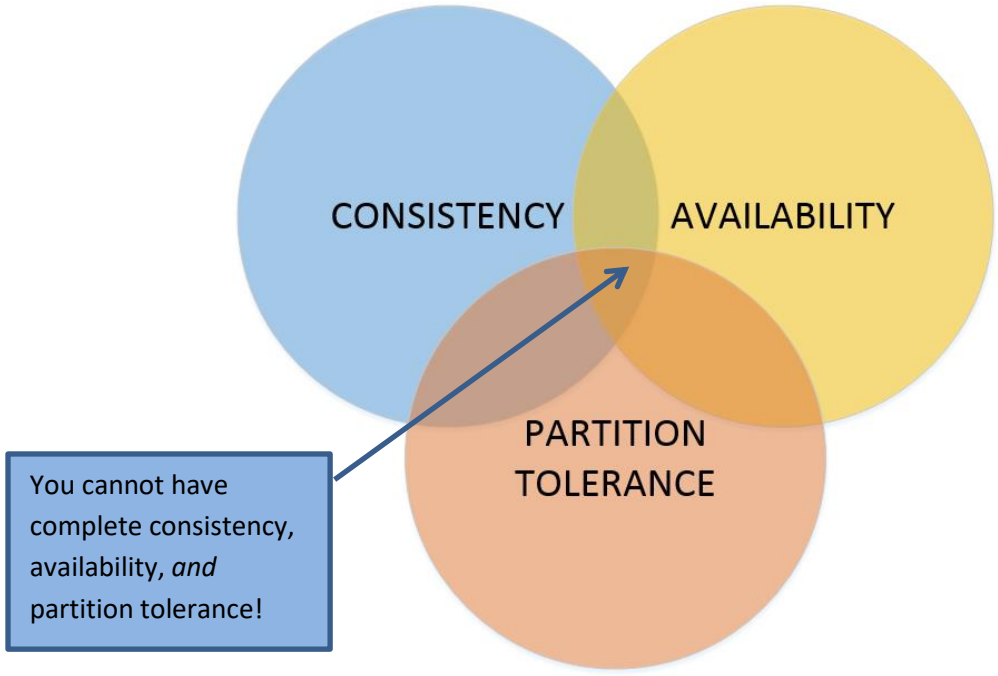


Figure K-2 — The CAP Theorem—You Can't Have All Three at the Same Time!

As Web sites evolved, organizations wanted to use Internet technology to display and update data from their organizational databases. Web developers began to take a serious interest in SQL, database performance, database security, and other aspects of database processing.

As Web developers invaded the database community, database practitioners wondered, "Who are these people, and what do they want?" Database practitioners began to learn about HTML, the language used to markup documents for display by Web browsers. At first, the database community scoffed at HTML because of its limitations, but it soon learned that HTML was the output of a more robust document markup language called **Standard Generalized Markup Language (SGML)**. SGML was clearly important, just as important to document processing as the relational model was to database processing. Obviously, this powerful language had some role to play in the display of database data, but what role?

In the early 1990s, the two communities began to meet, and the result of their work is a series of standards that describes a language called **Extensible Markup Language (XML)**. XML is a subset of SGML, but additional standards and capabilities have been added to XML, and today XML technology is a hybrid of document processing and database processing. In fact, as XML standards evolved, it became clear that the communities had been working on different aspects of the same problem for many years. They even used the same terms, but with different meanings. You will see later in this appendix how the term *schema* is used in XML for a concept that is different from the use of *schema* in the database world.

XML provides a standardized yet customizable way to describe the content of documents. As such, it can be used to describe any database view, but in a standardized way. In addition, when used with the XML Schema standard, XML documents can automatically be generated from database data. Further, database data can automatically be extracted from XML documents. In addition, there are standardized ways of defining how document components are mapped to database schema components, and vice versa.

Meanwhile, the rest of the computing community began to take notice of XML. **SOAP**, which originally stood for **Simple Object Access Protocol**, was defined as a standard for providing remote procedure calls over the Internet. Initially, SOAP assumed the use of HTTP as a transport mechanism. When Microsoft, IBM, Oracle, and other large companies joined forces in support of the SOAP standard, this assumption was removed, and SOAP was generalized to become a standard protocol for sending messages encoded in XML. With this change, SOAP no longer meant Simple Object Access Protocol, so now SOAP is just a name and not an acronym.

Today, XML is used for many purposes. One of the most important is its use as a standardized means to define, communicate, and validate documents for processing over the Internet. XML plays a key role in Microsoft's .NET initiative, and in 2001, Bill Gates called XML the "*lingua franca* of the Internet age."

We will begin the discussion of XML by describing its use for displaying Web pages. As you will learn, however, XML uses go far beyond Web page display. In fact, Web page display is one of the least important applications of XML. We begin with page display only because it is an easy way to introduce XML documents. After that, we will explain the XML Schema standard and discuss its use for database processing.

As you read this appendix, keep in mind that this area is at the leading edge of database processing. Standards, products, and product capabilities are frequently changing. You can keep abreast of these changes by checking the following Web sites: *www.w3c.org*, *www.xml.org*, *http://msdn.microsoft.com, www.oracle.com, www.ibm.com*, and *www.mysql.com*. Learning as much as you can about XML and database processing is one of the best ways you can prepare yourself for a successful career using Web and database technologies.

XML as a Markup Language

As a markup language, XML is significantly better than HTML in several ways. For one, XML provides a clean separation between document structure, content, and materialization—the way the document is displayed on a particular device. XML has facilities for dealing with each, and they cannot be confounded (mixed together), as they are with HTML.

Additionally, XML is standardized, but as its name implies, the standards allow for extension by developers. With XML, you are not limited to a fixed set of elements such as <p>, <H1>, and <H2>; you can create your own.

Third, XML eliminates the inconsistent tag use that is possible (and popular) with HTML. For example, consider the following HTML:

```
<H2>Hello World</H2>
```

Although the <H2> tag can be used to mark a level-two heading in an outline, it can be used for other purposes, too, such as causing "Hello World" to be displayed in a particular font size, weight, and color. Because a tag has potentially many uses, we cannot rely on tags to discern the structure of an HTML page. Tag use is too arbitrary; it may mean a heading, or it may mean nothing at all. HTML is also lax about the use of closing tags, leading to inconsistent display of Web pages on different browsers. Some browsers support vendor-specific extensions, making it more difficult to develop applications that operate well across all browsers. XML, XHTML and related languages go a long way to correct that problem.

As you will see, the structure of an XML document can be formally defined. Tags are defined in relationship to one another. In XML, if we find the tag <street>, we know exactly what data we have, where those data belong in the document, and how that tag relates to other tags.

Materializing XML Documents with XSLT

The XML document shown in Figure K-3 shows both the document's structure and content. The second line in the document refers to a stylesheet that indicates how it is to be materialized for display. The designers of XML created a clean separation among structure, content, and format. The most popular way to materialize XML documents is to use **XSLT**, or **Extensible Style Language: Transformations**. XSLT is a powerful and robust transformation language. It can be used to materialize XML documents into HTML, and it can be used for many other purposes as well.

One common application of XSLT is to transform an XML document in one format into a second XML document in another format. A company can, for example, use XSLT to transform an XML order document in its own format into an equivalent XML order document in its customer's format. We will be unable to discuss many of the features and functions of XSLT here. See *www.w3.org* for more information.

XSLT is a declarative transformation language. It is declarative because you create a set of rules that govern how the document is to be materialized instead of specifying a procedure for materializing document elements. It is transformational because it transforms the input document into another (output) document.

The stylesheet to display the data in Figure K-3 is shown in Figure K-4. A **stylesheet** is used by XSLT to indicate how to transform the elements of the XML document into another format, in this case an HTML document that will be acceptable to a browser.

The XSLT processor copies the elements of the stylesheet until it finds a command in the format {*item, action*}. When it finds such a command, it searches for an instance of the indicated item; when it finds one, it takes the indicated action. For example, when the XSLT processor encounters

```
<xsl:for-each select = "SeminarList/Seminar">
```

it starts a search in the document for an element named SeminarList. When it finds such an element, it looks further within the SeminarList element for an element named Seminar. If a match is found, it takes the actions indicated in the loop that ends with the last occurrence of </xsl:for-each> in the file. Within the loop, styles are set for each element in the Seminar document.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="DBC-e08-SeminarListStyleSheet.xsl"?>
<SeminarList>
    <Seminar>
        <SeminarID>1</SeminarID>
        <SeminarDate>10/10/2016</SeminarDate>
        <SeminarTime>11:00</SeminarTime>
        <Location>San Antonio Convention Center</Location>
        <SeminarTitle>Kitchen on a Budget</SeminarTitle>
    </Seminar>
    <Seminar>
        <SeminarID>2</SeminarID>
        <SeminarDate>10/26/2016</SeminarDate>
        <SeminarTime>16:00</SeminarTime>
        <Location>Dallas Convention Center</Location>
        <SeminarTitle>Kitchen on a Big D Budget</SeminarTitle>
    </Seminar>
</SeminarList>
```

Figure K-3 — SEMINAR XML Document with Reference to an XSLT Stylesheet

Here we are creating an XML document that can be viewed using a Web browser to display the list of Seminars at Heather Sweeney Designs, as shown in Figure K-5, which is the result of applying the stylesheet in Figure K-4 to the document in Figure K-3. XSLT processors are context oriented; each statement is evaluated in the context of matches that have already been made. Thus, the following statement:

```
<xsl:value-of select = "SeminarTitle"/>
```

operates in the context of the SeminarList/Seminar match that was made above.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
 version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns="http://www.w3.org/1999/xhtml">
<xsl:output method="xml" encoding="UTF-8"/>
<xsl:template match="/">
      <html>
      <head> <title>Seminar Data</title> </head>
      <style type="text/css">
      h1 {text-align:center; color:blue}
      body {font-family:ariel, sans-serif; font-size:12pt; background-color:#FFFFFF}
      div.seminardata {font-weight:bold; background-color:#3399FF; color:#FFFFFF;
padding:4px}
      </style>
            <body>
            <p>
            <h1>HSD Seminar Data</h1>
            </p><hr />
            <div class="seminardata">
            <xsl:for-each select="SeminarList/Seminar">
            <xsl:value-of select="SeminarTitle"/>
            <br/>
            <xsl:value-of select="Location"/> <br/>
            <xsl:value-of select="SeminarDate"/> <br/>
            <xsl:value-of select="SeminarTime"/>
            <br/><br/>
            </xsl:for-each>
            </div>
            </body>
      </html>
</xsl:template>
</xsl:stylesheet>
```

Figure K-4 — The Heather Sweeney Designs Seminar Stylesheet

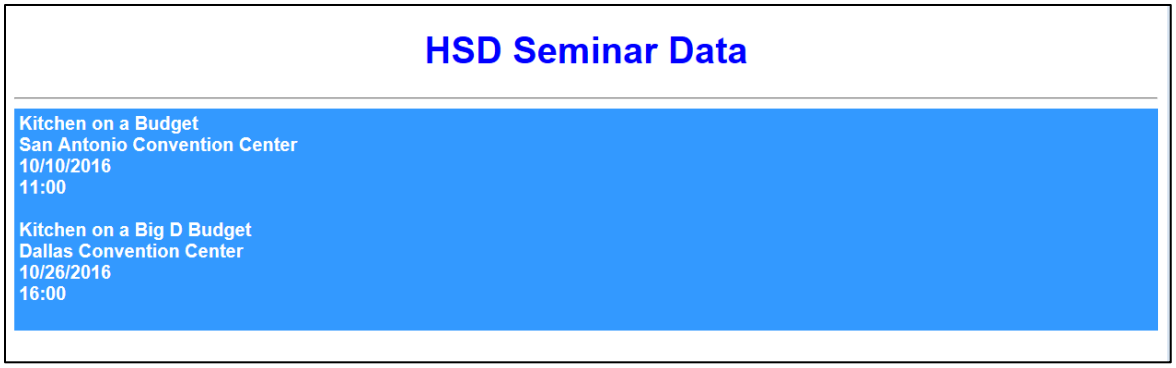


Figure K-5 — HTML Result from Application of Seminar Stylesheet

Browsers have built-in XSLT processors. You need only supply the document to the browser; it will locate the stylesheet and apply it to the document for you. In this example, the document in K-3 has a link to the stylesheet file in the same directory.

XML Document Type Declarations and XML Schema

Document Type Declarations (DTDs) were the XML community's first attempt at developing a document structure specification language. It is recognizable by the keyword DOCTYPE in the document. DTDs work, but they have some limitations, and, embarrassingly, DTD documents are not XML documents. To correct these problems, the W3C Committee defined another specification language called **XML Schema**. Today, XML Schema is the preferred method for defining and validating document structure, so we will skip DTDs and cover XML schemas instead.

XML Schemas are XML documents. This means that you use the same language to define an XML Schema as you would use to define any other XML document. It also means that you can validate an XML Schema document against its schema (a "meta-schema") just as you would any other XML document.

If you are following this discussion, then you realize that there is a chicken-and-the-egg problem here. If XML Schema documents are themselves XML documents, what document is used to validate them? What is the schema of all of the schemas? There is such a document; the mother of all schemas is located at *www.w3.org*. All XML Schema documents are validated against this document.

XML Schema is a broad and complex topic. Dozens of sizable books have been written just on XML Schema alone. Clearly, we will not be able to discuss even the major topics of XML Schema in this appendix. Instead, we will focus on a few basic terms and concepts and show how those terms and concepts are used with database processing. Given this introduction, you will then be able to learn more on your own.

XML Schema Validation

Figure K-6(a) shows an XML Schema document that can be used to represent the rules for a valid XML SeminarList file at Heather Sweeney Designs. Because the schema is itself an XML Schema document, it is to be validated against the mother of all schemas, the one at *www.w3.org*, as indicated on the second line of the schema document that starts with xsd:schema. This same reference will be used in all XML Schemas in every organization worldwide. (By the way, this reference address is used only for identification purposes. Because this schema is so widely used, most schema validation programs have their own built-in copy of it.)

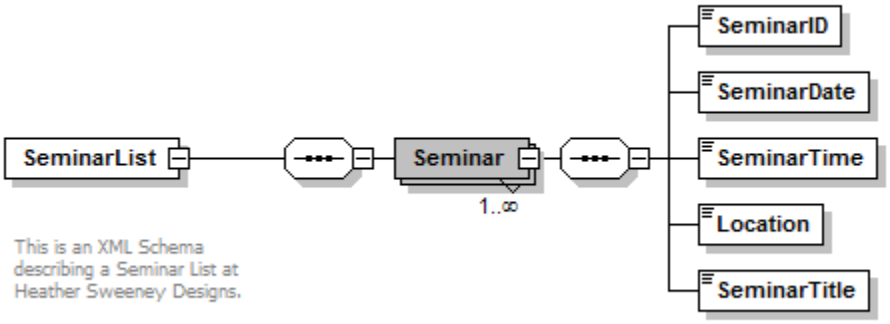
The *xsd:schema* statement not only specifies the document that is to be used for validation, it also establishes a labeled namespace. Namespaces are a complicated topic in their own right, and we will only give a brief overview here. In this first statement, the label *xsd* is defined by the expression *xmlns:xsd*. The first part of that expression stands for *xml namespace*, and the second part defines the label *xsd*. Notice that all of the other lines in the document use the label *xsd*. The expression *xsd:complexType* simply tells the validating program to look into the namespace called *xsd* (here, the one specified as *www.w3.org/2001/XMLSchema*) to find the definition of the term *complexType*.

The name of the label is up to the designer of the document. You could change *xmlns:xsd* to *xmlns:xse* or to *xmlns:mylabel*, and you would set *xse* or *mylabel* to point to the w3 document.

XML namespaces are used to combine different vocabularies into the same XML Schema. They can be used to define and support domains and to disambiguate terms. The need for the latter occurs when a document contains synonyms. For example, consider a document that has two different uses for the term Instrument. Suppose one usage of this term refers to musical instruments and has the sub-elements {Manufacturer, Model, Material}, as in {Horner, Bflat Clarinet, Wood}, and a second use of this term refers to electronic instruments and has the sub-elements {Manufacturer, Model, Voltage}, as in {RadioShack, Ohm-meter, 12-volt}. The author of the XML Schema for such a document can define two different namespaces where each contains one of these definitions. Then the complexType definition for each of these definitions of Instrument can be prefixed by the label of the namespace, as was done in our schema documents when we used the label xsd. There is more to XML namespaces, and you will undoubtedly learn more as you work with XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="SeminarList">
        <xsd:annotation>
            <xsd:documentation>
            This is an XML Schema describing a Seminar List at Heather Sweeney Designs.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="Seminar" maxOccurs="unbounded">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="SeminarID" type="xsd:integer"/>
                    <xsd:element name="SeminarDate" type="xsd:string"/>
                    <xsd:element name="SeminarTime" type="xsd:string"/>
                    <xsd:element name="Location" type="xsd:string"/>
                    <xsd:element name="SeminarTitle" type="xsd:string"/>
                </xsd:sequence>
            </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

(a) XML Schema



(b) Graphical Representation of XML Schema

Figure K-6 — Example XML Schema Document

Elements (and a Little About Attributes)

As shown in Figure K-6(a), XML Schemas consist of elements. **Elements** are either simple or complex. A **simple element** has a single data item. In Figure K-6(a), the elements SeminarID, SeminarDate, SeminarTime, Location, and SeminarTitle are all simple elements. A **complex element** can contain other elements that can be either simple or complex. In Figure K-6(a), Seminar is a complex element and is followed by (after the annotations section) a section with <xsd:complexType>…</xsd:complexType>

tags. It contains a sequence of the five simple elements SeminarID, SeminarDate, SeminarTime, Location, and SeminarTitle. It is possible to have complex types that contain other complex types.

Data types can be specified in simple elements. In Figure K-6(a), the element SeminarID is typed as xsd:integer and the element SeminarTitle is typed as xsd:string. By default, the cardinality of both simple and complex elements is 1.1, meaning that a single value is required, and no more than a single value can be specified. This is similar to a NOT NULL constraint in SQL schema definitions. If an element did not have a required value, we would add minOccurs="0" to the element definition. This is similar to the NULL statement in SQL schema definitions. There can be multiple occurrences of the complex type *Seminar* due to the MaxOccurs="unbounded" option on the Seminar element.

Schemas also contain **attributes**, which are usually used to provide additional information about elements. For example, we could choose to store the SeminarID as an attribute of Seminar by coding:

```
<Seminar  SeminarID="1">
```

A good rule of thumb is that elements represent data and attributes represent metadata, though this is not part of any XML standard.

The schema definition in Figure K-6(a) was used to validate the XML document in Figure K-3. There are several programs available that can validate schema. A free program is Notepad++ (see *https://notepad-plus-plus.org*) with the XML plugin enabled. Another excellent product is Altova's XMLSpy XML editing tool (see *http://www.altova.com*). Figure K-6(b) shows the XML Schema in a diagram format as drawn by XMLSpy. Being able to see an XML Schema as a diagram often makes it easier to interpret exactly what the XML Schema is specifying.

Why Is XML Important?

At this point, you should have some idea of the nature of XML and the XML standards. You know that XML makes a clear separation between structure, content, and materialization. Structure is defined by either a DTD or an XML Schema document. Content is expressed in an XML document, and the materializations of a document are expressed in an XSL document. As mentioned in Chapter 7, SQL statements can be used to create XML documents.

You may be asking, "These are interesting ideas, but why do they matter? What's so important about all of this?" The answer to these questions is that XML processing provides a standardized facility to describe, validate, and materialize any database view.

Suppose, for example, that the real estate industry agrees on an XML Schema document for property listings. Every real estate company that can produce data in the format of the schema can then exchange listings with every other real estate company. Given the schema, each company can ensure that it is transmitting valid documents, and it also can ensure that it is receiving valid documents. Further, each company can develop its own set of XSL documents to materialize property listings in whatever way it wants. Once the XSL documents have been prepared, any listing from any participating agent can be displayed in the local agency's materializations. Figure K-7 lists some XML standards work that is under way in various industries.

For another example, consider business-to-business e-commerce. Suppose that Wal-Mart wants to send orders to its vendors in a particular standardized format and that it wants to receive shipment responses to those orders in another particular standardized format. To do this, Wal-Mart can develop an XML Schema for Order documents and another for Shipment documents. It can then publish those XML Schemas on a Web site accessible to its vendors. In this way, all vendors can determine how they will receive orders from Wal-Mart and how they should send their Shipment notifications back.

The schemas can be used by Wal-Mart and all of its vendors to ensure that they are sending and receiving only valid XML documents. Further, Wal-Mart can develop XSL documents to cause the Order and Shipment documents to be transformed into the specific formats needed by its accounting, operations, marketing, and general management departments.

These XSL documents work for any Order or Shipment from any of its vendors. In all of these cases, once the XML Schema documents have been prepared and the XSL documents have been written, all validation and materialization are done via automated processes. Thus, there is no need for any human to touch the Order document between its origination at Wal-Mart and the picking of the inventory at the supplier.

So, the only challenge that remains is to populate the XML documents with database data in accordance with the relevant XML Schema. SQL can be used to populate simpler schemas, and newer technologies like ADO.NET can handle more sophisticated transformations of database data into XML documents. Both SQL and ADO.NET can also be used for the other direction, the transformation of XML documents into database data.

Industry Type	Example Standards
Accounting and Business	• American Institute of Certified Public Accountants (AICPA): eXtensible Business Reporting Language (XBRL) • Open Access Group: Open Access Group Integration Specification (OAGIS)
Architecture and Construction	• Construction Open Standards Alliance (COSA): Associated General Contractors XML (agcXML) and other related standards • Green Building XML Schema, Inc.: Green Building XML (gbXML)
Automotive	• Automotive Industry Action Group (AIAG): XML standards • Standards for Technology in Automotive Retail (STAR): STAR XML Schema Repository
Banking and Finance	• International Swaps and Derivatives Association (ISDA): Financial products Markup Language (FpML) • International Organization for Standardization (ISO): ISO 20022 (Universal financial industry message scheme, based on XML)
Electronic Data Interchange	• X12 (American National Standards Institute): XML schemas for electronic data interchange for business processes • Organization for the Advancement of Structured Information Standards (OASIS): electronic business XML (ebXML) core (ebCore)
Health and Medicine	• National Library of Medicine: Database DTDs • Health Level Seven International (HL7): virtual Medical Record for Clinical Decision Support (vMR-CDS)
Human Resources	• HR Open Standards Consortium: Human Resources XML (HR-XML)
Insurance	• Association for Cooperative Operations Research and Development (ACORD): XML standards for life insurance
Music and Entertainment	• MakeMusic, Inc.: musicXML • Entertainment Merchants Association (EMA): EMA video game metadata XML specifications
Real Estate	• Real Estate Standards Organization (RESO): RESO Data Dictionary XML files • Open Standards Consortium for Real Estate (OSCRE): XML standards for real estate
Workflow	• Workflow Management Coalition (WfMC): XML Process Definition Language (XPDL)

Figure K-7 — Example XML Industry Standards

Additional XML Standards

As you know, XML was developed as a series of standards. So far, we have mentioned XML, XSL, XSLT, and XML Schema. You will probably encounter a number of other XML standards, and you can find the standards, their documentation, and some tutorials on the *www.w3.org* and *www.xml.org* Web sites.

The XML Standards Committee continues its important work, and more standards will be developed as the needs arise. At present, work is under way for developing security standards. Keep checking *www.w3.org* for more information. XML will continue to be an important data interchange format for both relational and nonrelational (specifically NoSQL) databases.

Nonrelational Database Management Systems

XML is closely related to many nonrelational NoSQL database management systems. These systems can be divided into four major categories, which are described in the following subsections. Most NoSQL DBMSs, no matter which of the four categories they belong to, have one thing in common: There is often no schema for the database, as there is in a relational database. There are also fewer restrictions on the way data can be structured. Such systems are often based on XML (described above) or JSON (described below), but they may use other (or no) data structuring formats as well. These systems are also often called semi-structured databases or unstructured databases (depending on the level of structuring available) as opposed to the highly structured relational data model. For example, a STUDENT relation in a relational DBMS is highly structured: All rows have the same number of columns with the same column names and column data types, and every column value is a single value (first normal form).

JSON, which is used in several document databases, has some similarities to XML, but there is no notion of any schema external to the data. A document ("object" in JSON terminology) consists of a set of (field, value) pairs. Values can be simple values (strings or numbers) or other objects. Values can also be arrays of values or objects. Here is a JSON representation for student data:

```
{
      id: "student1",
      lastName: "Smith",
      firstName: "John",
      majors: ["History", "Computer Science"]
}
{
      id: "student2",
      firstName: "Mary",
      minor: "English",
      majors: ["Math"],
      GPA:  3.2
}
```

As you can see, the number, order, types, and (in some systems) names of columns are less restricted in a semi-structured database; in addition, fields can be multi-valued, such as the *majors* field above. These features, some of which were discussed above in the XML section, are common among NoSQL DBMSs. The data in the example above can be stored, in different ways, in each of the four kinds of nonrelational DBMSs described below.

Key-Value Databases

Key-value databases use a simple key and value pairing. Each key (similar to a relational DBMS primary key) appears only once in each database. An example is **DynamoDB**, which was developed by Amazon.com. Another key-value database is **MemcacheDB**. **Apache Cassandra** is typically classified as column family database, though it is built upon Amazon's DynamoDB and Google's BigTable, so it has some features of key-value databases as well.[11]

A key-value database is extremely simple and allows for easy distribution of the key-value pairs in the networked cluster. Note that the value can be anything: a string, a large binary object, a list of things, a JSON object, etc. Key-value databases are ideal for large amounts of data that need fast storage and retrieval of simple objects but do not need the full complexity of SQL queries, since the DBMS is unaware of the structures within the value object. Thus, queries cannot be made using internal fields. The basic operations available to query a key-value database are:

- **get (key)**: retrieves the value associated with a key
- **set (key, value)**: creates or updates a key-value pair
- **delete (key)**: removes a key-value pair

These commands are typically issued from within a program in some programming language or from a command-line interface in some systems. Here are some sample data that might appear in a typical key value database that stores information about email accounts and the last computer IP address from which those accounts were accessed:

```
("joe@somewhere.com", 172.13.233.1)
("mary@nowhere.com", 177.10.254.1)
```

The email addresses are the keys and the IP addresses are the values. Remember that the values can be much more complex or larger, but in a key value database, any structure within the values must be managed by the application after the key value database retrieves the value.

Document Databases

Document databases store data according to a document-oriented format, the two most popular of which are **XML (Extensible Markup Language)** and **JSON (Java Script Object Notation)**. An example is **Couchbase Server**, which uses JSON storage. Another example is Microsoft's **Azure DocumentDB**. Later in this appendix we will create and query a document database in JSON format using Microsoft Azure

[11] See the Wikipedia article **Apache_Cassandra** and the Apache Cassandra Web site.

and DocumentDB. Amazon's **DynamoDB** supports both key-value and document data storage. **MongoDB** is another popular document database; it uses **BSON (Binary JSON)** storage, which is a binary-encoded version of JSON documents, with a few additional data types available. In a document database, documents (e.g., JSON objects) are usually stored within a set or a "**collection**."

As with key-value databases, most interaction with the database is done either from within a program written in a programming language like Java or Python or via a command-line environment. Typical commands available in a document database (again using a generic syntax) include:

- **insert (doc, collection)**: put document "doc" into the collection
- **update (collection, doc_specifier, update_action)**: within the collection, update all documents matching the "doc_specifier" (e.g., all students with last name 'Smith'). The "update_action" can alter the values of multiple fields within each document.
- **delete (collection, doc_specifier)**: removes all documents from collection that match the "doc_specifier"
- **find (collection, doc_specifier)**: retrieves all documents in a collection matching the "doc_specifier"

In addition, most document databases provide some level of built-in support for map-reduce and other aggregation tasks. Here is some sample data that might appear in a document database regarding student and advisor information. The first document would be stored in a collection of similar "Advisor" documents, and the second would be stored in a collection of similar "Student" documents. The curly braces indicate a document (a set of (field, value) pairs), and the square brackets indicate an array. The JSON syntax is from MongoDB, but other systems have very similar syntax:

```
{
   _id:  "Advisor1",
   name:  "Shire, Robert",
   dept:  "History",
   yearHired:  1975
}

{
   _id:  555667777,
   name:  "Tierney, Doris",
   majors:  ["Music", "Spanish"],
   addresses:  [
      {
      street:  "14510 NE 4th Street",
      city:  "Bellevue",
      state:  "WA",
      zip:  "98005"
      },
      {
      street:  "335 Aloha Street",
      city:  "Seattle",
      state:  "WA",
      zip:  "98109"
      }
      ],
   advisorID:  "Advisor1"
}
```

The "_id" field is the globally unique key identifying the document. The advisor document is straightforward. The student has two majors (each of which is simple string) and two addresses (each of which is itself a document embedded within the student's "addresses" field). The student's "advisorID" field contains the identifier for the student's advisor, similar to a foreign key in a relational database. Note that many of these features (document references, embedded documents, arrays, embedded schema information) are unavailable in a simple key-value database. As you can see, document databases can be a good choice when your data have some structure, but that structure may be complex or inconsistent among the various documents. Unlike key-value databases, document databases support queries based on the internal structure of the document.

Column Family Databases

The basis for much of the development of column family databases was a structured storage mechanism developed by Google named **Bigtable**, and column family databases are now widely available, with a good example being the Apache Software Foundation's Cassandra project. Facebook did the original development work on Cassandra and then turned it over to the open source development community in 2008.

A generalized column family database storage system is shown in Figure K-8 (which duplicates Figure 8-22). The column family database storage equivalent of a relational DBMS (RDBMS) table has a very different construction. Although similar terms are used, they do *not mean* the same thing that they mean in a relational DBMS.

The smallest unit of storage is called a **column**, but it is really the equivalent of an RDBMS table cell (the intersection of an RDMBS row and column). A column consists of three elements: the *column name*, the *column value* or datum, and a *timestamp* to record when the value was stored in the column. This is shown in Figure K-8(a) by the LastName column, which stores the LastName value Able.

Columns can be grouped into sets referred to as **super columns**. This is shown in Figure K-8(b) by the CustomerName super column, which consists of a FirstName column and a LastName column and which stores the CustomerName value Ralph Able.

Columns and super columns are grouped to create **column families**, which are the column family database storage equivalent of RDBMS tables, as they are typically stored together. In a column family we have rows of grouped columns, and each row has RowKey, which is similar to the primary key used in an RDBMS table. However, unlike an RDBMS table, a row in a column family does not have to have the same number of columns as another row in the same column family. This is illustrated in Figure K-8(c) by the Customer column family, which consists of three rows of data on customers.

Figure K-8(c) clearly illustrates the difference between structured storage column families and RDBMS tables: Column families can have variable columns and data stored in each row in a way that is impossible in an RDBMS table. This storage column structure is definitely *not* in 1NF as defined in Chapter 2, let alone BCNF! For example, note that the first row has no Phone or City columns, while the third row not only has no FirstName, Phone, or City columns but also contains an EmailAddress column that does not exist in the other rows.

All the column families are contained in a **keyspace**, which provides the set of RowKey values that can be used in the data store. RowKey values from the keyspace are shown being used in Figure K-8 to identify each row in a column family. While this structure may seem odd at first, in practice it allows for great flexibility because columns to contain new data may be introduced at any time without modifying an existing table structure. It is up to the application, however, to handle the possibly different row structures.

As shown in Figure K-8(d), a **super column family** is similar to a column family but uses super columns (or a combination of columns and super columns) instead of columns. A super column is a named collection of related columns. The Cassandra DBMS supports super columns. Of course, there is more to

column family database storage than discussed here, but now you should have an understanding of the basic principles of column family databases.

Graph Databases

Based on mathematical graph theory, graph databases are composed of three elements:

1. Nodes
2. Properties
3. Edges

An example graph database management system is **Neo4j.**

Nodes are equivalent to entities in E-R data modeling and tables (or relations) in database design. They represent the things that we want to keep track of or about which we want to store data.

Properties are equivalent to attributes in E-R data modeling and columns (or fields) in database design. They represent the data items that we want to store for each node.

Edges are similar to, but not identical to, the relationships in E-R data models and database designs. They are similar because they connect nodes as relationships connect entities, but they are different because they also store data. Edges can also have a "direction."

Figure K-9 shows a graph database based on part of the HSD database. Note that the some of the data in the HSD SEMINAR, CUSTOMER, and SEMINAR_CUSTOMER tables are replicated in this database. CUSTOMER data are here, SEMINAR data are here, and the data about what customer attended which seminar are also here. However, the graph database adds some additional data. First, the edges labeled *Attendees* for a group named Attendees: While this grouping could be obtained by an SQL query in a relational database, it is a built-in component of the graph database. Similarly, the edge with ID 3001 and labeled *knows* adds entirely new data to the data set—additional data about the relationships between customers that does not exist in the original HSD database design.

Clearly, the graph database can easily extend our original data model and provide additional pathways between nodes.

Name: LastName
Value: Able
Timestamp: 40324081235

(a) A Column

Super Column Name:	CustomerName	
Super Column Values:	Name: FirstName	Name: LastName
	Value: Ralph	Value: Able
	Timestamp: 40324081235	Timestamp: 40324081235

(b) A Super Column

Column Family Name:	Customer			
RowKey001	Name: FirstName	Name: LastName		
	Value: Ralph	Value: Able		
	Timestamp: 40324081235	Timestamp: 40324081235		
RowKey002	Name: FirstName	Name: LastName	Name: Phone	Name: City
	Value: Nancy	Value: Jacobs	Value: 817-871-8123	Value: Fort Worth
	Timestamp: 40335091055	Timestamp: 40335091055	Timestamp: 40335091055	Timestamp: 40335091055
RowKey003	Name: LastName	Name: EmailAddress		
	Value: Baker	Value: Susan.Baker@elswhere.com		
	Timestamp: 40340103518	Timestamp: 40340103518		

(c) A Column Family

Super Column Family Name:	Customer			
Rowkey001	Customer Name		CustomerPhone	
	Name: FirstName	Name: LastName	Name: AreaCode	Name: PhoneNumber
	Value: Ralph	Value: Able	Value: 210	Value: 281–7987
	Timestamp: 40324081235	Timestamp: 40324081235	Timestamp: 40335091055	Timestamp: 40335091055
Rowkey002	Customer Name		CustomerPhone	
	Name: FirstName	Name: LastName	Name: AreaCode	Name: PhoneNumber
	Value: Nancy	Value: Jacobs	Value: 817	Value: 871–8123
	Timestamp: 40335091055	Timestamp: 40335091055	Timestamp: 40335091055	Timestamp: 40335091055
Rowkey003	Customer Name		CustomerPhone	
	Name: FirstName	Name: LastName	Name: AreaCode	Name: PhoneNumber
	Value: Susan	Value: Baker	Value: 210	Value: 281–7876
	Timestamp: 40340103518	Timestamp: 40340103518	Timestamp: 40340103518	Timestamp: 40340103518

(d) A Super Column Family

Figure K-8 — A Generalized Column Family Database Storage System

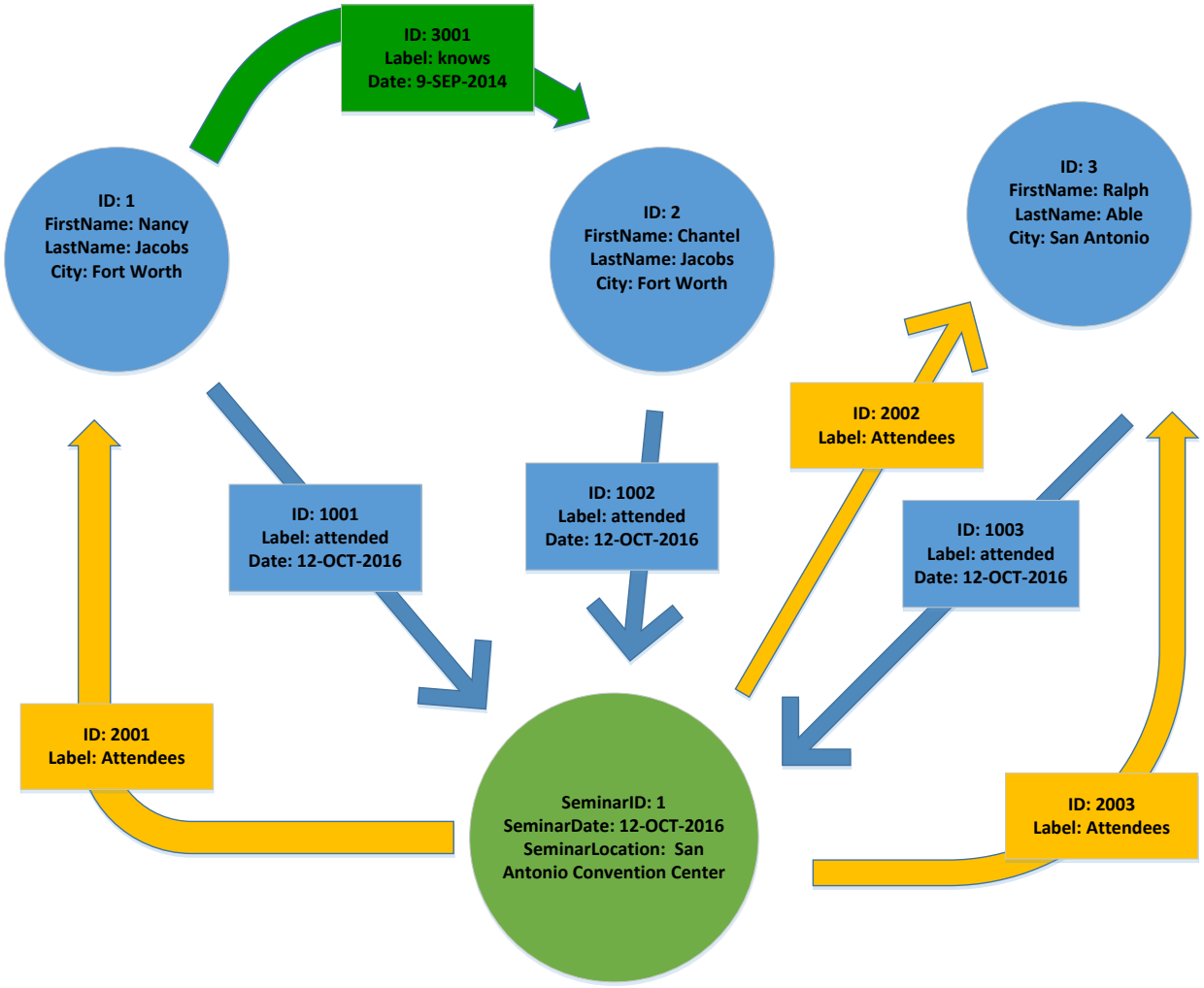


Figure K-9 — A Partial HSD Graph Database

Big Data, NoSQL Systems, and the Future

The usefulness and importance of these Big Data products to organizations such as Facebook demonstrate that we can look forward to the development of not only improvements to relational DBMSs, but also to a very different approach to data storage and information processing. Big Data and products associated with Big Data are rapidly changing and evolving, and you should expect many developments in this area in the near future. Many organizations today use both relational and NoSQL databases to fit specific needs—this is known as **polyglot persistence**.

By The Way

The Not only SQL world is an exciting one, but you should be aware that if you want to participate in it you will need to sharpen your programming skills. Whereas we can develop and manage databases in Microsoft Access, Microsoft SQL Server, Oracle Database, and Oracle MySQL using management and application development tools that are very user-friendly (Microsoft Access itself, Microsoft SQL Server Management Studio, Oracle SQL Developer, and MySQL Workbench), application development in the NoSQL world is currently done in programming languages.

This, of course, may change, and we look forward to seeing the future developments in the Not only SQL realm. For now, you'll need to sign up for that programming course!

Using the Database Features of Microsoft Azure

Now it is time to put your cloud and NoSQL knowledge to work. In this section, you will learn how to create a free account on Microsoft Azure (which we will, like Microsoft, sometimes refer to simply as Azure) then migrate an existing database from your PC to the Azure cloud. We will then create an SQL database on Azure using scripts. Later we will create and query a simple NoSQL DocumentDB database. For the SQL section, though, you will need to install Microsoft SQL Server 2016 and the **Microsoft SQL Server Management Studio (SSMS)** that functions as the graphical front-end to SQL Server AND to the SQL Database functionality on Azure. There are no install prerequisites for following our procedure to create an Azure account and use DocumentDB.

The SQL section assumes that you have performed the actions in Appendix A, "Getting Started with Microsoft Server 2016," to install SQL Server 2016 and SSMS and created the WP database. The SQL Database server on Azure is very similar to but not exactly the same as SQL Server 2016. Some of the advanced features of SQL Server 2016 are not supported on Azure's SQL Database, but for our purposes they are the same. If you have not performed the actions in Appendix A, a possible quick-start is to install SSMS and create a smaller database example in SQL scripts, then use SSMS to upload the SQL scripts and run a query. You must perform Steps 1–3 below whether you are migrating a database, creating one directly on Azure, or both.

Step 1: Connect to Microsoft Azure home page at azure.microsoft.com/en-us/free.

As shown in Figure K-10, select the **Start Free** button. Note, in the future, after you have created your account, use the **MY ACCOUNT** button on the top right to log in to manage your Azure services or to check your billing status.

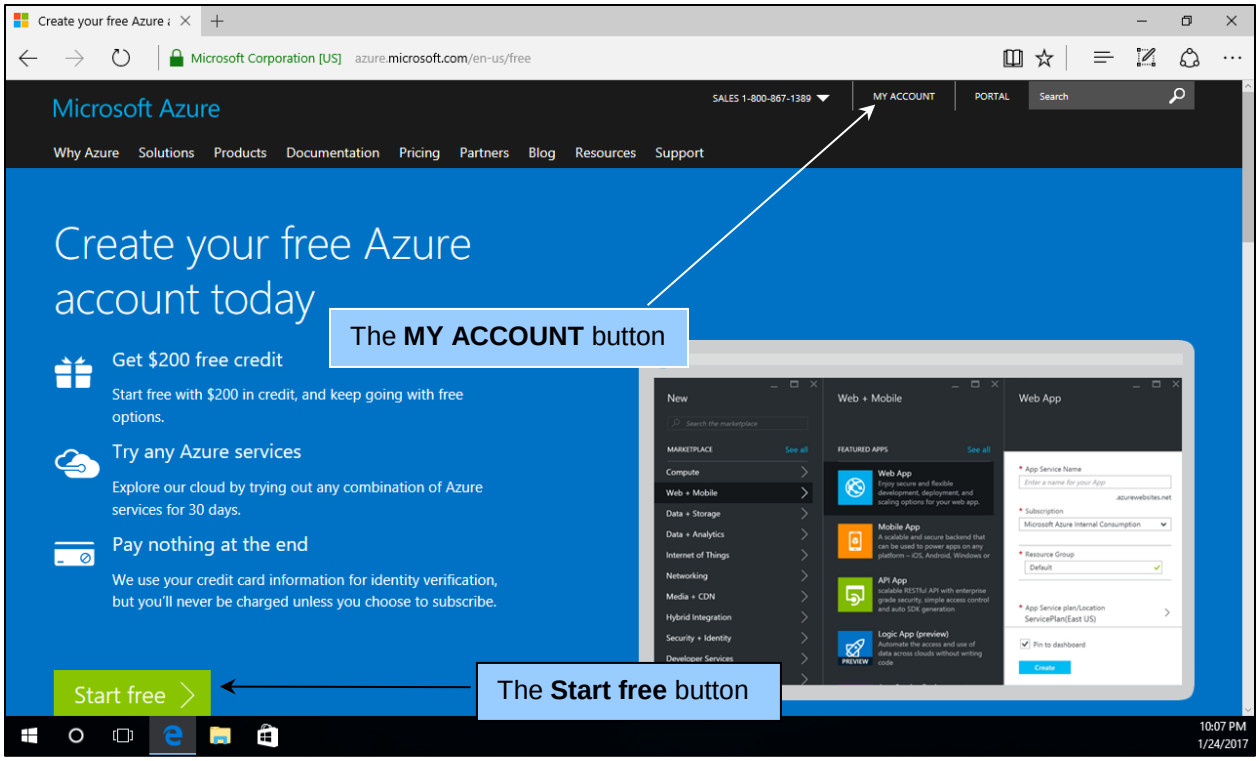


Figure K-10 — Create Your Free Azure Account

A screen requesting your email information is shown. Select your email account and enter your email password, then click the **Sign in** button to begin the process of setting up your Azure account. A screen displaying some of your personal information, such as your name, appears. Then enter your phone number to confirm your identity, as shown in Figure K-11. You may find it easiest to enter your cell phone number and receive a text message with your verification number that you then enter.

The next screen for Figure K-12 asks for your credit card information as another way to confirm your identity. Microsoft indicates that your card will not be charged. You should verify this, as Microsoft may change the terms of the agreement at its discretion. Then click the **Next** button. Finish the sign-up process by checking the **Agreement** box at the bottom of the screen.

Your account will be set up and you will get the Welcome to Microsoft Azure screen. You may choose tutorials or start managing your service.

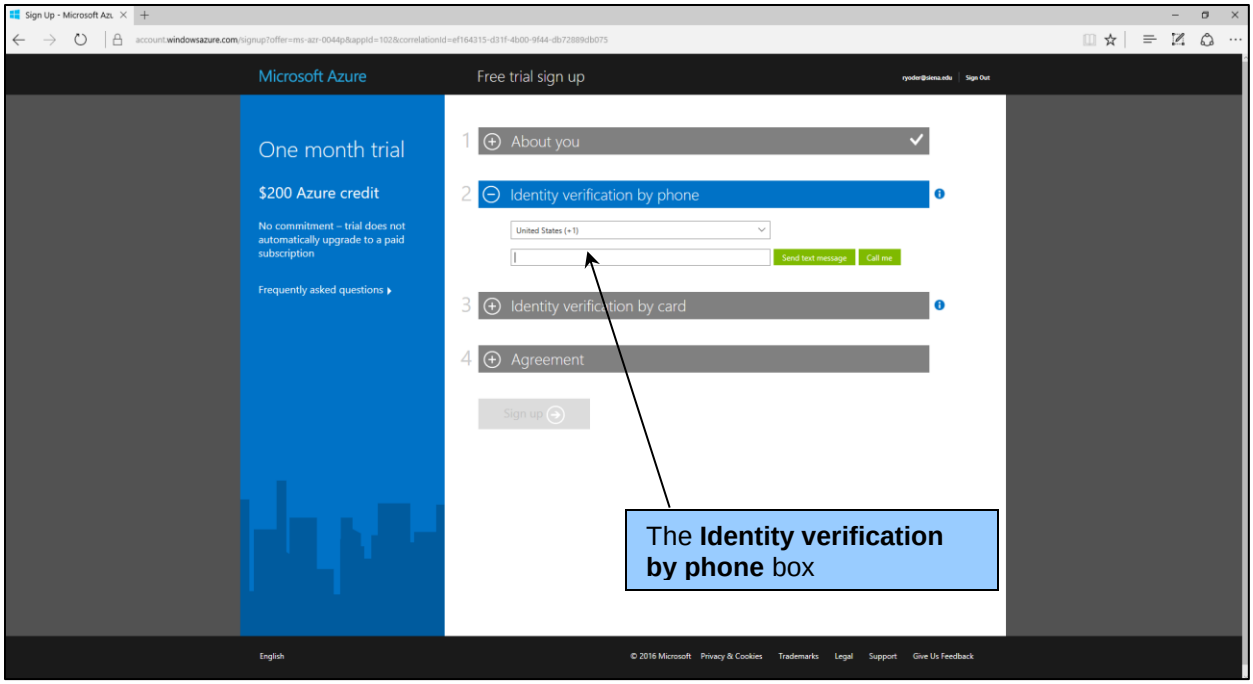


Figure K-11 — Trial Sign-up Information

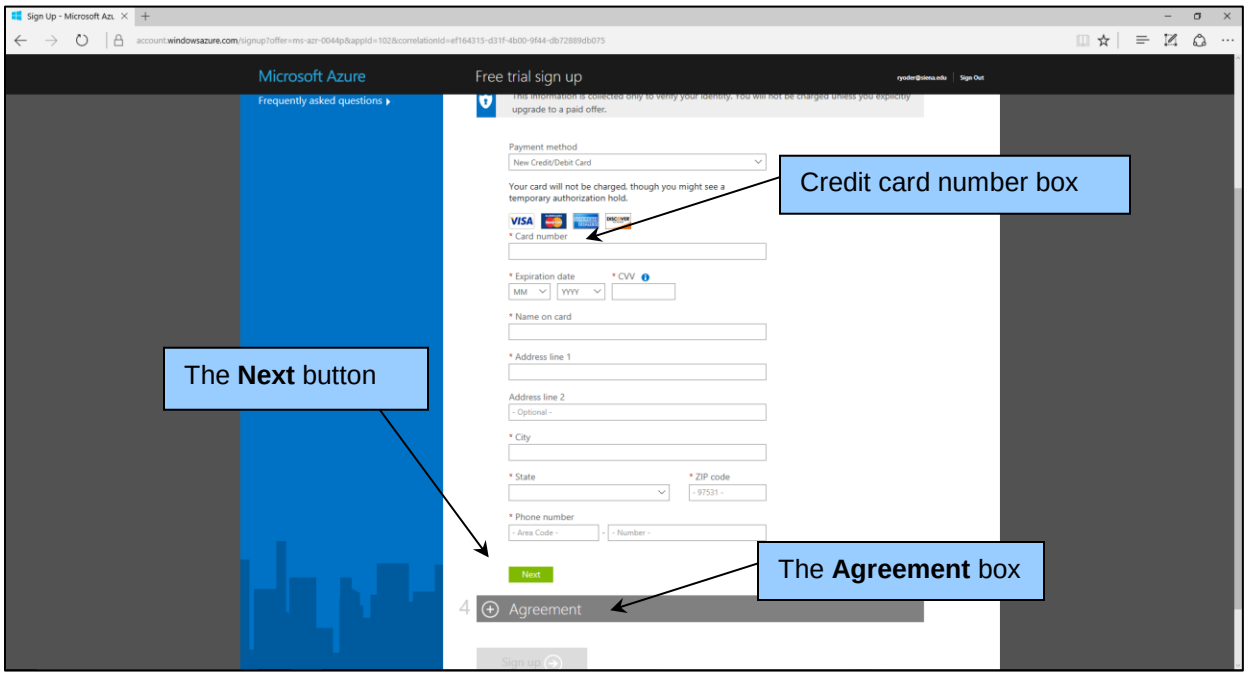


Figure K-12 — Credit Card Verification of Your Identity

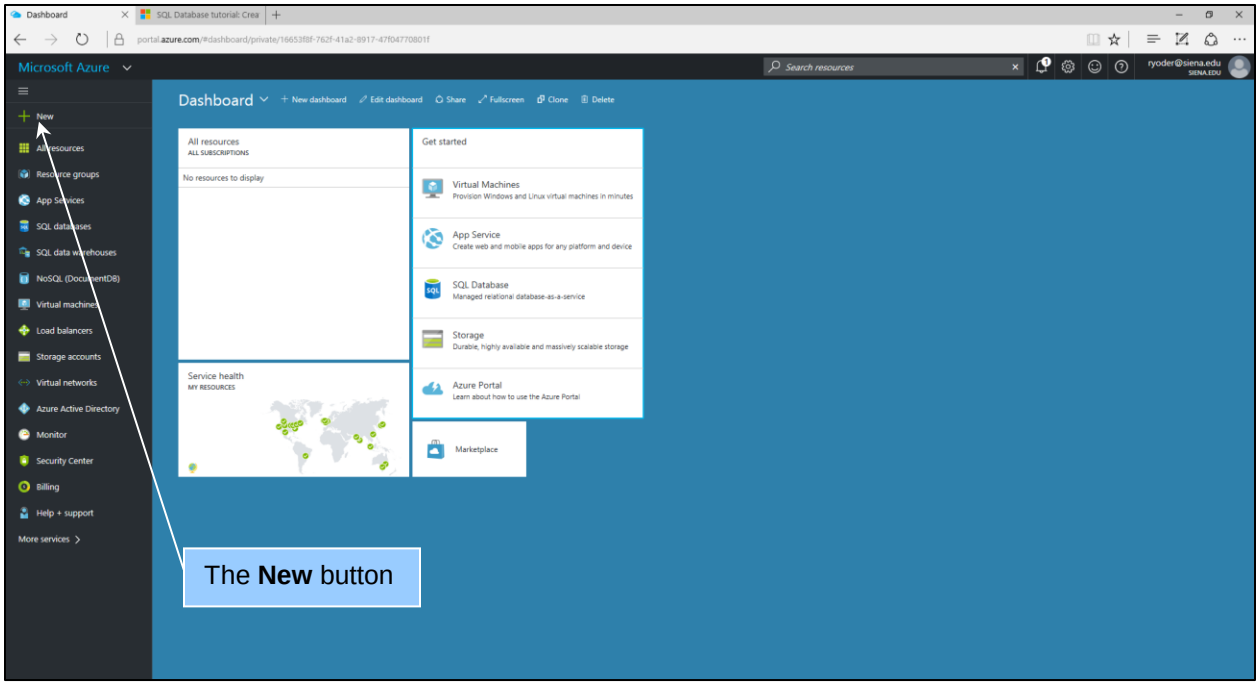


Figure K-13 — Azure Dashboard to Manage Your Services

Step 2: Create a new SQL database.

Click **Start managing my service** button. You will be directed to the Azure Dashboard screen as shown in Figure K-13. As you can see, Azure has a rich selection of other services, including virtual machines, data storage, and NoSQL Document database servers. Click the **New** button at the top of the left pane, then **Databases** as shown in Figure K-14.

In the New Database screen, select the **SQL Database** button under Featured Apps. You will need to create a new database server, set up a resource group, and create your administrator login and password, as shown in Figure K-15. Click **Create a new server** button, then fill in your information similar to that shown. Be sure to record your login and password somewhere! It is also a good idea to pick the geographic location of the server nearest you to minimize network transmission time. You should also create a blank database to establish a resource group, but we will be using SQL Server Management Studio (SSMS) to create our databases later in this appendix. To complete the process, click the **Select** button in the bottom right to select the new server and the **Create** button in the bottom left to create the database.

Figure K-14 — The New Databases Screen

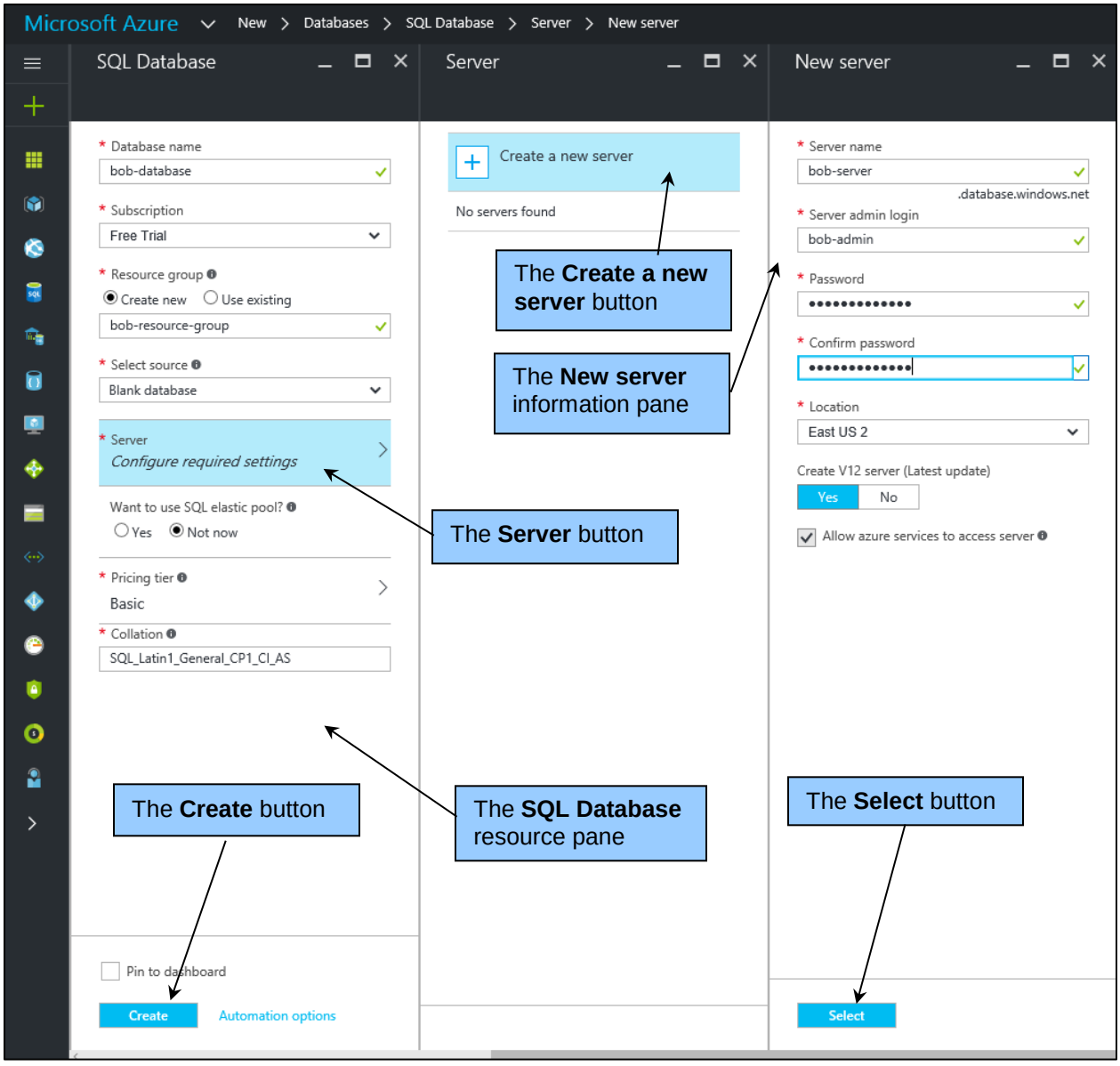


Figure K-15 — Create a New SQL Database Server

Step 3: Set up your database server firewall rules.

After you have established your administrator account and a blank database, you will need to configure your firewall to allow only certain computers (IP addresses) to access your databases. Security is very important in cloud computing. Select the **Firewall** button under SETTINGS (the gear-shaped button near the top will access settings), as shown in Figure K-16.

In the firewall connection dialog box as shown in Figure K-17, make up a **Rule Name** (I used "bob-firewall") and enter your IP address in the **Start IP** box. The dialog box may indicate your Client IP address. If not, Web browsers will tell you your IP address if you enter "My IP address" in the search box. You may need to enter your IP address again the first time you connect to your Azure server. In addition, your IP address may change if you are using a wireless connection.

Congratulations! Your Azure account is set up. From now on, you will mostly be interacting with Azure through SSMS as you create new SQL databases and run queries against them.

The next section shows you how to migrate the existing WP (Wedgewood Pacific) database from your local PC to Azure. The section after that shows how to create a new Microsoft Azure WP database instance directly in the cloud using SQL scripts. You may choose to do either or both procedures.

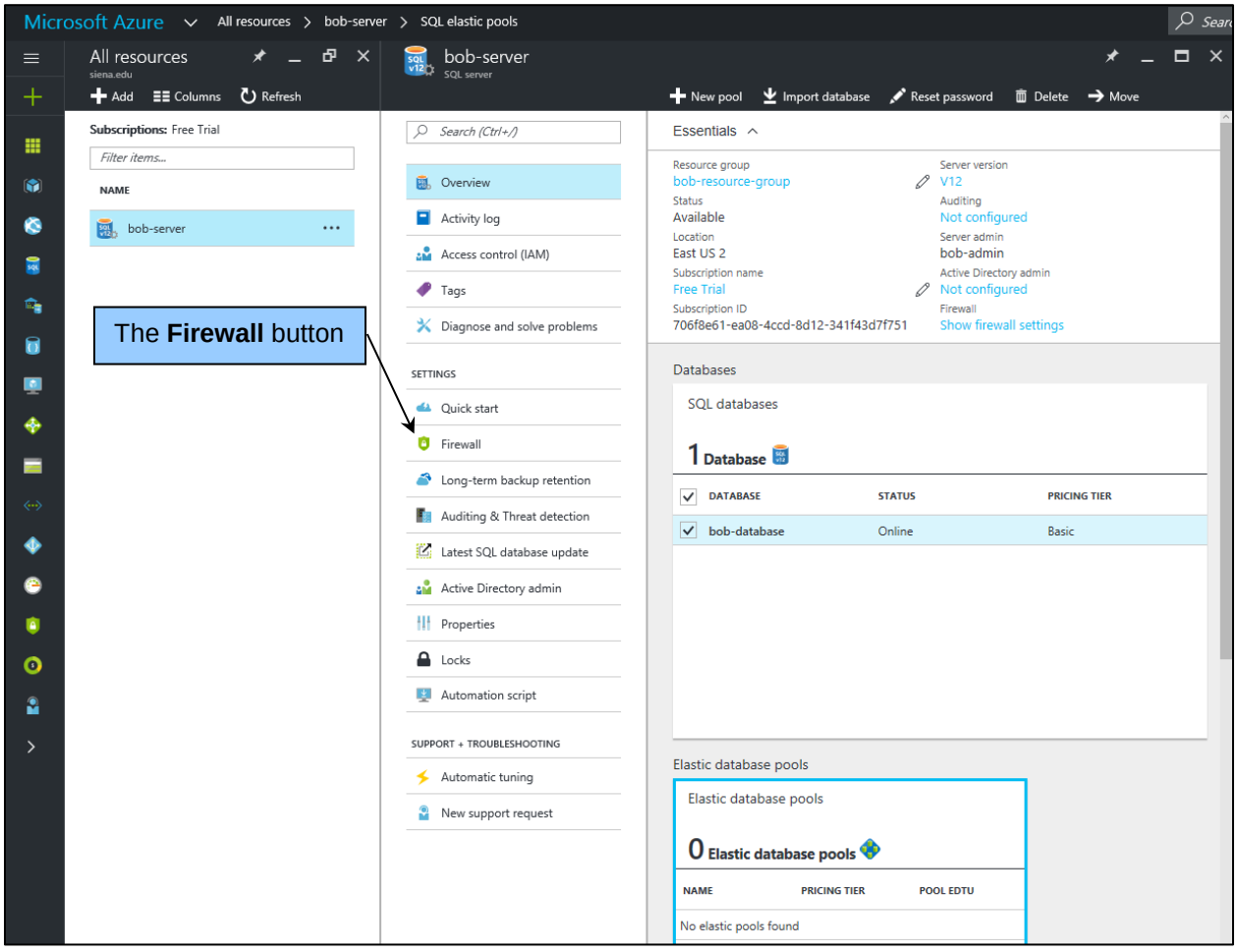


Figure K-16 — SQL Server Settings—Firewall

Migrating Your Existing Local WP Database to Azure

If you don't have an existing WP database on your PC, go back to Appendix A for instructions or skip this step. Startup SSMS, connect to your local SQL Server installation, then connect to Azure using **Connect | Database Engine** command from the Object Explorer window to get the dialog box shown in Figure K-18. Select or enter the name of your Microsoft Azure server and its administrator login and password, then click on the **Connect** button.

Now, in the Object Explorer, right-click the local WP database to show the sub-menu. Select **Tasks | Deploy Database to Microsoft Azure SQL Database** command. The **Deploy Database Introduction** screen appears, as shown in Figure K-19. Click the **Next** button.

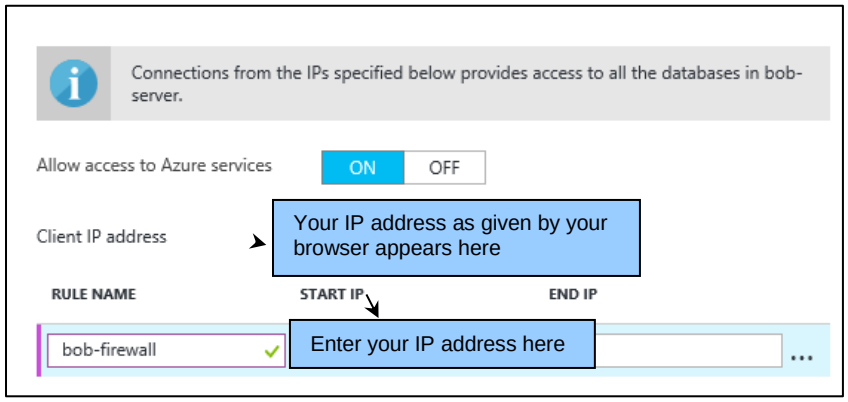


Figure K-17 — Configure the Firewall Dialog Box

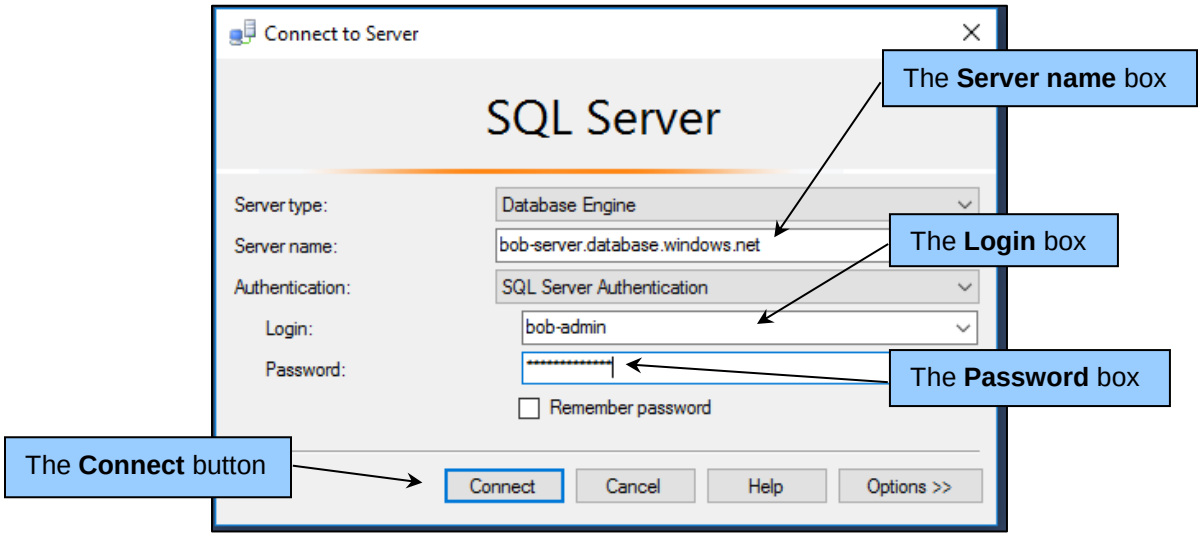


Figure K-18 — Connect to SQL Server Dialog Box

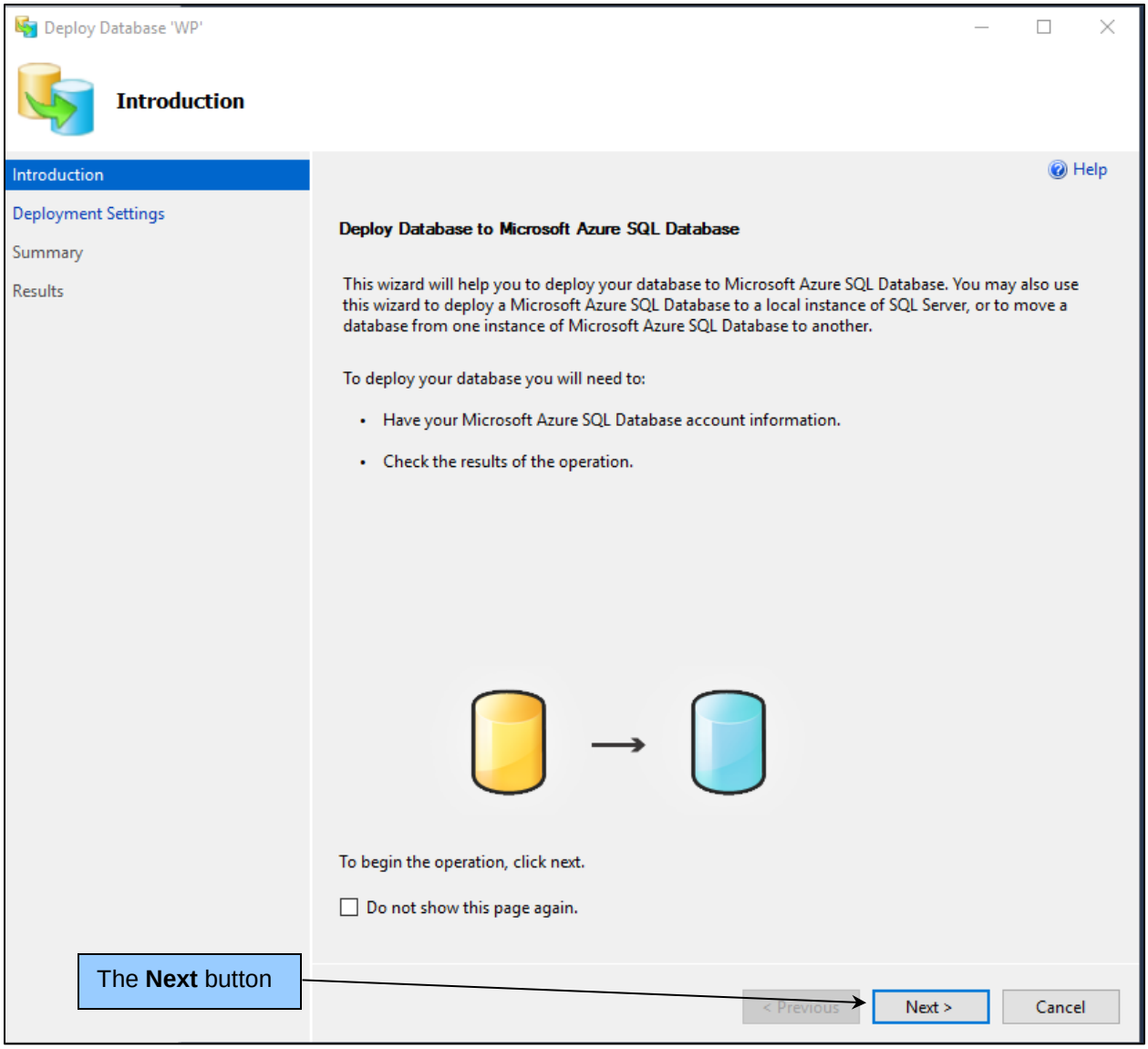


Figure K-19 — Deploy Database to Microsoft Azure SQL Database

In the **Deployment Settings** page as shown in Figure K-20, click the **Connect…** button to bring up the SQL Server **Connect to Server** dialog box again, as shown in Figure K-21. Enter the remote connection credentials to Azure, then click the **Connect** button, as shown in Figure K-22.

Once the login and password are provided, the deployment settings for the server connection are filled in, as shown in Figure K-22. Click the **Next** button. The **Summary** page will appear after a few seconds, as shown in Figure K-23.

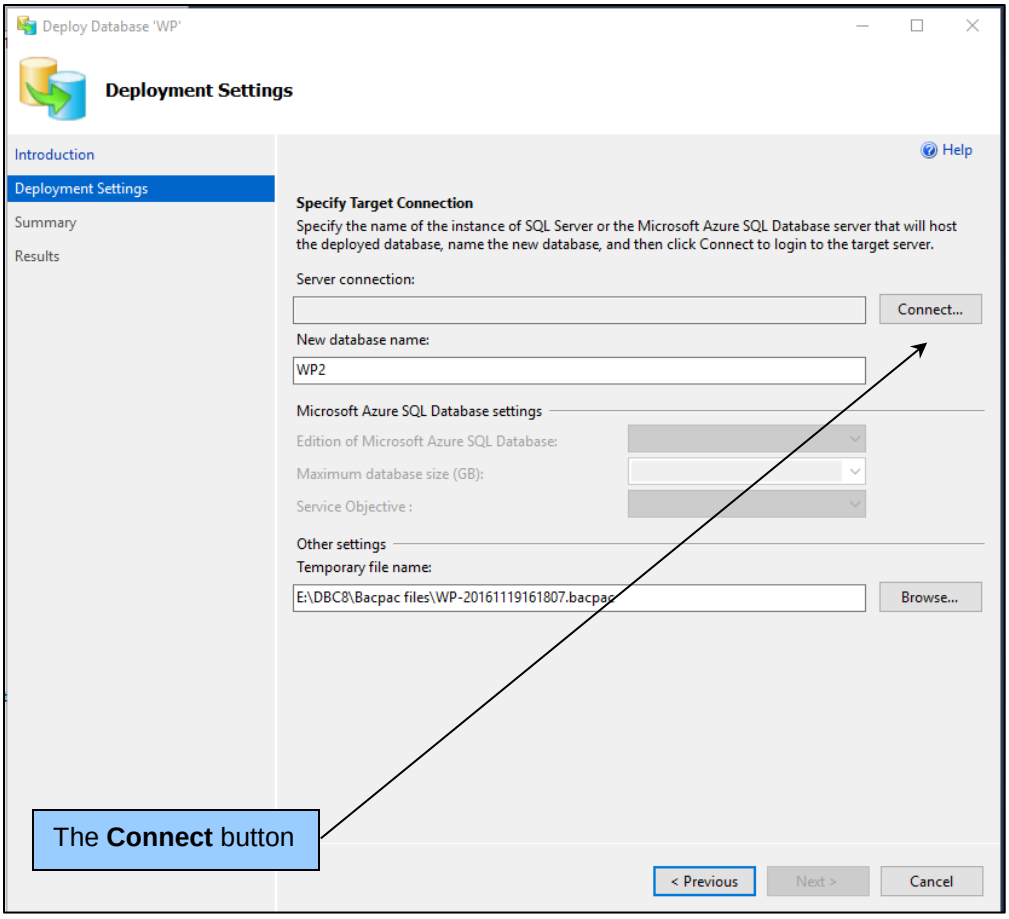


Figure K-20 — Deployment Settings

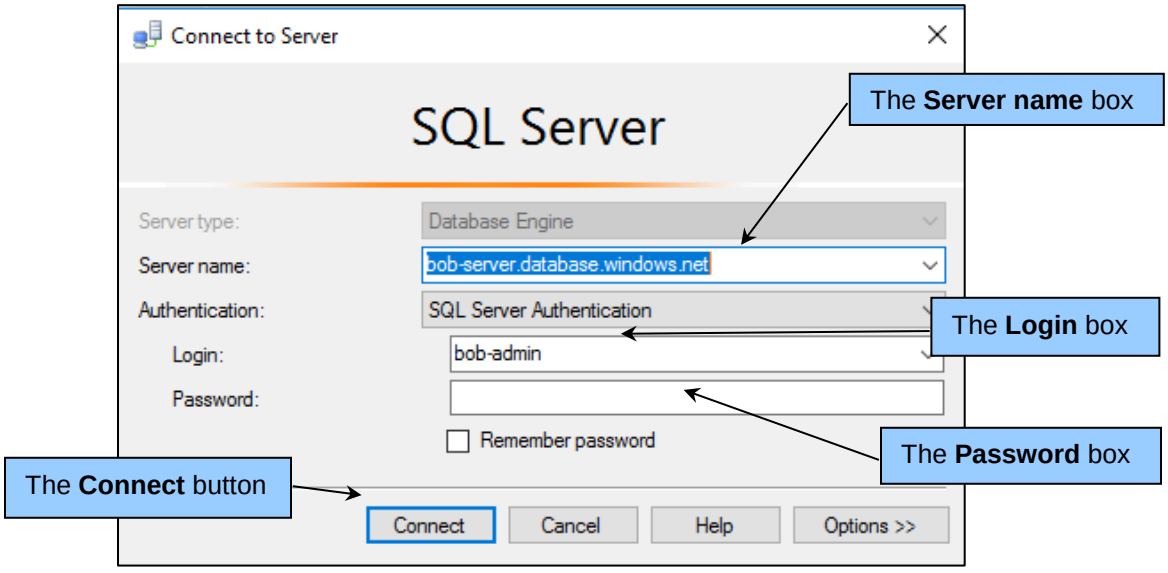


Figure K-21 — Connect to SQL Server

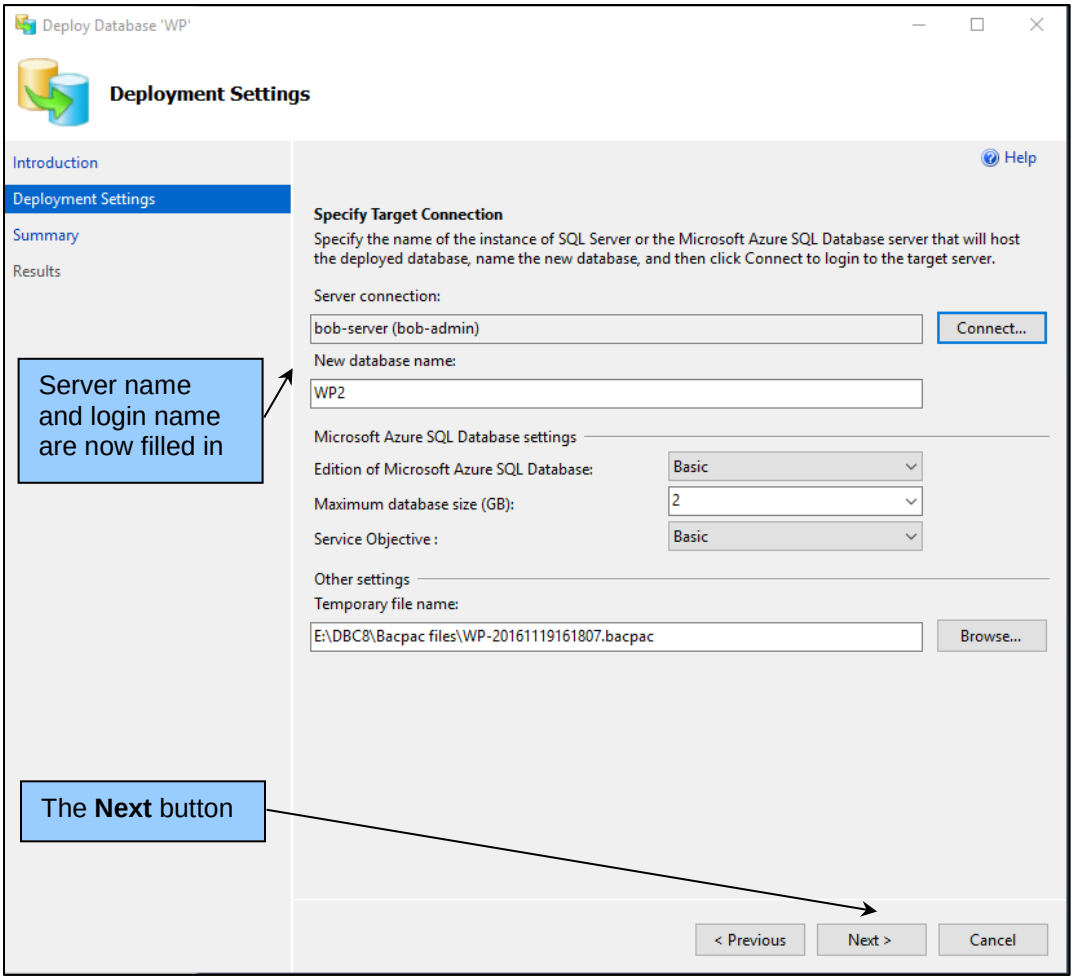


Figure K-22 — Deployment Settings

Click the **Finish** button. The Progress screen appears. Be patient, as it may take a minute or so to import the database to the remote Microsoft Azure server. Eventually the Results screen appears, as shown in Figure K-24. Click the **Close** Button. You will see the new database WP2 appear in SSMS under the remote server name, as shown in Figure K-25.

To test WP2 on Azure, select the WP2 icon, then the **New Query** button. A new SQL query blank window opens. Enter a simple SQL query such as:

```
SELECT * FROM DEPARTMENT;
```

then execute it.

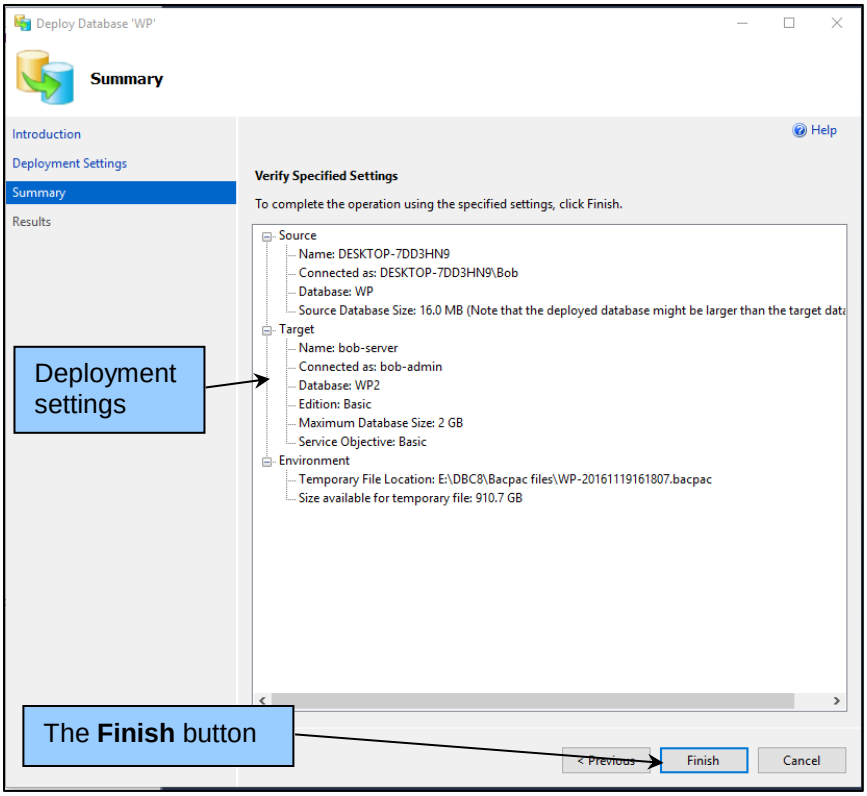


Figure K-23 — Deployment Summary

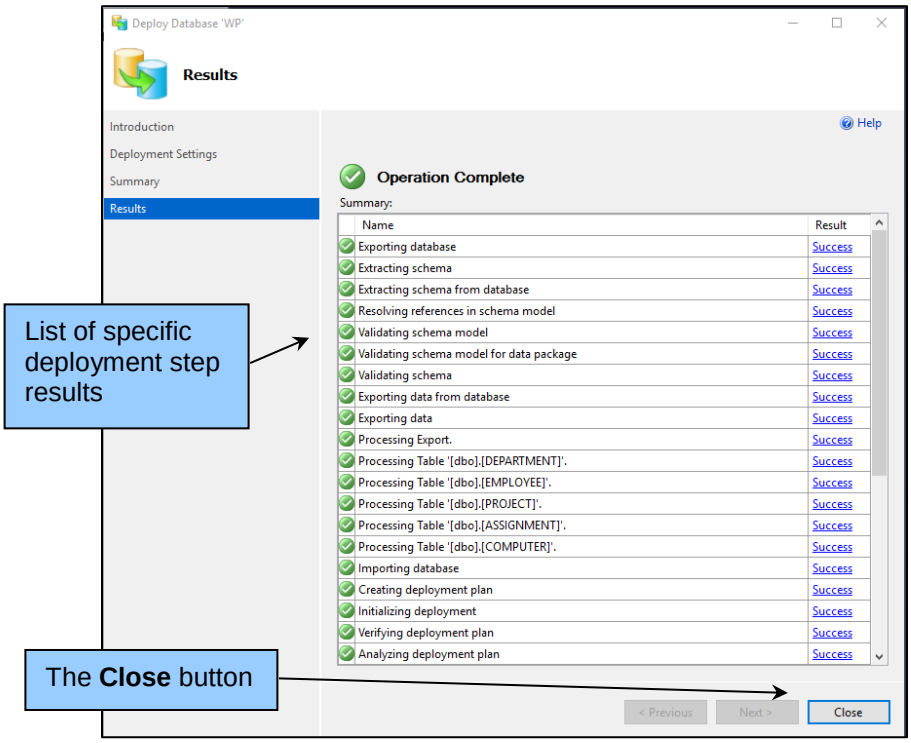


Figure K-24 — Results: Deployment Operation Complete

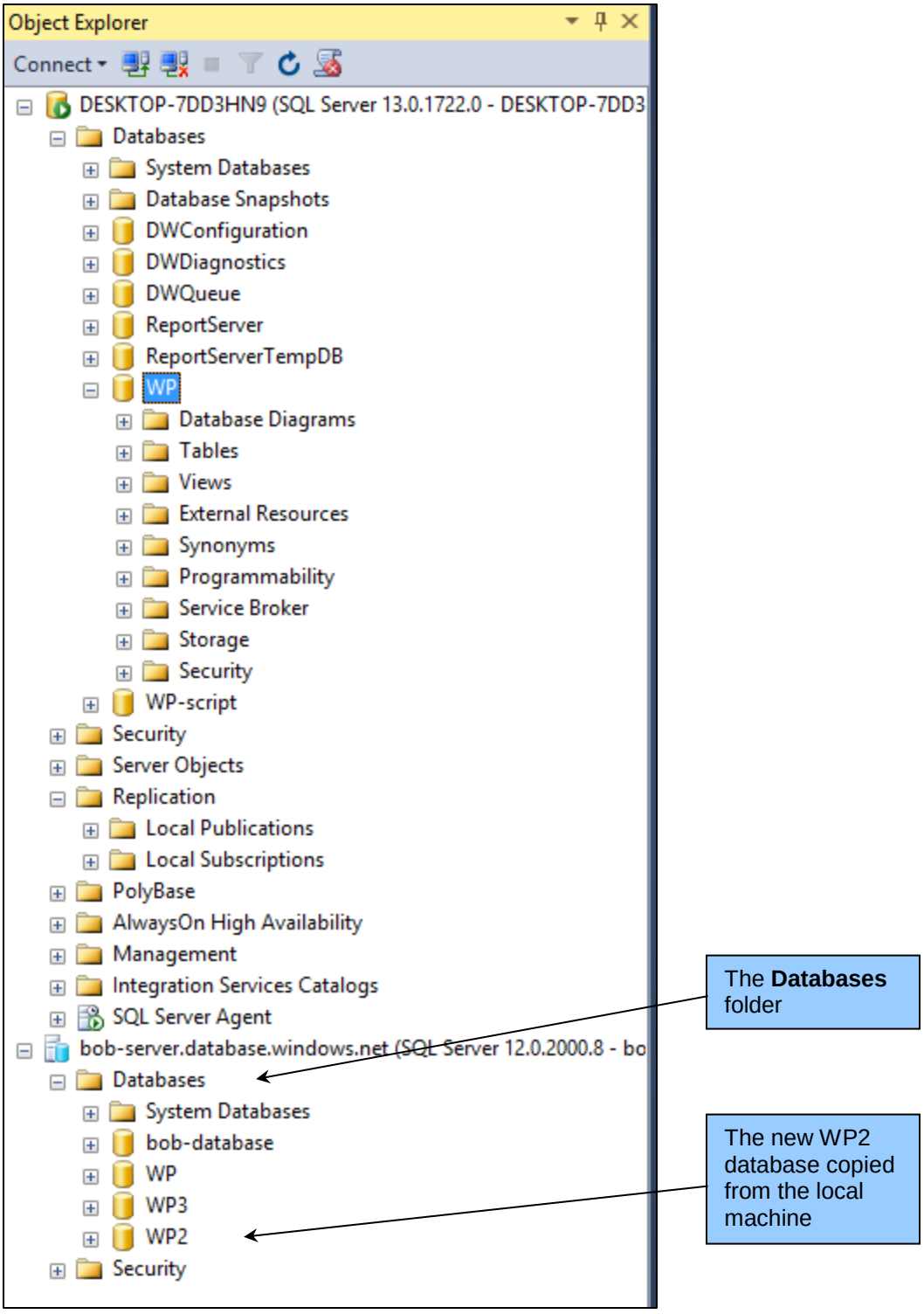


Figure K-25 — SSMS Object Explorer Showing New Azure WP2 Database

How Do I Delete the WP2 Database on Azure?

Click the remote Azure database server name in Object Explorer in SSMS, then click on the WP2 database icon to select it. Then right-click the WP2 icon on the remote server. Make sure you have selected the WP2 database on the correct server and not the local SQL server list, then select the **Delete** command. A confirmation box appears as shown partially in Figure K-26. Click the **OK** button. Note: Sometimes it takes a little while for SSMS to get an acknowledgment from Azure that the database was deleted properly, or you may even get a "Drop failed" error message. If that occurs, click the **OK** button at the bottom of the screen then close the window. To confirm deletion, simply disconnect then reconnect to the Azure server.

Using SQL Scripts to Create a New WP3 Database on Azure

Start SSMS. We will be connecting to BOTH our local Microsoft SQL Server and remote Microsoft Azure databases. Connect to the remote Azure Database engine and enter the password, as shown in Figure K-27. Then use the **File | Connect Object Explorer** command to connect to your local SQL Server 2016 database engine, as shown in Figure K-28. Note that for Microsoft Azure we use SQL Server authentication and for our local desktop instance we use Windows Authentication.

If you have previously created the WP database as outlined in Appendix A, then you already have the DBC-e08-MSSQL-Create-Tables.sql script file in your Projects | WP-Database folder. In not, refer to the Appendix A section "Creating an SQL Script in Microsoft SQL Server 2016" that creates the tables and the following section "How Do I Use SQL Statements to Insert Database Data?" that describes how to create a script to insert the data into the newly created tables. Hopefully you named it something like DBC-e08-MSSQL-Insert-Data.sql.

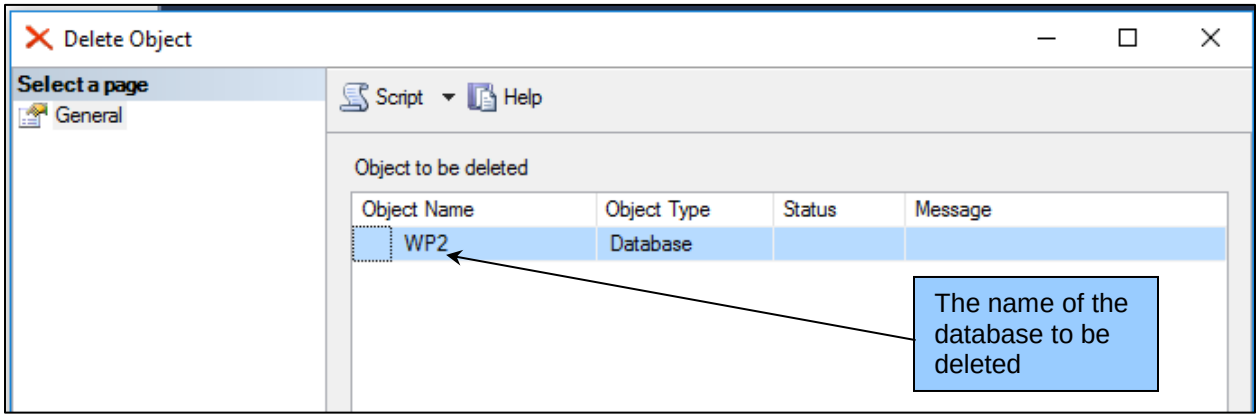


Figure K-26 — Confirm Deletion of Azure SQL Database

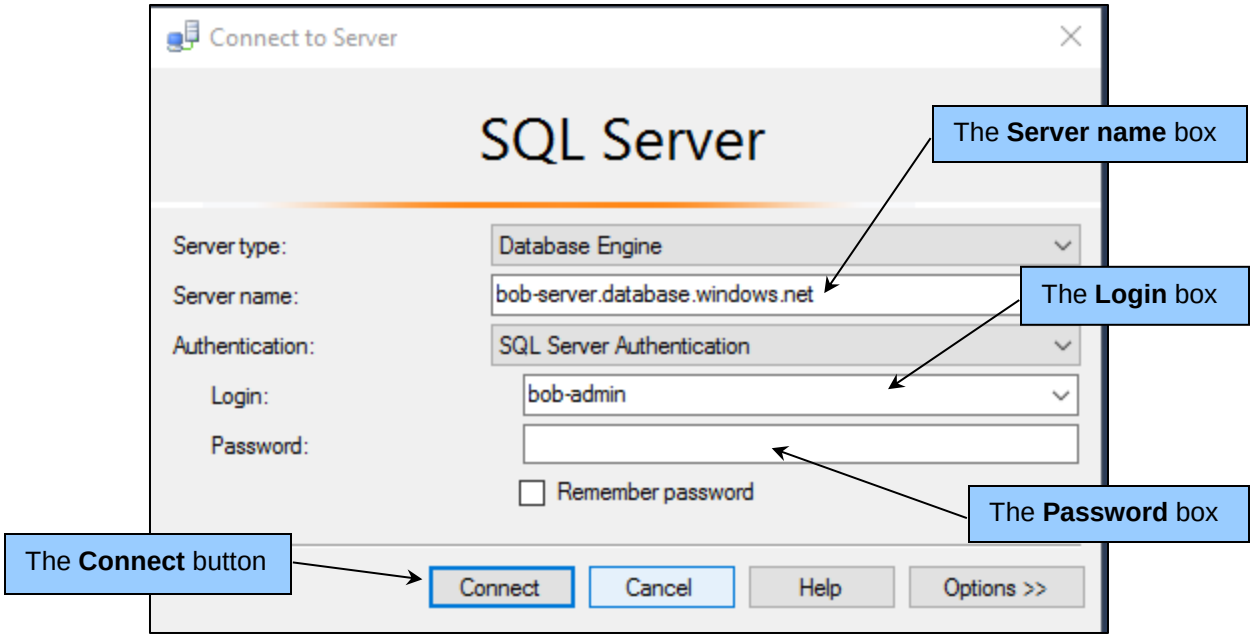


Figure K-27 — Connect to Remote Azure SQL Server

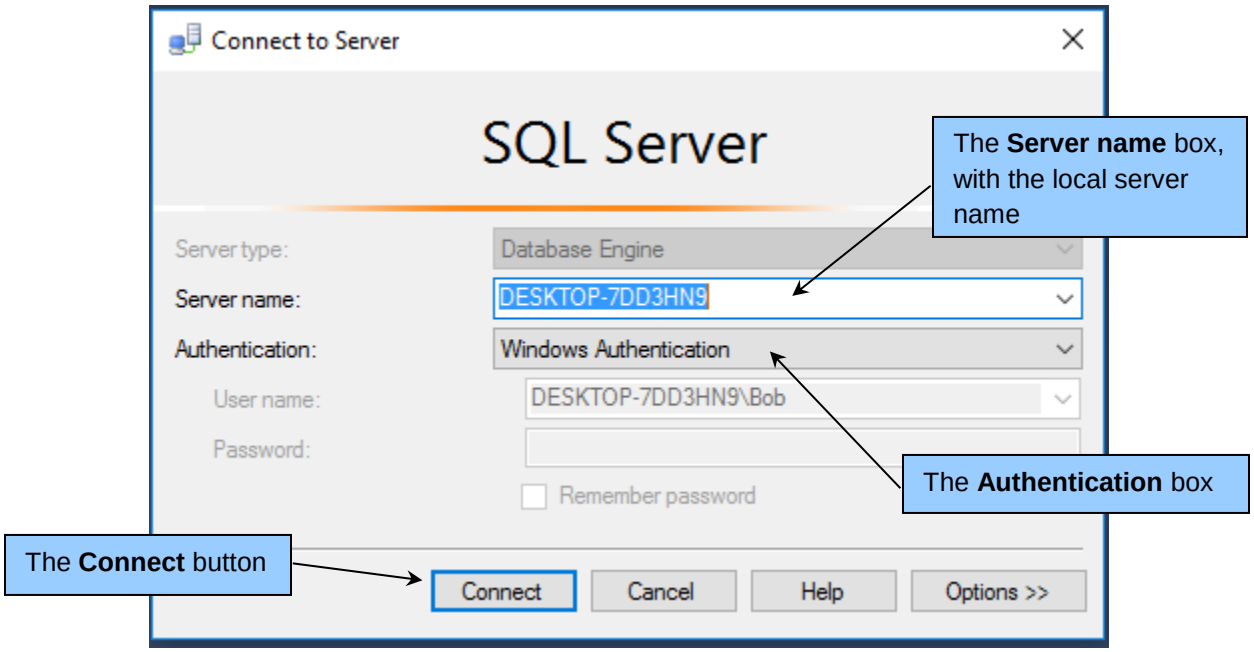


Figure K-28 — Connect to Local SQL Server

Before running our scripts to create our WP database on Microsoft Azure, please edit the scripts to add or change the first two lines of both scripts to be the following to ensure we are updating the correct database:

```
USE WP3
GO
```

That command will ensure that we run the script against the correct database. Don't run the scripts yet! We first need to create the WP3 database on Microsoft Azure. In SSMS, under the remote Microsoft Azure server section of Object Explorer, select the **Databases** folder, as shown at the bottom of Figure K-25. Right-click it and select the **New Database...** command. After a few seconds a **New Database** window appears, as shown in Figure K-29. Enter the new database name WP3 (to distinguish it from the WP2 database we created earlier in this appendix), then click the **OK** button at the bottom of the screen (not shown).

The new database name WP3 should appear in the Object Explorer window in SSMS under the remote server object name. Next select the **File | Open | File…** command, browse to DBC-e08-MSSQL-WP-Create-Tables.sql then open it. Be sure WP3 is selected as the current database in the top-left command bar in SSMS, as shown in Figure K-30.

Execute the file. If it runs correctly, open then execute DBC-e08-MSSQL-WP-Insert-Data, again making sure that WP3 is the active database.

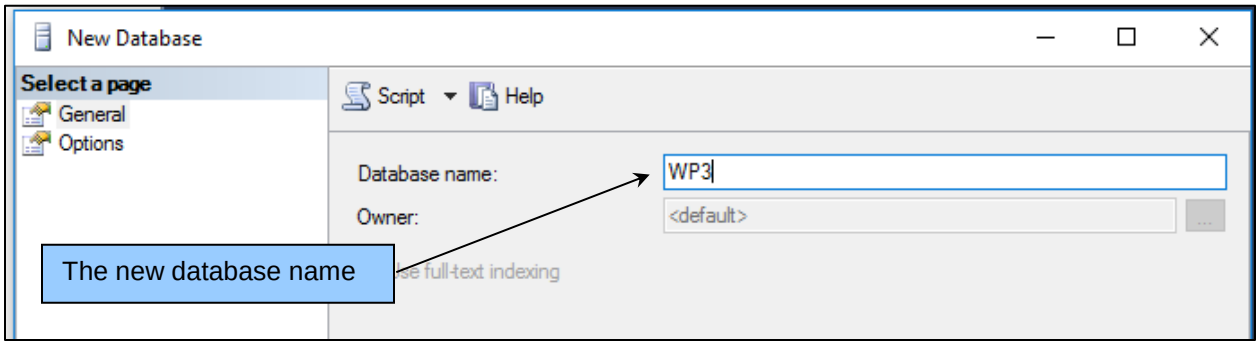


Figure K-29 — SSMS New Database Screen

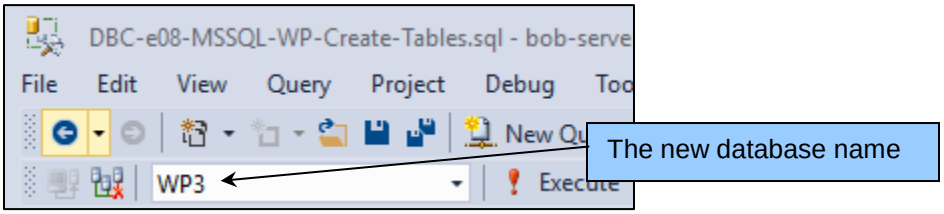


Figure K-30 — SSMS Object Explorer Showing Local WP Database

Test your new cloud-based relational database by using the **New Query** button on the Menu bar. Enter a simple query like:

```
SELECT * FROM DEPARTMENT;
```

and run it. If it worked, congratulations! If not, re-check your previous steps, making sure that the database names are correct and that WP3 is the active database. If desired, you can delete the WP3 database by using the Azure Database Delete instructions earlier in this appendix.

Creating and Using a NoSQL Document Database in Azure

The purpose of this section is to get you started with creating a simple NoSQL document database in Azure. We first create a Microsoft Azure DocumentDB account and blank database. Then we will create a collection of Students, upload two JSON documents into the Students collection (each representing a student), and run a simple query. Then you can read the documentation at your leisure and perhaps try some queries on a more sophisticated database.

Login to Microsoft Azure and enter the Azure Portal. If needed, expand the left pane using the Show Menu icon (with the three horizontal bars), as shown in Figure K-31. Select **NoSQL (DocumentDB)**.

You will see the NoSQL control panel, with no databases yet created, as shown in Figure K-32.

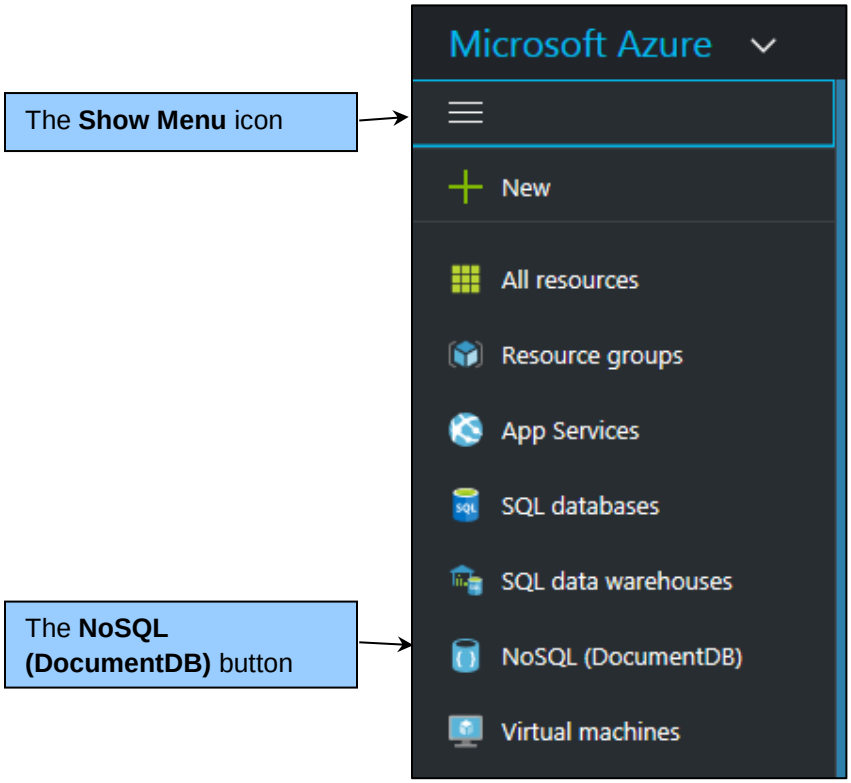


Figure K-31 — Microsoft Azure Menu Options

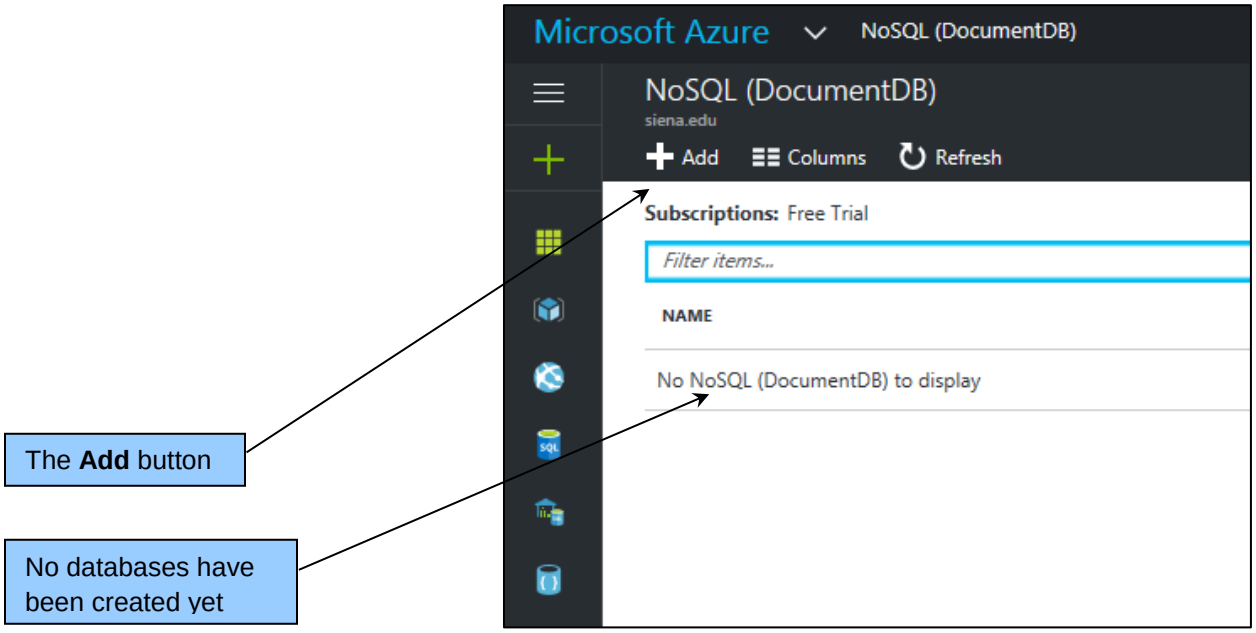


Figure K-32 — Microsoft Azure DocumentDB Control Panel

Click the white **Add** button on the NoSQL control panel to create a new DocumentDB database. Name your document database using lowercase letters and dashes only. We used dbc08-document-db and selected the DocumentDB API, as shown in Figure K-33. Use the resource group previously created for your Azure account, then click the **Create** button (not shown) at the bottom of the pane. Application Program Interfaces (APIs) are used by application programs to connect to and use services provided by other systems. A detailed discussion is beyond the scope of this appendix.

Be patient: It may take a minute or so to create and deploy your new database. It will show up in the NoSQL (DocumentDB) control panel. You may need to use the **Refresh** button to see it, as shown in Figure K-34. Select your new database. You will see the dbc08-document-db pane, showing its information and options, as in Figure K-35.

Click the **Add Collection** button at the top of the pane. The **Get Started** button looks tempting but does not allow us to name the collection. Enter the Collection Id "Students" and enter the database name at the bottom as shown in Figure K-36 (only the **Create New** button is enabled—that is OK). Click the **OK** button at the bottom (not shown) to create the collection.

You will then see the dbc08-document-db screen showing the list of collections, as shown in Figure K-37.

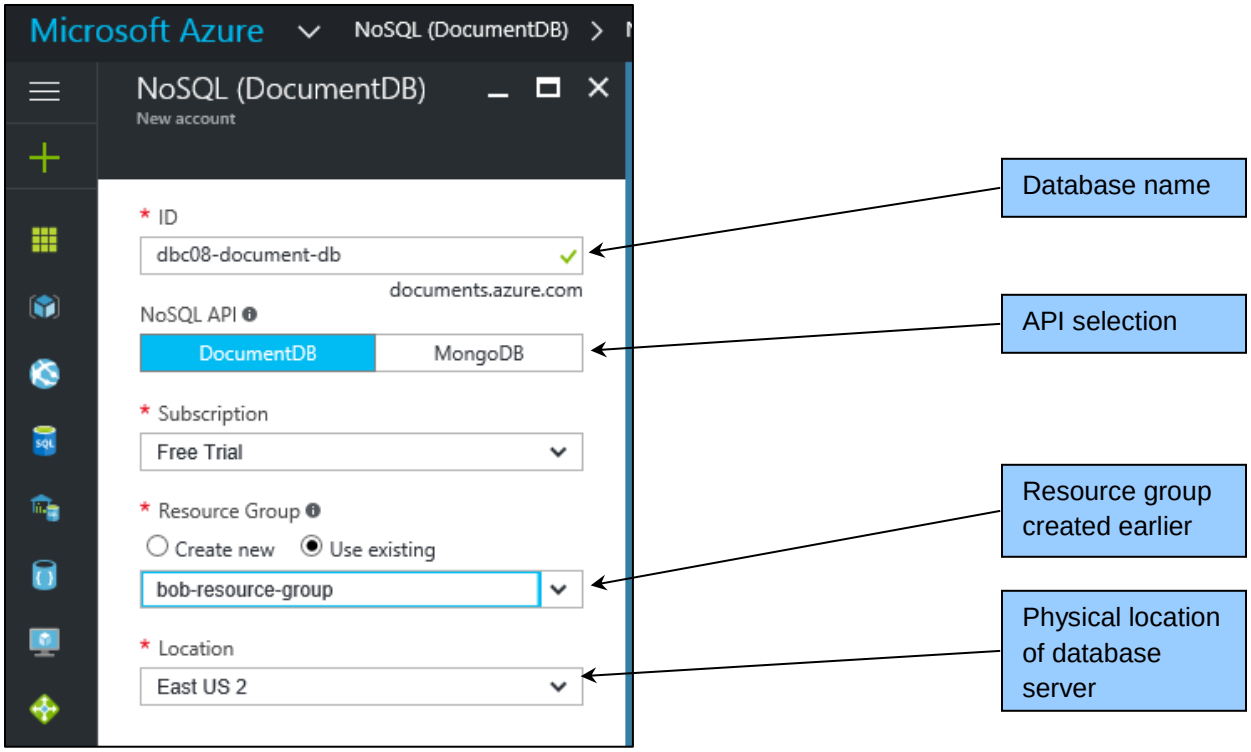


Figure K-33 — Create New DocumentDB Database

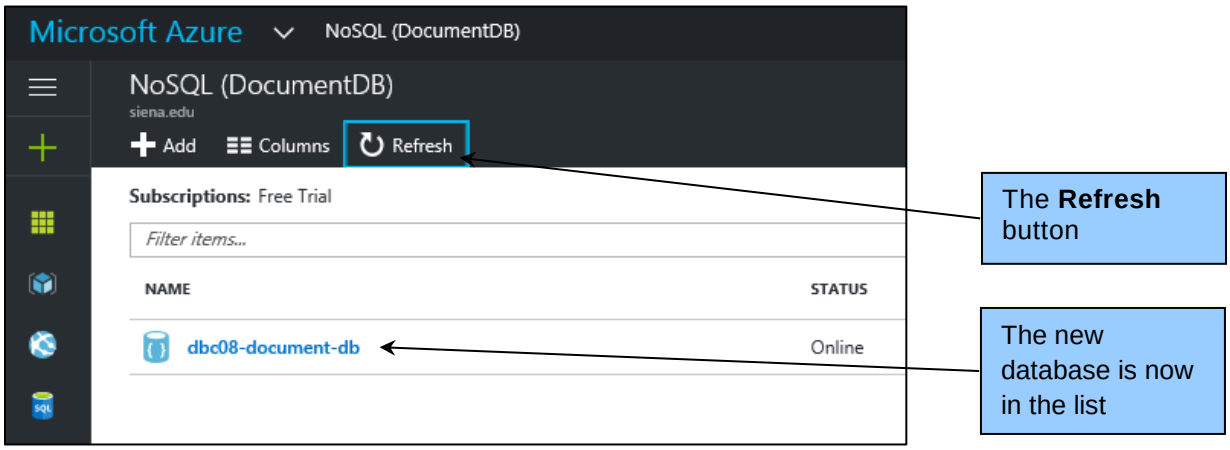


Figure K-34 — New DocumentDB Database Created

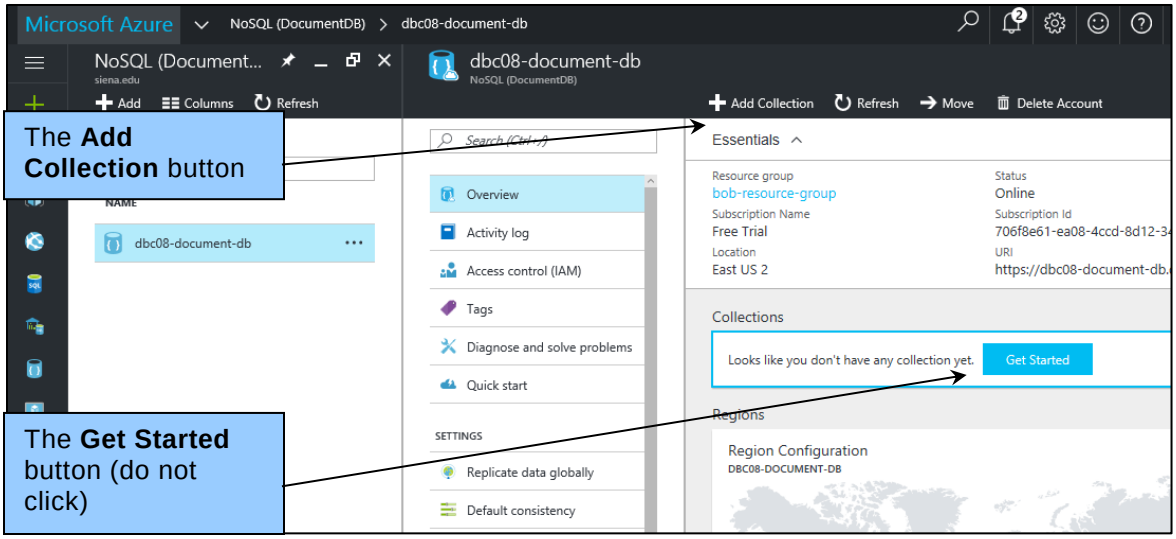


Figure K-35 — New DocumentDB Database Overview Pane

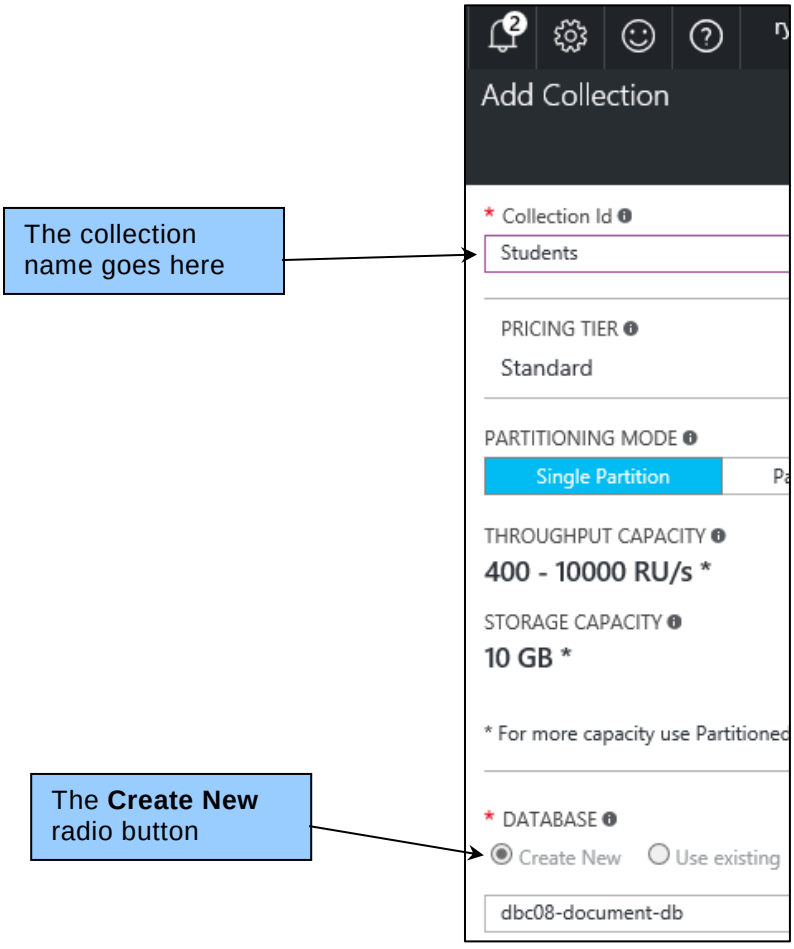


Figure K-36 — Creating a New DocumentDB Collection

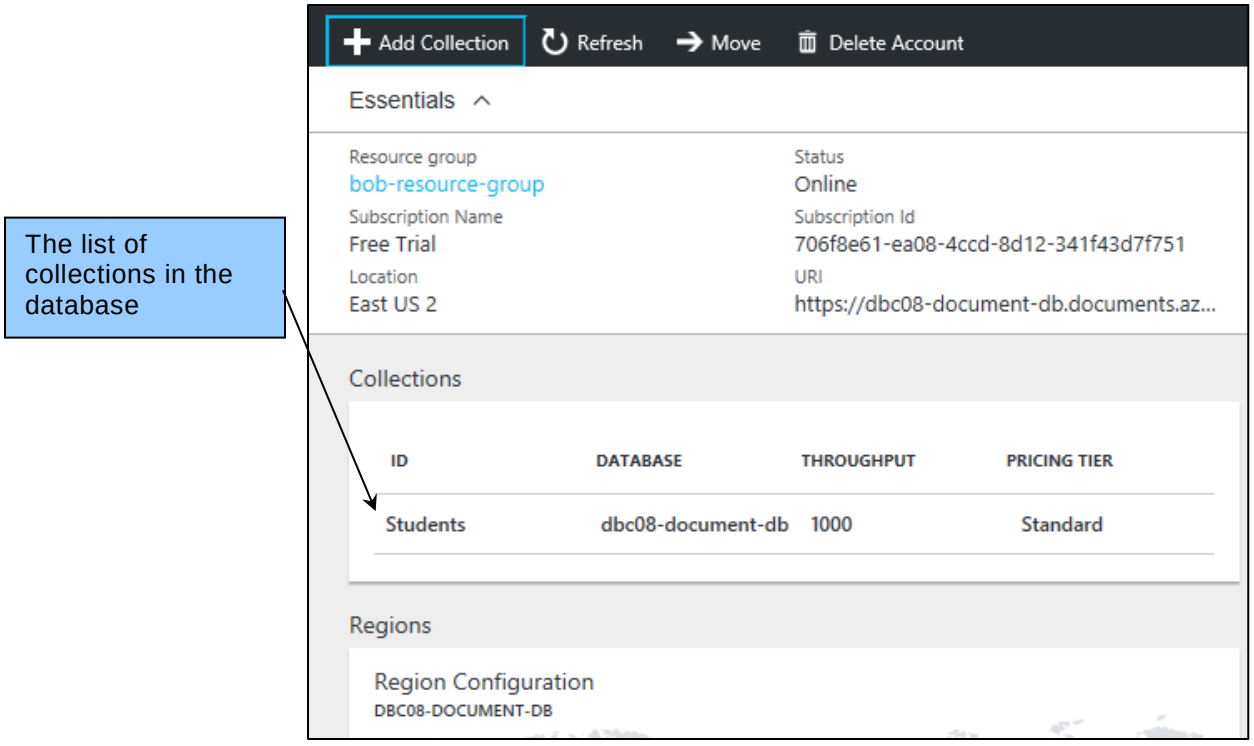


Figure K-37 — DocumentDB Database Collections List

Now, using NotePad or some other text editor, create a new file named **student1.json**. Put the following JSON code in it, and save it to a folder of your choice.

```
{
      "id":  "555667777",
      "name":  "Tierney, Doris",
      "majors":  ["Music", "Spanish"],
      "addresses":  [
            {
            "street":  "14510 NE 4th Street",
            "city":  "Bellevue",
            "state":  "WA",
            "zip":  "98005"
            },
            {
            "street":  "335 Aloha Street",
            "city":  "Seattle",
            "state":  "WA",
            "zip":  "98109"
            }
            ],
      "advisorID":  "Advisor1"
}
```

Next create another student and save it in **student2.json**:

```
{
        "id":  "901667777",
        "name":  "Orsini, Meg",
        "majors":  ["Math"],
        "addresses":  [
              {
              "street":  "21 Knob Ave",
              "city":  "Valley View",
              "state":  "NY",
              "zip":  "12943"
              }
              ],
        "advisorID":  "Advisor3"
}
```

Now we can use the Microsoft Azure Document Explorer to upload our two student documents into the Students collection. As shown in Figure K-38, the **Document Explorer** button is on the left pane in our dbc-e08-document-db control panel. Click the **Document Explorer** button. Next click the **Upload** button on the top of the panel, as shown in Figure K-39.

The **Upload Document** pane appears, as shown in Figure K-40. Click on the folder icon to select a file. In the **Open** dialog box as shown in Figure K-41, browse to the folder containing the student JSON files you created earlier, select both student1.json and student2.json, then click on the **Open** button (not shown). The files should appear in the **Upload Document** dialog box, as shown in Figure K-42.

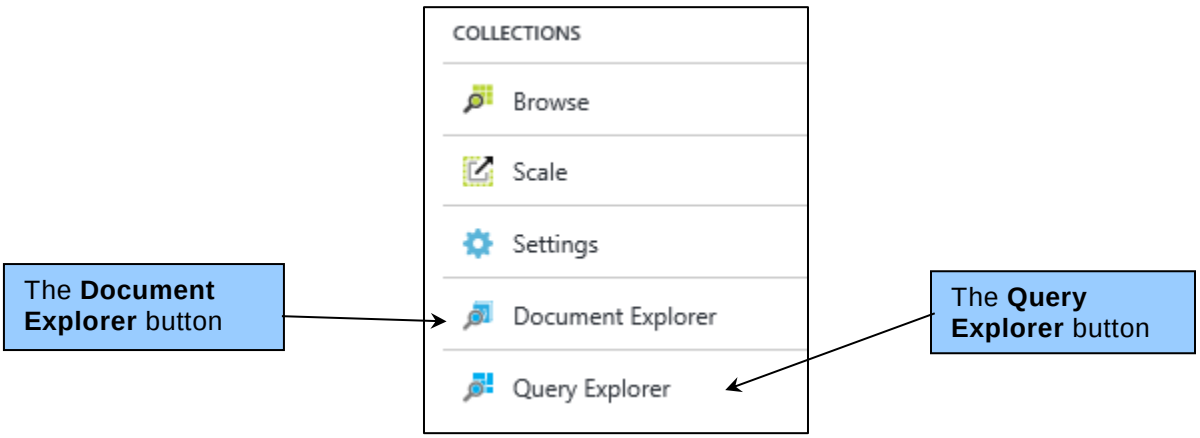


Figure K-38 — DocumentDB Control Panel with Document Explorer and Query Explorer Options

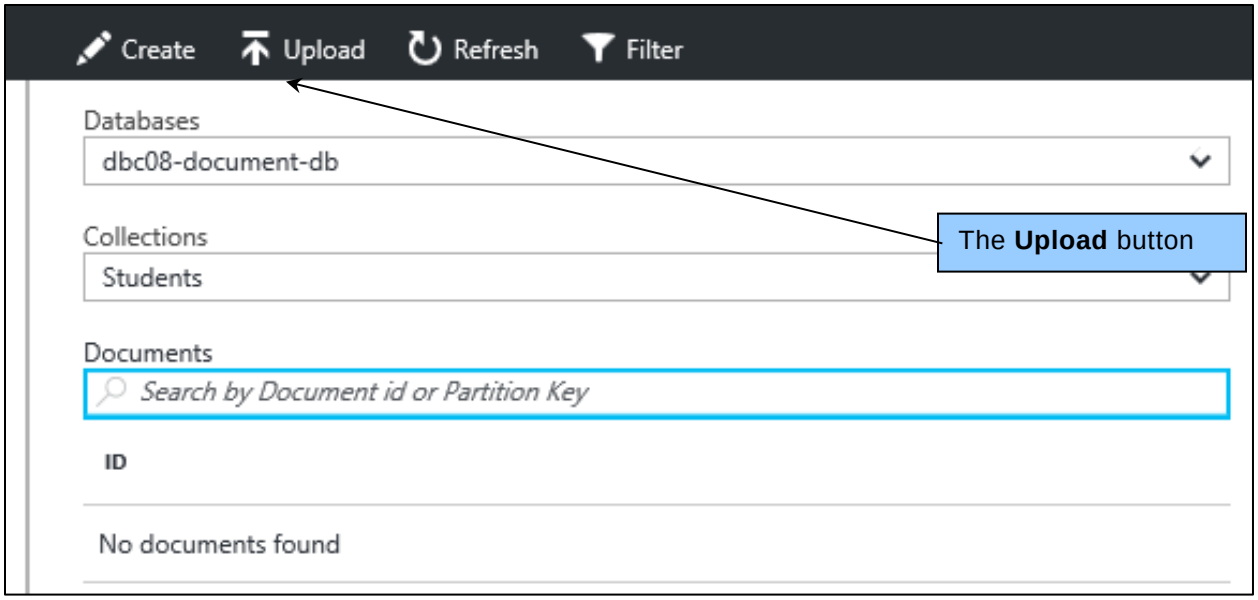


Figure K-39 — Document Explorer

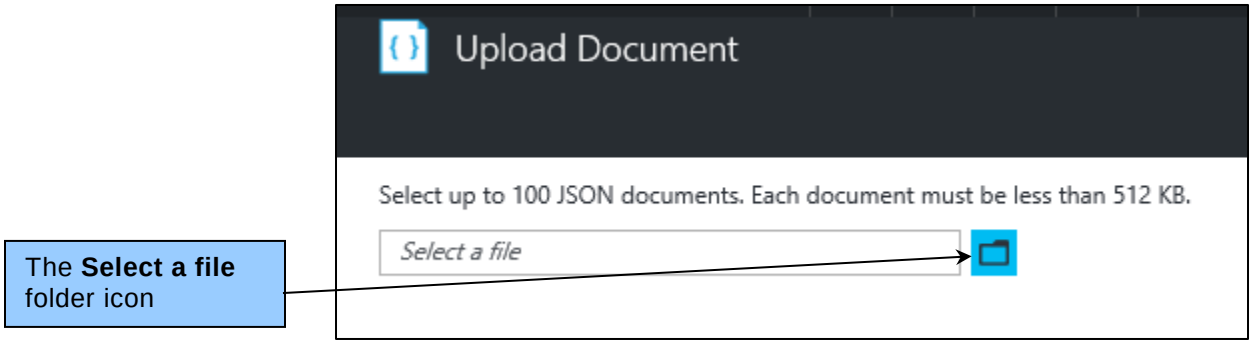


Figure K-40 — Document Upload: Select Documents

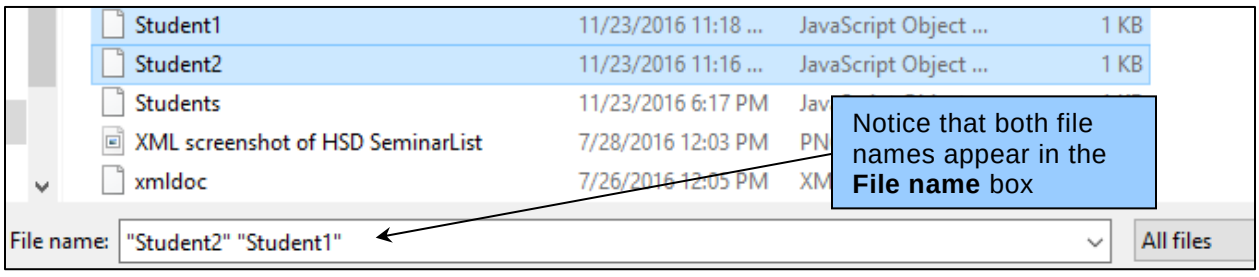


Figure K-41 — Open Dialog Box

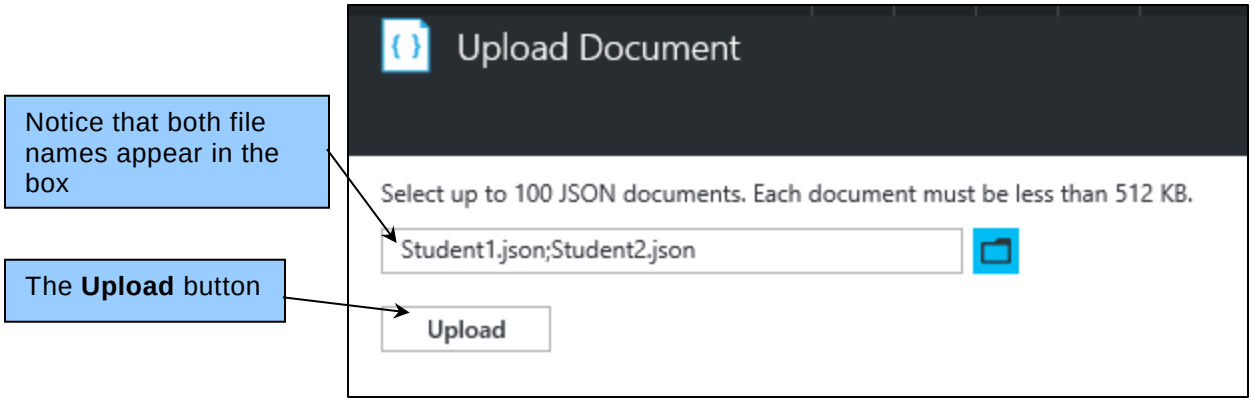


Figure K-42 — Upload Documents

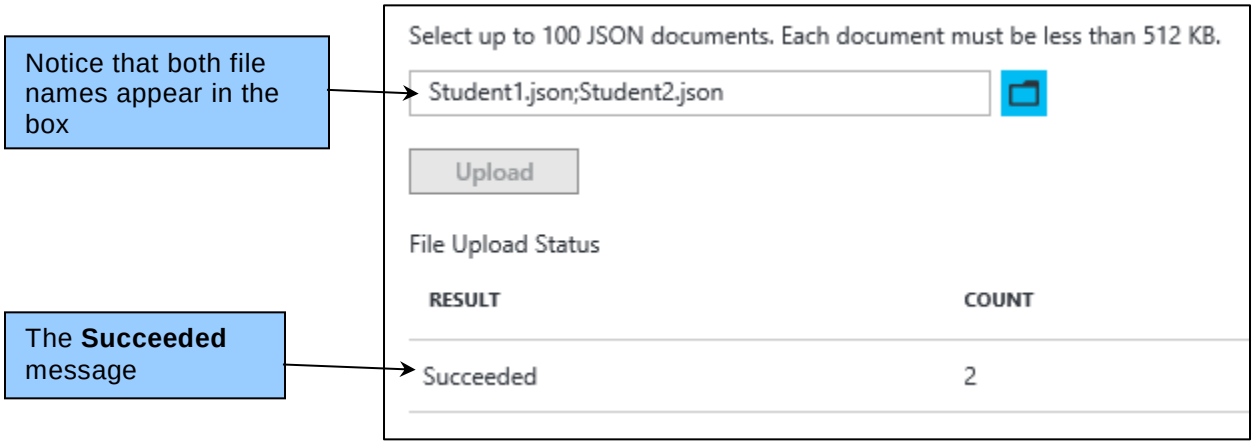


Figure K-43 — Upload Document—File Upload Status

Click the **Upload** button and hopefully you will get the **Succeeded** message for both, as shown in Figure K-43. If not, check the documents carefully for spelling or syntax errors and try again.

The new student ID numbers should also appear in the left pane in the Students collection.
We are almost done. Let's use Azure's Query Explorer to create and run an SQL-like DocumentDB query. You may need to scroll left to see the **Query Explorer** option (under Document Explorer), then select it. Enter in and run the query in Figure K-44. The results screen should appear, as shown in Figure K-45.

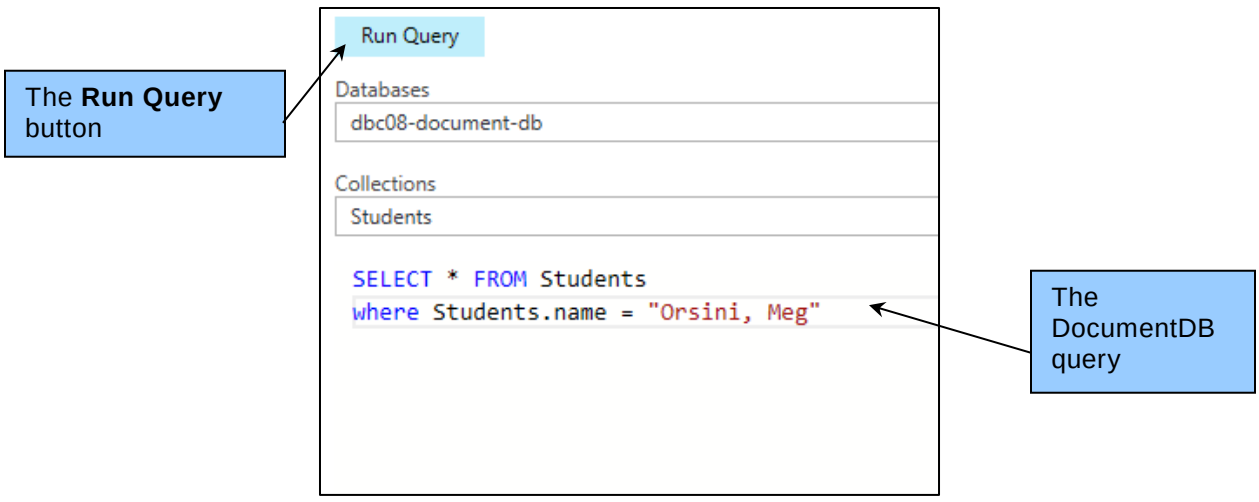


Figure K-44 — Query Explorer—Create Query

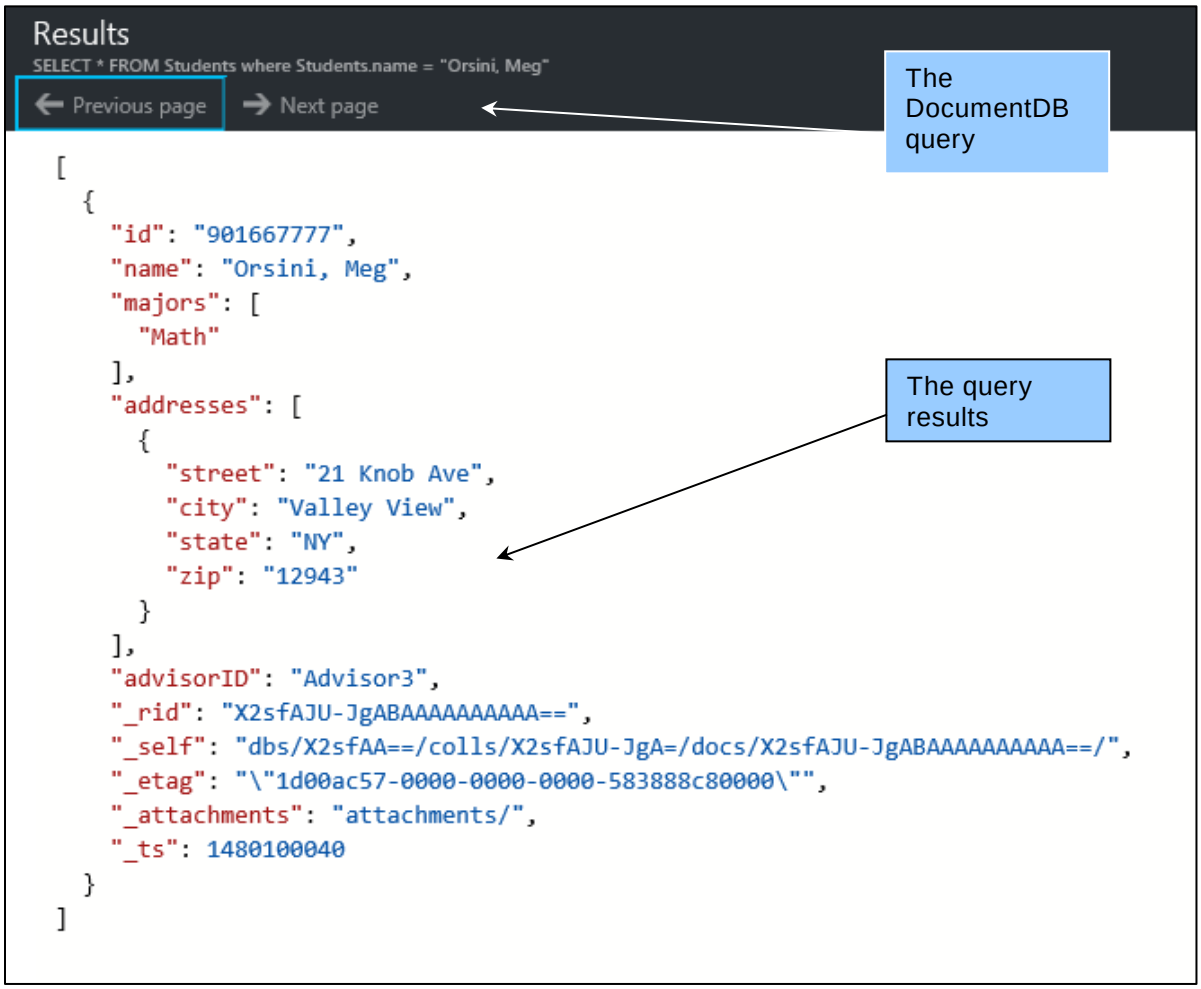


Figure K-45 — Query Explorer—Query Results of First Query

The following query illustrates how the Microsoft Azure DocumentDB query language can structure a JSON document collection in the SELECT clause and access nested collections of data:

```
SELECT {"StudentName": S.name, "StudentWashingtonCity": A.city} AS
      WashStudent
FROM Students S
JOIN A in S.addresses
WHERE A.state = "WA"
```

This query retrieves the name and city of all students with addresses in Washington state. The results are shown in Figure K-46. The result of the query is a JSON list, in which each element is an object with one member called "WashStudent." Each "WashStudent" in turn is an object with two members, called "StudentName" and "StudentWashingtonCity."

At this point, you have created a new DocumentDB database, created a new collection (similar to a RDBMS table), and uploaded two students (similar to rows in a table). You then ran a simple SQL-like query and a more complex query. You may wish to delete the database by right-clicking on the database name in the NoSQL (DocumentDB) pane, or simply log off from Azure by clicking on the account information at the upper-right corner of the screen then selecting Sign out.

Congratulations, you are finished and ready to continue your explorations on your own.

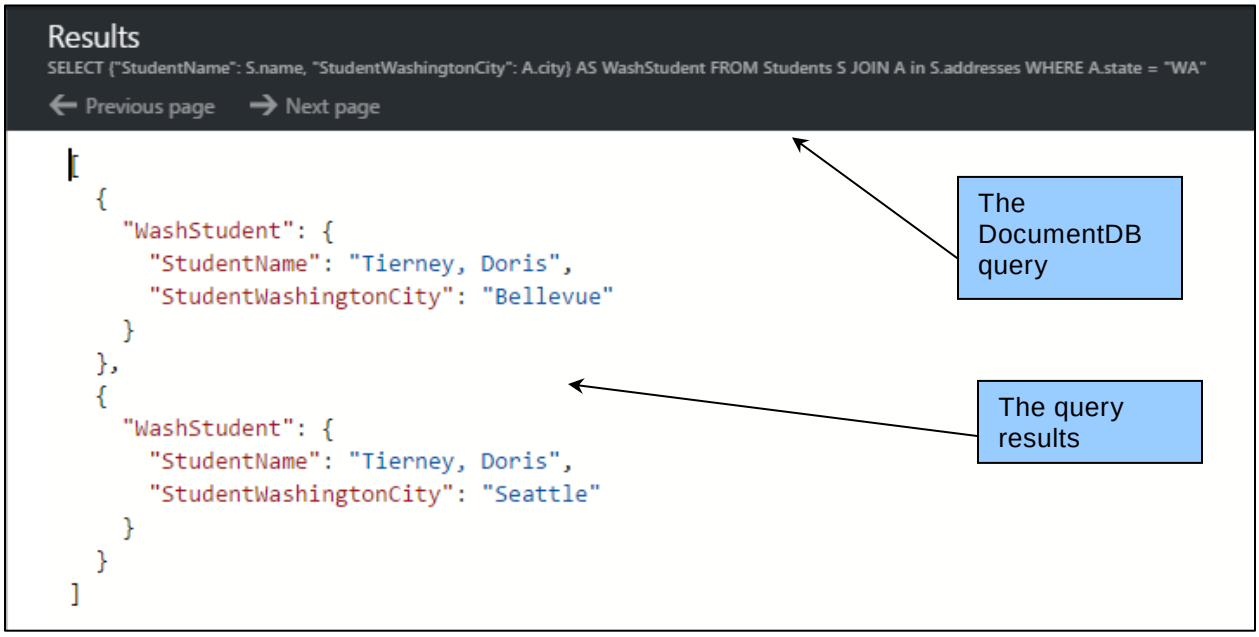


Figure K-46 — Query Explorer—Query Results of Second Query

Summary

The NoSQL movement (now often read as “not only SQL”) is built upon the need to meet the Big Data storage needs of companies such as Amazon.com, Google, and Facebook. The tools used to do this are nonrelational DBMSs known as structured storage or NoSQL databases. An early example was Bigtable, and a more recent popular example is Cassandra.

Many of these systems use data models based on document format standards such as XML or JSON. The confluence of database processing and document processing is one of the most important developments in information systems technology today. Database processing and document processing need each other. Database processing needs document processing for the representation and materialization of database views. Document processing needs database processing for the permanent storage of data.

SGML is as important to document processing as the relational model is to database processing. XML is a series of standards that were developed jointly by the database processing and document processing communities. XML provides a standardized yet customizable way to describe the contents of documents. XML documents can automatically be generated from database data, and database data can be automatically extracted from XML documents.

Although XML can be used to materialize Web pages, this is one of its least important uses. More important is its use for describing, representing, and materializing database views. XML is on the leading edge of database processing; see *www.w3.org* and *www.xml.org* for the latest developments. XML is a better markup language than HTML, primarily because XML provides a clear separation between document structure, content, and materialization. Also, XML tags are not ambiguous and always have a closing tag.

XML documents are transformed when an XSLT processor applies an XSL document to the XML document. A common transformation is to convert the XML document into HTML format. In the future, other transformations will be more important. For example, XSL documents can be written to transform the same Order document into different formats needed by different departments, say, for sales, accounting, or production. XSLT processing is context oriented; given a particular context, an action is taken when a particular item is located. Today, most Web browsers have built-in XSLT processors.

The content of XML documents can be described by the older Document Type Declarations (DTDs) and by XML Schemas. XML Schema is the newer standard for describing the content of an XML document. XML Schema can be used to define custom vocabularies. Documents that conform to an XML Schema are called *schema-valid*. Unlike DTDs, XML Schema documents are themselves XML documents and can be validated against their schema, which is maintained by the W3C.

Schemas consist of elements and attributes. There are two types of elements: simple and complex. Simple elements have one data value. ComplexType elements can have multiple elements nested within them. ComplexTypes may also have attributes. The elements contained in a ComplexType may be simple or other ComplexTypes. ComplexTypes may also define element sequences. XML Schemas (and documents) may have more structure than the columns of a table.

SQL Server, Oracle Database, and MySQL can produce XML documents from database data. The Oracle facilities require the use of Java (see *http://www.oracle.com* for more information).

XML is important because it facilitates the sharing of XML documents (and hence database data) among organizations. After an XML Schema has been defined, organizations can ensure that they are receiving and sending only schema-valid documents. Additionally, XSL documents can be coded to transform any schema-valid XML document, from any source, into other standardized formats. These advantages become even more important as industry groups standardize their own XML Schemas. XML thus facilitates business-to-business processing.

Nonrelational databases can be grouped into Key-value, Document, Column Family, and Graph databases. These products use a non-normalized semi-structured table structure.

Key-value databases use key-value pairs to store data. Document databases typically use data documents represented in XML or JSON. Column family databases use columns, super columns, column families, and super column families to store data. Column families and super column families use RowKeys to identify rows. Graph databases use nodes, properties, and edges. Unlike relational database relationships, edges can also have properties.

Much big data is stored in the cloud and in NoSQL databases, using either XML or JSON for their structuring capabilities. Microsoft Azure is a cloud-based platform that allows us to deploy relational databases using the SQL Server interface. These databases can be created by importing them from existing local databases or by creating them directly in the cloud on Microsoft Azure. The Azure platform also allows creation and querying of DocumentDB databases in JSON format.

KEY TERMS

3V framework	**Apache Cassandra**
aggregate	**attributes**
availability	**Azure DocumentDB**
Big Data	**Bigtable**
BSON (Binary JSON)	**CAP theorem**
Cassandra	**collection**
column	**column family**
complex element	**consistency**
Couchbase Server	**document**

DTD (document type declaration)	**DynamoDB**
edges	**element (simple and complex)**
Extensible Style Language Transformation (XLST)	**graph**
JSON (JavaScript Object Notation)	**keyspace**
key-value	**MemcacheDB,**
namespaces	**Neo4j**
nodes	**NoSQL**
Not only SQL	**Not only SQL movement**
not-type-valid document	**partition tolerance**
polyglot persistence	**properties**
quorum	**replica set**
replication	**Rowkey**
SGML (Standard Generalized Markup Language)	**sharding**
simple element	**SOAP (Simple Object Access Protocol)**
SSMS (Microsoft SQL Server Management Studio)	**stylesheet**
super column	**super column family**
type-valid document	**variety**
velocity	**volume**
www.w3.org/2001/XMLSchema	**XLST (Extensible Style Language:Transformations)**
XML (Extensible Markup Language)	**XML Schema**

REVIEW QUESTIONS

K.1 What is the *NoSQL movement*?

K.2 What is the difference between *sharding* and *replication* in a distributed data store?

K.3 What is Big Data?

K.4 What are the original three Vs? Define each term.

K.5 What is a *quorum*, and what is its purpose for a distributed data store?

K.6 What are the four categories of NoSQL databases used in this book?

K.7 What is the CAP theorem? How has it stood up over time?

K.8 Why do database processing and document processing need each other?

K.9 How are HTML, SGML, and XML related?

K.10 Explain the phrase "standardized but customizable."

K.11 What is SOAP? What did it stand for originally? What does it stand for today?

K.12 What are the problems in interpreting a tag such as <H2> in HTML?

K.13 What is an *aggregate*? What are some of the NoSQL databases that support aggregates?

K.14 Why is it too limiting to say that XML is just the next version of HTML?

K.15 How are XML, XSL, and XSLT related?

K.16 Explain the use of the pattern *{item, action}* in the processing of an XSL document.

K.17 What is the purpose of XML Schema?

K.18 (new) What is the relationship between aggregates and XML or JSON?

K.19 What is a *schema-valid* document?

K.20 Explain the chicken-and-egg problem concerning the validation of XML Schema documents.

K.21 Explain the difference between *simple* and *complex* elements.

K.22 Explain the difference between *elements* and *attributes*.

K.23 Explain, in your own words, why XML is important to database processing.

K.24 Why is XML Schema important for interorganizational document sharing?

K.25 What was the first nonrelational data store to be developed, and who developed it?

K.26 As illustrated in Figure K-8, what is column family database storage and how are such systems organized? How do column family database storage systems compare to RDBMS systems?

K.27 What is a graph database? What are nodes, properties and edges?

K.28 What is a key-value database? Under what circumstances is one most useful? Where does processing relating to the structure of data values take place?

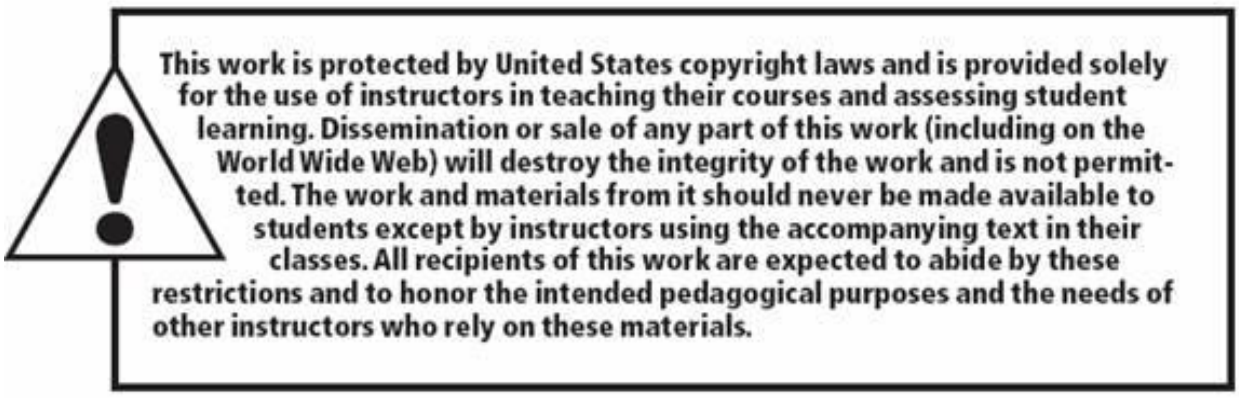
K.29 What are the main features of data in a document database? What are the basic operations and utilities provided by a document DBMS?

EXERCISES

K.30 Develop an XML Schema for a document that uses the data in the HSD SEMINAR, SEMINAR_CUSTOMER, and CUSTOMER tables. If you need more information about XML Schema, go to W3C and read tutorials on XML and XML Schema. Create an example of an XML document for your schema.[12]

K.31 Develop a graph database based on the WP EMPLOYEE and PROJECT tables. Data from the ASSIGNMENT table should appear as edge properties.

K.32 Develop a document database for the data in the HSD SEMINAR, SEMINAR_CUSTOMER, and CUSTOMER tables. Your database should have two collections. Justify your design decisions, and use the JSON syntax used in the examples in this appendix. You do not need to include all data; at a minimum, include data for seminars 2–4 and customers 6–12.



[12] A trial version of Altova's XMLSpy is available at from Altova. This is an excellent program for XML work.